MATLAB for M151B

MATLAB for M151B

P. Howard

Fall 2008

Contents

1 Introduction

1.1 The Origin of MATLAB

MATLAB, which stands for MATrix LABoratory, is a software package developed by MathWorks, Inc. to facillitate numerical computations as well as some symbolic manipulation. The collection of programs (originally written in Fortran) that eventually became MATLAB were developed in the late 1970s by Cleve Moler, who used them in a numerical analysis course he was teaching at the University of New Mexico. Jack Little and Steve Bangert later reprogrammed these routines in C, and added M-files, toolboxes, and more powerful graphics (original versions created plots by printing asterisks on the screen). Moler, Little, and Bangert founded MathWorks in California in 1984.

1.2 Our Course Goal

In recent years mathematical computations have begun to play a larger role in the biological sciences, and MATLAB is a useful platform for carrying out such calculations with relative ease. While our focus will naturally be on concepts that arise in calculus, and on the applications of these concepts, it is also important that students gain some basic familiarity with MATLAB's capabilities as a general computing tool. In particular, we will begin the semester with some background material on arithmetic, algebra, and graphing, and consequently the MATLAB material will lag several weeks behind the class lecture material. For example, while we will begin with limits of functions on the first day of class, we will not get to limits in MATLAB until Week 6.

1.3 Starting MATLAB at Texas A&M University

Your NetID and password should access your calclab account. Log in and click on the six pointed geometric figure in the bottom left corner of your screen. Go to **Mathematics** and choose **Matlab**. Congratulations! (Alternatively, click on the surface plot icon at the foot of your screen.)

1.4 The MATLAB Interface

The (default) MATLAB screen is divided into three windows, with a large Command Window on the right, and two smaller windows stacked one atop the other on the left. (If these windows aren't maximized on your screen, you can maximize them by selecting the middle button at the top right corner of the MATLAB box.) The Command Window is where calculations are carried out in MATLAB, while the smaller windows display information about your current MATLAB session, your previous MATLAB sessions, and your computer account. Your options for these smaller windows are *Command History*, which displays the commands you've typed in from both the current and previous sessions, *Current Directory*, which shows which directory you're currently in and what files are in that directory, and *Workspace*, which displays information about each variable defined in your current session. You can choose which of these options you would like to have displayed by selecting **Desktop**

from the main MATLAB window. Occasionally, it will be important that you are working in a certain directory. Notice that you can change MATLAB's working directory by double-clicking on a directory in the *Current Directory* window. In order to go backwards a directory, click on the folder with a black arrow on it in the top left corner of the *Current Directory* window.

1.5 Basic Computations

At the prompt designated by two arrows, >>, type *sin(0)* and press **Enter**. You should find that the answer has been assigned to the default variable *ans.* Next, type *sin(0);* and hit **Enter** (there is now a semicolon at the end of the line). Notice that the semicolon suppresses screen output.

We will refer to a series of commands as a MATLAB *script*. For example, we might type

```
>>t=4;
>>s=sin(t)
s =
-0.7568
```

where in this example and those that follow, commands typed into MATLAB will be designated by the prompt >>. (Notice that MATLAB assumes that t is in radians, not degrees.) While we're at it, type the up arrow key on your keyboard, and notice that the command *s=sin(t)* comes back up on your screen. Hit the up arrow key again and *t=4;* will appear at the prompt. Using the down arrow, you can scroll back the other way, giving you a convenient way to bring up old commands without retyping them.

Example 1.1. Compute

$$r = \frac{7^3\sqrt{17}}{e^2},$$

where e is the base of the natural logarithm, an irrational number whose value is approximately $e = 2.7182818$.

At the MATLAB prompt, type

```
>>r=7^3*sqrt(17)/exp(2)
r =
191.3946
```

△

1.6 Diary Files

For many of the assignments this semester, you will need to turn in a log of MATLAB commands typed and of MATLAB's responses. (See, for example, Example 1.1.) This is straightforward in MATLAB with the *diary* command.

Example 1.2. Write a MATLAB script that sets $x = 1$ and computes $\tan^{-1} x$ (or $\arctan x$). Save the script to a file called *script1.txt* and print it.

In order to accomplish this, we use the following MATLAB commands.

```
>>diary script1.txt
>>x=1
x =
1
>>atan(1)
ans =
0.7854
>>diary off
```

In this script, the command *diary script.txt* creates the file *script1.txt,* and MATLAB begins recording the commands that follow, along with MATLAB's responses. When the command *diary off* is typed, MATLAB writes the commands and responses to the file *script1.txt.* Commands typed after the *diary off* command will no longer be recorded, but the file *script1.txt* can be reopened either with the command *diary on* or with *diary script1.txt.* Finally, the diary file *script1.txt* can be deleted with the command *delete script1.txt.*

In order to print *script1.txt*, follow the *xprint* instructions posted in the Blocker lab. More precisely, open a terminal window by selecting the terminal icon from the bottom of your screen and use the *xprint* command

xprint -d blocker script1.txt

You will be prompted to give your NetID and password. The file will be printed in Blocker 133. △

1.7 Getting Help

MATLAB has extensive documentation explaining the use of all of its built-in functions (such as *sin*, *atan*, *diary*, etc.). For example, we can look at the help file for the *diary* command by typing *help diary* in the Command Window. More generally, the help documentation can be accessed by typing *helpdesk* in the Command Window. In this case, four options are available (on the upper left corner of the screen): a contents page, an index option (probably the most useful), a search option, and a Demo option (i.e., a list of available deomonstrations).

1.8 Assignments

Use MATLAB to make each of the following calculations. Record your calculations in a diary file and turn in a printout of this file.

1. [2 pts] (Hint: MATLAB understands *pi* as π.)

$$r = \sqrt{1 - \frac{2}{\pi^5}}.$$

2. [2 pts] (Hint: MATLAB's convention is log() for natural log.)

$$r = e^2 \ln 5.$$

3. [2 pts]

$$r = \sin^2 2 + \cos^2 4,$$

with 2 and 4 measured in radians, MATLAB's default.

4. [2 pts]

$$r = \sin^2 30 + \cos^2 40,$$

30 and 40 measured in degrees. (This requires a conversion.)

5. [2 pts] Use MATLAB's built-in help to look up the function *lambertw*. Compute $w(2)$ and write down the equation this value solves.

2 Symbolic Calculations in MATLAB

Though MATLAB has not been designed with symbolic calculations in mind, it can carry them out with the Symbolic Math Toolbox, which is standard with student versions. (In order to check if this, or any other toolbox is on a particular version of MATLAB, type *ver* at the MATLAB prompt.) In carrying out these calculations, MATLAB uses Maple software[1], but the user interface is significantly different.

2.1 Defining Symbolic Objects

Symbolic manipulations in MATLAB are carried out on symbolic variables, which can be either particular numbers or unspecified variables. The easiest way in which to define a variable as symbolic is with the *syms* command.

Example 2.1. Suppose we would like to symbolically define the logistic model

$$R(N) = aN(1 - \frac{N}{K}),$$

where N denotes the number of individuals in a population and R denotes the growth rate of the population. First, we define both the variables and the parameters as symbolic objects, and then we write the equation with standard MATLAB operations:

```
>>syms N R a K
>>R=a*N*(1-N/K)
R =
a*N*(1-N/K)
```

Here, the expressions preceded by >> have been typed at the command prompt and the others have been returned by MATLAB. △

Symbolic objects can also be defined to take on particular numeric values.

Example 2.2. Suppose we want a general form for the logistic model, but we know that the value of K (the "carrying capacity") is 10, and we want to specify this. We can use the following commands:

```
>>K=sym(10)
K =
10
>>R=a*N*(1-N/K)
R =
a*N*(1-1/10*N)
```

[1]Maple is one of a number of alternatives to MATLAB. These alternative packages tend to have some advantages and some disadvanges relative to MATLAB, and this leads to a lot of senseless bickering in the scientific community. FYI.

2.1.1 Complex Numbers

You can also define and manipulate symbolic complex numbers in MATLAB. Recall that a complex number has the form

$$z = x + iy,$$

where x and y are both real numbers and $i = \sqrt{-1}$. The *complex conjugate* of a complex number, denoted $\bar{z}$, is defined by

$$\bar{z} = x - iy.$$

Example 2.3. Suppose we would like to define the complex number $z = x + iy$ and compute z^2 and $z\bar{z}$. We use

```
>>syms x y real
>>z=x+i*y
z =
x+i*y
>>square=expand(z^2)
square =
x^2+2*i*x*y-y^2
>>zzbar=expand(z*conj(z))
zzbar =
x^2+y^2
```

Here, we have particularly specified that x and y be real, as is consistent with complex notation. The built-in MATLAB command *conj* computes the complex conjugate of its input, and the *expand* command is required in order to force MATLAB to multiply out the expressions. (The *expand* command is discussed more below in Subsubsection 2.2.2.)

2.1.2 The *Clear* Command

You can clear variable definitions with the *clear* command. For example, if x is defined as a symbolic variable, you can type *clear x* at the MATLAB prompt, and this definition will be removed. (*Clear* will also clear other MATLAB data types.) If you have set a symbolic variable to be *real*, you will additionally need to use *syms x unreal* or the Maple kernel that MATLAB calls will still consider the variable real.

2.2 Manipulating Symbolic Expressions

Once an expression has been defined symbolically, MATLAB can manipulate it in various ways.

2.2.1 The *Collect* Command

The *collect* command gathers all terms together that have a variable to the same power.

Example 2.4. Suppose that we would like organize the expression

$$f(x) = x(\sin x + x^3)(e^x + x^2)$$

by powers of x. We use

```
>>syms x
>>f=x*(sin(x)+x^3)*(exp(x)+x^2)
f =
x*(sin(x)+x^3)*(exp(x)+x^2)
>>collect(f)
ans =
x^6+exp(x)*x^4+sin(x)*x^3+sin(x)*exp(x)*x
```

△

2.2.2 The *Expand* Command

The *expand* command carries out products by distributing through parentheses, and it also expands logarithmic and trigonometric expressions.

Example 2.5. Suppose we would like to expand the expression

$$f(x) = e^{x+x^2}.$$

We use

```
>>syms x
>>f=exp(x+x^2)
f =
exp(x+x^2)
>>expand(f)
ans =
exp(x)*exp(x^2)
```

△

2.2.3 The *Factor* Command

The *factor* command can be used to factor polynomials.

Example 2.6. Suppose we would like to factor the polynomial

$$f(x) = x^4 - 2x^2 + 1.$$

We use

```
>syms x
>f=x^4-2*x^2+1
f =
x^4-2*x^2+1
>factor(f)
ans =
(x-1)^2*(x+1)^2
```

△

2.2.4 The *Horner* Command

The *horner* command is useful in preparing an expression for repeated numerical evaluation. In particular, it puts the expression in a form that requires the least number of arithmetic operations to evaluate.

Example 2.7. Re-write the polynomial from Example 2.6 in Horner form.

```
>>syms x
>>f=x^4-2*x^2+1
f =
x^4-2*x^2+1
>>horner(f)
ans =
1+(-2+x^2)*x^2
```

2.2.5 The *Simple* Command

The *simple* command takes a symbolic expression and re-writes it with the least possible number of characters. (It runs through MATLAB's various manipulation programs such as *collect*, *expand*, and *factor* and returns the result of these that has the least possible number of characters.)

Example 2.8. Suppose we would like a reduced expression for the function

$$f(x) = (1 + \frac{1}{x} + \frac{1}{x^2})(1 + x + x^2).$$

We use

```
>>syms x f
>>f=(1+1/x+1/x^2)*(1+x+x^2)
f =
(1+1/x+1/x^2)*(x+1+x^2)
>>simple(f)
simplify:
(x+1+x^2)^2/x^2
radsimp:
(x+1+x^2)^2/x^2
combine(trig):
(3*x^2+2*x+2*x^3+1+x^4)/x^2
factor:
(x+1+x^2)^2/x^2
expand:
2*x+3+x^2+2/x+1/x^2
combine:
(1+1/x+1/x^2)*(x+1+x^2)
convert(exp):
(1+1/x+1/x^2)*(x+1+x^2)
convert(sincos):
(1+1/x+1/x^2)*(x+1+x^2)
convert(tan):
(1+1/x+1/x^2)*(x+1+x^2)
collect(x):
2*x+3+x^2+2/x+1/x^2
mwcos2sin:
(1+1/x+1/x^2)*(x+1+x^2)
ans =
(x+1+x^2)^2/x^2
```

In this example, three lines have been typed, and the rest is MATLAB output as it tries various possibilities. It returns the expression in *ans*, in this case from the *factor* command. △

2.2.6 The *Pretty* Command

MATLAB's *pretty* command simply re-writes a symbolic expression in a form that appears more like typeset mathematics than does MATLAB syntax.

Example 2.9. Suppose we would like to re-write the expression from Example 2.8 in a more readable format. Assuming, we have already defined f as in Example 2.8, we use *pretty(f)* at the MATLAB prompt. (The output of this command doesn't translate well into a printed document, so I won't give it here.)

2.3 Solving Algebraic Equations

MATLAB's built-in function for solving equations symbolically is *solve*.

Example 2.10. Suppose we would like to solve the quadratic equation

$$ax^2 + bx + c = 0.$$

We use

```
>>syms a b c x
>>eqn=a*x^2+b*x+c
eqn =
a*x^2+b*x+c
>>roots=solve(eqn)
roots =
1/2/a*(-b+(b^2-4*a*c)^(1/2))
1/2/a*(-b-(b^2-4*a*c)^(1/2))
```

Observe that we only defined the expression on the left-hand side of our equality. By default, MATLAB's *solve* command sets this expression to 0. Also, notice that MATLAB knew which variable to solve for. (It takes x as a default variable.) Suppose that in lieu of solving for x, we know x and would like to solve for a. We can specify this with the following commands:

```
>>a=solve(eqn,a)
a =
-(b*x+c)/x^2
```

In this case, we have particularly specified in the *solve* command that we are solving for a. Alternatively, we can type an entire equation directly into the solve command. For example:

```
>>syms a
>>roots=solve(a*x^2+b*x+c)
roots =
1/2/a*(-b+(b^2-4*a*c)^(1/2))
1/2/a*(-b-(b^2-4*a*c)^(1/2))
```

Here, the *syms* command has been used again because a has been redefined in the code above. Finally, we need not first make our variables symbolic if we put the expression in *solve* in single quotes. We could simply use *solve('a*x^2+b*x+c')*. △

MATLAB's *solve* command can also solve systems of equations.

Example 2.11. For a population of prey x with growth rate R_x and a population of predators y with growth rate R_y, the the Lotka–Volterra predator–prey model is

$$\begin{aligned} R_x &= ax - bxy \\ R_y &= -cy + dxy. \end{aligned}$$

In this example, we would like to determine whether or not there is a pair of population values (x, y) for which neither population is either growing or decaying (the rates are both 0). We call such a point an equilibrium point. The equations we need to solve are:

$$\begin{aligned} 0 &= ax - bxy \\ 0 &= -cy + dxy. \end{aligned}$$

In MATLAB

```
>>syms a b c d x y
>>Rx=a*x-b*x*y
Rx =
a*x-b*x*y
>>Ry=-c*y+d*x*y
Ry =
-c*y+d*x*y
>>[prey pred]=solve(Rx,Ry)
prey =
0
1/d*c
pred =
0
1/b*a
```

Again, MATLAB knows to set each of the expression Rx and Ry to 0. In this case, MATLAB has returned two solutions, one with $(0, 0)$ and one with $(\frac{c}{d}, \frac{a}{b})$. In this example, the appearance of *[prey pred]* particularly requests that MATLAB return its solution as a vector with two components. Alternatively, we have the following:

```
>>pops=solve(Rx,Ry)
pops =
x: [2x1 sym]
y: [2x1 sym]
>>pops.x
ans =
0
1/d*c
>>pops.y
ans =
0
1/b*a
```

In this case, MATLAB has returned its solution as a MATLAB *structure*, which is a data array that can store a combination of different data types: symbolic variables, numeric values, strings etc. In order to access the value in a *structure*, the format is

```
structure_name.variable_identification
```

△

2.4 Numerical Calculations with Symbolic Expressions

In many cases, we would like to combine symbolic manipulation with numerical calculation.

2.4.1 The *Double* and *Eval* commands

The *double* and *eval* commands change a symbolic variable into an appropriate double variable (i.e., a numeric value).

Example 2.12. Suppose we would like to symbolically solve the equation $x^3 + 2x - 1 = 0$, and then evaluate the result numerically. We use

```
>>syms x
>>r=solve(x^3+2*x-1);
>>eval(r)
ans =
0.4534
-0.2267 + 1.4677i
-0.2267 - 1.4677i
>>double(r)
ans =
0.4534
-0.2267 + 1.4677i
-0.2267 - 1.4677i
```

MATLAB's symbolic expression for r is long, so I haven't included it here, but you should take a look at it by leaving the semicolon off the *solve* line. △

2.4.2 The Subs Command

In any symbolic expression, values can be substituted for symbolic variables with the *subs* command.

Example 2.13. Suppose that in our logistic model

$$R(N) = aN(1 - \frac{N}{K}),$$

we would like to substitute the values $a = .1$ and $K = 10$. We use

```
>>syms a K N
>>R=a*N*(1-N/K)
R =
a*N*(1-N/K)
>>R=subs(R,a,.1)
R =
1/10*N*(1-N/K)
>>R=subs(R,K,10)
R =
1/10*N*(1-1/10*N)
```

Alternatively, numeric values can be substitued in. We can accomplish the same result as above with the commands

```
>>syms a K N
>>R=a*N*(1-N/K)
R =
a*N*(1-N/K)
>>a=.1
a =
0.1000
>>K=10
K =
10
>>R=subs(R)
R =
1/10*N*(1-1/10*N)
```

In this case, the specifications $a = .1$ and $K = 10$ have defined a and K as numeric values. The *subs* command, however, places them into the symbolic expression.

2.5 Assignments

For each problem, turn in a diary file containing your MATLAB script along with MATLAB's output.

1. [2 pts] Factor the polynomial

$$x^5 - 15x^4 + 85x^3 - 225x^2 + 274x - 120.$$

2. [2 pts] Find an inverse for the function

$$f(x) = \frac{1-x}{1+x}; \quad x \neq -1.$$

3. [2 pts] Solve the equation

$$x - \frac{y}{x} - y^2 = 0$$

for y as a function of x. (Hint. This is a good problem to use the *pretty* command on.)

4. [2 pts] Find all solutions for the system of algebraic equations

$$\begin{aligned} x^2 - y^2 &= 0 \\ 2y - x &= 1. \end{aligned}$$

5. [2 pts] Write down expressions for the three solutions for

$$ax^3 + c = 0.$$

Use the *subs* command to evaluate your solution for $a = 1$ and $c = 2$.

3 Plots and Graphs in MATLAB

3.1 The *Plot* Command

The primary tool we will use for plotting in MATLAB is *plot()*.

Example 3.1. Plot the line that passes through the points $\{(1,4),(3,6)\}$.

We first define the x values (1 for the first point and 3 for the second) as a single variable $x = (1,3)$ (typically referred to as a *vector*) and the y values as the vector $y = (4,6)$, and then we plot these points, connecting them with a line. The following commands (accompanied by MATLAB's output) suffice:

```
>>x=[1 3]
x =
     1     3
>>y=[4 6]
y =
     4     6
>>plot(x,y)
```

The output we obtain is the plot given as Figure 3.1.

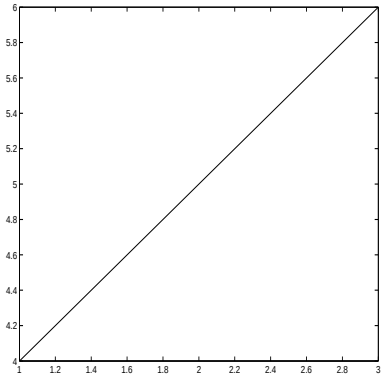

Figure 3.1: A very simple linear plot.

In MATLAB it's particularly easy to decorate a plot. For example, minimize your plot by clicking on the left button on the upper right corner of your window, then add the following lines in the Command Window:

```
>>xlabel('Here is a label for the x-axis')
>>ylabel('Here is a label for the y-axis')
>>title('Useless Plot')
>>axis([0 4 2 10])
```

The only command here that needs explanation is the last. It simply tells MATLAB to plot the x-axis from 0 to 4, and the y-axis from 2 to 10. If you now click on the plot's button at the bottom of the screen, you will get the labeled figure, Figure 3.2.

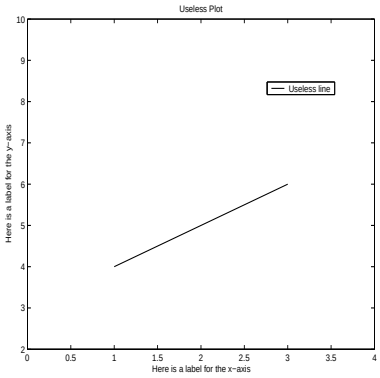

Figure 3.2: A still pretty much ridiculously simple linear plot.

I added the legend after the graph was printed, using the menu options. Notice that all this labeling can be carried out and edited from these menu options. After experimenting a little, your plots will be looking great (or at least better than the default-setting figures displayed here). Not only can you label and detail your plots, you can write and draw on them directly from the MATLAB window. One warning: If you retype *plot(x,y)* after labeling, MATLAB will think you want to start over and will give you a clear figure with nothing except the line. To get your labeling back, use the up arrow key to scroll back through your commands and re-issue them at the command prompt. (Unless you labeled your plots using menu options, in which case you're out of luck, though this might be a good time to consult Section 3.7 on saving plots.) △

Defining vectors as in the example above can be tedious if the vector has many components, so MATLAB has a number of ways to shorten your work. For example, you might try:

```
>>X=1:9
X =
    1    2    3    4    5    6    7    8    9
>>X=0:2:10
X =
    0    2    4    6    8   10
```

3.2 Plotting Functions with the *plot* command

In order to plot a function with the *plot* command, we proceed by evaluating the function at a number of x-values $x_1, x_2, ..., x_n$ and drawing a curve that passes through the points $\{(x_k, y_k)\}_{k=1}^{n}$, where $y_k = f(x_k)$.

Example 3.2. Use the *plot* command to plot the function $f(x) = x^2$ for $x \in [0, 1]$.

First, we will partition the interval [0,1] into twenty evenly spaced points with the command, *linspace(0, 1, 20)*. (The command *linspace(a,b,n)* defines a vector with n evenly spaced points, beginning with left endpoint a and terminating with right endpoint b.) Then at each point, we will define f to be x^2. We have

```
>>x=linspace(0,1,20)
x =
  Columns 1 through 8
         0    0.0526    0.1053    0.1579    0.2105    0.2632    0.3158    0.3684
  Columns 9 through 16
    0.4211    0.4737    0.5263    0.5789    0.6316    0.6842    0.7368    0.7895
  Columns 17 through 20
    0.8421    0.8947    0.9474    1.0000
>>f=x.^2
f =
  Columns 1 through 8
         0    0.0028    0.0111    0.0249    0.0443    0.0693    0.0997    0.1357
  Columns 9 through 16
    0.1773    0.2244    0.2770    0.3352    0.3989    0.4681    0.5429    0.6233
  Columns 17 through 20
    0.7091    0.8006    0.8975    1.0000
>>plot(x,f)
```

Only three commands have been typed; MATLAB has done the rest. One thing you should pay close attention to is the line f=x.^2, where we have used the *array operation* .^. This operation .^ signifies that the vector x is not to be squared (it's not entirely clear at this point what we might even mean by the square of a vector), but rather that each component of x is to be squared and the result is to be defined as a component of f, another vector. Similar commands are .* and ./. These are referred to as *array operations,* and you will need to become comfortable with their use. △

Example 3.3. In our section on symbolic algebra, we encountered the logistic population model, which relates the number of individuals in a population N with the rate of growth of the population R through the relationship

$$R(N) = aN(1 - \frac{N}{K}) = -\frac{a}{K}N^2 + aN.$$

Taking $a = 1$ and $K = 10$, we have

$$R(N) = -.1N^2 + N.$$

In order to plot this for populations between 0 and 20, we use the following MATLAB code, which creates Figure 3.3.

```
>>N=linspace(0,20,1000);
>>R=-.1*N.^2+N;
>>plot(N,R)
```

Observe that the rate of growth is positive until the population achieves its "carrying capacity" of $K = 10$ and is negative for all populations beyond this. In this way, if the population is initially below its carrying capacity, then it will increase toward its carrying capacity, but will never exceed it. If the population is initially above the carrying capacity, it will decrease toward the carrying capacity. The carrying capacity is interpreted as the maximum number of individuals the environment can sustain. △

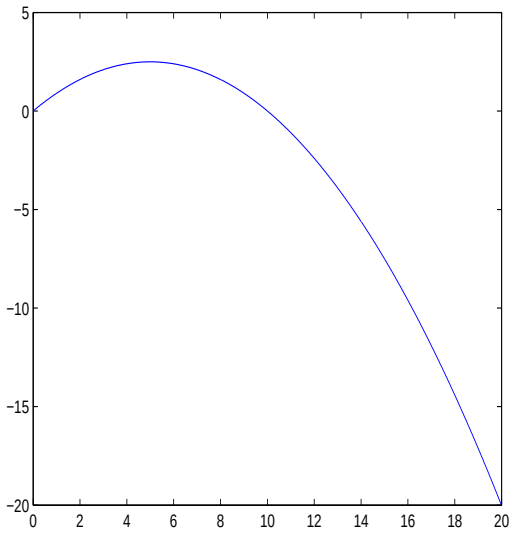

Figure 3.3: Growth rate for the logistic model.

3.3 Parametric Curves

In certain cases the relationship between x and y can be described in terms of a third variable, say t. In such cases, t is a parameter, and we refer to a plot of the points (x, y) as a parametric curve.

Example 3.4. Plot a curve in the x-y plane corresponding with $x(t) = t^2 + 1$ and $y(t) = e^t$, for $t \in [-1, 1]$. One way to accomplish this is through solving for t in terms of x and substituing your result into $y(t)$ to get y as a function of x. Here, rather, we will simply get values of x and y at the same values of t. Using semicolons to suppress MATLAB's output, we use the following script, which creates Figure 3.4.

```
>>t=linspace(-1,1,100);
>>x=t.^2 + 1;
>>y=exp(t);
>>plot(x,y)
```

△

3.4 Juxtaposing One Plot On Top of Another

Example 3.5. For the functions $x(t) = t^2 + 1$ and $y(t) = e^t$, plot $x(t)$ and $y(t)$ on the same figure, both versus t.

The easiest way to accomplish this is with the single command

```
>>plot(t,x,t,y);
```

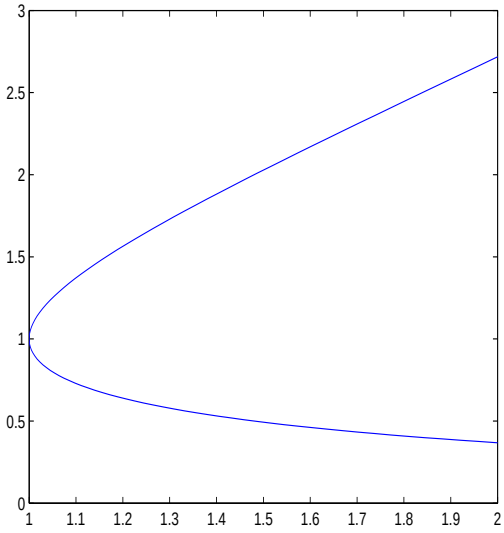

Figure 3.4: Plot of $x(t) = t^2 + 1$ and $y(t) = e^t$ for $t \in [-1, 1]$.

The color and style of the graphs can be specified in single quotes directly after the pair of values. For example, if we would like the plot of $x(t)$ to be red, and the plot of $y(t)$ to be green and dashed, we would use

```
>>plot(t,x,'r',t,y,'g–')
```

For more information on the various options, type *help plot*.

Another way to accomplish this same thing is through the *hold on* command. After typing *hold on*, further plots will be typed one over the other until the command *hold off* is typed. For example,

```
>>plot(t,x)[2]
>>hold on
>>plot (t,y)
>>title('One plot over the other')
>>u=[-1 0 1];
>>v=[1 0 -1]
>>plot(u,v)
```

△

3.5 Multiple Plots

Often, we will want MATLAB to draw two or more plots at the same time so that we can compare the behavior of various functions.

[2]If a plot window pops up here, minimize it and bring it back up at the end.

Example 3.6. Plot the three functions $f(x) = x$, $g(x) = x^2$, and $h(x) = x^3$. The following sequence of commands produces the plot given in Figure 3.5.

```
>>x = linspace(0,1,20);
>>f = x;
>>g = x.^2;
>>h = x.^3;
>>subplot(3,1,1);
>>plot(x,f);
>>subplot(3,1,2);
>>plot(x,g);
>>subplot(3,1,3);
>>plot(x,h);
```

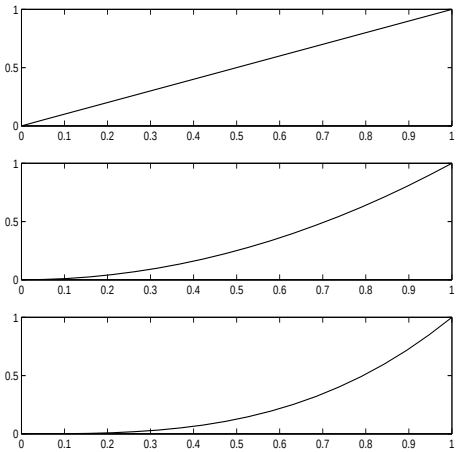

Figure 3.5: Algebraic functions on parade.

The only new command here is *subplot(m,n,p)*. This command creates m rows and n columns of graphs and places the current figure in position p (counted left to right, top to bottom).

3.6 Ezplot

In most of our plotting for M151B, we will use the *plot* command, but another option is the built-in function *ezplot*, which can be used along with symbolic variables.

Example 3.7. Plot the function

$$f(x) = x^4 + 2x^3 - 7x^2.$$

We can use

```
>>syms f x
>>f=x^4+2*x^3-7*x
f =
x^4+2*x^3-7*x
>>ezplot(f)
```

In this case, MATLAB chooses appropriate axes, and we obtain the plot in Figure 3.6.

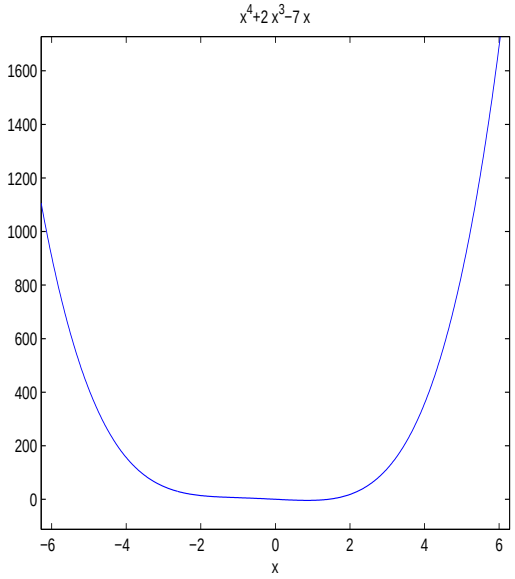

Figure 3.6: Default plot from *ezplot*.

We can also specify the domain on which to plot with *ezplot(f,xmin,xmax).* For example, *ezplot(f,-1,1)* creates Figure 3.7.

Alternatively, the variables need not be defined symbolically if they are placed in single quotes. We could also plot this example using respectively

```
>>ezplot('x^4+2*x^3-7*x')
```

or

```
>>ezplot('x^4+2*x^3-7*x',-1,1)
```

△

The ezplot command can also be a good way for plotting implicitly defined relations, by which we mean relations between x and y than cannot be solved for one variable in terms of the other.

Example 3.8. Plot y versus x given the relation

$$\frac{x^2}{9} + \frac{y^2}{4} = 1.$$

This is, of course, the equation of an ellipse, and it can be plotted by separately graphing each of the two solution curves

$$y = \pm 2\sqrt{1 - \frac{x^2}{9}}.$$

Alternatively, we can use the following single command to create Figure 3.8.

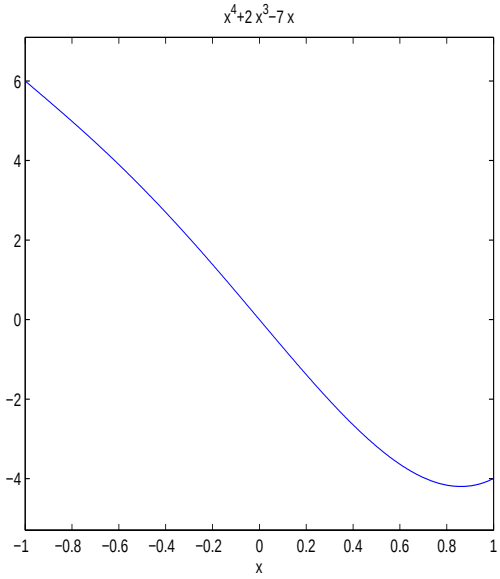

Figure 3.7: Domain specified plot with *ezplot*.

```
>>ezplot('xˆ2/9+yˆ2/4=1',[-3,3],[-2,2])
```

Here, observe that the first interval specifies the values of x and the second specifies the values for y. △

Finally, we can use *ezplot* to plot parametrically defined relations.

Example 3.9. Use *ezplot* to plot y versus x, given $x(t) = t^2+1$ and $y(t) = e^t$, for $t \in [-1, 1]$.

We can accomplish this with the single command

```
>>ezplot('tˆ2+1','exp(t)',[-1,1])
```

△

3.7 Saving Plots as Encapsulated Postscript Files

In order to print a plot, first save it as an encapsulated postscript file. From the options in your graphics box, choose **File**, **Save As**, and change **Save as type** to **EPS file**. Finally, click on the **Save** button. The plot can now be printed using the *xprint* command.

Once saved as an encapsulated postscript file, the plot cannot be edited, so it should also be saved as a MATLAB figure. This is accomplished by choosing **File**, **Save As**, and saving the plot as a .fig file (which is MATLAB's default).

3.8 Assignments

For each of these assignments, turn in the indicated plot.

1. [2 pts] Use the *plot* command to plot the function $f(x) = x + \sin x$ for $x \in [0, 2\pi]$. Label your x and y axes and add a title to your plot.

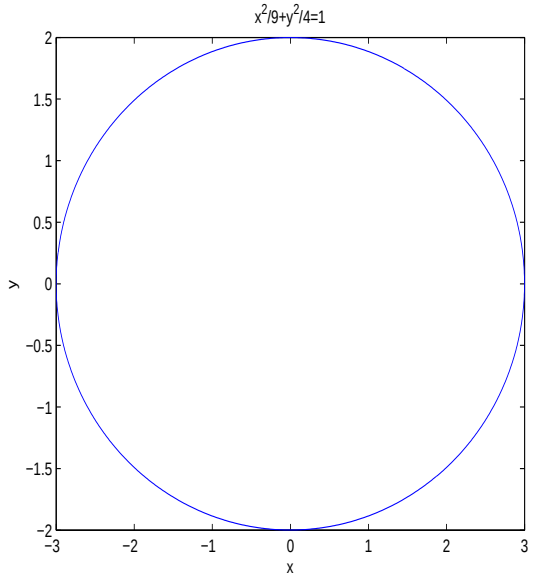

Figure 3.8: The ellipse described by $\frac{x^2}{9} + \frac{y^2}{4} = 1$.

2. [2 pts] Use the *plot* command to plot the parametric curve described by the functions $x(t) = \tan t$ and $y(t) = \cos t$ for $t \in (-\frac{\pi}{4}, \frac{\pi}{4})$.

3. [2 pts] The Gompertz population model has the form

$$R(N) = -aN \ln(\frac{N}{K}),$$

where as in the logistic model N denotes the number of individuals in a population and R denotes the rate of growth of the population. For $a = 1$ and $K = 10$, and for $N \in [0, 20]$ use the *plot* command to plot the logistic model and the Gompertz model on the same figure. Which model has a higher maximum growth rate?

4. [2 pts] Use the *plot* command to create a stacked plot with $f(x) = \sin x$ on the top plot and $f(x) = \cos x$ on the bottom plot. Take $x \in [0, 2\pi]$.

5. [2 pts] Use the *ezplot* command to plot the hyperbola described by the equation

$$\frac{x^2}{9} - \frac{y^2}{4} = 1.$$

Take $x \in [-10, 10]$ and $y \in [-6, 6]$.

4 Semilog and Double-log Plots

In many applications, the values of data points can range significantly, and it can become convenient to work with $\log_{10}$ values of the original data. In such cases, we often work with *semilog* or *double-log* (or *log-log*) plots.

4.1 Semilog Plots

Consider the following data (real and estimated) for world populations in certain years.

Year	Population
-4000	7×10^6
-2000	2.7×10^7
1	1.7×10^8
2000	6.1×10^9

We can plot these values in MATLAB with the following commands, which produce Figure 4.1.

```
>>years=[-4000 -2000 1 2000];
>>pops=[7e+6 2.7e+7 1.7e+8 6.1e+9];
>>plot(years,pops,'o')
```

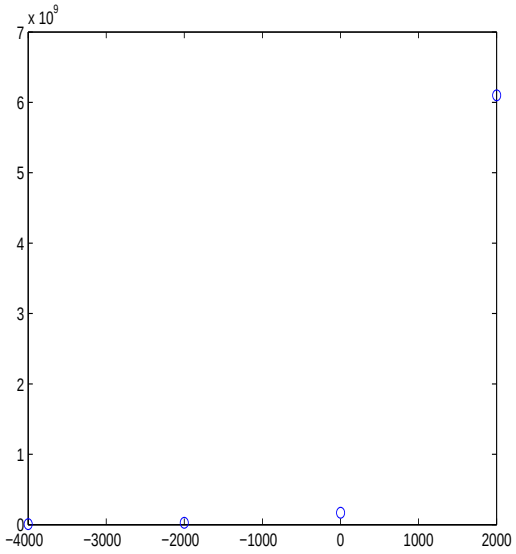

Figure 4.1: Standard plot for populations versus year.

Looking at Figure 4.1, we immediately see a problem: the final data point is so large that the remaining points are effectively zero on the scale of our graph. In order to overcome this

problem, we can take a base 10 logarithm of each of the population values. That is,

$$\begin{aligned}
\log_{10} 7 \times 10^6 &= \log_{10} 7 + 6 \\
\log_{10} 2.7 \times 10^7 &= \log_{10} 2.7 + 7 \\
\log_{10} 1.7 \times 10^8 &= \log_{10} 1.7 + 8 \\
\log_{10} 6.1 \times 10^9 &= \log_{10} 6.1 + 9.
\end{aligned}$$

We can plot these new values with the following commands.

```
>>logpops=log10(pops);
>>plot(years,logpops,'o')
```

In this case, we obtain Figure 4.2.

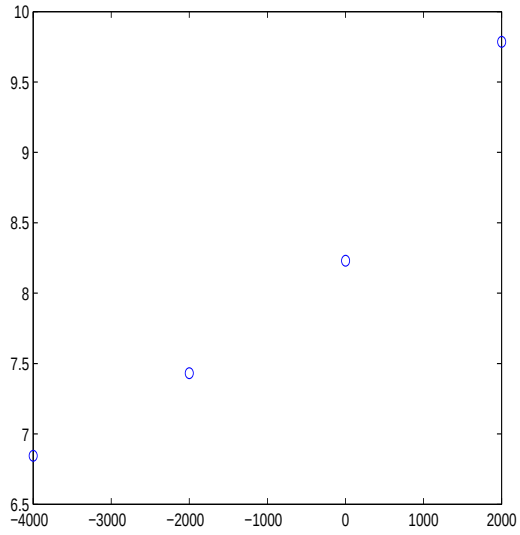

Figure 4.2: Plot of the log of populations versus years.

We can improve this slightly with MATLAB's built-in function *semilogy*. This function carries out the same calculation we just did, but MATLAB adds appropriate marks on the vertical axis to make the scale easier to read. We use

```
>>semilogy(years,pops,'o')
```

The result is shown in Figure 4.3. Observe that there are precisely eight marks in Figure 4.3 between 10^7 and 10^8. The first of these marks 2×10^7, the second 3×10^7 etc. up to the eighth, which is 9×10^7. At that point, we have reached the mark for 10^8.

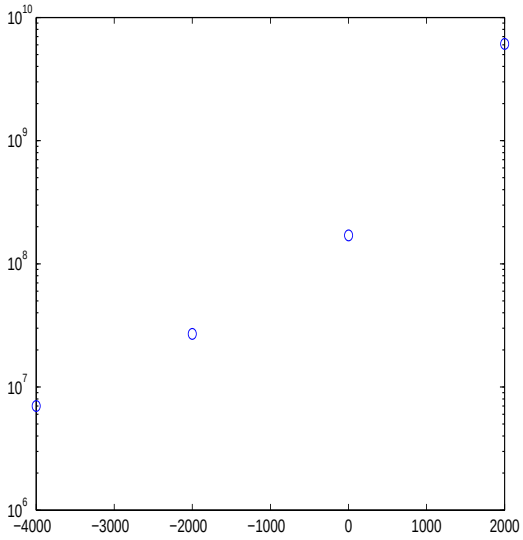

Figure 4.3: Semilog plot of world population data.

4.1.1 Deriving Functional Relations from a Semilog Plot

Having plotted our population data, suppose we would like to find a relationship of the form

$$N = f(x),$$

where N denotes the number of individuals in the population during year x. We proceed by observing that the four points in Figure 4.3 all lie fairly close to the same straight line. In Section **??**, we will discuss how calculus can be used to find the exact form for such a line, but for now we simply allow MATLAB to carry out the computation. From the graphics window for Figure 4.2 (the figure created prior to the use of *semilogy*), choose **Tools**, **Basic Fitting**. From the **Basic Fitting** menu, choose a **Linear** fit and check the box next to **Show Equations**. This produces Figure 4.4.

This line suggests that the relationship between N and x is

$$\log_{10} N = .00048x + 8.6.$$

(Recall that we obtained this figure by taking $\log_{10}$ of our data.) Taking each side of this last expression as an exponent for the base 10, we find

$$10^{\log_{10} N} = 10^{.00048x+8.6} = 10^{.00048x}10^{8.6}.$$

We conclude with the functional relation

$$N(x) = 10^{.00048x}10^{8.6},$$

which is the form we were looking for.

Finally, we note that MATLAB's built-in function *semilogx* plots the x-axis on a logarithmic scaling while leaving the y-axis in its original form.

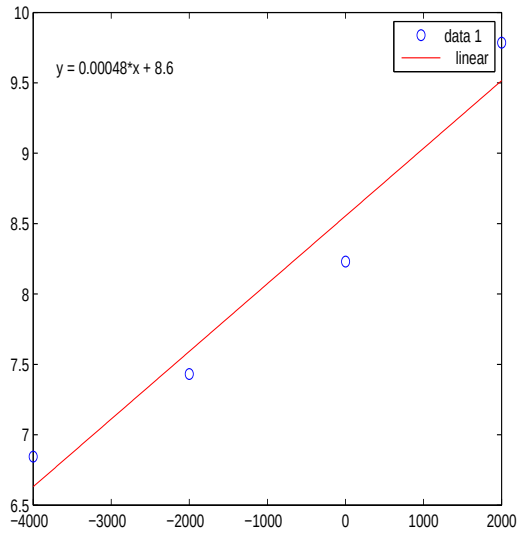

Figure 4.4: Best line fit for the population data.

4.2 Double-log Plots

In the case that we take the base 10 logarithm of both variables in the problem, we say that the plot is a *double-log* or *log-log* plot.

Example 4.1. In certain cases, the number of plants in an area will decrease as the average size of the individual plants increases. (Since each plant is using more resources, fewer plants can be sustained.) In order to find a quantitative relationship between the number of plants N and the average plant size S, consider the data given in Table 1.

N	S
1	10000
10	316.23
50	28.28
100	10

Table 1: Number of plants N and average plant size S.

In this case, we will find a relationship between N and S of the form

$$S = f(N).$$

We proceed by taking the base 10 logarithm of all the data and creating a double-log plot of the resulting values. The following MATLAB code produces Figure 4.5.

```
>>N=[1 10 50 100];
>>S=[10000 316.23 28.28 10];
>>loglog(N,S)
```

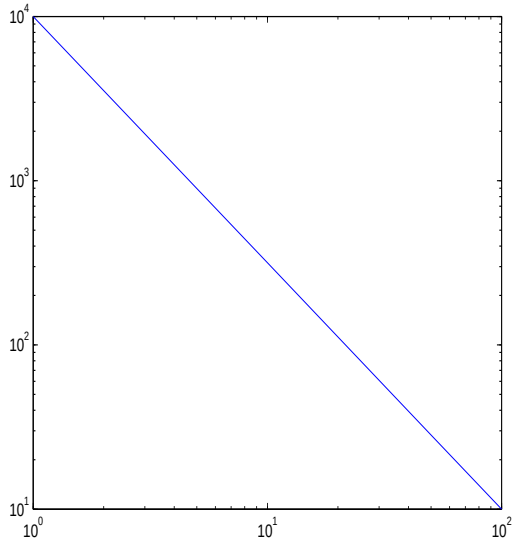

Figure 4.5: Double-log plot of average plant size S versus number of plants N.

Since the graph of the data is a straight line in this case,[3] we can compute the slope and intercept from standard formulas. In standard slope-intercept form, we can write the equation for our line as

$$\log_{10} S = m \log_{10} N + b.$$

The slope is

$$m = \frac{y_2 - y_1}{x_2 - x_1},$$

where (x_1, y_1) and (x_2, y_2) denote two points on the line, and b is the value of $\log_{10} S$ when $N = 1$ (because $\log_{10} 1 = 0$). In reading the plot, notice that values 10^k should be interpreted simply as k. That is,

$$m = \frac{4 - 1}{0 - 2} = -\frac{3}{2},$$

and

$$b = 4.$$

We conclude

$$\log_{10} S = -\frac{3}{2} \log_{10} N + 4.$$

In order to get a functional relationship of the type we are interested in, we take each side of this last expression as an exponent for the base 10. That is,

$$10^{\log_{10} S} = 10^{-\frac{3}{2} \log_{10} N + 4} = 10^{\log N^{-\frac{3}{2}}} 10^4 \Rightarrow S = 10^4 N^{-\frac{3}{2}}.$$

In practice, the multiplication factor 10^4 varies from situation to situation, but the power law $N^{-\frac{3}{2}}$ is fairly common. We often write

$$S \propto N^{-\frac{3}{2}}.$$

[3]Cooked up, admittedly, though the relationship we'll get in the end is fairly general.

△

4.3 Assignments

For each of the following data sets find a relationship between x and y of the form

$$y = f(x).$$

1. [3 pts]

x	y
0	5.000
10	5.0061
100	5.0609
1000	5.6430

Table 2: Data for Problem 1.

2. [3 pts]

x	y
1	-.5
10	0
100	.5

Table 3: Data for Problem 2.

3. [4 pts]

x	y
2	7.9370
3	10.4004
4	12.5992
5	14.6201

Table 4: Data for Problem 3.

5 Inline Functions and M-Files

Functions can be defined in MATLAB either in line (that is, at the command prompt) or as M-files (separate text files).

5.1 Inline Functions

Example 5.1. Define the function $f(x) = e^x$ in MATLAB and compute $f(1)$.

We can accomplish this, as follows, with MATLAB's built-in *inline* function.

```
>>f=inline('exp(x)')
>>f(1)
ans =
2.7183
```

Observe, in particular, the difference between $f(1)$ when f is a function and $f(1)$ when f is a vector: if f is a vector, then $f(1)$ is the first component of f, *not* the function f evaluated at 1. △

In a similar manner, we can define a function of several variables.

Example 5.2. Define the function $f(x, y) = x^2 + y^2$ in MATLAB and compute $f(1, 2)$.

In this case, we use

```
>>f=inline('x^2 + y^2','x','y')
f =
    Inline function:
    f(x,y) = x^2 + y^2
>>f(1,2)
ans =
    5
```

Notice that in the case of multiple variables we specify the order in which the variables will appear as arguments of f. Compare the previous code with the following, in which MATLAB expects y as the first input of f and x as the second.[4]

```
f=inline('x^2+y^2','y','x')
f =
Inline function:
f(y,x) = x^2+y^2
```

△

In many cases we would like to define functions that use MATLAB's array operations .^, .*, and ./. This can be accomplished either by typing the array operations in by hand or by using the *vectorize* command.

Example 5.3. Define the function $f(x) = x^2$ in MATLAB in such a way that MATLAB can take vector input and return vector output. Compute $f(x)$ if x is the vector $x = [1, 2]$.

We use

[4] Granted, in this example order doesn't matter.

```
>>f=inline(vectorize('x^2'))
f =
     Inline function:
     f1(x) = x.^2
>>x=[1 2]
x =
     1     2
>>f(x)
ans =
     1     4
```

△

Finally, in some cases it is convenient to define an inline function when the variables are symbolic. Since the *inline* function expects a string, or character, as input, we first convert the symbolic expression into a string expression.

Example 5.4. Compute the inverse of the function

$$f(x) = \frac{1}{x+1}, \quad x > -1,$$

and define the result as a MATLAB inline function. Compute $f^{-1}(5)$.

We use

```
>>finv=solve('1/(x+1)=y')
finv =
-(y-1)/y
>>finv=inline(char(finv))
finv =
Inline function:
finv(y) = -(y-1)/y
>>finv(5)
ans =
-0.8000
```

Observe that the variable *finv* is originally defined symbolically even though the expression MATLAB solves is given as a string. The *char* command converts *finv* into a string, which is appropriate as input for *inline*. △

Inline functions can be plotted with either the *ezplot* command or the *fplot* (function plot) command.

Example 5.5. Define the function $f(x) = x + \sin x$ as an inline function and plot if for $x \in [0, 2\pi]$ using first the *ezplot* command and second the *fplot* command.

The following commands create, respectively, Figure 5.1 and Figure 5.2.

```
>>f=inline('x+sin(x)')
f =
Inline function:
```

```
f(x) = x+sin(x)
>>ezplot(f,[0 2*pi])
>>fplot(f,[0 2*pi])
```

△

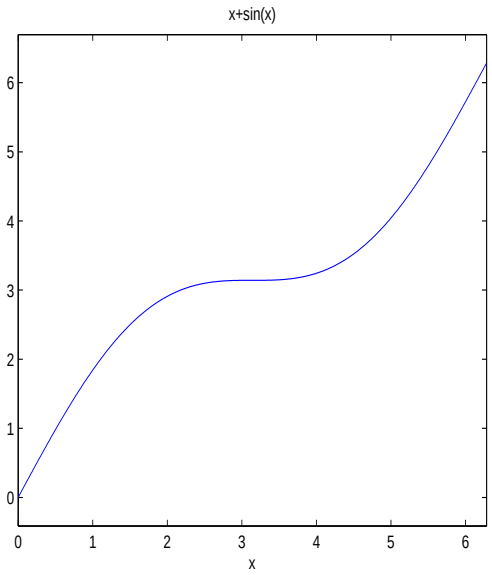

Figure 5.1: Plot of $f(x) = x + \sin x$ using *ezplot*.

5.2 Script M-Files

The heart of MATLAB lies in its use of M-files. We will begin with a *script* M-file, which is simply a text file that contains a list of valid MATLAB commands. To create an M-file, click on **File** at the upper left corner of your MATLAB window, then select **New**, followed by **M-file**. A window will appear in the upper left corner of your screen with MATLAB's default editor. (You are free to use an editor of your own choice, but for the brief demonstration here, let's stick with MATLAB's.) In this window, type the following lines:

```
x = linspace(0,2*pi,50);
f = sin(x);
plot(x,f)
```

Save this file by choosing **File**, **Save As** from the main menu. In this case, save the file as *sineplot.m*, and then close or minimize your editor window. Back at the command line, type *sineplot* at the prompt, and MATLAB will plot the sine function on the domain $[0, 2\pi]$. It has simply gone through your file line by line and executed each command as it came to it.

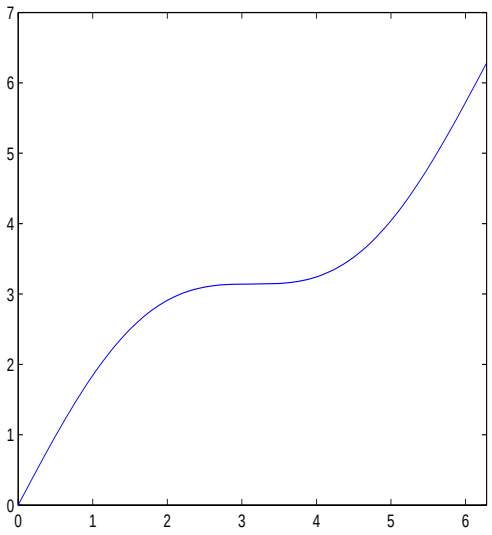

Figure 5.2: Plot of $f(x) = x + \sin x$ using *fplot*

5.3 Function M-files

The second type of M-file is called a *function* M-file and typically (though not inevitably) these will involve some variable or variables sent to the M-file and processed. As our first example, we will write a function M-file that takes as input the number of points for our sine plot from the previous section and then plots the sine curve. We can begin by typing

```
>>edit sineplot
```

In MATLAB's editor, revise your file sineplot.m so that it has the following form:

```
function sineplot(n)
x = linspace(0,2*pi,n);
f = sin(x);
plot(x,f)
```

Every function M-file begins with the command *function,* and the input is always placed in parentheses after the name of the function M-file. Save this file as before and then run it with 5 points by typing

```
>>sineplot(5)
```

In this case, the plot should be fairly poor, so try it with 50 points (i.e., use *sineplot(50)*).

We can also take several inputs into our function at once. As an example, suppose that we want to take the left and right endpoints of our plotting interval as input (as well as the number of points). We use

```
function sineplot(a,b,n)
x = linspace(a,b,n);
f = sin(x);
plot(x,f)
```

Here, observe that order is important, so when you call the function you will need to put your inputs in the same order as they are read by the M-file. For example, to again plot sine on $[0, 2\pi]$, we use

```
>>sineplot(0,2*pi,50).
```

MATLAB can also take multiple inputs as a vector. Suppose the three values 0, 2π, and 50 are stored in the vector v. That is, in MATLAB you have typed

```
>>v=[0,2*pi,50];
```

In this case, we write a function M-file that takes v as input and appropriately places its components.

```
function sineplot(v)
x = linspace(v(1),v(2),v(3));
f = sin(x);
plot(x,f)
```

5.4 Functions that Return Values

In the function M-files we have considered so far, the files have taken data as input, but they have not returned values. In order to see how MATLAB returns values, suppose we want to compute the maximum value of $\sin(x)$ on the interval over which we are plotting it. Change *sineplot.m* as follows:

```
function maxvalue = sineplot(v)
x = linspace(v(1),v(2),v(3));
f = sin(x);
maxvalue = max(f);
plot(x,f)
```

In this new version, we have made two important changes. First, we have added *maxvalue* = to our first line, specifying that the value we want MATLAB to return is the one we compute as maxvalue. Second, we have added a line to the code that computes the maximum of f and assigns its value to the variable *maxvalue*. (The MATLAB function *max* takes vector input and returns the largest component.) When running an M-file that returns data from the command window, you will typically want to assign the returned value a designation. Here, you might use

```
>>m=sineplot(v)
```

The maximum of $\sin(x)$ on this interval will be recorded as the value of m.

MATLAB can also return multiple values. Suppose we would like to return both the maximum and the minimum of f in this example. We use

```
function [minvalue,maxvalue] = sineplot(v)
x = linspace(v(1),v(2),v(3));
f = sin(x);
minvalue = min(f);
maxvalue = max(f);
plot(x,f)
```

In this case, at the command prompt, type

```
>>[m,n]=sineplot(v)
```

The value of m will now be the minimum of $\sin(x)$ on this interval, while n will be the maximum.

As our last example, we will write a function M-file that takes vector input and returns vector output. In this case, the input will be as before, and we will record the minimum and maximum of f in a vector. We have

```
function w = sineplot(v)
x = linspace(v(1),v(2),v(3));
f = sin(x);
w = [min(f),max(f)];
plot(x,f)
```

This function can be called with

```
>>b=sineplot(v)
```

where it is now understood that b is a vector with two components.

5.5 Debugging M-files

Since MATLAB views M-files as computer programs, it offers a handful of tools for debugging. First, from the M-file edit window, an M-file can be saved and run by clicking on the icon with the white sheet and downward-directed blue arrow (alternatively, choose **Debug, Run** or simply type **F5**). By setting your cursor on a line and clicking on the icon with the white sheet and the red dot, you can set a marker at which MATLAB's execution will stop. A green arrow will appear, marking the point where MATLAB's execution has paused. At this point, you can step through the rest of your M-file one line at a time by choosing the *Step* icon (alternatively **Debug, Step** or **F6**).

Unless you're a phenomenal programmer, you will occasionally write a MATLAB program (M-file) that has no intention of stopping any time in the near future. You can always abort your program by typing **Control-c**, though you must be in the MATLAB Command Window for MATLAB to pay any attention to this. If all else fails, **Control-Alt-Backspace** will end your session on a calclab account.

5.6 Assignments

For the logistic population model

$$R(N) = aN(1 - \frac{N}{K}),$$

the number of individuals in the population can be written as a function of time t (and the parameters of the model) as

$$N(t, a, K, N_0) = \frac{N_0 K}{N_0 + (K - N_0)e^{-at}},$$

where N_0 is the number of individuals at time 0.

1. [2.5 pts]Define the function $N(t, a, K, N_0)$ as an inline function and compute the population for $t = 10$, $a = .1$, $K = 100$, and $N_0 = 5$.

2. [2.5 pts] Write a script M-file that plots N as a function of t for the parameter values given in problem 1. Take $t \in [0, 5]$.

3. [2.5 pts] Write a function M-file that takes the parameters a, K, and N_0 as input and plots N as a function of t for $t \in [0, 5]$. Use your M-file to plot N for the values $a = .2$, $K = 100$, $N_0 = 150$.

4. [2.5] Write a function M-file that takes the parameters a, K, and N_0 as input and returns the value of the population at $t = 5$.

6 Function Limits in MATLAB

6.1 Symbolic Limits of Functions

MATLAB's symbolic toolbox has a function *limit* that can symbolically compute limits. The syntax for computing the limit

$$\lim_{x\to a} f(x) = L$$

is

limit(f(x),x,a),

where x is a symbolic variable.

Example 6.1. Compute the limit

$$\lim_{x\to 0} \frac{\sin x}{x} = 1.$$

In MATLAB,

```
>>syms x;
>>limit(sin(x)/x,x,0)
ans =
1
```

△

Example 6.2. Compute the limit

$$\lim_{x\to\infty} \frac{x^4 + x^2 - 3}{3x^4 - \log x} = \frac{1}{3},$$

In MATLAB,

```
>>limit((x^4 + x^2 - 3)/(3*x^4 - log(x)),x,Inf)
ans =
1/3
```

△

For left and right limits, the option 'left' or 'right' can be added at the end of the function statement.

Example 6.3. Compute the left and right limits

$$\lim_{x\to 0^-} \frac{|x|}{x} = -1; \qquad \lim_{x\to 0^+} \frac{|x|}{x} = +1.$$

In MATLAB,

```
>>syms x;
>>limit(abs(x)/x,x,0,'left')
ans =
-1
>>limit(abs(x)/x,x,0,'right')
ans =
1
```

△

6.2 Divergence by Oscillation

In class we have observed that in certain cases a limit can fail to exist because the graph of the function continues to oscillate as the independent variable approaches its limit. In this subsection, we will consider an example of such a case.

Example 6.4. Consider the limit

$$\lim_{x\to 0} \sin\frac{1}{x}.$$

In order to study this limit, we will plot the function $\sin\frac{1}{x}$ for x near 0. We use

```
>>x=linspace(.03, .5, 1000);
>>f=sin(1./x);
>>plot(x,f)
```

We see in Figure 6.1 that as x goes toward 0, $\sin\frac{1}{x}$ continues to oscillate between -1 and +1, with the period of oscillation becoming shorter and shorter. △

6.3 The Sandwich Theorem

In class, we considered the following *Sandwich Theorem (*sometimes called *the Squeeze Theorem)*:

Sandwich Theorem. If $f(x) \le g(x) \le h(x)$ for all x in an open interval containing c (except possibly at c) and

$$\lim_{x\to c} f(x) = \lim_{x\to c} h(x) = L,$$

then

$$\lim_{x\to 0} g(x) = L.$$

Example 6.5. Consider the limit

$$\lim_{x\to 0} x\sin\frac{1}{x}.$$

Our approach to this limit in class was to apply with Sandwich Theorem with the inequalities

$$-|x| \le x\sin\frac{1}{x} \le |x|.$$

In order to see how this looks graphically, let's plot all three of these functions together on a MATLAB figure. We use the commands

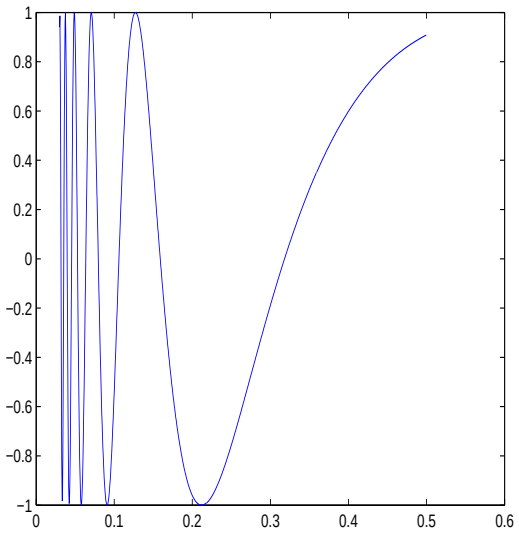

Figure 6.1: Plot of $\sin\frac{1}{x}$.

```
>>x=linspace(-1,1,100);
>>f=-abs(x);
>>g=x.*sin(1./x);
>>h=abs(x);
>>plot(x,f,'r',x,g,x,h,'r')
```

Here, the bounding functions $f(x)$ and $h(x)$ have been drawn in red with the option 'r' (this won't appear in the black and white figure in these notes). △

Example 6.6. We also use the Sandwich Theorem to establish the limit

$$\lim_{x\to 0}\frac{\sin x}{x} = 1.$$

In order to see this limit graphically, we use the following MATLAB code.

```
>>x=linspace(-1, 1, 1000);
>>f=sin(x)./x;
>>plot(x,f)
```

This creates Figure 6.3, in which we see that for x either above or below 0, the limit is approaching 1. △

6.4 The Bisection Method

In class, we used the Intermediate Value Theorem to develop the bisection method for finding roots of functions.

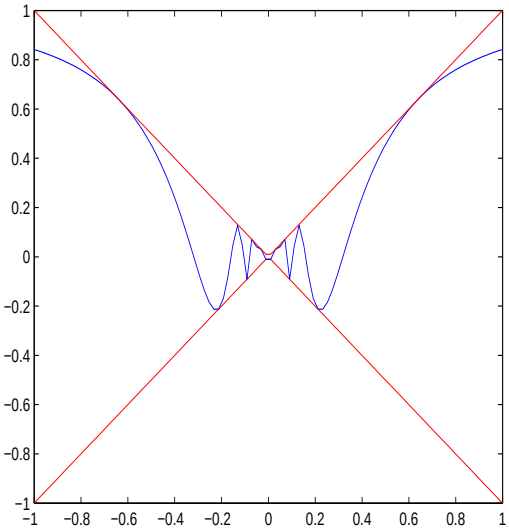

Figure 6.2: Sandwich Theorem plot of $x \sin \frac{1}{x}$.

Intermediate Value Theorem. Suppose $f(x)$ is continuous on some closed interval $[a, b]$, and that either $f(a) < L < f(b)$ or $f(b) < L < f(a)$. There there must exist some point $c \in (a, b)$ so that $f(c) = L$.

We can use this theorem to find roots for equations of the form

$$f(x) = 0.$$

We proceed by finding two points $a < b$ so that either $f(a) < 0 < f(b)$ or $f(b) < 0 < f(a)$. In either case, the Intermediate Value Theorem asserts that there must be some value $c \in (a, b)$ so that $f(c) = 0$. That is, c is a root of the equation. Since we only know that the root is between a and b, our best estimate of its location is the midpoint

$$m_1 = \frac{a+b}{2},$$

and this is our first approximation of the root. In order to compute a second approximation, we begin by evaluating $f(m_1)$. Of course, if $f(m_1) = 0$, we have found a root. Suppose alternatively that $f(m_1) > 0$ and that $f(a) < 0$. In this case, we now know that the root is located in the interval (a, m_1), which is half the size of the interval (a, b). Our second approximation becomes the midpoint of this interval

$$m_2 = \frac{m_1 + a}{2}.$$

By repeating this procedure, we can approximate the root between a and b as closely as we like. In particular, since we reduce our interval by a factor of 2 at each iteration, our error at the n^{th} iteration is

$$E_n = (b - a)(\frac{1}{2})^n.$$

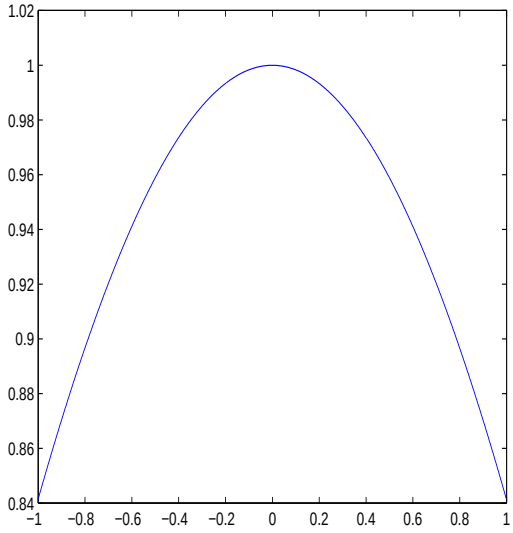

Figure 6.3: Plot of $\frac{\sin x}{x}$.

Example 6.7. (Problem 8 from Section 3.5 of our text.) Use the bisection method to find a solution of

$$\cos x = x$$

that is accurate to two decimal places (i.e., that has an error smaller than .01).

In this case, the function of interest is $f(x) = \cos x - x$, and we begin with the endpoints $a = 0$ and $b = 1$, for which $f(0) = 1$ and $f(1) = -.4597$. Our first estimate is

$$m_1 = \frac{1}{2}.$$

In order to find an estimate within two decimal places, we require the number of iterations n to be large enough so that

$$(\frac{1}{2})^n < .01.$$

That is, we insist that our error is less than $1/100$, even though rounding errors could actually make our two-decimal approximation differ from the correct answer by $1/100$. Solving this inequality for n we find that n must be large enough so that

$$n > \frac{\log_{10} .01}{\log_{10} .5} = 6.6439.$$

In other words, we require 7 iterations. We will carry this out in MATLAB. For the first few iterations, we can proceed explicitly

```
>>cos(1/2)-1/2
ans =
0.3776
```

```
>>m2=(1+1/2)/2
m2 =
0.7500
>>cos(m2)-m2
ans =
-0.0183
>>m3=(.75+.5)/2
m3 =
0.6250
```

Repeating this 7 times will get old fast, so let's finish the calculation with an M-file, *bisection.m*.[5]

```
function value = bisection(a,b,n)
f=inline('cos(x)-x','x');
fleft = f(a);
fright = f(b);
for k=1:n
m(k) = (a+b)/2;
fmid = f(m(k));
if fmid*fleft > 0
a = m(k);
else
b = m(k);
end
end
value = m;
```

In this M-file, we take left and right endpoints as input, as well as the number of iterations n. At step k, our approximation is m_k, which in MATLAB looks like $m(k)$. The file returns a vector containing the approximation at each iteration.

```
>>bisection(0,1,7)
ans =
0.5000 0.7500 0.6250 0.6875 0.7188 0.7344 0.7422
```

Here, the approximation at step 7 is $m_7 = .7422$, which we compare with the exact solution $m_{\text{exact}} = .7391$ (to four decimal places). We see indeed that our approximation is good to two decimal places (i.e., $|m_1 - m_{\text{exact}}| = .0031 < .01$). $\triangle$

[5]This M-file, along with several others, is available on the course web site.

6.5 Experimenting with the Formal Definition of Limits

The formal definition of a (finite) limit is given as follows:

Definition. The statement

$$\lim_{x \to c} f(x) = L$$

means that for any $\epsilon > 0$ there exists some $\delta > 0$ so that if $0 < |x - c| < \delta$ then $|f(x) - L| < \epsilon$. (Recall that we take $|x - c| > 0$, or equivalently $x \neq c$, because $f(x)$ may not be defined at $x = c$.) That is, by taking x sufficiently close to its limiting value c, we can force $f(x)$ to be arbitrarily close to its limiting value L.

Example 6.8. Consider the limit

$$\lim_{x \to 1} x^2 + 1 = 2.$$

The formal definition says that in order to prove that this is true, we must show that given any $\epsilon > 0$ there exists some $\delta > 0$ so that if $|x - 1| < \delta$ then $|(x^2 + 1) - 2| < \epsilon$. Suppose, for example, that $\epsilon = .01$. How small must we choose δ? In order to examine this in MATLAB, we will plot our function $x^2 + 1$ only for those values of x for which $|x - 1| < \delta$. Let's begin with the case $\delta = .1$. In order to specify $|x - 1| < .1$, we use *x=linspace(1-.1,1+.1,50).* We create Figure 6.4 with the following code.

```
>>x=linspace(.9,1.1,50);
>>f=x.^2+1;
>> plot(x,f)
```

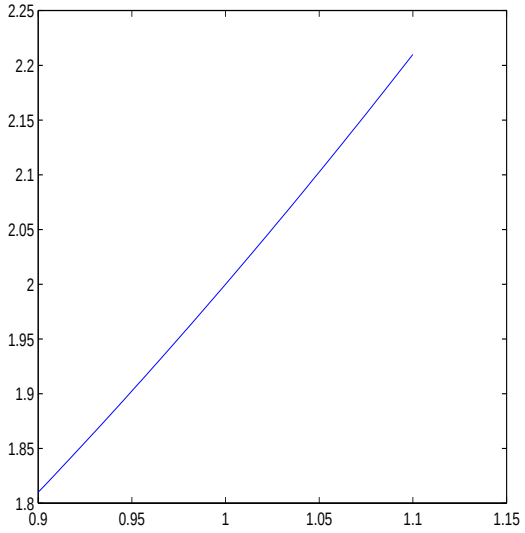

Figure 6.4: Figure for $\delta = .1$.

Notice that it's clear from Figure 6.4 that this value of δ is not small enough. That is, we need to insure that the plot is never farther than .01 from 2, and in this figure it gets as

far as .2 from 2. (Look at values on the y-axis.) This means that we must take a smaller value of δ. For $\delta = .01$, we have

```
x=linspace(.99,1.01,50);
f=x.^2+1;
plot(x,f)
```

We observe in Figure 6.5 that this is almost sufficient. That is, the values of $x^2 + 1$ now only differ from 2 by about .02.

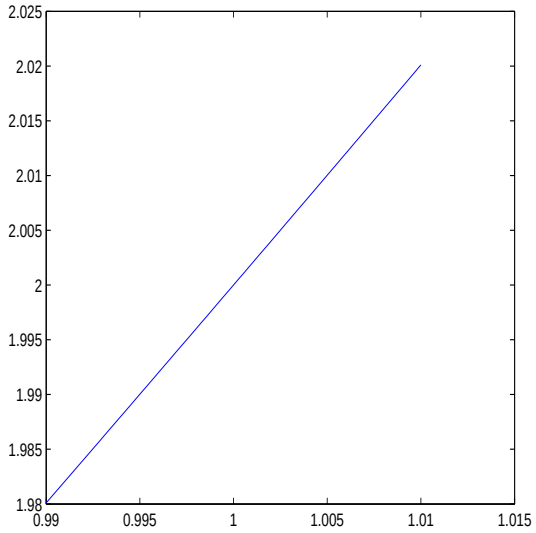

Figure 6.5: Figure for $\delta = .01$.

Finally, let's take $\delta = .001$. In this case, we use

```
>>x=linspace(.999,1.001,50);
>>f=x.^2+1;
>>plot(x,f)
```

This creates Figure 6.6, from which it is clear that for these values of x, $|(x^2 + 1) - 2| < .01$. In particular, this difference appears to be bounded by about .002, so it is even smaller than we require. △

6.6 Assignments

1. [2 pts] Use MATLAB's *limit* function to compute the following limits:

1a.

$$\lim_{x \to 0} \frac{\tan x}{x}.$$

1b.

$$\lim_{x \to \infty} x^3 e^{-x}.$$

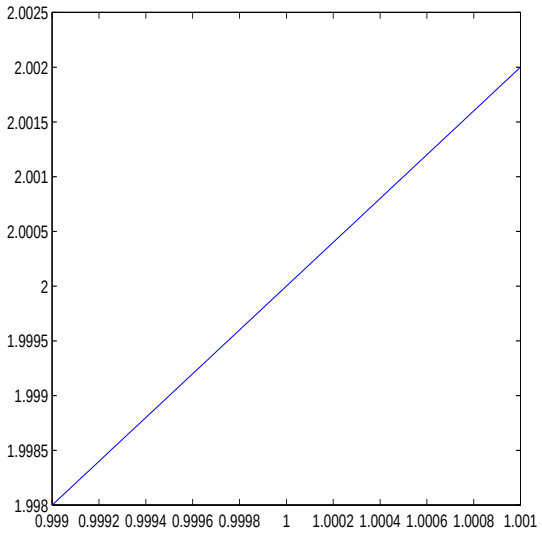

Figure 6.6: Figure for $\delta = .001$.

1c.

$$\lim_{x \to 1^+} e^{-\frac{1}{1-x}}.$$

2. [2 pts] The limit

$$\lim_{x \to 0^+} \frac{\sin x}{x} = 1$$

can be established by using the Squeeze Theorem and the inequality

$$\cos x \leq \frac{\sin x}{x} \leq \frac{1}{\cos x},$$

for $x \in (0, \frac{\pi}{2})$. Depict this graphically by proceeding similarly as in Example 6.5.

3. [2 pts] Find the number of iterations n that will insure that the solution to

$$x = \tan x,$$

for $x \in (\frac{\pi}{8}, \frac{3\pi}{8})$ is accurate to two decimal places.

4. [2 pts] Alter *bisection.m* so that the function $f(x)$ is taken (in the form of an inline function) as input in the function call statement. Use your new M-file to find the root for

$$f(x) = x - \tan x$$

for $x \in (\frac{\pi}{8}, \frac{3\pi}{8})$. Your error should be smaller than .01.

5. [2 pts] The limit

$$\lim_{x \to 1} \frac{(1 - .5^{x-1})}{x - 1} = \ln(2)$$

means that given any $\epsilon > 0$ there exists some $\delta > 0$ so that

$$0 < |x - 1| < \delta \Rightarrow |\frac{(1 - .5^{x-1})}{x - 1} - \ln(2)| < \epsilon.$$

Proceeding as in Example 6.8 find an appropriate value for δ if $\epsilon = .01$.

7 Symbolic Derivatives in MATLAB

Symbolic derivatives can be computed in MATLAB with the command *diff()*.

Example 7.1. Use the *diff()* command to compute the derivative of x^3.

We can compute this three different ways. First, we can proceed directly:

```
>>syms x;
>>diff(x^3)
ans =
3*x^2
```

Alternatively, we can first define $f(x) = x^3$ as an *inline* function.

```
>>syms x;
>>f=inline('x^3','x');
>>diff(f(x))
ans =
3*x^2
```

Finally, we can use *diff* without first specifying x as a symbolic variable by using single quotes in the argument of *diff*.

```
>>diff('x^3')
ans =
3*x^2
```

△

Example 7.2. Compute the derivative of $f(x) = e^x \sin x$. In MATLAB,

```
>>syms x;
>>diff(exp(x)*sin(x))
ans =
exp(x)*sin(x)+exp(x)*cos(x)
```

△

7.1 Computing Higher Order Derivatives in MATLAB

The derivative of a function $f(x)$ is again a function $f'(x)$. If we now take a derivative of $f'(x)$, we have

$$\frac{d}{dx} f'(x) = f''(x),$$

which we refer to as the *second* derivative of f. Alternatively, we have the notation

$$f''(x) = \frac{d^2 f}{dx^2}.$$

Similarly, the third derivative of f is defined as

$$f'''(x) = \frac{d}{dx} f''(x),$$

and can also be written as

$$f'''(x) = \frac{d^3 f}{dx^3}.$$

Proceeding similarly, we can define the derivative of a function to any order. Eventually, the prime notation becomes unwieldy, and so we often write the n^{th} derivative as

$$f^{(n)}(x) = \frac{d^n f}{dx^n}.$$

Example 7.3. For $f(x) = x^3$, compute $f''(x)$ in MATLAB.

As with first order derivatives, there are three ways in which to do this in MATLAB. First, we can simply use the *diff* command, and specify in the second entry that we want a second order derivative.

```
>>syms x;
>>diff(x^3,2)
ans =
6*x
```

Alternatively, we can first specify $f(x)$ as an inline function.

```
>>syms x;
>>f=inline('x^3','x')
f =
Inline function:
f(x) = x^3
>>diff(f(x),2)
ans =
6*x
```

Finally, we can again use the diff command with single quotes.

```
>>diff('x^3',2)
ans =
6*x
```

△

Example 7.4. Compute the third x-derivative of the function

$$f(x) = \sin(ax).$$

Again, we can proceed in one of three different ways, though now we must specify which variable we want MATLAB to differentiate with respect to. First, we use

```
>>syms a x;
>>diff(sin(a*x),x,3)
ans =
-cos(a*x)*a^3
```

(In fact, MATLAB takes x as its independent variable by default, so the statement *diff(sin(a*x),3)* would also work here, but in general you can use any letter (or combination of letters) as an independent variable, so long as you specify it as such.) Alternatively, we can define an inline function as a function of both x and a, and then specify that we want to differentiate with respect to x.

```
>>syms a x;
>>f=inline('sin(a*x)','a','x')
f =
Inline function:
f(a,x) = sin(a*x)
>>diff(f(a,x),x,3)
ans =
-cos(a*x)*a^3
```

Finally, we can still use *diff* with single quotes bracing the argument. In this case, notice that x must be placed in single quotes when being distinguished as the independent variable.

```
>>diff('sin(a*x)','x',3)
ans =
-cos(a*x)*a^3
```

△

7.2 Computing Derivatives as Limits

According to our definition, the derivative of a function $f(x)$ can be computed as the limit

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

Example 7.5. Use the definition of derivative to compute the derivative of $f(x) = x^2$. In MATLAB,

```
>>syms x h;
>>limit(((x+h)^2-x^2)/h,h,0)
ans =
2*x
```

△

Example 7.6. Use the definition of derivative to compute the derivatives of $\sin x$ and $\cos x$. In MATLAB

```
>>syms x h;
>>limit((sin(x+h)-sin(x))/h,h,0)
ans =
cos(x)
>>limit((cos(x+h)-cos(x))/h,h,0)
ans =
-sin(x)
```

△

7.3 Dynamic and Geometric Interpretations of the Derivative

In studying calculus and its applications, it is extremely important to keep in mind both the dynamic interpretation of the derivative as the rate of change of some process, and the geometric interpretation of the derivative as the slope of the line that is tangent to the graph of the function at the specified point.

Example 7.7. The logistic population model relates the rate of growth R of a population to the size of the population N by the relation

$$R(N) = aN(1 - \frac{N}{K}).$$

If N_0 is the population at time 0, it is possible to show that

$$N(t) = \frac{K}{1 + (\frac{K}{N_0} - 1)e^{-at}}.$$

In this example, we will investigate the behavior of this function for the values $a = 1$, $N_0 = 1$, and $K = 10$, for which we have

$$N(t) = \frac{10}{1 + 9e^{-t}}.$$

We can plot this function in MATLAB with the following commands, which create Figure 7.1.

```
>>t=linspace(0,5,50);
>>N=10./(1+9*exp(-t));
>>plot(t,N)
```

The average growth rate of this population over any time period is given by

$$\text{Average growth rate } = \frac{\text{Change in population}}{\text{Length of time period}}.$$

For example, the average growth rate of this population from year 3 to year 5 is

$$\text{Average growth rate from year 3 to year 5 } = \frac{N(5) - N(3)}{2}.$$

In particular, notice that this value $\frac{N(5)-N(3)}{2}$ is precisely the slope of the line connecting the points $(3, N(3))$ and $(5, N(5))$ (see Figure 7.2). This line is referred to as a *secant* line and has slope $m = 1.2613$ and y-intercept $b = 3.1218$. It can be added to our figure with the following commands, which create Figure 7.2.

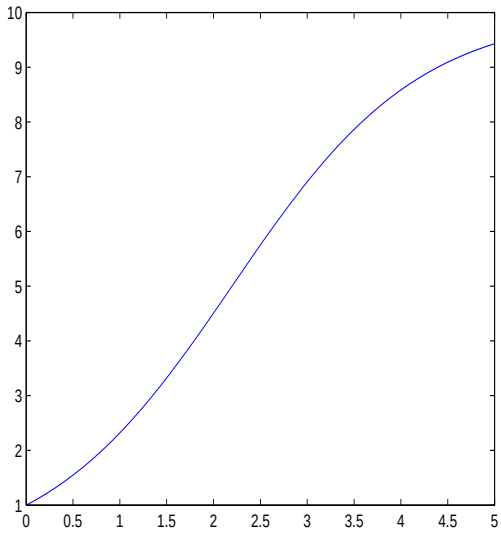

Figure 7.1: The logistic curve for $a = 1$, $N_0 = 1$, and $K = 10$.

```
>>s1=1.2613*t+3.1218;
>>hold on
>>plot(t,s1,'--')
```

Next, consider the average growth rate of the population between years 3 and 4. In this case, we have

$$\text{Average growth rate from year 3 to year 4} = \frac{N(4) - N(3)}{1},$$

where this is now the slope of the line connecting the points $(3, N(3))$ and $(4, N(4))$. The slope of this new line is $m = 1.6792$ and the y-intercept is $b = 1.8681$. We can add this line to our plot with the following commands, which create Figure 7.3.

```
>>s2=1.6792*t+1.8681;
>>plot(t,s2,'-.')
```

More generally, the average growth rate between time $t = 3$ and any later time $t = 3 + h$ is given by

$$\text{Average growth rate from year 3 to year 3+h} = \frac{N(3+h) - N(3)}{h}.$$

Suppose we would like to know the growth rate of the population precisely at the time $t = 3$. We expect that for h small (that is, for time intervals very close to $t = 3$), the average growth rate will be a fairly good approximation of this *instantaneous* growth rate. Moreover, we can find the exact growth rate at $t = 3$ by taking a limit as $h \to 0$. That is

$$\text{Instantaneous growth rate at year 3} = \lim_{h \to 0} \frac{N(3+h) - N(3)}{h} = 2.1368.$$

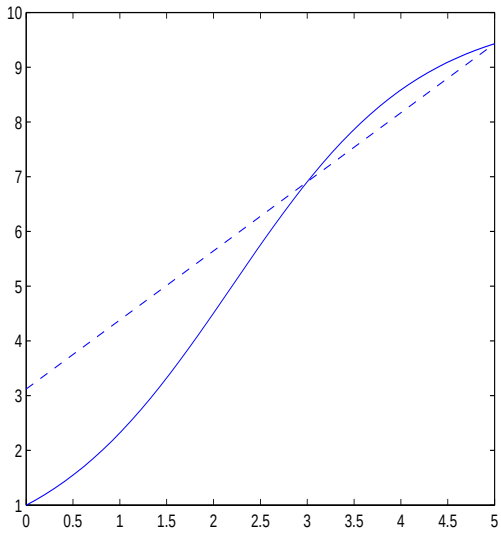

Figure 7.2: The logistic curve with a secant line.

This limit is the derivative of $N(t)$ at the time $t = 3$ and corresponds precisely with the slope of the line tangent to the curve of $N(t)$ at the time $t = 3$. The y-interept associated with this time is $b = .4953$, and so this line can be added to our plot with the following MATLAB commands, which create Figure 7.4. (The tangent line will appear in red on your figure.)

```
>>tangent=2.1368*t+.4953
>>plot(t,tangent,'r')
```

Generally speaking, we define a derivative of $N(t)$ at any time t by the relation

$$\frac{dN}{dt} = \lim_{h \to 0} \frac{N(t+h) - N(t)}{h}.$$

The fundamental point regarding the geometry of the derivative is that it corresponds with the slope of the line that is tangent to the curve at that point. △

7.4 Computing the Equation of the Tangent Line

It is clear from our geometric interpretation of derivative that for a differentiable function $f(x)$, if we want to compute the slope of the tangent line at some point value $x = c$, we need only compute $f'(c)$. In the final example of this section, we will use this information to compute the equation for the tangent line.

Example 7.8. Compute the equation of the line tangent to $f(x) = \cos x$ at the point $x_0 = \frac{\pi}{4}$.

We begin by computing the slope of this line, which is $f'(\frac{\pi}{4})$. In MATLAB,

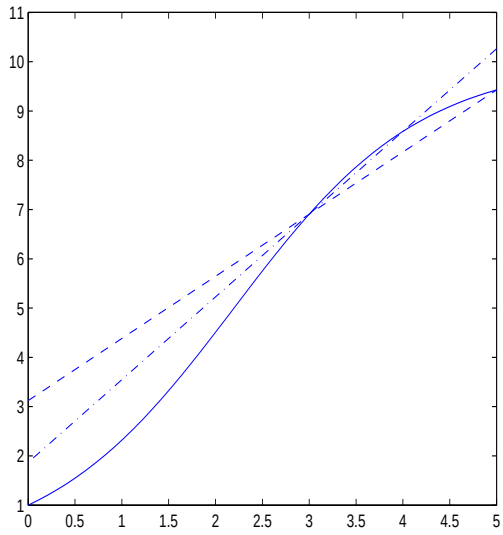

Figure 7.3: The logistic curve with two secant lines.

```
>>syms x
>>fprime=diff(cos(x))
fprime =
-sin(x)
>>subs(fprime,x,pi/4)
ans =
-0.7071
```

(You may recall that $\sin\frac{\pi}{4} = \frac{\sqrt{2}}{2}$, which is approximated to four decimal places by .7071.) In order to write the equation in point-slope form $y - y_0 = m(x - x_0)$, we compute $y_0 = \cos\frac{\pi}{4} = .7071$. Finally, we have

$$y - .7071 = -.7071(x - \frac{\pi}{4}),$$

which can be put into slope-intercept form $y = mx + b$

$$y = -.7071x + 1.2625.$$

We can plot $\cos x$ and this tangent line with the following MATLAB code, which creates Figure 7.5.

```
>>x=linspace(0,pi/2,100);
>>f=cos(x);
>>tangent=-.7071*x+1.2625;
>>plot(x,f,x,tangent,'–')
```

△

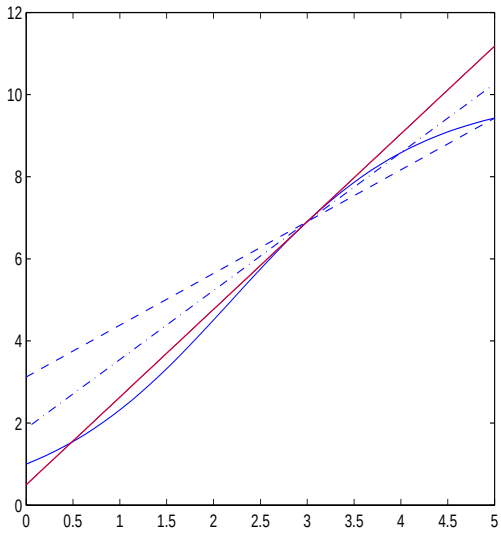

Figure 7.4: The logistic curve with a tangent line at $t = 3$.

7.5 Assignments

1. [2 pts] Use MATLAB's *diff()* function to compute the derivative of each of the following functions.

1a.

$$f(x) = x^{\tan x}.$$

1b.

$$f(x) = \cos^{-1} x.$$

2. [2 pts] Use MATLAB's *limit()* function to compute the derivative of

$$f(x) = x^r,$$

for any real number $r \neq 0$.

3. [6 pts] The Malthusian population model $R(N) = aN$ has the solution

$$N(t) = N_0 e^{at},$$

where N_0 denotes the number of individuals in the population at time t and a denotes the growth rate of the population. Take $N_0 = 1$ and $a = 1$, and plot $N(t)$ for $t \in [0, 5]$ along with (a) the secant line between $(3, N(3))$ and $(5, N(5))$; (b) the secant line between $(3, N(3))$ and $(4, N(4))$; and (c) the line tangent to the graph of N at the point $(3, N(3))$. (Note: This problem combines the ideas of Examples 7.7 and 7.8. You will need to use the ideas of Example 7.8 to find the equation for this tangent line.)

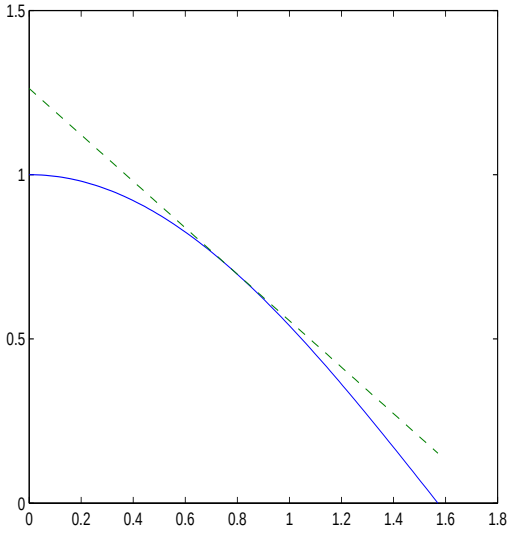

Figure 7.5: Plot of $\cos x$ along with tangent line at $x = \frac{\pi}{4}$.

8 Implicitly Defined Functions

In many cases we find that we have an equation involving both x and y that cannot be solved explicitly for y as a function of x (see, for example, Example 8.1 below). It may be the case, however, that given any x there corresponds precisely one y, and in such cases we say that the resulting function $y(x)$ is *implicitly* defined by the equation. In the event that there correspond multiple values of y to certain values of x, it is typically possible to extract multiple functions from the equation (see Example 8.2).

8.1 Plotting Implicitly Defined Functions

We can plot implicitly defined functions in MATLAB with *ezplot*.

Example 8.1. Plot the curve described by the relationship

$$x^2 y = e^{xy^2}.$$

With *ezplot*, the syntax is

$$\text{ezplot(FUN,[xmin xmax ymin ymax]),}$$

where FUN needs to be the relationship between the variables that is equal to zero; i.e., in this case

$$\text{FUN} = x^2 y - e^{xy^2}.$$

For this example, we can create Figure 8.1 with the following MATLAB code:

```
>>ezplot('x^2*y-exp(x*y^2)',[-2 0 0 2])
```

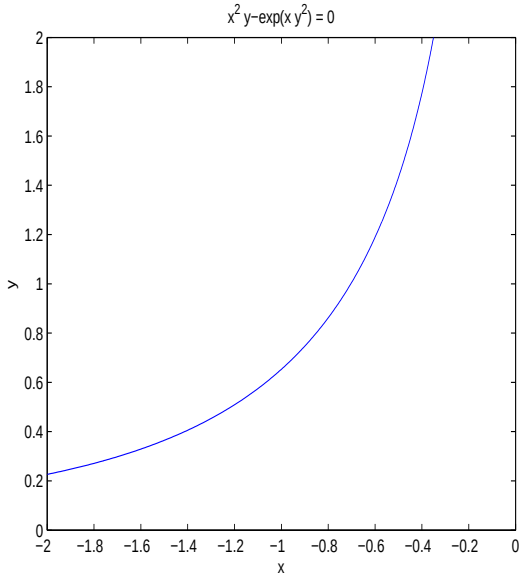

Figure 8.1: Plot of the relationship $x^2y = e^{xy^2}$.

△

Example 8.2. Write down the two functions $y_1(x)$ and $y_2(x)$ associated with the equation

$$x^2 + y^2 = 1.$$

In this case, we can solve for y to find

$$y(x) = \pm\sqrt{1 - x^2},$$

which is not a function since there correspond two values of y to each value of x. In this case, we have two functions

$$y_1(x) = -\sqrt{1 - x^2}$$
$$y_2(x) = \sqrt{1 - x^2}.$$

△

8.2 Tangent Lines for Implicitly Defined Relations

In order to compute the tangent line at a point for an implicitly defined relation, we require a method for computing $\frac{dy}{dx}$ that does not require an explicit formula for y as a function of x. We accomplish this with implicit differentiation.

Example 8.3. Plot the line tangent to the curve described in Example 8.1 at the point $x = -1$. First, we need a value of y associated with $x = -1$, and we observe that from our relation we have

$$y = e^{-y^2}.$$

Though we cannot solve this exactly, we can solve it in MATLAB with the following code.

```
>>y=solve('y-exp(-y^2)','y')
y =
exp(-1/2*lambertw(2))
>>eval(y)
ans =
0.6529
```

Our point is $(-1, .6529)$. In order to find the slope of the tangent line, we compute $\frac{dy}{dx}$ by taking an x-derivative of our entire equation. We have

$$\frac{d}{dx}(x^2y) = \frac{d}{dx}e^{xy^2} \Rightarrow 2xy + x^2\frac{dy}{dx} = e^{xy^2}(y^2 + x2y\frac{dy}{dx}).$$

Solving for $\frac{dy}{dx}$, we find

$$\frac{dy}{dx} = \frac{e^{xy^2}y^2 - 2xy}{x^2 - 2xye^{xy^2}}. \tag{8.1}$$

In MATLAB,

```
>>x=-1;
>>y=.6529;
>>yprime=(exp(x*y^2)*y^2-2*x*y)/(x^2-2*x*y*exp(x*y^2))
yprime =
0.8551
```

We conclude that our line has the form

$$y = .8551x + b.$$

We now find b by substituting our point $(-1, .6529)$ for x and y. We have

$$b = .6529 + .8551 = 1.5080,$$

and so our tangent line has the equation

$$y = .8551x + 1.5080.$$

(Alternatively, we can proceed from point-slope form; see Example 8.8.) We plot this along with with original curve with the following code, which creates Figure 8.2.

```
>>ezplot('x^2*y-exp(x*y^2)',[-2 0 0 2])
>>hold on
>>ezplot('y-.8551*x-1.5080',[-2 0 0 2])
```

△

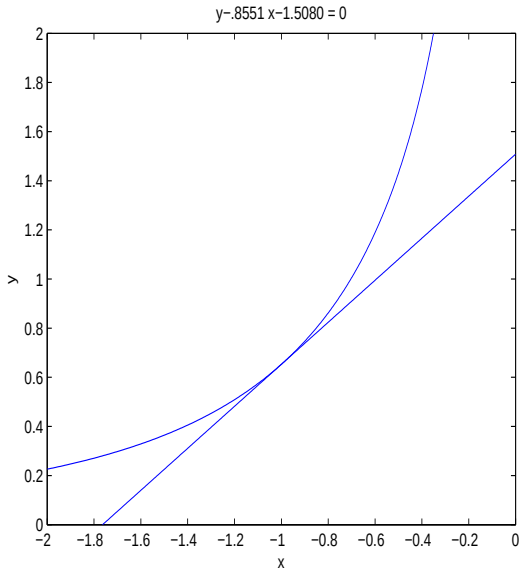

Figure 8.2: Plot of $x^2y = e^{xy^2}$ along with the tangent line at $(-1, .6529)$.

8.3 Implicit Differentiation in MATLAB (sort of)

Though MATLAB does not have a built-in function that implicitly differentiates both sides of an equation, expressions such as (8.1) can be obtained by the application of an important result involving partial differentiation. Since we haven't yet discussed partial differentiation, I will only give the appropriate MATLAB commands here, and we will talk about what they mean later in the course (next semester). Suppose, then, that for a relationship between x and y described by the equation

$$F(x, y) = 0,$$

we would like to find an expression for $y' = \frac{dy}{dx}$ using MATLAB. We use

$$\text{yprime} = -\text{diff(F(x,y),x)/diff(F(x,y),y)}.$$

Example 8.4. Use MATLAB to find $\frac{dy}{dx}$ for x and y described as in Example 9.1.

We use

```
>>syms x y;
>>F=inline('x^2*y-exp(x*y^2)','x','y')
F =
Inline function:
F(x,y) = x^2*y-exp(x*y^2)
>>yprime=-diff(F(x,y),x)/diff(F(x,y),y)
yprime =
(-2*x*y+y^2*exp(x*y^2))/(x^2-2*x*y*exp(x*y^2))
```

In checking that this agrees with (8.1), you might find the *pretty* function useful. △

8.4 Assignments

1. [5 pts] Plot the relation between x and y described by the equation

$$x^4 + y^4 = 1.$$

Use the limits $x \in [-2, 2]$ and $y \in [-2, 2]$. On the same figure, add the lines that are tangent to this curve at the points $(\frac{1}{2}, \frac{\sqrt[4]{15}}{2})$ and $(\frac{1}{2}, \frac{\sqrt[4]{15}}{2})$.

2. [5 pts] Plot the relation between x and y described by the equation

$$x^3 + y^3 = 6xy.$$

Use the limits $x \in [-4, 4]$ and $y \in [-4, 4]$. On the same figure, add the line that is tangent to this curve at the point $(3, 3)$.

9 Derivatives of Exponential and Inverse Functions

9.1 Approximating e from the Definition

One definition for the natural base e is the base for which the following limit holds:

$$\lim_{h\to 0}\frac{e^h-1}{h}=1.$$

In order to approximate a value of e from this definition, we must recall what we mean by this limit. By our formal definition of limits, for any $\epsilon>0$ there exists some $\delta>0$ so that if $|h|<\delta$ then

$$|\frac{e^h-1}{h}-1|<\epsilon.$$

(Keep in mind here that we are not proving this; we are taking advantage of it.) We have, then

$$-\epsilon<\frac{e^h-1}{h}-1<\epsilon\Rightarrow(1-\epsilon)h+1<e^h<(1+\epsilon)h+1.$$

Taking the $\frac{1}{h}$ power of each term in the inequality, we find that e is bounded by

$$((1-\epsilon)h+1)^{\frac{1}{h}}<e<((1+\epsilon)h+1)^{\frac{1}{h}}. \tag{9.1}$$

Given any $\epsilon>0$, we can choose h sufficiently small to make these true, so we will start with $\epsilon=.0001$, and study values of these expression as $h\to 0$. We can compute an upper bound with the following MATLAB code, in which the function $R(\epsilon,h)$ is simply the right-hand side of (9.1). We have

```
>>R=inline('((1+ep)*h+1)^(1/h)','ep','h')
R =
Inline function:
R(ep,h) = ((1+ep)*h+1)^(1/h)
>>R(.0001,.01)
ans =
2.7051
>>R(.0001,.001)
ans =
2.7172
>>R(.0001,.0001)
ans =
2.7184
>>R(.0001,.00001)
ans =
2.7185
```

We conclude that $e<2.7185$. On the other hand, we can get a lower bound with the following similar code:

```
>>L=inline('((1-ep)*h+1)^(1/h)','ep','h')
L =
Inline function:
L(ep,h) = ((1-ep)*h+1)^(1/h)
>>L(.0001,.01)
ans =
2.7045
>>L(.0001,.001)
ans =
2.7167
>>L(.0001,.0001)
ans =
2.7179
>>L(.0001,.00001)
ans =
2.7180
```

In this way, we conclude

$$2.7180 < e < 2.7185.$$

Of course, we can make this approximation better by choosing ϵ smaller to begin with, and then taking smaller values of h. The exact value of e to four decimal places is $e = 2.7183$.

9.2 The Derivative of an Inverse Function

If a function $f(x)$ is one-to-one and differentiable on some open interval (a, b), then the inverse function $f^{-1}(x)$ is one-to-one and differentiable on this same interval, and for all $x \in (a, b)$

$$\frac{df^{-1}}{dx}(x) = \frac{1}{f'(f^{-1}(x))}. \tag{9.2}$$

Example 9.1. Show that $f(x) = x^3 + x + 1$ is invertible on $(-1, 1)$, and compute

$$\frac{df^{-1}}{dx}(0).$$

First, we can check the invertibility of this function in MATLAB by plotting it and applying the horizontal line test. The following code created Figure 9.1, from which it is clear that no horizontal line crosses the function more than once.

```
>>x=linspace(-1,1,100);
>>f=x.^3+x+1;
>>plot(x,f)
```

According to (9.2), we can now compute $\frac{df^{-1}}{dx}(0)$ as

$$\frac{df^{-1}}{dx}(0) = \frac{1}{f'(f^{-1}(0))},$$

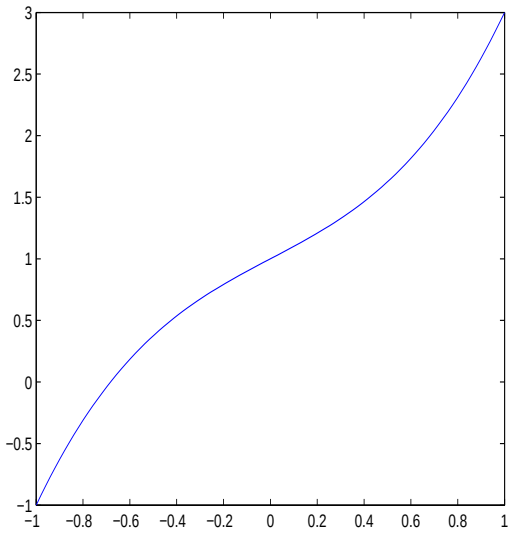

Figure 9.1: Plot of $f(x) = x^3 + x + 1$.

where

$$f'(x) = 3x^2 + 1.$$

All that remains is to compute $f^{-1}(0)$, and we observe that this is the x value associated with $y = 0$, and hence it is the real value of x that satisfies the equation

$$0 = x^3 + x + 1.$$

We solve this in MATLAB with the following code:

```
>>solve('x^3+x+1')
ans =
-1/6*(108+12*93^(1/2))^(1/3)+2/(108+12*93^(1/2))^(1/3)
1/12*(108+12*93^(1/2))^(1/3)-1/(108+12*93^(1/2))^(1/3)
+1/2*i*3^(1/2)*(-1/6*(108+12*93^(1/2))^(1/3)-2/(108+12*93^(1/2))^(1/3))
1/12*(108+12*93^(1/2))^(1/3)-1/(108+12*93^(1/2))^(1/3)
-1/2*i*3^(1/2)*(-1/6*(108+12*93^(1/2))^(1/3)-2/(108+12*93^(1/2))^(1/3))
>>eval(ans)
ans =
-0.6823
0.3412 - 1.1615i
0.3412 + 1.1615i
```

Though the cubic equation has three roots (as expected), only one is real. (If two of the roots were real, the function would not have an inverse on this interval.) We conclude that (to four decimal places of accuracy)

$$f(-.6823) = 0,$$

and consequently

$$f^{-1}(0) = -.6823.$$

We have, then

$$\frac{df^{-1}}{dx}(0) = \frac{1}{f'(f^{-1}(0))} = \frac{1}{3(-.6823)^2 + 1} = .4173.$$

△

Example 9.2. For $f(x)$ as in Example 1, plot $f^{-1}(x)$ along with $f(x)$ on $x \in [-1, 3]$ and plot the line tangent to $f(x)$ at the point $(-.6823, 0)$ and the line tangent to $f^{-1}(x)$ at the point $(0, -.6823)$. (The choice of interval $x \in [-1, 3]$ is taken so that the symmetry between $f(x)$ and $f^{-1}(x)$ will be apparent.)

First, we can invert $f(x)$ in MATLAB by solving the equation $y = x^3 + x + 1$ for x. In MATLAB, we use.

```
>>finv=solve('xˆ3+x+1=y','x');
```

This statement returns three solutions, but as in Example 1 we are only concerned with the first, which corresponds with the unique real solution. In order to plot this function along with $f(x)$, we use the following MATLAB code, which creates Figure 9.2.

```
>>y=linspace(-1,3,100);
>>finv=eval(vectorize(finv(1)));
>>plot(x,f,y,finv,'–')
```

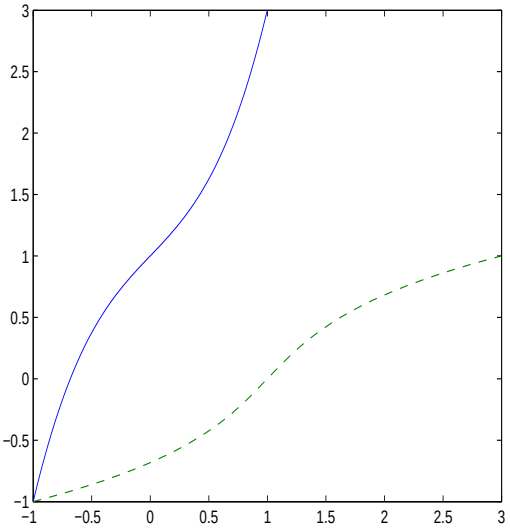

Figure 9.2: Plot of $f(x) = x^3 + x + 1$ along with $f^{-1}(x)$.

Here, we have used *finv(1)* to reference only the first component of *finv*, and we have used the *vectorize* command to insert appropriate array operations into the expression for *finv(1)*.

We see from Figure 9.2 that the inverse function is simply a mirror image of the original function about the line $y = x$. In order to plot the tangent line to $f(x)$, we observe from our calculations above that the slope of this line is

$$m_1 = f'(-.6823) = 3(-.6823)^2 + 1 = 2.3966.$$

The y-intercept can be found from the point $(-.6823, 0)$,

$$0 = 2.3966(-.6823) + b_1 \Rightarrow b_1 = 1.6352$$

That is, the equation for the line tangent to $f(x)$ at $(-.6823, 0)$ is

$$y = 2.3966x + 1.6352.$$

On the other hand, the slope of the line tangent to $f^{-1}(x)$ at $(0, -.6823)$ is $m_2 = .4173$, as computed in Example 1. (This is also simply $1/2.3966$; that is, these equations have reciprocal slopes.) In this case, we find the y-intercept b_2 from

$$-.6823 = .4173(0) + b_2 \Rightarrow b_2 = -.6823.$$

That is, the equation for the line tangent to $f^{-1}(x)$ at the point $(0, -.6823)$ is

$$y = .4173x - .6823.$$

Finally, we add these lines to our figure with the following MATLAB code.

```
>>tangent1=2.3866*x+1.6352;
>>tangent2=.4173*y-.6823;
>>plot(x,f,x,tangent1,'b',y,finv,'–',y,tangent2,'g')
>>axis([-1 3 -1 3])
```

The *axis* command has been added at the end so that the figure will have the same scaling as Figure 9.2. In order to add this command, the figure created with *plot* should be minimized, and then the command typed into the command window. The new figure is given as Figure 9.3. △

9.3 Assignments

1. [3 pts] Proceed as in Section 9.1 and find the value of a so that

$$\lim_{h \to 0} \frac{a^h - 1}{h} = 1.1447.$$

2. [2 pts] Use the derivative formula

$$\frac{d}{dx} a^x = a^x \ln a,$$

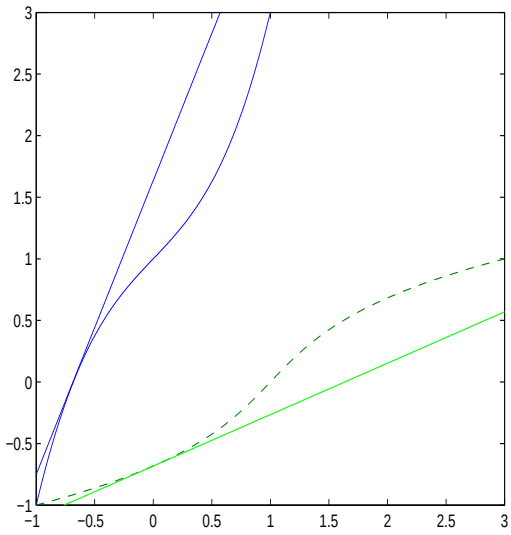

Figure 9.3: Plot of $f(x)$ and $f^{-1}(x)$ along with tangent lines.

and the limit definition of derivative to verify your solution to Problem 1. Why couldn't we use this easier calculation in Section 9.1?

3. [2 pts] Show that

$$f(x) = e^x - x$$

is invertible for $x \in (0, 2)$, and compute

$$\frac{df^{-1}}{dx}(2).$$

4. [3 pts] For $f(x)$ as in Problem 3 plot $f(x)$ along with $f^{-1}(x)$ for $x \in [0, 3]$ and plot the line tangent to $f(x)$ at the point $(f^{-1}(2), 2)$ and the line tangent to $f^{-1}(x)$ at the point $(2, f^{-1}(2))$.

10 Extrema and the Mean Value Theorem

10.1 The Mean Value Theorem

One of the most useful theorems in calculus is the Mean Value Theorem.

Theorem (Mean Value Theorem). Suppose $f(x)$ is continuous on the closed interval $[a, b]$ and differentiable on the open interval (a, b). Then there exists at least one number $c \in (a, b)$ so that

$$f'(c) = \frac{f(b) - f(a)}{b - a}.$$

Example 10.1. For $f(x) = x + e^{3x}$ on $[0, 1]$, find c so that

$$f'(c) = \frac{f(1) - f(0)}{1 - 0}. \tag{10.1}$$

In this case, $f'(x) = 1 + 3e^{3x}$, and so we must find c so that

$$1 + 3e^{3c} = \frac{1 + e^3 - 1}{1} = e^3.$$

If we subtract 1 from both sides, divide by 3, and take a natural logarithm, we find

$$c = \frac{1}{3}\ln(\frac{e^3 - 1}{3}) = .6168,$$

which is indeed on the open interval $(0, 1)$.

In order to understand this example graphically, we observe that (10.1) relates the slope of the tangent line at $x = c$ with the slope of the secant line connecting the points $(0, f(0))$ and $(1, f(1))$. In the following MATLAB code, we plot three things: (1) the function $f(x)$ on $(0, 1)$, (2) the line tangent to $f(x)$ at $x = .6168$, and (3) the secant line connecting the points $(0, 1)$ and $(1, 1 + e^3)$. In order to plot the tangent line, we will require the y-intercept, which (observing from above that $f'(.6168) = e^3$) can be computed from

$$y = e^3 x + b,$$

and the observation that $f(.6168) = 6.9786$ so that $(.6168, 6.9786)$ is a point on the line. That is,

$$6.9786 = .6168e^3 + b \Rightarrow b = -5.4101.$$

Figure 10.1 is now created with the following MATLAB code.

```
>>x=linspace(0,1,100);
>>f=x+exp(3*x);
>>tangent=exp(3)*x-5.4101;
>>plot(x,f,x,tangent,[0 1],[1 1+exp(3)])
```

We observe that the Mean Value Theorem simply asserts that there must be some point between 0 and 1 so that the slope of the tangent line at that point is the same as the slope of the secant line between the endpoints of the function. △

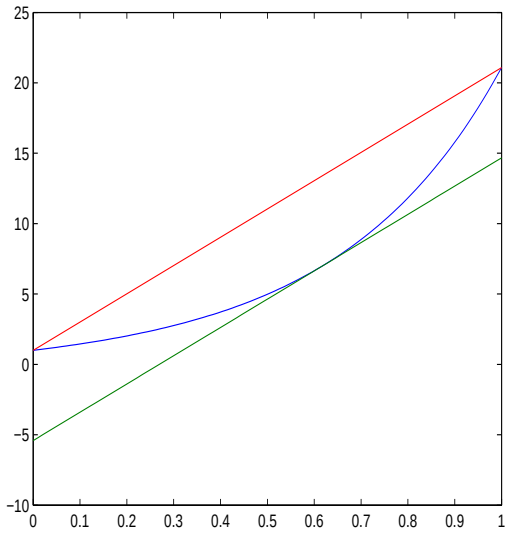

Figure 10.1: Plot of $f(x) = x + e^{3x}$ along with tangent and secant lines.

10.2 Monotonicity

According to our geometric interpretation of the derivative of a function at a point as the slope of the line tangent to that function at the point, it's clear that whenever $f'(x) < 0$ the function must be decreasing (these tangent lines are directed downward), while whenever $f'(x) > 0$ the function must be increasing. In the event that a function does not change direction on an interval (a, b) (that is, the function is either is always increasing or always decreasing) we call the function *strictly monotonic*.

Example 10.2. Plot both the function

$$f(x) = x^4 - 3x^3 - 7x^2 + 1$$

and its derivative, and observe that $f(x)$ increases whenever $f'(x)$ is positive and decreases whenever $f'(x)$ is negative.

We can create Figure 10.2 with the following MATLAB code.

```
>>syms x;
>>f=x^4-3*x^3-7*x^2+1;
>>fprime=diff(f)
fprime =
4*x^3-9*x^2-14*x
>>x=linspace(-2,4,100);
>>f=eval(vectorize(f));
>>fprime=eval(vectorize(fprime));
>>plot(x,f)
>>figure
>>plot(x,fprime,[-2 4],[0 0],'k')
```

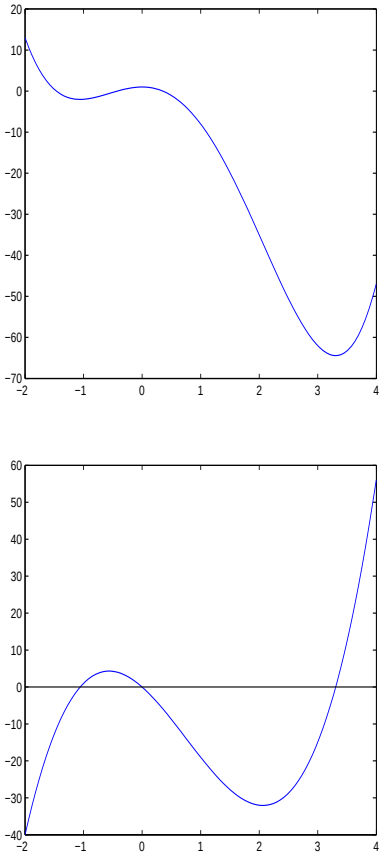

Figure 10.2: Plot of $f(x) = x^4 - 3x^3 - 7x^2 + 1$ (top), along with $f'(x)$ (bottom).

In particular, the zeros of $f'(x)$ are located at approximately the values -1.0580, 0, and 3.3080. Observe that $f'(x)$ is negative for $x \in (-2, -1.0580)$ and also for $x \in (0, 3.3080)$, and $f(x)$ is decreasing on precisely these same intervals. Also, observe that at the endpoints of these intervals (that is, at the values for which $f'(x) = 0$), $f(x)$ is either a local maximum or a local minimum. △

10.3 Concavity

Loosely speaking, we say that a function is *concave up* if it bends upward and *concave down* if it bends downward. Observe from Figure 10.2 that whenever $f(x)$ is concave up, $f'(x)$ is increasing, while whenever $f(x)$ is concave down $f'(x)$ is decreasing. We have the following definition.

Definition (Concavity). A differentiable function $f(x)$ is *concave up/down* on an interval (a, b) if its first derivative is an increasing/decreasing function on (a, b).

From our monotonicity discussion, we know that if $f'(x)$ is increasing, then its derivative $f''(x)$ must be positive, while if $f'(x)$ is decreasing, then its derivative must be negative. This leads to the following test for concavity.

Concavity Test. Suppose $f(x)$ is twice differentiable on some interval (a, b). If $f''(x) > 0$ on (a, b), then f is concave up on (a, b), while if $f''(x) < 0$ on (a, b), then f is concave down on (a, b).

Example 10.3. For $f(x)$ as in Example 11.2, plot $f(x)$ and $f''(x)$, and observe that $f(x)$ is concave up whenever $f''(x) > 0$ and that $f(x)$ is concave down whenever $f''(x) < 0$. We can create Figure 10.3 with the following MATLAB code.

```
>>syms x;
>>f=x^4-3*x^3-7*x^2+1;
>>fprime2=diff(f,2)
fprime2 =
12*x^2-18*x-14
>>x=linspace(-2,4,100);
>>f=eval(vectorize(f));
>>fprime2=eval(vectorize(fprime2));
>>plot(x,f)
>>figure
>>plot(x,fprime2,[-2 4],[0 0],'k')
```

△

10.4 Maximization and Minimization

In many applications, we are interested in finding the maximum or the minimum value of a function over some specified interval. For example, x might denote the number of widgets made in some industrial process and $f(x)$ might denote the profit drawn from the sale of these widgets. Since money cannot be made without the production of some widgets, we know that $f(0) = 0$, while if we create too many widgets we won't be able to sell them all, and we will lose our production costs. The goal is to find precisely the right number of widgets to maximize our profit. The derivative is an extremely useful tool for solving problems of this type. Along these lines, we have the following theorem.

Fermat's Theorem. Suppose $f'(x)$ is defined on some interval (a, b) and f has a local minimum or a local maximum at $c \in (a, b)$. Then $f'(c) = 0$.

If we combine Fermat's Theorem with the Extreme Value Theorem, we can conclude the following about the location of the global maximum and the global minimum for a function $f(x)$ that is continuous on a closed interval $[a, b]$.

Categorization of Extrema. For a continuous function f on a bounded interval $[a, b]$, suppose c is a global extremum (a global minimum or a global maximum). Then c must satisfy one of the following:

1. $f'(c) = 0$.
2. $f'(c)$ does not exist.
3. c is an endpoint.

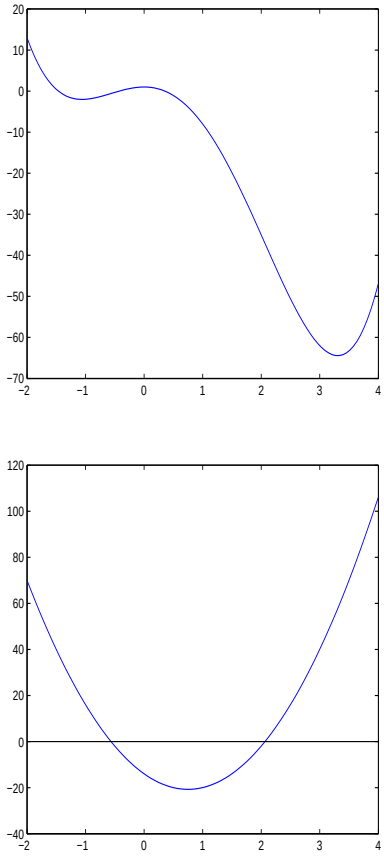

Figure 10.3: Graphs of $f(x)$ and $f'(x)$ for Example 3.

In practice, we can use this result to identify all possible candidates for extrema, and then we can determine the maximum and minimum values of the function by evaluating it at each of these values.

Example 10.4. Find the global extrema of the function

$$f(x) = x^2 e^{\sin x} - \frac{x}{x^3 + 1},$$

on the interval $[0, 5]$.

First, we plot this function for $x \in [0, 5]$. Figure 10.4 is created with the following MATLAB code.

```
>>f=inline('x^2*exp(sin(x))-x/(x^3+1)')
f =
Inline function:
f(x) = x^2*exp(sin(x))-x/(x^3+1)
>>ezplot(f,[0 5])
```

We can see from this graph the rough locations of the local maxima and minima, and we must use this information as initial guesses for MATLAB's function *fzero*, which numerically

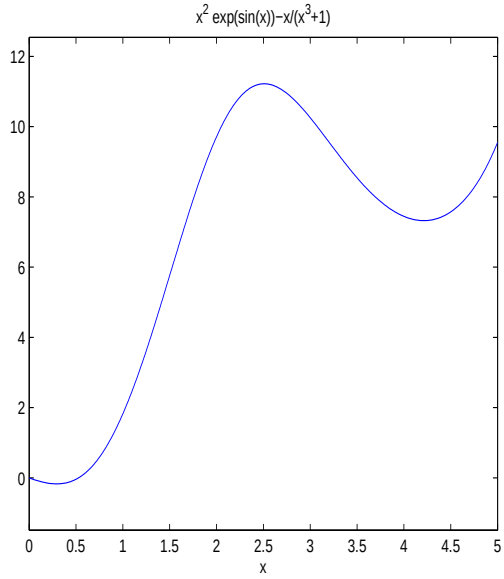

Figure 10.4: Plot of $f(x) = x^2 e^{\sin x} - \frac{x}{x^3+1}$.

locates the zeros of a function. (In this case, *solve* won't work because MATLAB cannot solve the equation $f'(x) = 0$ exactly.) The syntax for *fzero* is

fzero (f,guess),

where *guess* denotes an initial guess of the location of a root of the function f.

In order to find all zeros of $f'(x)$, we use the following MATLAB code:

```
>>syms x;
>>fprime=diff(f(x))
fprime =
2*x*exp(sin(x))+xˆ2*cos(x)*exp(sin(x))-1/(xˆ3+1)+3*xˆ3/(xˆ3+1)ˆ2
>>fprime=inline(char(fprime))
fprime =
Inline function:
fprime(x) = 2*x*exp(sin(x))+xˆ2*cos(x)*exp(sin(x))-1/(xˆ3+1)+3*xˆ3/(xˆ3+1)ˆ2
>>fzero(fprime,0)
ans =
0.2953
>>fzero(fprime,2.5)
ans =
2.5092
>>fzero(fprime,4.5)
ans =
4.2139
```

Here, the command *fprime=inline(char(fprime))* converts *fprime* into an inline function so that it can be analyzed with the *fzero* command. The *char* command is required to convert *fprime* from a symbolic object into a string, which is the required input for *inline*. We must now evaluate $f(x)$ at the five possible extrema, 0, .2953, 2.5092, 4.2139, and 5, where of course the middle values are all approximate. Assuming the previous code for this example has already been typed in, we can evaluate $f(x)$ at these values as follows:

```
>>f(0)
ans =
0
>>f(.2953)
ans =
-0.1712
>>f(2.5092)
ans =
11.2209
>>f(4.2139)
ans =
7.3222
>>f(5)
ans =
9.5429
```

We conclude that the global minimum occurs at (approximately) $x = .2953$, with $f(.2963) = -.1712$ (again approximate), and that the global maximum occurs at $x = 2.5092$, with $f(2.5092) = 11.2209$, both approximate. Of course, given our figure, we could reasonably evaluate f only at .2953 and 2.5092, but I wanted to proceed here with a general algorithm, which consists of checking every candidate. △

10.5 Assignments

1. [3 pts] For the function $f(x) = x + \sin x$, find the value $c \in (0, \pi)$ so that

$$f'(c) = \frac{f(\pi) - f(0)}{\pi}.$$

Plot $f(x)$ along with the line tangent to f at the point $(c, f(c))$ and the secant line between $(0, 0)$ and (π, π).

2. [2 pts] Plot $f(x) = \sin(2x^2 - 1)$ along with its first derivative (as in Example 10.2) for $x \in [0, 2\pi]$. Specify the intervals on which f is increasing and the intervals on which f is decreasing.

3. [2 pts] For the function from Problem 2 plot $f(x)$ along with $f''(x)$ (as in Example 10.3) for $x \in [0, 2\pi]$. Specify the intervals on which f is concave up and the intervals on which f is concave down.

4. [3 pts] Determine the absolute maximum and absolute minimum values of the function

$$f(x) = x^5 e^{-x^2} - \frac{\sin x}{x^2 + 1},$$

for $x \in [0, 5]$.

11 Limits of Sequences

MATLAB's symbolic toolbox has a function *limit* that can symbolically compute limits.

Example 11.1. Consider the exponential sequence

$$a_n = 3 \cdot 2^n, \quad n = 0, 1, 2, 3, ...$$

We can compute

$$\lim_{n\to\infty} a_n = \infty$$

with the following MATLAB code:

```
>>syms n
>>limit(3*2^n,n,Inf)
ans =
Inf
```

△

Example 11.2. We can compute the limit

$$\lim_{n\to\infty} \frac{1}{n+1} = 0$$

with the following MATLAB code:

```
>>limit(1/(n+1),n,Inf)
ans =
0
```

△

Example 11.3. We can compute the Zeno limit

$$\lim_{n\to\infty} \frac{2^n - 1}{2^n} = 1$$

with the following MATLAB code:

```
>>limit((2^n-1)/2^n,n,Inf)
ans =
1
```

△

11.1 Experimenting with the Limit Definition

In class, we made the following precise definition of a limit:

Definition. The sequence $\{a_n\}_{n=1}^{\infty}$ converges to a limit a, written

$$\lim_{n\to\infty} a_n = a,$$

if for any $\epsilon > 0$, there exists an integer N so that if $n > N$ then $|a_n - a| < \epsilon$.

Example 11.4. Consider the Zeno limit

$$\lim_{n\to\infty} \frac{2^n - 1}{2^n} = 1.$$

In proving that this sequence converges to 1, we must show that for any $\epsilon > 0$ there exists an integer N so that if $n > M$ then

$$|\frac{2^n - 1}{2^n} - 1| < \epsilon.$$

We can view this graphically in MATLAB by plotting the sequence a_n as a function of its index n. At the MATLAB prompt, type

```
>>n=1:10;
>>a=(2.^n-1)./(2.^n);
>>plot(n,a,'o')
```

The plot this creates is given below in Figure 11.1.

We can see from Figure 11.1 that if $\epsilon = .1$, then $N = 4$ is clearly sufficient. Likewise, if $\epsilon = .05$, then $N = 5$ appears to be sufficient. The choice of N depends on ϵ, and the smaller we take ϵ to be, the larger N must be. In fact, recalling from class that in the proof that Zeno's limit converges we took N to be the first integer larger than $-\log_2 \epsilon$, we can see the dependence explicitly. In particular, we plot the function

$$f(\epsilon) = -\log_2 \epsilon.$$

```
>>epsilon=.001:.01:.1;
>>f=-log2(epsilon);
>>plot(epsilon,f)
```

It is clear from Figure 11.2 that as ϵ gets small, this value $-\log_2 \epsilon$ grows.

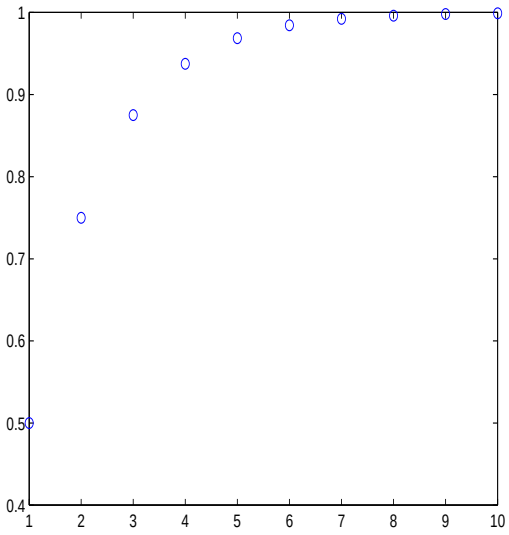

Figure 11.1: Figure for Example 4.

11.2 Recursions

We have seen in class that if we are computing population values at discrete times, we typically find recursive schemes of calculation.

Example 11.5. Consider the Beverton–Holt recruitment curve

$$N_{t+1} = \frac{RN_t}{1 + \frac{R-1}{K}N_t}; \quad N(0) = N_0,$$

where we will take the growth rate R to be 2, the carrying capacity K to be 10, and the initial population N_0 to be 1. Write a MATLAB function M-file that takes an integer value of t as input and returns the value of the population at that time.

```
function value = bevholt(t)
R = 2; K = 10; N0 = 1;
N(1) = N0;
for k = 1:t
N(k+1) = R*N(k)/(1 + ((R-1)/K)*N(k));
end
value = N(t+1);
```

Observe that since MATLAB does not allow a 0 index, we have had to associate the index 1 with the time 0, the index 2 with the time 1 etc. We can now calculuate populations according to this model as follows:

```
>>bevholt(0)
```

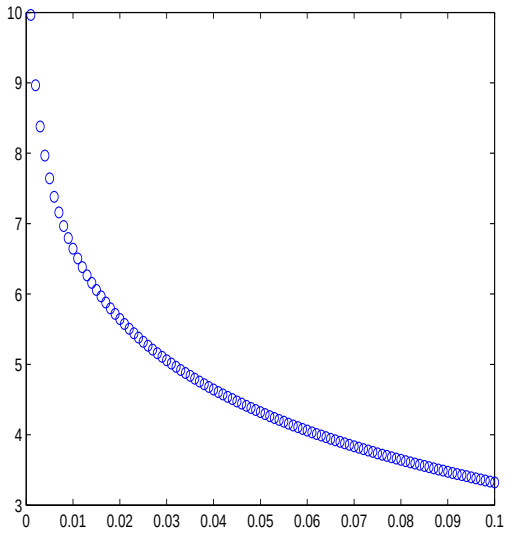

Figure 11.2: Plot of $-\log_2 \epsilon$.

```
ans =
1
>>bevholt(1)
ans =
1.8182
>>bevholt(2)
ans =
3.0769
```

Next, we plot solutions to this model for $t = 0, 1, 2, ..., 10$. We use

```
>>times=0:10;
>>for k=1:11
N(k)=bevholt(times(k));
end
>>plot(times,N)
```

The lines can either be written into an M-file or typed directly into the MATLAB prompt. If you type them directly in at the MATLAB prompt, observe that MATLAB does not give a prompt after the *for* statement, but does allow you to write the loop material just below it. The figure this creates is given in Figure 11.3.

We will show in class that $N = K$ is a *fixed point* of the Beverton–Holt recursion, and thus a natural candidate for this limit.

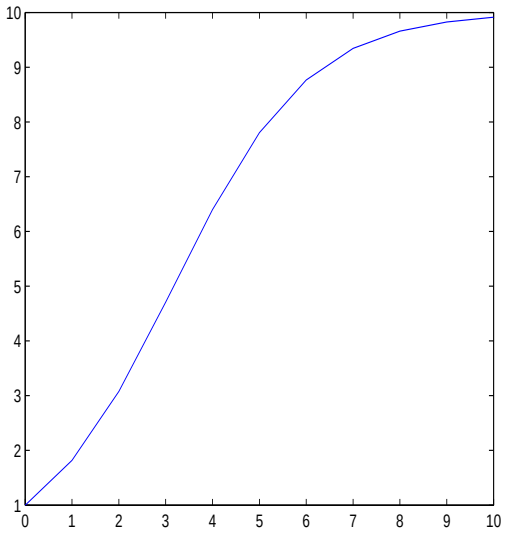

Figure 11.3: Plot of the Beverton–Holt recruitment curve.

Assignments

1. [2 pts] Compute the following limits in MATLAB:

1a.

$$\lim_{N\to\infty}(\sqrt{N}-\sqrt{N-1}).$$

1b.

$$\lim_{N\to\infty}(1+\frac{1}{N})^N.$$

2. [3 pts] Show that

$$\lim_{N\to\infty} N(1-.5^{\frac{1}{N}}) = \ln 2$$

(keeping in mind that MATLAB denotes ln by log). This means that for any $\epsilon > 0$ there exists some integer N large enough so that

$$|N(1-.5^{\frac{1}{N}}) - \ln 2| < \epsilon.$$

Find such values of N for $\epsilon = .1$ and for $\epsilon = .01$.

3. [5 pts] Write a MATLAB function M-file that takes an integer time t as input and returns the population at time t according to the discrete logistic equation

$$N_{t+1} = N_t\Big[1 + R\Big(1 - \frac{N_t}{K}\Big)\Big].$$

Take the growth rate R to be 2, the carrying capacity K to be 10, and the initial population N_0 to be 1. Turn in a plot depicting $N(t)$ at the times $t = 0, 1, 2, ..., 10$.

12 Cobwebbing and Newton's Method

12.1 Cobwebbing

For single recursion equations $x_{n+1} = f(x_n)$ we can graphically depict the iteration process by the method of cobwebbing.

Example 12.1. Use the method of cobwebbing to depict the solution to the recursion equation

$$x_{n+1} = \frac{1}{2}x_n; \quad x_0 = 1$$

for $n = 10$ iterations (i.e., up to x_{10}).

In order to apply the method of cobwebbing, we plot $f(x) = \frac{1}{2}x$ and $g(x) = x$ on a single figure, and beginning with the point $(1, \frac{1}{2})$ (on the graph of $f(x)$), we draw a line horizontally to the graph of $g(x)$, then vertically to the graph of $f(x)$ and so on. Each iteration requires two lines, one horizontal and one vertical. After k iterations we arrive at the point (x_k, x_{k+1}). In order to automate this process we will use the MATLAB file *cobweb.m*.

```
function values = cobweb(f,x0,n)
%COBWEB: MATLAB function M-file that takes as input an
%inline function f, an initial point x0, and a number of
%iterations n, and returns a sequence of values x(1) = x0,
%x(2) = x1, ... x(n+1) = xn for the recursion x(n+1)=f(x(n)),
%and plots the cobwebbing associated with this recursion.
%
x(1) = x0;
for k=2:n+1
x(k)=f(x(k-1));
end
%Specify the interval of x values. The gap variable will ensure
%that no lines appear on the boundary of the figure.
a = min(x);
b = max(x);
gap = (b-a)/10;
%Plot the graphs of f(x) and g(x) = x
y=linspace(a-gap,b+gap,round((b-a)*25));
fvals = f(y);
plot(y,y,y,fvals);
%Plot the cobwebbing
hold on
for k=1:n
plot([x(k) f(x(k))], [x(k+1) x(k+1)], '–');
plot([x(k+1) x(k+1)], [x(k+1) f(x(k+1))], '–');
end
%Place a circle on the final point
plot(x(n+1),f(x(n+1)),'o','MarkerFaceColor','k','MarkerSize',5)
```

```
values = x;
```

Notice that while this file is a bit long, most of the code is for commenting and plotting; the iteration is carried out entirely in the first *for* loop. We implement this file at the MATLAB prompt with the following commands, which create Figure 12.1.

```
>>f = inline('.5*x')
f =
Inline function:
f(x) = .5*x
>>cobweb(f,1,10)
ans =
Columns 1 through 7
1.0000 0.5000 0.2500 0.1250 0.0625 0.0312 0.0156
Columns 8 through 11
0.0078 0.0039 0.0020 0.0010
```

△

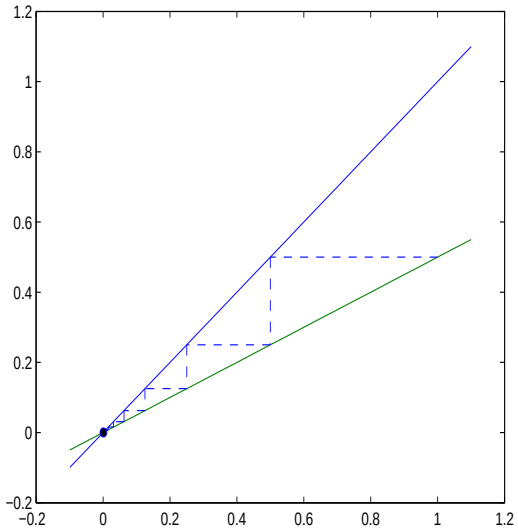

Figure 12.1: Cobwebbing for the recursion equation $x_{n+1} = \frac{1}{2}x_n; \quad x_0 = 1.$

Example 12.2. Use the method of cobwebbing to depict the solution to the recursion equation

$$x_{n+1} = 1 - \frac{1}{2}x_n; \quad x_0 = 1$$

for $n = 10$ iterations.

Proceeding almost precisely as in Example 12.1, we use the following MATLAB code to create Figure 12.2.

```
>>f = inline('1-.5*x')
f =
Inline function:
f(x) = 1-.5*x
>>cobweb(f,1,10)
ans =
Columns 1 through 7
1.0000 0.5000 0.7500 0.6250 0.6875 0.6562 0.6719
Columns 8 through 11
0.6641 0.6680 0.6660 0.6670
```

We see from the output that the values of x_n appear to be approaching $\frac{2}{3}$, which is a stable fixed point. △

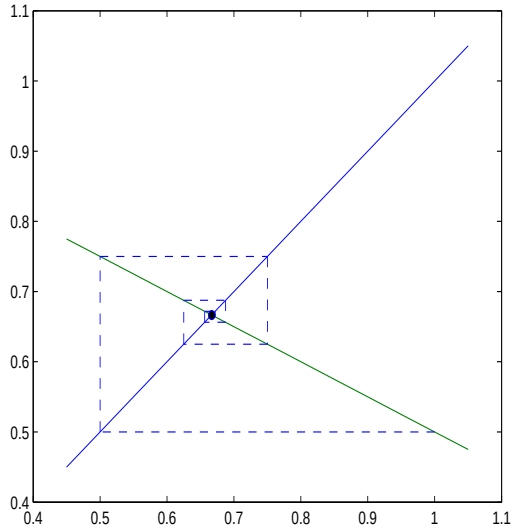

Figure 12.2: Cobwebbing for the recursion equation $x_{n+1} = 1 - \frac{1}{2}x\, x_0 = 1$.

Example 12.3. Use the method of cobwebbing to depict the solution to the recursion equation

$$x_{n+1} = 1.8(x_n - x_n^3)$$

for $n = 10$ iterations and three case $x_0 = 1$, $x_0 = .99$, and $x_0 = 1.01$.

For the first, we use the following MATLAB code. Note, in particular, that the cube must be an array operation.

```
>>f = inline('1.8*(x-x.^3)')
f =
Inline function:
f(x) = 1.8*(x-x.^3)
>>cobweb(f,1,10)
```

```
ans =
1 0 0 0 0 0 0 0 0 0 0
```

In this case $x_1 = 0$, and 0 is recovered at every subsequent step.

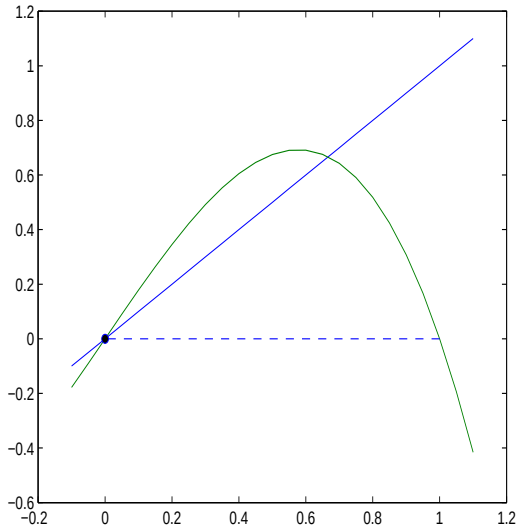

Figure 12.3: Cobwebbing for the recursion equation $x_{n+1} = 1.8(x_n - x_n^3)$, $x_0 = 1$.

Next, we use the command *cobweb(f,.99,10)*, which creates Figure 12.4. In this case, we find that a small change in the initial value x_0 leads to a relatively large change in the behavior of the system. This is a hallmark of the type of behavior defined as *chaos*.[6] The system is approaching the fixed point $\frac{2}{3}$.

Finally, we use the command *cobweb(f,1.01,10)* to create Figure 12.5. We see that again the behavior is entirely different, and in this case the system approaches the fixed point $-\frac{2}{3}$.

12.2 Newton's Method

In Section 6 we used the bisection method to find the roots of an algebraic equation. As second, typically much faster method, is known as Newton's method. Suppose we would like to find a particular root for the equation

$$f(x) = 0.$$

We begin with an initial guess x_0, and consider the line tangent to $y = f(x)$ at x_0, given in point-slope form by

$$y - f(x_0) = f'(x_0)(x - x_0).$$

[6]Technically, a recursion equation must have three properties to be classified as chaotic and this is only one of them. We haven't developed enough theory to discuss the other two.

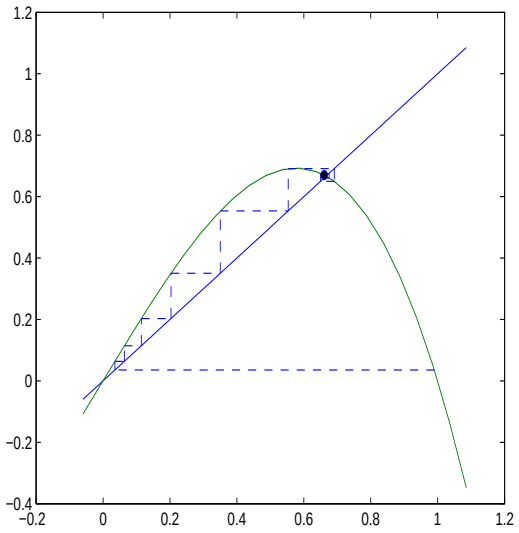

Figure 12.4: Cobwebbing for the recursion equation $x_{n+1} = 1.8(x_n - x_n^3)\, x_0 = .99$.

Since our goal is to $y = 0$, we set $y = 0$ in this linear approximation and obtain (solving for x in terms of x_0) the equation

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

The idea of Newton's method is simply to keep repeating this process, so that we arrive at a recursion relation of the form

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Example 12.4. Use Newton's method to approximate $\sqrt{2}$.

We begin by observing that $\sqrt{2}$ is a solution to the algebraic equation

$$x^2 - 2 = 0.$$

If we set $f(x) = x^2 - 2$ then we can apply Newton's method to obtain the iteration

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n} = \frac{x_n^2 + 2}{2x_n}.$$

We can now use *cobweb.m* to carry out the iteration. Since $\sqrt{2}$ is somewhere between 1 and 2 (the square roots of 1 and 4 respectively), we will take 1.5 as our initial guess. The following MATLAB code creates Figure ....

```
>>f=inline('(x.^2+2)./(2*x)')
f =
Inline function:
```

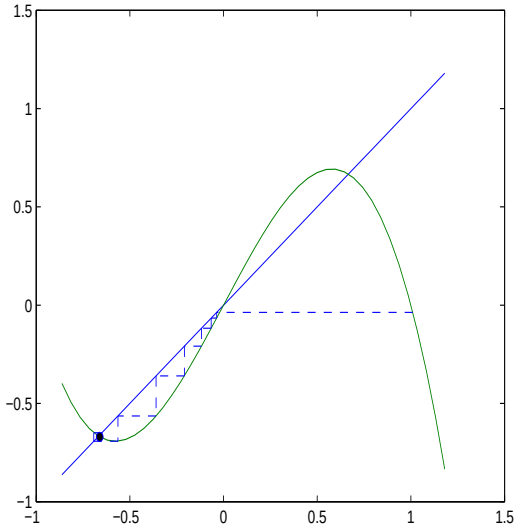

Figure 12.5: Cobwebbing for the recursion equation $x_{n+1} = 1.8(x_n - x_n^3)\, x_0 = 1.01$.

```
f(x) = (x.^2+2)./(2*x)
root2=cobweb(f,1.5,10)
root2 =
Columns 1 through 7
1.5000 1.4167 1.4142 1.4142 1.4142 1.4142 1.4142
Columns 8 through 11
1.4142 1.4142 1.4142 1.4142
```

Notice that the convergence is very fast: four decimal places of accuracy have been obtained by the second iteration.

12.3 Assignments

1. [4 pts] In Secton 2.3 of the course text, the author studies the recursion relation

$$x_{n+1} = rx_n(1 - x_n), \quad x_0 = .2.$$

for several different values of r (see particularly Figures 2.18 through 2.22 in the text). Use *cobweb.m* to create cobweb diagrams corresponding with $r = 2, 3.2, 3.52, 3.8$, with 10 iterations each. (These correspond respectively with the author's Figures 2.18 through 2.21.) In each case explain how the behavior of your cobwebbing diagram corresponds with the author's figure.

2. [3 pts] Use Newton's method to approximate a value for $\sqrt{149}$ to four decimal places.

3. [3 pts] Find the minimum value of $f(x) = (\ln x)^2 + x$ to four decimal places.

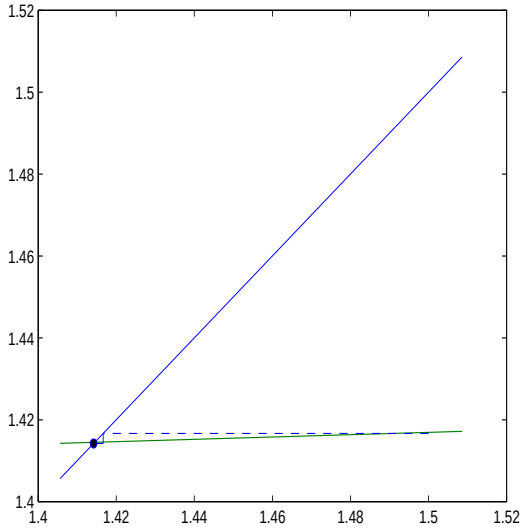

Figure 12.6: Cobwebbing for the iteration $x_{n+1} = \frac{x_n^2+2}{2x_n}$, $x_0 = \frac{1}{2}$.

13 Riemann Sums in MATLAB

In this section we will review the concept of Riemann summation and use MATLAB to directly compute a few example Riemann sums.

13.1 Riemann Sums

Definition 13.1. Let $f(x)$ be a function on an interval $[a, b]$, and suppose this interval is partitioned by the values $a = x_0 < x_1 < ... < x_{n-1} < x_n = b$. Any sum of the form

$$R = \sum_{k=1}^{n} f(c_k) \triangle x_k,$$

where $\triangle x_k = x_k - x_{k-1}$ and $c_k \in [x_{k-1}, x_k]$ is referred to as a *Riemann sum* of f.

If we let P denote our choice of partition (the choice of values x_0, x_1, ..., x_n), and we let

$$\|P\| := \max(\triangle x_1, \triangle x_2, ..., \triangle x_n)$$

denote the norm of this partition, then we say f is *Riemann integrable* if

$$\lim_{\|P\| \to 0} \sum_{k=1}^{n} f(c_k) \triangle x_k$$

exists. Following Leibniz, our notation for this limit is

$$\int_a^b f(x)dx = \lim_{\|P\| \to 0} \sum_{k=1}^{n} f(c_k) \triangle x_k.$$

Example 13.1. As our first example, we will consider the case in which c_k is always chosen as the right endpoint of the interval $[x_{k-1}, x_k]$. If we take a regular partition with n intervals, then each interval has length $\triangle x = \frac{b-a}{n}$, and the k^{th} endpoint is

$$x_k = a + k\triangle x.$$

The Riemann sum becomes

$$R = \sum_{k=1}^{n} f(a + k\triangle x)\triangle x.$$

Suppose we would like to approximate the integral

$$\int_0^2 e^{-x^2} dx$$

with $n = 4$. We have $\triangle x = \frac{2-0}{4} = .5$ and the values

$$\begin{aligned} x_0 &= 0 \\ x_1 &= .5 \\ x_2 &= 1 \\ x_3 &= 1.5 \\ x_4 &= 2.0. \end{aligned}$$

The Riemann sum is

$$R = \sum_{k=1}^{4} f(0 + .5k).5 = .5(e^{-.5^2} + e^{-1^2} + e^{-1.5^2} + e^{-2^2}) = .6352.$$

More generally, we can write a MATLAB function M-file that carries out this calculation for any function f (defined as an inline function), endpoints a and b and regular partition with n points. See *rsum1.m*.[7]

```
function value=rsum1(f,a,b,n)
%RSUM1: Computes a Riemann Sum for the function f on
%the interval [a,b] with a regular partition of n points.
%The points on the intervals are chosen as the right endpoints.
value = 0;
dx = (b-a)/n;
for k=1:n
c = a+k*dx;
value = value + f(c);
end
value = dx*value;
```

We run this in MATLAB with the following lines in the Command Window.

[7]The MATLAB M-files used in these notes are available from the course web site.

```
>>f=inline('exp(-x^2)')
f =
Inline function:
f(x) = exp(-x^2)
>>rsum1(f,0,2,4)
ans =
0.6352
>>rsum1(f,0,2,10)
ans =
0.7837
>>rsum1(f,0,2,100)
ans =
0.8723
>>rsum1(f,0,2,1000)
ans =
0.8811
```

To four decimal places, the correct value is .8821. △

One interesting aspect of the Riemann sum is that the points c_k need not be chosen in the same place on each interval. That is, suppose we partition the interval $[0, 1]$ with $0 = x_0 < x_1 = \frac{1}{2} < x_2 = 1$. In this case, a possible Riemann sum is

$$f(0)\frac{1}{2} + f(1)\frac{1}{2}.$$

Here $\triangle x_1 = \triangle x_2 = \frac{1}{2}$, and we have chosen c_1 as the left endpoint of the interval $[0, \frac{1}{2}]$ and c_2 as the right endpoint of the interval $[\frac{1}{2}, 1]$.

Example 13.2. As our second example, we will consider the case in which c_k is randomly selected on the interval $[x_{k-1}, x_k]$. In this case, we revise *rsum1.m* into *rsum2.m*.

```
function value=rsum2(f,a,b,n)
%RSUM2: Computes a Riemann Sum for the function f on
%the interval [a,b] with a regular partition of n points.
%The points on the intervals are chosen randomly.
value = 0;
dx = (b-a)/n;
for k=1:n
c = dx*rand + (a + (k-1)*dx)
value = value + f(c);
end
value = dx*value;
```

The only tricky line here is the one that defines c_k as a random number on the interval $[x_{k-1}, x_k]$. In order to avoid a digression on the simulation of random variables, let's simply

accept that this line works.[8] The implementation is as follows (assuming $f(x) = e^{-x^2}$ has already been defined as an inline function).

```
>>rsum2(f,0,2,5)
c =
0.0390
c =
0.5114
c =
1.0188
c =
1.5830
c =
1.9860
ans =
0.8894
>>rsum2(f,0,2,5)
c =
0.0630
c =
0.7882
c =
1.1829
c =
1.3942
c =
1.9201
ans =
0.7793
```

Observe that the values of c differ for each run of the program. The first value lies in the interval $[0, .4]$, the second in $[.4, .8]$ etc. We can get better accuracy by increasing the size of the partition. (In this case a semicolon was added to the line defining c to suppress the output.)

```
>>rsum2(f,0,2,10)
ans =
0.8665
>>rsum2(f,0,2,100)
ans =
0.8818
```

[8] Okay, if you're really curious. Given any interval $[a, b]$, the points $c = (1 - r)a + rb$ move from a to b as r goes from 0 to 1. Our generic interval is $[a + (k - 1)\triangle x, a + k\triangle x]$ and *rand* denotes a value randomly chosen between 0 and 1. Therefore $c = (1 - \text{rand})(a + (k - 1)\triangle x) + \text{rand}(a + k\triangle x)$, which gives the formula we're using.

```
>>rsum2(f,0,2,1000)
ans =
0.8821
```

Notice that your values may vary slightly from these due to the random nature of the program. Nonetheless, you should find that Riemann sums based on random values are actually converging more rapidly to the correct solution than did Riemann sums based on right endpoints. The reason for this is that when the c_k are chosen randomly the area approximations on some subintervals are too large (a positive error), while the area approximations on others are too small (a negative error). Under summation, these errors can cancel, leading to a better approximation of the total area.

13.2 Assignments

1. [5 pts] Alter the M-file *rsum1.m* so that it computes Riemann sums of the given function by taking the values c_k as the left endpoints of each interval. Use your M-file to estimate

$$\int_0^2 e^{-x^2} dx$$

for regular partitions with $n = 10, 100, 1000$ and compare your result with those obtained using right endpoints and those obtained using randomly selected points.

2. [5 pts] Alter the M-file *rsum1.m* so that it computes Riemann sums of the given function by taking the values c_k as the midpoints of each interval. Use your M-file to estimate

$$\int_0^2 e^{-x^2} dx$$

for regular partitions with $n = 10, 100, 1000$ and compare your result with those obtained using right endpoints and those obtained using randomly selected points.

14 Symbolic and Numerical Integration in MATLAB

In this section we introduce MATLAB's built-in functions for symbolic and numerical integration.

14.1 Symbolic Integration in MATLAB

Certain functions can be symbolically integrated in MATLAB with the *int* command.

Example 14.1. Find an antiderivative for the function

$$f(x) = x^2.$$

We can do this in (at least) three different ways. The shortest is:

```
>>int('x^2')
ans =
1/3*x^3
```

Alternatively, we can define x symbolically first, and then leave off the single quotes in the *int* statement.

```
>>syms x
>>int(x^2)
ans =
1/3*x^3
```

Finally, we can first define f as an inline function, and then integrate the inline function.

```
>>syms x
>>f=inline('x^2')
f =
Inline function:
>>f(x) = x^2
>>int(f(x))
ans =
1/3*x^3
```

In certain calculations, it is useful to define the antiderivative as an inline function. Given that the preceding lines of code have already been typed, we can accomplish this with the following commands:

```
>>intoff=int(f(x))
intoff =
1/3*x^3
>>intoff=inline(char(intoff))
intoff =
Inline function:
intoff(x) = 1/3*x^3
```

The inline function *intoff(x)* has now been defined as the antiderivative of $f(x) = x^2$. △

The *int* command can also be used with limits of integration.

Example 14.2. Evaluate the integral

$$\int_1^2 x \cos x dx.$$

In this case, we will only use the first method from Example 2.1, though the other two methods will work as well. We have

```
>>int('x*cos(x)',1,2)
ans =
cos(2)+2*sin(2)-cos(1)-sin(1)
>>eval(ans)
ans =
0.0207
```

Notice that since MATLAB is working symbolically here the answer it gives is in terms of the sine and cosine of 1 and 2 radians. In order to force MATLAB to evaluate this, we use the *eval* command. △

For many functions, the antiderivative cannot be written down in a closed form (as the sum of a finite number of terms), and so the *int* command cannot give a result. As an example, the function

$$f(x) = e^{-x^2}$$

falls into this category of functions. If we try *int* on this function, we get:

```
int('exp(-xˆ2)')
ans =
1/2*piˆ(1/2)*erf(x)
```

where by *erf(x)* MATLAB is referring to the function

$$\text{erf}(x) := \frac{2}{\sqrt{\pi}} \int_0^x e^{-y^2} dy,$$

which is to say, MATLAB hasn't actually told us anything. In cases like this, we can proceed by evaluating the integral numerically.

14.2 Numerical Integration in MATLAB

MATLAB has two primary tools for the numerical evaluation of integrals of real-valued functions, the *quad* command which uses an adaptive Simpson's method (we will discuss Simpson's method in Section 4 of these notes) and the *quadl* command which uses an adaptive Lobatto method (we won't discuss the Lobatto method in these notes).

Example 14.3. Evaluate the integral

$$\int_1^2 e^{-x^2} dx.$$

We use

```
quad('exp(-x.^2)',1,2)
ans =
0.1353
```

The *quad* command requires an input function that can be appropriately evaluated for vector values of the argument, and so we have used an array operation. △

The *quad* command can also be used to evaluate functions defined in M-files. In this way it's possible to integrate functions that have no convenient closed form expression.

Example 14.4. Evaluate the integral

$$\int_0^{10} y(x)dx,$$

where y is implicitly defined by the relationship

$$x = y^3 + e^y.$$

In this case, we cannot solve explicity for y as a function of x, and so we will write an M-file that takes values of x as input and returns the associated values of y as output by numerically solving the algebraic equation.

```
function value = yfunction(x)
f=inline('x-y^3-exp(y)','x','y');
for k=1:length(x)
value(k) = fzero(@(y) f(x(k),y), .5);
end
```

The *for* loop is necessary so that the function *yfunction* can be evaluated at vector values for the independent variable x, as required by the *quad* command. We find

```
quad(@yfunction,0,10)
ans =
9.9943
```

(There is also an alternative approach to this type of problem that involves relating the integral of $y(x)$ to the integral of $x(y)$, but that's not the topic of this section.) △

14.3 Assignments

1. [2 pts] Find an antiderivative for the function

$$f(x) = x \sin^2 x.$$

2. [2 pts] Evaluate the integral

$$\int_1^2 x \sin^2 x dx.$$

3. [2 pts] Evaluate the integral

$$\int_1^2 \sin(x^2)dx.$$

4. [4 pts] Evaluate the integral

$$\int_0^2 y(x)dx,$$

where y is defined implicitly by the relation

$$x = y + \sin y.$$

Index