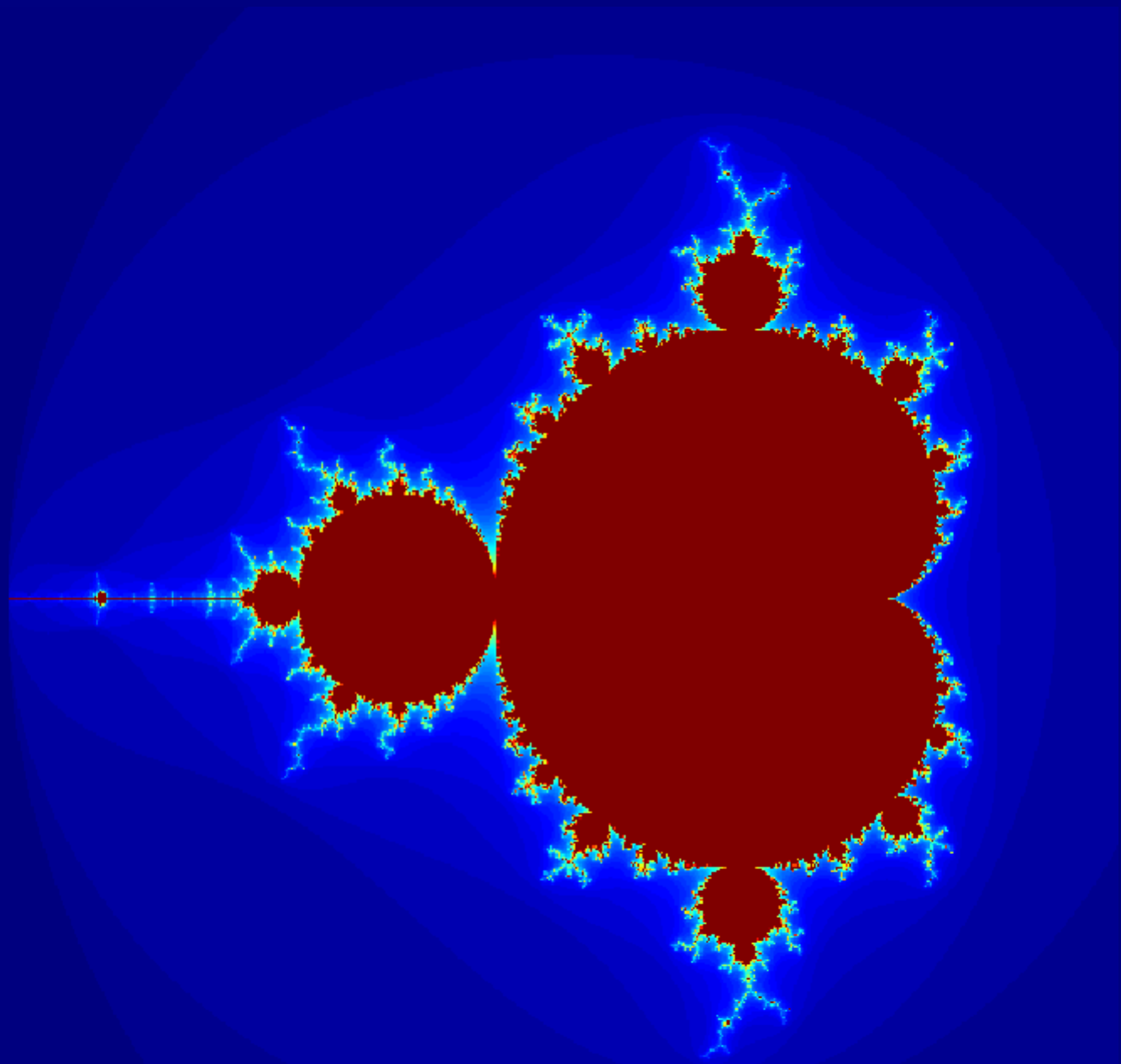
Introduction to GNU Octave

A brief tutorial for linear algebra and calculus students



by Jason Lachniet

Introduction to GNU Octave

A brief tutorial for linear algebra and calculus students

Jason Lachniet
Wytheville Community College
jlachniet@wcc.vccs.edu

First Edition

ISBN 978-1-365-98319-1

Corrected 1st Ed. (May 12, 2018)

Contents

Preface

These notes are not intended as a comprehensive manual. Instead, what follows is a tutorial that puts Octave to work solving a selection of applied problems in linear algebra and calculus. The goal is to learn enough of the basics to begin solving problems with minimum frustration. Note that minimum frustration does not mean no frustration. Be patient!

Features of the text

To get the most out of this book, you should read it alongside an open Octave window where you can follow along with the computations (you will want paper and pencil, too, as well as your math books). Blocks of Octave commands are indented and printed with special formatting as follows.

```
>> % example Octave commands:
>>
>> x = [-3 : 0.1 : 3];
>> plot(x, x.^2);
>> title('Example plot')
```

Comments used to explain the code are preceded by a "%" sign and shown in green. Key words are highlighted in magenta. Strings (text variables) are highlighted in purple. The same formatting is used for commands that appear inline in the text. The Octave prompt is shown as ">>".

Octave scripts (.m-files) are shown between horizontal rules and are labeled with a title, as in the following example. These are short programs in the Octave language.

Octave Script 1: Example

```
1 % This is an example Octave script (.m-file)
2 t = linspace(0, 2*pi, 50);
3 x = cos(t);
4 y = sin(t);
5
6 % plot the graph of a unit circle
7 plot(x, y);
8 grid on;
```

The line numbers are for reference purposes and are not part of the code.

The color coding is not essential to understand the text. Thus the text can be printed in black and white to save on printing costs.

If you are reading the electronic PDF version, there are numerous hyperlinks throughout the text that link back to other parts of the text, or to external urls. There is a set of bookmarks to each chapter and section that can be used to easily navigate from section to section. Open the bookmark link at the left side of the screen in your PDF viewer to use this feature (not visible in a web browser view; use a full PDF reader, like https://get.adobe.com/reader/).

> Solutions to the many example problems are offset with a bar along the left side of the page, as shown here. A box signifies the end of the example. □

MATLAB

The majority of the code shown in this book will work in MATLAB. This guide can therefore also be used an introduction to that software package. Refer to Appendix A for some notes on MATLAB compatibility.

Scope and purpose

This guide is heavy on linear algebra and makes a good supplement to a linear algebra textbook. But, it is assumed that any college student studying linear algebra will also be studying calculus and differential equations, maybe statistics. Therefore it makes sense to apply the Octave skills learned for linear algebra to these subjects as well. Chapters 3 and 5 have several applications to calculus, differential equations, and statistics. The overarching objective is to enhance our understanding of calculus and linear algebra using Octave as a tool for computations. For the most part, we will not address issues of accuracy and round-off error in machine arithmetic. For more details about numerical issues, refer to [1], which also contains many useful Octave examples.

To get started, read Chapter 1, without worrying too much about any of the mathematics you don't yet understand. After grasping the basics, you should be able to move into any of the chapters or sections that interest you.

Every chapter concludes with a set of problems, some of which are routine practice, and some of which are more extended applied projects.

Most examples assume the reader is familiar with the mathematics involved. In a few cases, more detailed explanation of relevant theorems is given by way of motivation, but there are no proofs. Refer to the linear algebra and calculus books listed in the references for background on the underlying mathematics. In the spirit of openness, all references listed are available for free under GNU or Creative Commons licenses and can be accessed using the links provided.

Chapter 1

Basic operation

1.1 Introduction

1.1.1 What is GNU Octave?

GNU Octave is free software designed for scientific computing. It is intended primarily for solving numerical problems. In linear algebra, we will use Octave's capabilities to solve systems of linear equations and to work with matrices and vectors. Octave can also generate sophisticated plots. For example, we will use it in vector calculus to plot vector fields, space curves, and three dimensional surfaces. Octave is mostly compatible with the popular "industry standard" commercial software package MATLAB, so the skills you learn here can be applied to MATLAB programming as well. In fact, while this guide is written and intended as an introduction to Octave, it can serve equally well as a basic introduction to MATLAB.

What is GNU? A gnu is a type of antelope, but GNU is a free, UNIX-like computer operating system. GNU is a recursive acronym that stands for "GNU's not Unix." GNU Octave (and many other free programs) are licensed under the GNU General Public License: http://www.gnu.org/licenses/gpl.html.

From www.gnu.org/software/octave:

> GNU Octave is a high-level interpreted language, primarily intended for numerical computations. It provides capabilities for the numerical solution of linear and non-linear problems, and for performing other numerical experiments. It also provides extensive graphics capabilities for data visualization and manipulation. Octave is normally used through its interactive command line interface, but it can also be used to write non-interactive programs. The Octave language is quite similar to MATLAB so that most programs are easily portable.

Octave is a fully functioning programming language, but it is not a general purpose programming language (like C or Java). Octave is numerical, not symbolic; it is not a computer algebra system

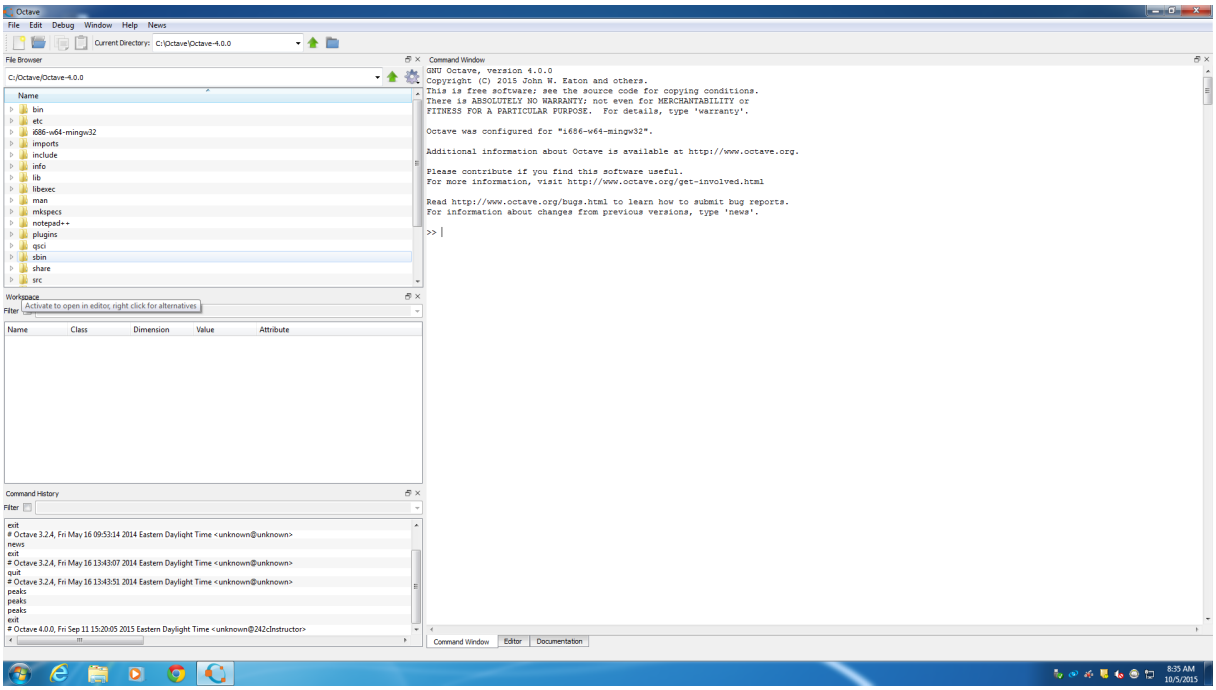


Figure 1.1: Windows Octave GUI

(like Maple, Mathematica, or Sage). However, Octave is ideally suited to all types of numeric calculations and simulations. Matrices are the basic variable type and the software is optimized for vectorized operations.

1.1.2 Installing Octave

It's free! Octave will work with Windows, Macs, or Linux. Go to https://www.gnu.org/software/octave/download.html and look for the download that matches your system. For example, Windows users can find an installer for the current Windows version at https://ftp.gnu.org/gnu/octave/windows/. Manual installation can be tricky, so look for the most recent .exe installer file and run that. Installation in most Linux systems is easy. For example, in Debian/Ubuntu, run the command `sudo apt-get install octave`. If you find Octave useful, consider making a donation to support the project at https://www.gnu.org/software/octave/donate.html.

Beginning with version 4.0, Octave uses a graphical user interface (GUI) by default. When you start Octave, you should see something like Figure 1.1.

The user can customize the arrangement of windows. By default, you will have a large command window, which is where commands are entered and run, a file browser, a workspace window displaying the variables in the current scope, and a command history.

1.1.3 Getting started

There are several good help resources on the web, and built-in help functions within Octave. The shell command help can be used at the Octave prompt. In particular, if you know the name of the command you want to use, help NAME will give the correct syntax.

Here are two good free, online resources:

- The Octave Manual [3]:
  http://www.gnu.org/software/octave/octave.pdf
- Wikibooks Tutorial:
  https://en.wikibooks.org/wiki/Octave_Programming_Tutorial

Additional help can be found with internet searches. Depending on what you are looking for, searches for Octave commands and searches for MATLAB commands can both be useful. Numerous commercial user's guides and textbooks for Octave and/or MATLAB are available. Linear algebra textbooks sometimes contain MATLAB code examples and these generally work in Octave as well.

The best way to get started is to try some simple problems. Use the following examples as a tutorial to learn your way around the program. Octave knows about basic arithmetic. Try something simple like:

```
>> 2*6 + (7 - 4)^2
ans = 21
```

Octave ignores white space, so 2∗6 and 2 ∗ 6 are interpreted the same way. You can't take shortcuts and leave out implied operations, though. For example, 2(5 − 1) will give an error. Use 2∗(5 − 1).

Vectors and matrices are basic variable types, so it is easier to learn Octave syntax if you already know a little linear algebra. Try this example to enter a row vector and name it **u**. You do not need to enter the comments (indicated by the % sign).

```
>> u = [1 -4 6]    % row vector
u =                % variable name

   1   -4    6     % output
```

The code u = ... assigns the result of the operation that follows to the variable u, which can then be recalled and used in further calculations.

To create a column vector instead, use semicolons:

```
>> u = [1; -4; 6]    % column vector
u =                  % variable name

   1                 % output
  -4
   6
```

Notice that the function of the semicolon is to begin a new row. The same basic syntax is used to enter matrices. For example, let's see how to enter a matrix:

```
>> A = [1  2  -3; 2  4  0; 1  1  1]    % matrix
A =                                     % variable name

   1   2  -3                           % output
   2   4   0
   1   1   1
```

1.2 Matrices and vectors

Matrices are the basic variable type in Octave. In fact, a scalar is treated as a 1×1 matrix. Similarly, a row vector is a $1 \times n$ matrix and a column vector is an $m \times 1$ matrix.

1.2.1 Vector operations

We'll start with some simple examples. First, enter the column vector **u** from above, if it is not already in memory.

```
>> u = [1; -4; 6]
u =

    1
   -4
    6
```

Now enter another column vector **v** and try the following vector operations which illustrate linear combinations, dot product, cross product, and norm.

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

  2
  1
 -1

>> 2*v + 3*u
ans =

   7
 -10
  16

>> dot(u, v)          % dot product
ans = -8


>> cross(u, v)        % cross product
```

```
ans =

  -2
  13
   9

>> norm(u)          % length of vector u
ans =   7.2801
```

Try a few more operations:

- Find cross(v, u). How does that compare to $\mathbf{u} \times \mathbf{v}$?
- Calculate the length of $\mathbf{v}$, $||\mathbf{v}||$, using norm(v).
- Create a unit vector $\mathbf{v}_1$ that points in the direction of $\mathbf{v}$.

1.2.2 Projections

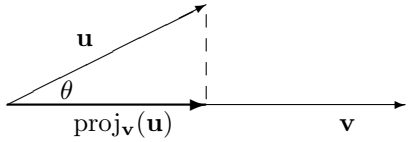


Figure 1.2: Vector projection

The *projection of* $\mathbf{u}$ *onto* $\mathbf{v}$, denoted $\text{proj}_\mathbf{v}(\mathbf{u})$, is the component of $\mathbf{u}$ that points in the direction of $\mathbf{v}$. This can be thought of as the shadow $\mathbf{u}$ casts onto $\mathbf{v}$ from a direction orthogonal to $\mathbf{v}$, as shown in the figure. To find the magnitude of the projection, use basic right-triangle trigonometry:

$$\|\text{proj}_\mathbf{v}(\mathbf{u})\| = \|\mathbf{u}\| \cos(\theta)$$

Then, since $\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\| \cos(\theta)$,

$$\begin{aligned} \|\text{proj}_\mathbf{v}(\mathbf{u})\| &= \|\mathbf{u}\| \cos(\theta) \\ &= \|\mathbf{u}\| \frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|} \\ &= \frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{v}\|} \end{aligned}$$

This is known as the *scalar projection* of $\mathbf{u}$ onto $\mathbf{v}$. The *vector projection* onto $\mathbf{v}$ is obtained by multiplying the scalar projection by a unit vector that points in the direction of $\mathbf{v}$. Thus,

$$\begin{aligned} \text{proj}_\mathbf{v}(\mathbf{u}) &= \frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{v}\|} \frac{\mathbf{v}}{\|\mathbf{v}\|} \\ &= \frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{v}\|^2} (\mathbf{v}) \end{aligned}$$

Since $\mathbf{v} \cdot \mathbf{v} = \|\mathbf{v}\|^2$, this can also be written as:

$$\text{proj}_\mathbf{v}(\mathbf{u}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\mathbf{v} \cdot \mathbf{v}} (\mathbf{v})$$

Example 1.2.1. Find the projection of $\langle 3, 5\rangle$ onto $\langle 7, 2\rangle$.

Solution. The operations needed for vector projection are easily carried out in Octave.

```
>> u = [3  5]
u =

   3   5

>> v = [7  2]
v =

   7   2

>> proj = dot(u,v)/(norm(v))^2*v
proj =

   4.0943   1.1698
```

Thus $\text{proj}_{\mathbf{v}}(\mathbf{u}) = \langle 4.0943, 1.1698\rangle$. □

1.2.3 Matrix operations

Matrix operations are carried out very easily. We'll start with matrix multiplication.

Example 1.2.2. Let $A = \begin{bmatrix} 1 & 2 & -3 \\ 2 & 4 & 0 \\ 1 & 1 & 1 \end{bmatrix}$ and $B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & -2 & -4 & 6 \\ 1 & -1 & 0 & 0 \end{bmatrix}$. Find AB.

Solution.

```
>> A = [1  2  -3; 2  4  0; 1  1  1]    % matrix
A =                                     % variable name

   1   2  -3                            % output
   2   4   0
   1   1   1

>> B = [1  2  3  4; 0  -2  -4  6; 1  -1  0  0]
B =

   1   2   3   4
   0  -2  -4   6
   1  -1   0   0

>> A*B                                  % multiply A and B
ans =                                   % result stored as 'ans'

  -2   1  -5  16                        % answer
   2  -4 -10  32
   2  -1  -1  10
```

Notice that the result is stored in the temporary variable ans. □

Arithmetic operations in Octave are always assumed to be matrix operations. Therefore, for A and B defined as above, we can compute things like $4A$ or AB by entering 4*A or A*B, but operations like B*A or A+B give errors (why?).

To get the transpose of a matrix, use the single quote. For example, try calculating $B^T A$.

```
>> B'*A          % B' is the transpose of B
ans =

   2    3   -2
  -3   -5   -7
  -5  -10   -9
  16   32  -12
```

To perform basic matrix arithmetic, we also need the identity matrix. This is easy to do in Octave with the eye(n) command, where n is the dimension of the matrix. Let's find $2A - 4I$.

```
>> 2*A - 4*eye(3)        % eye(3) is a 3x3 identity matrix
ans =

  -2   4  -6
   4   4   0
   2   2  -2
```

Octave can also find determinants, inverses, and eigenvalues. For example, try these commands.

```
>> det(A)                              % determinant
ans = 6

>> inv(A)                              % matrix inverse
ans =

   0.66667   -0.83333    2.00000
   0.33333    0.66667   -1.00000
   0.33333    0.16667    0.00000

>> eig(A)                              % eigenvalues
ans =

   4.52510 + 0.00000i
   0.73745 + 0.88437i
   0.73745 - 0.88437i
```

Notice that our matrix has one real and two complex eigenvalues. Octave handles complex numbers, of course! Eigenvalues will be discussed in more detail in Chapter 4. Octave can also compute many other matrix values, such as rank:

```
>> rank(A)     % matrix rank
ans =  3
```

1.2.4 Saving your work

If we have solved some problems, we are going to want some way to save our work and maybe reload it later. In Octave, you can save variables that you defined in your session, but you cannot save the commands you used or a whole worksheet. Octave does have a command history that persists between sessions, so past commands can be brought up using the up arrow key, or using the command history list in the GUI. If you want to document how you did something, use copy and paste to copy your commands into a Word document or text file.

Within the Octave graphical user interface, you should see your current directory listed near the top left. You can click the folder button to navigate to a different directory, such as the desktop or your personal flash drive. Under the file menu, the option "save workspace as" will save all of your current variables in a file of your choosing. You can see a list of the variables currently defined listed under "workspace" on the left side of the screen. You can use the "load workspace" option under the file menu to load previously saved variables.

Another approach is to use the manual save and load commands at the command line. If you type save FILENAME var1 var2 ..., Octave will save the specified variables in the file FILENAME. If you do not supply a list of variables, then all variables in the current scope will be saved. You can then reload the saved variable(s) at another time by navigating to the appropriate directory and using load FILENAME. You can also load a variable or workspace by double-clicking on its name in the file browser.

If you want to save a series of commands that can be reopened and run again, you can create an Octave script, also known as an .m-file. This will be described in more detail in Chapter 3.

1.3 Plotting

Basic two-dimensional plotting of functions in Octave is accomplished by creating a vector for the independent variable and a second vector for the range of the function. There are several forms for the syntax and we will attempt to outline the simplest methods here. See also:

- http://www.gnu.org/software/octave/doc/interpreter/Plotting.html
- http://en.wikibooks.org/wiki/Octave_Programming_Tutorial/Plotting

Let's start by plotting the graph of the function $\sin(x)$ on the interval $[0, 2\pi]$. Like a typical graphing calculator, Octave will simply plot a series of points and connect the dots to represent the curve. The process is less automated in Octave (but in the end, much more powerful). We begin by creating a vector of x-values.

```
>> x = linspace(0, 2*pi, 50);
```

Notice the format linspace(start_val , end_val, n). This creates a row vector of 50 evenly spaced values beginning at 0 and going up to 2π. The smaller the increment, the smoother the curve will look. In this case, 50 points should be suitable. The semicolon at the end of the line is to

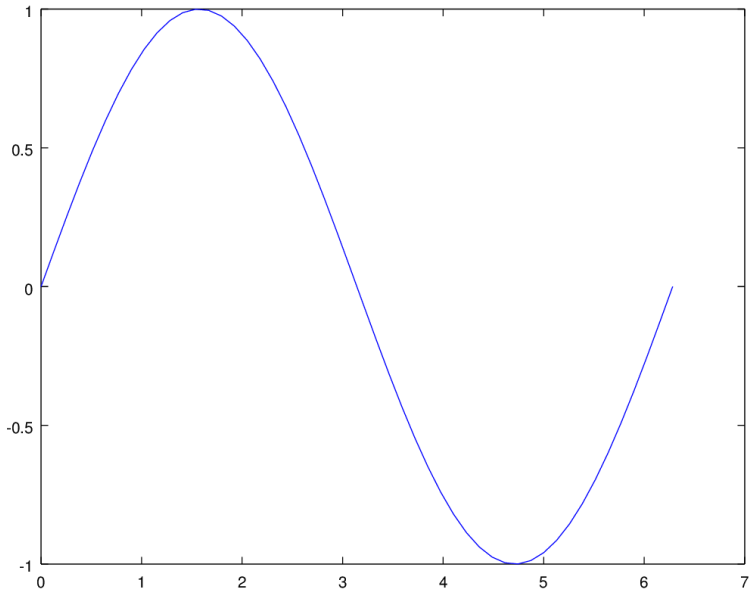


Figure 1.3: Default graph of $y = \sin(x)$ on $[0, 2\pi]$

suppress the output to the screen, since we don't need to see all the values in the vector. Now, we want to create a vector of the corresponding y-values. Use this command:

```
>> y = sin(x);
```

Now, to plot the function, use the plot command:

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

You should see the graph of $f(x) = \sin(x)$ as a thin blue line pop up in a new window (like Figure 1.3).

This is the default graph. You may wish to customize it a little bit. For example, the x-axis extends too far. We can set the window with the axis command. The window is controlled by a vector of the form [Xmin Xmax Ymin Ymax]. Let's set the axes to match the domain and range of the function.

```
>> axis([0 2*pi -1 1]);
```

We may want to change the color (to, say, red) or make the line thicker. We can add a grid to help guide our eye. In addition, a graph should usually be labeled with a title, axis labels, and legend. Try these options to get the improved graph shown in Figure 1.4.

```
>> plot(x, y, 'r', 'linewidth', 3)
>> grid on
>> xlabel('x');
>> ylabel('y');
>> title('Sine graph');
>> legend('y=sin(x)');
```

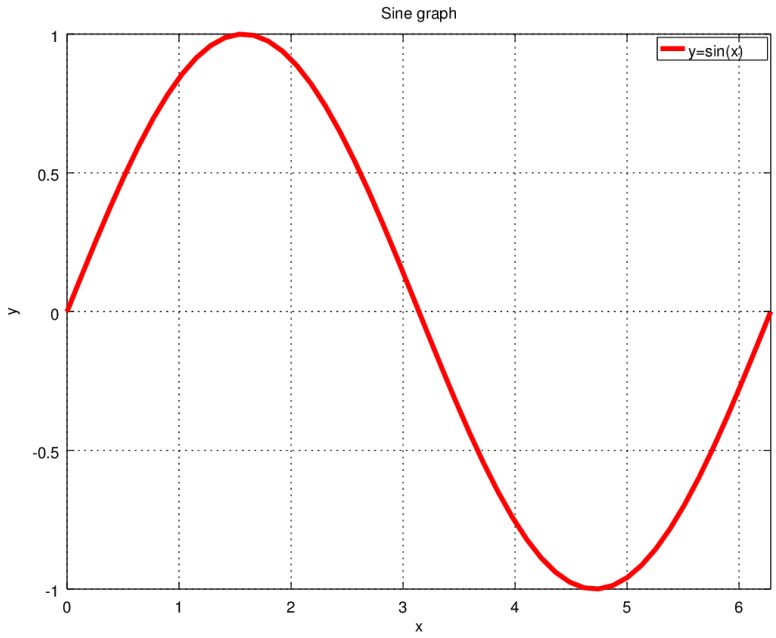


Figure 1.4: Improved graph of $y = \sin(x)$ on $[0, 2\pi]$

Note that some adjustments, like zooming in, or turning on the grid, can be done within the graph window using the controls provided. Some standard color options are red, green, blue, cyan, and magenta, which can be specified with their first letter in single quotes.

Now, let's try plotting points. The procedure is the same, but we use an option to specify the marker we want. Some marker options are o, +, or *. We will plot the set of points $\{(1,1),(2,2),(3,5),(4,4)\}$ using circles as our marker. First, clear the variables from the workspace and clear any existing graphs. Then define a vector of x-values and a vector of y-values and use the plot command.

```
>> clear; clf;
>> x = [1 2 3 4]
>> y = [1 2 5 4]
>> plot(x, y, 'o')
```

Now suppose we want to graph the line $y = 1.2x$ on the same set of axes (this is the line of best fit for this data). To add to our current graph we need to use the command hold on. Then any new plots will be drawn onto the current axes. We can switch back later with hold off.

```
>> hold on
>> plot(x, 1.2*x)
```

Now we should see four points and the graph of the line. Alternately, we can create multiple plots within a single plot command. Try this, for example:

```
>> clear; clf;
>> x = [1 2 3 4];
>> y1 = [1 2 5 4];
```

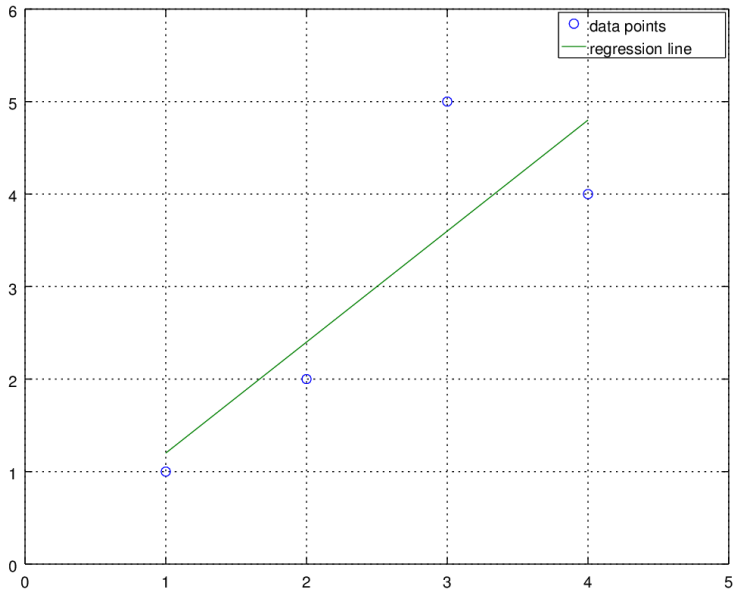


Figure 1.5: Scatter plot with regression line

```
>> y2 = 1.2*x;
>> plot(x, y1, 'o', x, y2)
>> axis([0 5 0 6]);
>> grid on;
>> legend('data points', 'regression line');
```

Notice that sets of input and output variables come in pairs, followed by any options that apply to that pair. The result is shown in Figure 1.5.

It would be good practice for you to try graphing some other functions. One thing to remember is that we are defining the independent variable x as a vector, so when we multiply, Octave will regard multiplication as matrix multiplication, unless we indicate otherwise. Likewise, division and exponentiation are interpreted as matrix operations. To graph a function such as $y = x^2 \sin(x)$, we need to use *elementwise* exponentiation and multiplication. This is done by preceding the operation with a period (as in, .^ or .*). For example, these commands will give an error:

```
>> x = linspace(-10, 10, 100);
>> plot(x, x^2*sin(x))
error: for A^b, A must be a square matrix. Use .^ for elementwise power.
error: evaluating argument list element number 2
```

But this will do the trick:

```
>> x = linspace(-10, 10, 100);
>> plot(x, x.^2.*sin(x))
```

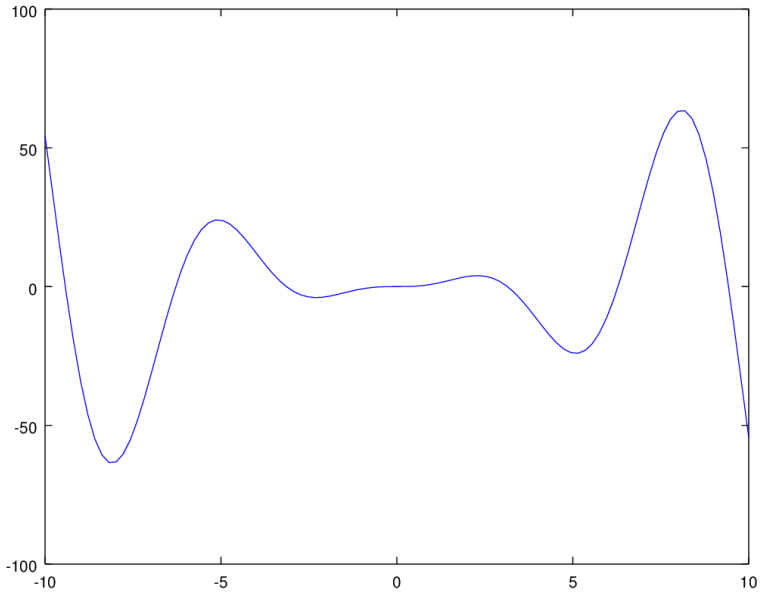


Figure 1.6: Graph of $y = x^2 \sin(x)$

Remember to use elementwise multiplication, division, and exponentiation! This is the source of many errors and frustration for beginning Octave users. The result is in Figure 1.6.

1.3.1 Plot options

The following table summarizes some standard plot options.

PLOT OPTIONS

MARKER	'+'	crosshair	COLOR	'k'	black
	'o'	circle		'r'	red
	'*'	star		'g'	green
	'.'	point		'b'	blue
	's'	square		'm'	magenta
	'^'	triangle		'c'	cyan
SIZE	'linewidth', n (where n is a positive value)				
	'markersize', n (where n is a positive value)				
LINE STYLE	'−'	solid line (default)			
	'−−'	dashed line			
	':'	dotted line			

Several options may be combined. For example, plot(x, y, 'ro:') indicates red color with circle markers joined by dotted lines.

1.3.2 Saving plots

If we have created a good plot, we probably want to save it. The easiest option is to use copy and paste from the plot window. You can also use the "save as" option under the file menu to save the plot as a PDF.

An alternate method is to save the plot directly by "printing" it to a file. Octave supports several image formats. In the example below, the PNG format is used. To save the current graph as a PNG, use this syntax:

```
>> print filename.png -dpng
```

Here "filename" is whatever file name you want. You can replace "png" with other image formats, such as "jpg" or "eps." Your file will be saved in your current working directory.

Chapter 1 Exercises

Begin each problem with no variables stored. You can clear any previous results with the command clear.

1. For practice saving and loading variables, try the following.
   (a) Create a new directory called "octave_projects".
   (b) Change to the octave_projects directory.
   (c) Save the example matrices A and B from above in a text file named "matrices.txt".
   (d) Quit Octave.
   (e) Restart Octave and reload the saved matrices.

2. Let $\mathbf{a} = \langle 2, -4, 0\rangle$ and $\mathbf{b} = \langle 3, 1.5, -7\rangle$. Find each of the following.
   (a) $\mathbf{x} = 2\mathbf{a} + 5\mathbf{b}$
   (b) $d = \mathbf{a} \cdot \mathbf{b}$
   (c) $l = ||\mathbf{a}||$
   (d) Find a vector $\mathbf{n}$ orthogonal to both $\mathbf{a}$ and $\mathbf{b}$.
   (e) Find $\text{proj}_{\mathbf{b}}(\mathbf{a})$.

   Be sure to use the variable names indicated to store your answers. Save your workspace including all of the required variables. What does the dot product reveal about $\mathbf{a}$ and $\mathbf{b}$? How did you produce a vector mutually orthogonal to $\mathbf{a}$ and $\mathbf{b}$?

3. Begin this problem with no variables stored. Enter the following matrices.

$$A = \begin{bmatrix} 1 & -3 & 5 \\ 2 & -4 & 3 \\ 0 & 1 & -1 \end{bmatrix}, B = \begin{bmatrix} 1 & -1 & 0 & 0 \\ -3 & 0 & 7 & -6 \\ 2 & 1 & -2 & -1 \end{bmatrix}, \text{ and } I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

   Use Octave to compute each of the following, if possible, or explain why the operation is undefined.
   (a) $d = \det(A)$
   (b) $C = 2A + 4I$
   (c) $D = A^{-1}$
   (d) $E = B^{-1}$
   (e) $F = BA$
   (f) $G = (AB)^T$
   (g) $H = B^T A^T$

   Use the variable names indicated to store your answers. Save your workspace including all of the required variables. Which of the operations were undefined and why? Did you notice anything about $(AB)^T$ and $B^T A^T$? If so, explain the relationship between these quantities.

4. Modify the plot of $y = x^2 \sin(x)$ given in Figure 1.6 as follows:
   (a) Make the graph of $y = x^2 \sin(x)$ a thick red line.
   (b) Graph $y = x^2$ and $y = -x^2$ on the same axes, as thin black dotted lines.
   (c) Use a legend to identify each curve.
   (d) Add a title.
   (e) Add a grid.
   (f) Save the plot as a PNG or JPG image file.

Chapter 2

Matrices and linear systems

Octave is a powerful tool for many problems in linear algebra. We have already seen some of the basics in Section 1.2. In this chapter, we will consider systems of linear equations, polynomial curve fitting, and rotation matrices.

2.1 Linear systems

2.1.1 Gaussian elimination

Octave has sophisticated algorithms built in for solving systems of linear equations, but it is useful to start with the more basic process of Gaussian elimination. Using Octave for Gaussian elimination lets us practice the procedure, without the inevitable arithmetic errors that come when doing elimination by hand. It also teaches useful Octave syntax and methods for manipulating matrices.

Row operations are easy to carry out. But first, we need to see how matrices and vectors are indexed in Octave. Consider the following augmented matrix.

```
>> B = [1  2  3  4; 0 -2 -4  6; 1 -1  0  0]
B =

   1   2   3  4
   0  -2  -4  6
   1  -1   0  0
```

If we enter B(2, 3), then the result given is -4. This is the scalar stored in row 2, column 3. We can also pull out an entire row vector or column vector using the colon operator. A colon can be used to specify a limited range, or if no starting or ending value is specified, it gives the full range. For example, B(1, :) will give every entry out of the first row.

```
>> B(1, :)
ans =
```

```
1   2   3   4
```

Now, let's use this notation to carry out basic row operations on B to reach row-echelon form.

Example 2.1.1. Let

$$B = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 0 & -2 & -4 & 6 \\ 1 & -1 & 0 & 0 \end{array}\right]$$

Use row operations to put B into row-echelon form, then solve by backward substitution. Compare to the row-reduced echelon form computed by Octave.

Solution. The first operation is to replace row 3 with -1 times row 1, added to row 3.

```
>> B(3, :) = (-1)*B(1, :) + B(3, :)
ans =

   1   2   3   4
   0  -2  -4   6
   0  -3  -3  -4
```

Next, we will replace row 3 with -1.5 times row 2, added to row 3.

```
>> B(3, :) = -1.5*B(2, :) + B(3, :)
ans =

   1   2   3    4
   0  -2  -4    6
   0   0   3  -13
```

The matrix is now in row echelon form. We could continue using row operations to reach row-reduced echelon form, but it is more efficient to simply write out the corresponding linear system on paper and solve by backward substitution. Do it! The solution vector is $\langle \frac{17}{3}, \frac{17}{3}, -\frac{13}{3} \rangle$. Of course, Octave also has a built-in command to find the row-reduced echelon form of the matrix directly. Try rref(B) to see the result.

```
>> rref(B)
ans =

   1.00000   0.00000   0.00000   5.66667
   0.00000   1.00000   0.00000   5.66667
   0.00000   0.00000   1.00000  -4.33333
```

From here, the solution to the system is evident. Notice that everything is now expressed as floating point numbers (i.e., decimals). Five decimal places are displayed by default. The variables are actually stored with higher precision and it is possible to display more decimal places, if desired (type format(long)). □

2.1.2 Left division

The built-in operation for solving linear systems of the form $A\mathbf{x} = \mathbf{b}$ in Octave is called *left division* and is entered as A\b. This is conceptually equivalent to the product $A^{-1}\mathbf{b}$. For example, let's go back to the matrix B given previously, which represents an augmented matrix.

Example 2.1.2. Use left division to solve the system of equations with augmented matrix B.

$$B = \left[\begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 0 & -2 & -4 & 6 \\ 1 & -1 & 0 & 0 \end{array}\right]$$

Solution. To use left division, we need to extract the coefficient matrix and vector of right-side constants. Let's call the coefficient matrix A and the right-side constants $\mathbf{b}$. (You have probably already noticed that Octave is case-sensitive.)

```
>> B = [1  2  3  4; 0 -2 -4  6; 1 -1 0  0]
B =

   1   2   3  4
   0  -2  -4  6
   1  -1   0  0

>> A = B(:, 1:3) % extract coefficient matrix
A =

   1   2   3
   0  -2  -4
   1  -1   0

>> b = B(:, 4)   % extract right side constants
b =

   4
   6
   0

>> A\b           % solve system Ax = b
ans =

   5.6667
   5.6667
  -4.3333
```

The solution vector matches what we found by Gaussian elimination. □

2.1.3 LU decomposition

LU decomposition is a matrix factorization that encodes the results of the Gaussian elimination algorithm. The goal is to write

$$A = LU$$

where L is a unit lower triangular matrix and U is an upper triangular matrix. We will see that this factored form can be used to easily solve $A\mathbf{x} = \mathbf{b}$.

The process is best explained with an example. We will not attempt to justify why the algorithm works; refer to [5] for the underlying theory.

Example 2.1.3. Find an LU decomposition for

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -2 & -4 \\ 1 & -1 & 0 \end{bmatrix}$$

Solution. This is the same coefficient matrix we row-reduced in Example 2.1.1. We proceed the same way, carefully noting the multiplier used to obtain each 0. The lower triangular L starts as an identity matrix, then the negative of each multiplier used in the elimination process is placed into the corresponding entry of L.

The first zero in position $(2, 1)$ is already there, so we put 0 for that multiplier in the corresponding position of L. Then we replace row 3 with -1 times row 1 plus row 3. The negative of this multiplier is $-(-1) = 1$, which is entered in L at the point where the 0 was obtained.

At this point, we have two entries for L along with a partly reduced A:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -2 & -4 \\ 1 & -1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & -2 & -4 \\ 0 & -3 & 4 \end{bmatrix}; L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

The next step is to replace row 3 using -1.5 times row 2. Thus we put $-(-1.5) = 1.5$ in the corresponding position of L. Once A has reached row echelon form, we have the desired upper triangular matrix U.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -2 & -4 \\ 1 & -1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & -2 & -4 \\ 0 & -3 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & -2 & -4 \\ 0 & 0 & 3 \end{bmatrix} = U$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1.5 & 1 \end{bmatrix}, U = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -2 & -4 \\ 0 & 0 & 3 \end{bmatrix}$$

So, to review, U is the row-echelon form of A and L is an identity matrix with the negatives of the Gaussian elimination multipliers placed into the corresponding positions where they were used to obtain zeros.

Let's check to see if it worked.

```
>> L = [1 0 0; 0 1 0; 1 1.5 1]
L =

    1.00000    0.00000    0.00000
    0.00000    1.00000    0.00000
    1.00000    1.50000    1.00000
```

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

    1    2    3
    0   -2   -4
    0    0    3

>> L*U
ans =

    1    2    3
    0   -2   -4
    1   -1    0
```

It worked! In fact, the procedure outlined in this example will work anytime Gaussian elimination can be performed *without row interchanges*. □

Now, let's see how the LU form can be used to solve linear systems $A\mathbf{x} = \mathbf{b}$. If $A = LU$, then the system $A\mathbf{x} = \mathbf{b}$ can be written as $LU\mathbf{x} = \mathbf{b}$. Let $U\mathbf{x} = \mathbf{y}$. Then we can proceed in two steps:

1. Solve $L\mathbf{y} = \mathbf{b}$.
2. Solve $U\mathbf{x} = \mathbf{y}$.

Since we are dealing with triangular matrices, each step is easy.

Example 2.1.4. Solve $A\mathbf{x} = \mathbf{b}$, where $A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -2 & -4 \\ 1 & -1 & 0 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} 4 \\ 6 \\ 0 \end{bmatrix}$, using LU decomposition.

Solution. We already have the LU decomposition. Since $L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1.5 & 1 \end{bmatrix}$, the first step is to solve:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1.5 & 1 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \\ 0 \end{bmatrix}$$

The corresponding systems of equations is

$$\begin{array}{rcrcrcl} y_1 & & & & & = & 4 \\ & & y_2 & & & = & 6 \\ y_1 & + & 1.5y_2 & + & y_3 & = & 0 \end{array}$$

Starting with the first row and working down, this system is easily solved by *forward substitution*. We can see that $y_1 = 4$ and $y_2 = 6$. Substituting these values into the third equation and solving for y_3 gives $y_3 = -13$. Thus the intermediate solution for $\mathbf{y}$ is $\begin{bmatrix} 4 \\ 6 \\ -13 \end{bmatrix}$.

Step two is to solve $U\mathbf{x} = \mathbf{y}$, which looks like:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -2 & -4 \\ 0 & 0 & 3 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 6 \\ -13 \end{bmatrix}$$

This is easily solved by backward substitution to get $\mathbf{x} = \begin{bmatrix} 17/3 \\ 17/3 \\ -13/3 \end{bmatrix}$. □

If row interchanges are used, then A is multiplied by a *permutation matrix* and the decomposition takes the form $PA = LU$. This is the default form of the LU decomposition given by Octave using the command [L U P] = lu(A).

Example 2.1.5. Find an LU decomposition (with permutation) for

$$A = \begin{bmatrix} -7 & -2 & 9 & 4 \\ -4 & -9 & 3 & 0 \\ -3 & 4 & 6 & -2 \\ 6 & 7 & -4 & -8 \end{bmatrix}$$

Solution. We will use Octave for this.

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

   -7   -2    9    4
   -4   -9    3    0
   -3    4    6   -2
    6    7   -4   -8

>> [L U P] = lu(A)
L =

    1.00000    0.00000    0.00000    0.00000
    0.57143    1.00000    0.00000    0.00000
   -0.85714   -0.67273    1.00000    0.00000
    0.42857   -0.61818    0.36000    1.00000

U =

   -7.00000   -2.00000    9.00000    4.00000
    0.00000   -7.85714   -2.14286   -2.28571
    0.00000    0.00000    2.27273   -6.10909
    0.00000    0.00000    0.00000   -2.92800

P =

Permutation Matrix

    1    0    0    0
    0    1    0    0
```

```
0     0     0     1
0     0     1     0
```

□

Refer to Exercise 4 to see how $PA = LU$ can be used to solve a linear system, using a method almost to identical to what we did in Example 2.1.4.

LU decomposition is widely used in numerical linear algebra. In fact, it is the basis of how Octave's left division operation works. It is especially efficient to use LU decomposition when one is solving several systems of equations that all have the same coefficient matrix, but different right side constants. The LU decomposition only needs to be done once for all of the systems with that coefficient matrix.

2.2 Polynomial curve fitting

In statistics, the problem of fitting a straight line to a set of data is often considered. We tackle the more general problem of fitting a polynomial to a set of points.

Example 2.2.1. Find the least-squares parabola for the set of points in the following 6×2 data matrix D.

$$D = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 5 \\ 4 & 4 \\ 5 & 2 \\ 6 & -3 \end{bmatrix}$$

The matrix shows x-values in column 1 and y-values in column 2.

Solution. Enter the data matrix in Octave and extract the x- and y-data to column vectors. Then plot the points to get a sense of what the data look like.

```
>> D = [1  1; 2  2; 3  5; 4  4; 5  2; 6  -3]
>> xdata = D(:, 1)
>> ydata = D(:, 2)
>> plot(xdata,ydata, 'o-')  % plot line segments with circle markers
```

In this case, we are constructing a model of the form $y = a + bx + cx^2$, but it is easy to see how our approach generalizes to polynomials of any degree (including linear functions). By plugging in the given data to the proposed equation, we obtain the following system of linear equations.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \\ 1 & 5 & 25 \\ 1 & 6 & 36 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 5 \\ 4 \\ 2 \\ -3 \end{bmatrix}$$

Notice the form of the coefficient matrix, which we'll call A. The first column is all ones, the second column is the x-values, and the third column is the square of the x-values (this column would not appear if we were using a linear model). The right-side constants are the y-values. There are several ways to construct the coefficient matrix in Octave. One approach is to use the ones command to create a matrix of ones of the appropriate size, and then overwrite the second and third columns with the correct data.

```
>> A = ones(6, 3);
>> A(:, 2) = xdata;
>> A(:, 3) = xdata.^2
A =

   1   1    1
   1   2    4
   1   3    9
   1   4   16
   1   5   25
   1   6   36
```

Note the use of elementwise exponentiation to square each value of the vector xdata. Our system is inconsistent. It can be shown that the least-squares solution comes from solving the *normal equations*, $A^TA\mathbf{b} = A^T\mathbf{y}$, where $\mathbf{b}$ is the vector $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ of polynomial coefficients.

We can use Octave to construct the normal equations.

```
>> A'*A
ans =

     6    21    91
    21    91   441
    91   441  2275

>> A'*ydata
   ans =

   11
   28
   60
```

The corresponding augmented matrix is:

$$\left[\begin{array}{ccc|c} 6 & 21 & 91 & 11 \\ 21 & 91 & 441 & 28 \\ 91 & 441 & 2275 & 60 \end{array}\right]$$

We can then solve the problem using Gaussian elimination. Here is one way to create the augmented matrix and row-reduce it:

```
>> B = A'*A;
>> B(:, 4) = A'*ydata;
```

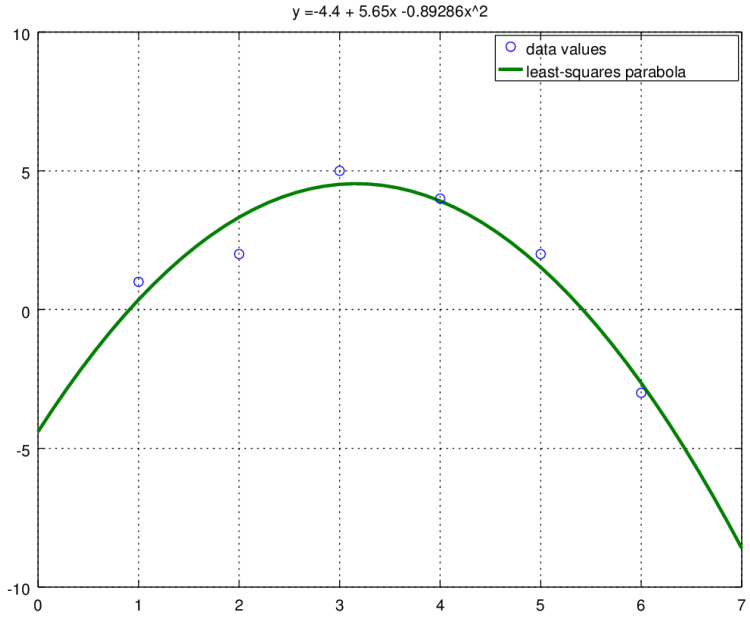


Figure 2.1: Least-squares parabola

```
>> rref(B)
ans =

   1.00000   0.00000   0.00000   -4.40000
   0.00000   1.00000   0.00000    5.65000
   0.00000   0.00000   1.00000   -0.89286
```

Thus the correct quadratic equation is $y = -4.4 + 5.65x - 0.89286x^2$. Figure 2.1 shows a graph of this parabola together with our original data points. □

These are the commands used to create Figure 2.1:

```
>> x = linspace(0, 7, 50);
>> y = -4.4 + 5.65*x - 0.89286*x.^2;
>> plot(xdata, ydata, 'o', x, y, 'linewidth', 2)
>> grid on;
>> legend('data values', 'least-squares parabola')
>> title('y = -4.4 + 5.65x - 0.89286x^2')
```

You may be wondering if any of this process can be "automated" by built-in Octave functions. Yes! If we want Octave to do all of the work for us, we can use the built-in function for polynomial fitting, polyfit. The syntax is polyfit(x, y, order), where "order" is the degree of the polynomial desired. For example, our previous problem can be directly solved as follows.

```
>> polyfit(xdata, ydata, 2)
ans =

   -0.89286    5.65000   -4.40000
```

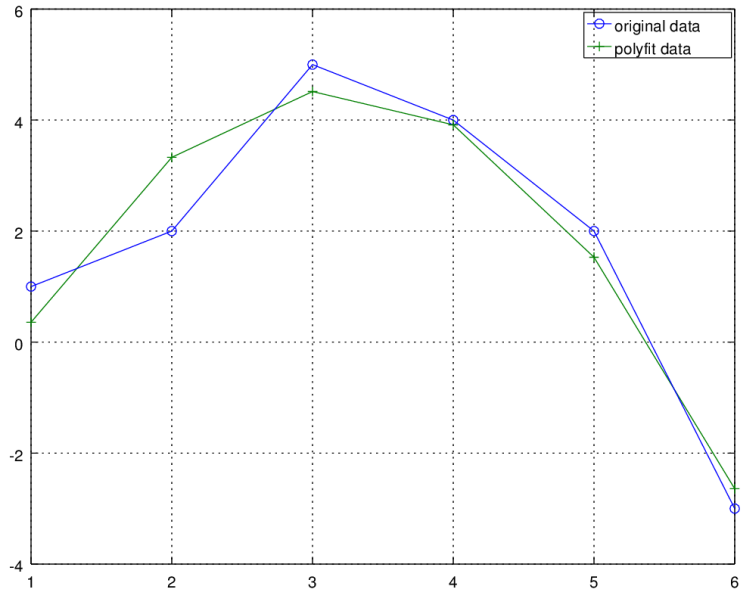


Figure 2.2: Plot of original data vs. polyfit data

Notice that the order of the coefficients is reversed, relative to how we had originally set up the problem. We can also use the built-in function polyval to evaluate the polynomial at the given x-values. This code, for example, will create the least-squares polynomial and plot the graph of the original data vs. the polyfit data, as shown in Figure 2.2:

```
>> P = polyfit(xdata, ydata, 2);
>> y = polyval(P, xdata);
>> plot(xdata, ydata, 'o-', xdata, y, '+-');
>> grid on;
>> legend('original data', 'polyfit data');
```

2.3 Matrix transformations

Matrices and matrix transformations play a key role in computer graphics. There are several ways to represent an image as a matrix. The approach we take here is to list a series of vertices that are connected sequentially to produce the edges of a simple graph. We write this as a $2 \times n$ matrix where each column represents a point in the figure. As a simple example, let's try to encode a 'house graph.' First, we draw the figure on a grid and record the coordinates of the points, as in Figure 2.3.

There are many ways to encode this in a matrix. An efficient method is to choose a path that traverses each edge exactly once, if possible[1]. Here is one such matrix, starting from $(1, 2)$ and

[1]This is called an *Eulerian path*. Such a path exists if the graph has exactly 0 or 2 vertices with odd degree.

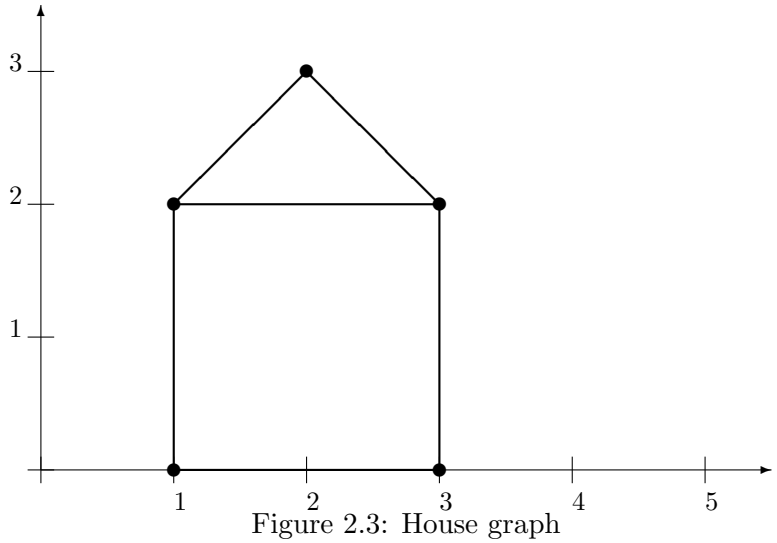


Figure 2.3: House graph

traversing counterclockwise.

$$D = \begin{bmatrix} 1 & 1 & 3 & 3 & 2 & 1 & 3 \\ 2 & 0 & 0 & 2 & 3 & 2 & 2 \end{bmatrix}$$

Try plotting it in Octave and see if it worked.

```
>> D = [1 1 3 3 2 1 3; 2 0 0 2 3 2 2]
D =

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

>> x = D(1, :);
>> y = D(2, :);
>> plot(x, y);
>> grid on
```

You may want to zoom out to see the origin. Then the graph appears correct (Figure 2.4).

2.3.1 Rotation matrices

Now that we have a representation of a digital image, we consider various ways to transform it. Rotations can be obtained using multiplication by a special matrix.

A rotation of the point (x, y) about the origin is given by

$$R \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

where

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

and θ is the angle of rotation (measured counterclockwise).

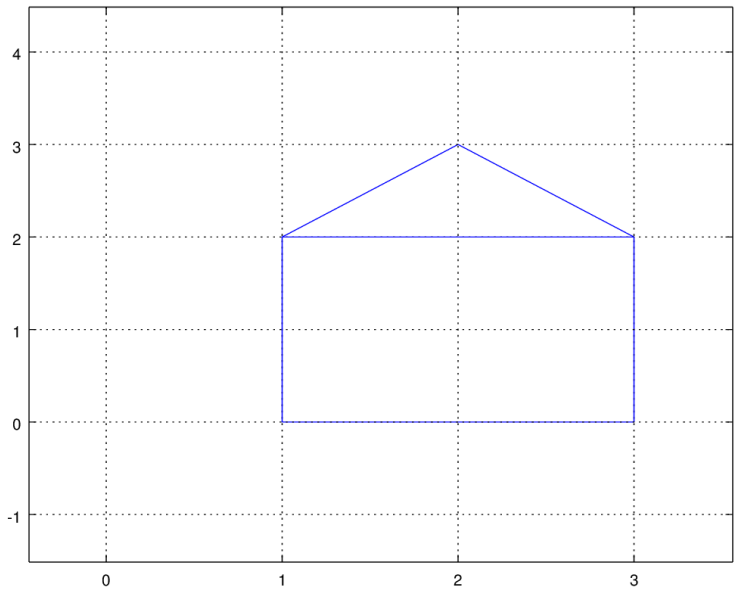


Figure 2.4: House graph

For example, what happens to the point $(1, 0)$ under a $90°$ rotation?

$$\begin{bmatrix} \cos(90°) & -\sin(90°) \\ \sin(90°) & \cos(90°) \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The rotation appears to work, at least in this case. Try a few more points to convince yourself. Notice that a rotation about the origin corresponds to moving along a circle, thus the trigonometry is fairly straightforward to work out.

Now, to produce rotations of a data matrix D, encoded as above, we only need to compute the matrix product RD.

Example 2.3.1. Rotate the house graph through $90°$ and $225°$.

Solution. Note that θ must be converted to radians. Here we go:

```
>> D = [1 1 3 3 2 1 3; 2 0 0 2 3 2 2];
>> x = D(1, :);
>> y = D(2, :);
>>
>> theta1 = 90*pi/180;
>> R1 = [cos(theta1) -sin(theta1); sin(theta1) cos(theta1)];
>> RD1 = R1*D;
>> x1 = RD1(1, :);
>> y1 = RD1(2, :);
>>
>> theta2 = 225*pi/180;
```

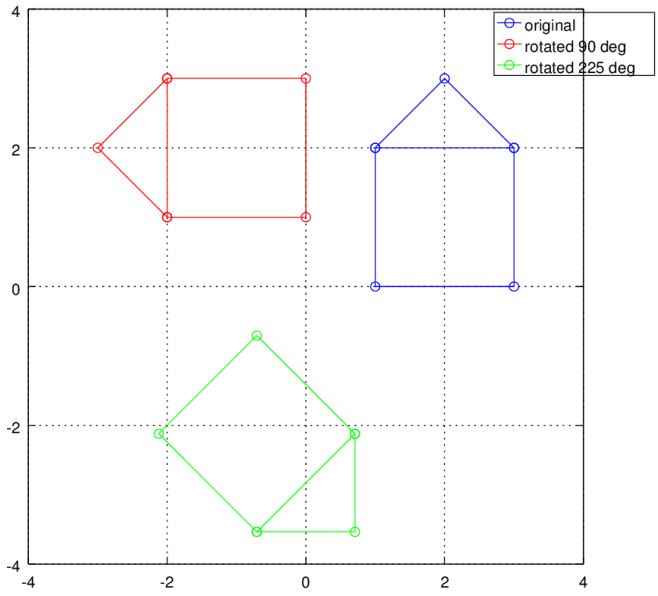


Figure 2.5: Rotations of the 'house graph'

```
>> R2 = [cos(theta2) -sin(theta2); sin(theta2) cos(theta2)];
>> RD2 = R2*D;
>> x2 = RD2(1, :);
>> y2 = RD2(2, :);
>>
>> plot(x, y, 'bo-', x1, y1, 'ro-', x2, y2, 'go-')
>> axis([-4 4 -4 4], 'equal');
>> grid on;
>> legend('original', 'rotated 90 deg', 'rotated 225 deg');
```

Note the combined plot options to set color, marker and line styles. The original and rotated graphs are shown in Figure 2.5. Notice that the rotation is about the origin. For rotations about an arbitrary point, see Exercise 11. □

2.3.2 Other transformations

While some transformations, such as translations, could be accomplished by addition, in practice even these operations are completed using matrix multiplication. The advantage to using multiplication is that the composition of several transformations can be handled with the relatively simple operation of matrix multiplication. Furthermore, inverse transformations are easily produced by inverting the original transformation matrix.

For example, if T is a translation, R is a rotation, and S is a stretch, the combined operations

of first translating, then rotating, then stretching can be completed with the matrix SRT and a data matrix D can be transformed with the product $(SRT)D$. The inverse of these combined operations is $(SRT)^{-1} = T^{-1}R^{-1}S^{-1}$.

Refer to Exercises 10–11 for some of the details and an example.

Chapter 2 Exercises

1. Solve the system of equations using Gaussian elimination row operations

$$\begin{cases} -x_1 + x_2 - 2x_3 &= 1 \\ x_1 + x_2 + 2x_3 &= -1 \\ x_1 + 3x_2 + 2x_3 &= -11 \end{cases}$$

   To document your work in Octave, click "select all," then "copy" under the edit menu, and paste your work into a Word or text document. After you have the row-echelon form, solve the system by hand on paper, using backward substitution.

2. Use the Gaussian elimination multipliers from Exercise 1 to write an LU decomposition for $A = \begin{bmatrix} -1 & 1 & -2 \\ 1 & 1 & 2 \\ 1 & 3 & 2 \end{bmatrix}$. Use this factorization to solve the system from Exercise 1.

3. Let $A = \begin{bmatrix} 1 & -3 & 5 \\ 2 & -4 & 3 \\ 0 & 1 & -1 \end{bmatrix}$ be the coefficient matrix for a system of linear equations $A\mathbf{x} = \mathbf{b}$, where $\mathbf{b} = \langle 1, -1, 3 \rangle$. Solve the system using left division. Then, construct an augmented matrix A_1 and use rref to row-reduce it. Compare the results.

4. Use LU decomposition to solve the system from Exercise 3. Use Octave's [L U P] = lu(A) command. To use $PA = LU$ to solve $A\mathbf{x} = \mathbf{b}$, first multiply through by P, then replace PA with LU:

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ PA\mathbf{x} &= P\mathbf{b} \\ LU\mathbf{x} &= P\mathbf{b} \end{aligned}$$

   First solve $L\mathbf{y} = P\mathbf{b}$, then solve $U\mathbf{x} = \mathbf{y}$.

5. So far we have only looked at consistent systems. How does Octave handle inconsistent systems? Let's turn our previous system into an inconsistent one. Let $A\mathbf{x} = \mathbf{b}$ be the system from Exercise 3. To make this into a inconsistent system, we will make one row of the coefficient matrix into a linear combination of some other rows, without making the corresponding adjustment to the right-side constants. Do the following:

```
>> A(1, :) = 3*A(2, :) - 4*A(3, :)
```

   Now $A\mathbf{x} = \mathbf{b}$ should be an inconsistent system. Try solving it and see what Octave does. Compare the results of left-division with the row-reduced echelon form. How can you see that the system is inconsistent?

6. Octave can easily solve large problems that we would never consider working by-hand. Let's try constructing and solving a larger systems of equations. The command rand(m, n) will generate an $m \times n$ matrix with entries uniformly distributed from the interval $(0, 1)$. If we want integer entries, we can multiply by 10 and use the floor function to chop off the decimal. Use this command to generate an augmented matrix M for a system of 25 equations in 25 unknowns:

```
>> M = floor(10*rand(25, 26));
```

Note the semicolon. This suppresses the output to the screen, since the matrix is now too large to display conveniently. Solve the system of equations using rref and/or left division and save the solution as a column vector **x**.

7. On July 4, 2006, during a launch of the space shuttle Discovery, NASA recorded the following altitude data[2].

Time (s)	Altitude (ft)
0	7
10	938
20	4,160
30	9,872
40	17,635
50	26,969
60	37,746
70	50,548
80	66,033
90	83,966
100	103,911
110	125,512
120	147,411

   (a) Find the quadratic polynomial that best fits this data. Use Octave to set-up and solve the normal equations. After you have the equations set up, you may solve using either the rref command or the left-division operator. Do not use polyfit.

   (b) Plot the best-fitting parabola together with the given data points. Save or print the plot. Your plot should have labeled axes and include a legend.

   (c) Use the height model to determine models for the velocity and acceleration of the shuttle. Estimate the velocity two minutes into the flight.

8. There are many situations where the polynomial models we have considered so far are not appropriate. However, sometimes we can use a simple transformation to linearize the data. For example, if the points (x, y) lie on an exponential curve, then the points $(x, \ln y)$ should lie on a straight line. To see this, assume that $y = Ce^{kx}$ and take the logarithm of both sides of the equation:

$$\begin{aligned} y &= Ce^{kx} \\ \ln y &= \ln Ce^{kx} \\ &= \ln C + \ln e^{kx} \\ &= kx + \ln C \end{aligned}$$

   Make the change of variables $Y = \ln y$ and $A = \ln C$. Then we have a linear function of the form

$$Y = kx + A$$

[2] https://www.nasa.gov/pdf/585034main_ALG_ED_SSA-Altitude.pdf

We can find the line that best fits the (x, Y)-data and then use inverse transformations to obtain the exponential model we need:

$$y = Ce^{kx}$$

where

$$C = e^A$$

Consider the following world population data[3]:

$x =$ year	$y =$ population (in millions)	$Y = \ln y$
1900	1650	7.4085
1910	1750	
1920	1860	
1930	2070	
1940	2300	
1950	2525	
1960	3018	
1970	3682	
1980	4440	
1990	5310	
2000	6127	
2010	6930	

(a) Fill in the blanks in the table with the values for $\ln y$. Note that in Octave, the log(x) command is used for the natural logarithm. Make a scatter plot of x vs. Y. This is called a semi-log plot. Is the trend approximately linear?

(b) Use the polyfit function to find the best-fitting line for the (x, Y)-data and add the graph of the line to your scatter plot from part (a). Save or print the plot. Your plot should have labeled axes and include a legend. Note that the vertical axis is the logarithm of the population. Give the plot the title "Semi-log plot."

(c) Use the data from part (b) to determine the exponential model $y = Ce^{kx}$. Plot the original data and the exponential function on the same set of axes. Save or print the plot. Your plot should have labeled axes and include a legend. Give the plot the title "Exponential plot."

(d) Use the model from part (c) to predict the date when the global population will (or did) reach 7 billion.

9. Create a data matrix that corresponds to a picture of your own design, containing six or more edges. Plot it, then plot two rotations of the same image.

10. Let the point (x, y) be represented by the column vector $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. These are known as *homogeneous coordinates*. Then the translation matrix

$$T = \begin{bmatrix} 1 & 0 & h \\ 0 & 1 & k \\ 0 & 0 & 1 \end{bmatrix}$$

[3] https://esa.un.org/unpd/wpp/

is used to move the point (x, y) to $(x + h, y + k)$ as follows:

$$\begin{bmatrix} 1 & 0 & h \\ 0 & 1 & k \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+h \\ y+k \\ 1 \end{bmatrix}$$

Use a translation matrix and homogeneous coordinates to shift the graph you created in problem 9 as follows: shift 3 units left and 2 units up.

11. The translation method described in problem 10 can be combined with a rotation matrix to give rotations around an arbitrary point. Suppose for example that we wished to rotate the house graph from Figure 2.3 about the center of the rectangular portion (coordinates $(2, 1)$ in the original figure). This can be done by using homogeneous coordinates and a translation T to move the figure, then a rotation matrix R for the rotation. The form of R is now

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The shifted and rotated figure is then given by $(RT)D$. To shift back to the original position, an inverse transformation T^{-1} is used. Thus the rotated image can be found by computing $(T^{-1}RT)D$. Use this method to rotate the house graph 90° about the point $(1, 2)$. Show the combined transformation matrix $T^{-1}RT$ and the results.

Chapter 3

Calculus

3.1 Limits, sequences, and series

Octave is an excellent tool for many types of numerical experiments. Octave is a full-fledged programming language supporting many types of loops and conditional statements. However, since it is a vector-based language, many things that would be done using loops in FORTRAN or other languages can be "vectorized." As an example, let's construct some numerical evidence to guess at the value of the following limit:

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$$

We simply want to evaluate the expression for a series of larger and larger n-values. This is what we mean by vectorized code: instead of writing a loop to evaluate the function multiple times, we will generate a vector of input values, then evaluate the function using the vector input. This produces code that is easier to read and understand, and executes faster, due to Octave's underlying efficient algorithms for matrix operations.

First, we need to define the function. There are a number of ways to do this. The method we use here is known as an *anonymous function*. This is a good way to quickly define a simple function.

```
>> f = @(n) (1 + 1./n).^n;  % anonymous function
```

Note the use of elementwise operations. We have named the function f. The input variable is designated by the @-sign followed by the variable in parentheses. The expression that follows gives the rule to be used when the function is evaluated. Now f can be used like any function in Octave.

Next we create an index variable, consisting of the integers from 0 to 9:

```
>> k = [0:1:9]'  % index variable
k =
```

```
     0
     1
     2
     3
     4
     5
     6
     7
     8
     9
```

The syntax [0:1:9] produces a row vector that starts at 0 and increases by an increment of 1 up to 9 (linspace can also be used). Notice that we have used the transpose operation, simply because our results will be easier to read as column vectors. Now, we'll take increasing powers of 10, which will be the input values, then evaluate $f(n)$.

```
>> format long % display additional decimal places
>> n = 10.^k
n =

             1
            10
           100
          1000
         10000
        100000
       1000000
      10000000
     100000000
    1000000000

>> f(n)
ans =

   2.00000000000000
   2.59374246010000
   2.70481382942153
   2.71692393223552
   2.71814592682436
   2.71826823719753
   2.71828046915643
   2.71828169398037
   2.71828178639580
   2.71828203081451

>> format       % return to standard 5-digit display
```

This is good evidence that the limit converges to a finite value that is approximately 2.71828... You (hopefully!) recognize the number as e.

Similar methods can be used for numerical exploration of sequences and series, as we show in the following examples.

Example 3.1.1. Let $\sum_{n=2}^{\infty} a_n$ be the series whose nth term is $a_n = \dfrac{1}{n(n+2)}$. Find the first ten terms, the first ten partial sums, and plot the sequence and partial sums.

Solution. To do this, we will define an index vector n from 2 to 11, then calculate the terms.

```
>> n = [2:1:11]';        % index
>> a = 1./(n.*(n + 2)) % terms of the sequence
a =

   0.1250000
   0.0666667
   0.0416667
   0.0285714
   0.0208333
   0.0158730
   0.0125000
   0.0101010
   0.0083333
   0.0069930
```

If we want to know the 10th partial sum, we need only type sum(a). If we want to produce the sequence of partial sums, we need to make careful use of a loop. We will use a "for loop" with index i from 1 to 10. For each i, we produce a partial sum of the sequence a_n from the first term to the ith term. The output is a 10-element vector of these partial sums.

```
>> for i = 1:10
     s(i) = sum(a(1:i));
   end
>> s' % sequence of partial sums, displayed as a column
ans =

   0.12500
   0.19167
   0.23333
   0.26190
   0.28274
   0.29861
   0.31111
   0.32121
   0.32955
   0.33654
```

Finally, we will plot the terms and partial sums, for $2 \leq n \leq 11$.

```
>> plot(n, a, 'o', n, s, '+')
>> grid on
>> legend('terms', 'partial sums')
```

The result is shown in Figure 3.1. □

An advantage of using a language like Octave is that it is simple to determine the sum of many

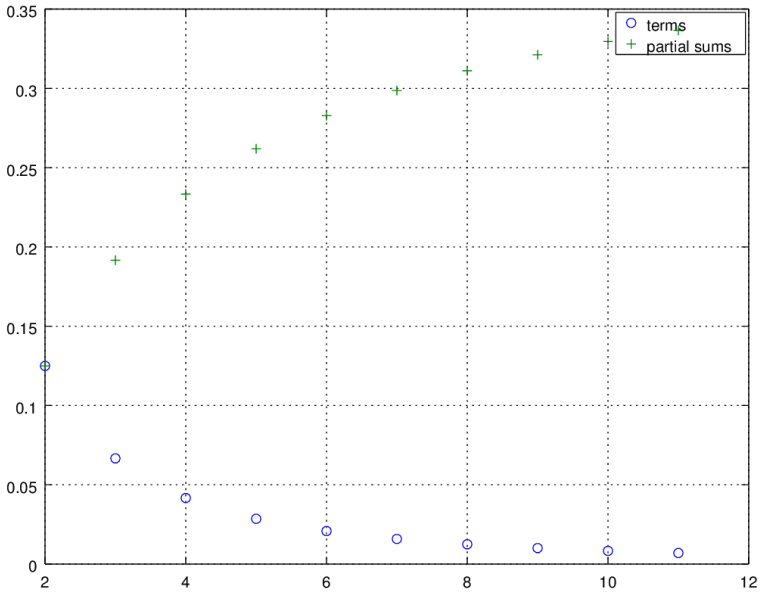


Figure 3.1: Plot of a sequence and its partial sums

terms of a series. If the series is known to converge, this can help give an estimate for the sum.

Example 3.1.2. Find the sum of the first 1000 terms of the harmonic series.

$$\sum_{n=1}^{1000} \frac{1}{n}$$

Solution. We only need to generate the terms as a vector, then take its sum.

```
>> n = [1:1:1000];
>> a = 1./n;
>> sum(a)
ans =  7.4855
```

Of course, Octave cannot tell us if this figure is a good estimate for the sum of the infinite series. In this case it is not, since, by the integral test, we know the series diverges. We can use Octave to easily document how slowly this particular series diverges. The first 1000 terms sum to only about 7.5. If we look at the $1,000,000$th partial sum, it is still only about 14.4. □

3.2 Numerical integration

When it is not possible to find an explicit antiderivative, numerical methods are used to find definite integrals. We will use Octave's built-in numerical integration capability, then try writing

our own scripts to apply the midpoint rule, trapezoid rule, and Simpson's rule.

3.2.1 Quadrature

Octave has several built in functions to calculate definite integrals. We will use the quad command. 'Quad' is short for *quadrature*, which refers to the process of numeric integration.

Example 3.2.1. Estimate $\int_0^{\pi/2} e^{x^2}\cos(x)\,dx$ using Octave's quad algorithm.

Solution. The correct syntax is quad('f', a, b). We need to first define the function.

```
>> function y = f(x)
     y = exp(x.^2).*cos(x);
   end
>> quad('f', 0, pi/2)
ans =  1.8757
```

Note that the function exp(x) is used for e^x. In this example, we used the function ... end construction to define f. This is a versatile format that allows for multiple operations and outputs. We could have also used an anonymous function. Note that no quotes are used around f if using an anonymous function. □

3.2.2 Approximating sums

The midpoint rule, trapezoid rule, and Simpson's rule are common algorithms used for numerical integration. Ideally these are implemented in a computer program and Octave is well suited for this purpose.

Let $\{a = x_0, x_1, x_2, \ldots, x_n = b\}$ be a partition of $[a, b]$ into n subintervals, each of width $\Delta x = \dfrac{b-a}{n}$. Then $\int_a^b f(x)\,dx$ can be approximated as follows.

Midpoint rule:

$$\Delta x\,[f(m_1) + f(m_2) + \cdots + f(m_n)]$$

where m_i is the midpoint of the ith subinterval.

Trapezoid rule:

$$\frac{\Delta x}{2}\,[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + 2f(x_n)]$$

where $x_i = a + i\Delta x$.

Simpson's rule:

$$\frac{\Delta x}{3}\,[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]$$

where $x_i = a + i\Delta x$.

Notice that the sum in the midpoint formula has n terms, while the trapezoid and Simpson's rules have $n+1$ (the index i goes from 0 to n). To implement these rules in Octave, we will write *script* files, which are plain text files containing a series of Octave commands. Use a text editor, such as Notepad, Notepad++, or Emacs. A script file needs to have a ".m" extension (not the .txt used by default in Windows for text files) and cannot begin with the keyword function. The Octave GUI has its own built in text editor which can be accessed by changing to the "Editor" tab option displayed below the main command window. This editor is ideal for creating, editing, and running .m files and will automatically color code comments and key words.

Example 3.2.2. Write an Octave script to calculate a midpoint rule approximation of

$$\int_0^{\pi/2} e^{x^2} \cos(x)\, dx$$

using $n = 100$.

Solution. The basic strategy is to use a loop that adds an additional function value to a running total with each iteration. Then the final answer is found by multiplying the sum by Δx.

The following code can be used. Enter the code in a plain text file and name it "midpoint.m". It must be placed in your working directory, then it can be run by typing 'midpoint' at the command prompt.

Octave Script 3.1: Midpoint rule approximation

```
% file 'midpoint.m'
% calculates a midpoint rule approximation of
%   the integral from 0 to pi/2 of f(x) = exp(x^2)cos(x)
% traditional looped code

% set limits of integration, number of terms and delta x
a = 0
b = pi/2
n = 100
dx = (b - a)/n

% define function to integrate
function y = f(x)
  y = exp(x.^2).*cos(x);
end

msum = 0;       % initialize sum
m1 = a + dx/2; % first midpoint

% loop to create sum of function values
for i = 1:n
  m = m1 + (i - 1)*dx; % calculate midpoint
  msum = msum + f(m);  % add to midpoint sum
end

% midpoint approximation to the integral
approx = msum*dx
```

Now run midpoint.m.

```
>> midpoint
a = 0
b =  1.5708
n =  100
dx =  0.015708
approx =  1.8758
```

□

The traditional code works fine, but because Octave is a vector-based language, it is also possible to write vectorized code that does not require any loops.

Example 3.2.3. Write a vectorized Octave script to calculate a midpoint rule approximation of

$$\int_0^{\pi/2} e^{x^2} \cos(x)\, dx$$

using $n = 100$.

Solution. Now our strategy is to create a vector of the x-coordinates of the midpoints. Then we evaluate f over this midpoint vector to obtain a vector of function values. The midpoint approximation is the sum of the components of the vector, multiplied by Δx.

Octave Script 3.2: Midpoint rule approximation - vectorized

```
% file 'midpoint2.m'
% calculates a midpoint rule approximation of
%   the integral from 0 to pi/2 of f(x) = exp(x^2)cos(x)
% vectorized code

% set limits of integration, number of terms and delta x
a = 0
b = pi/2
n = 100
dx = (b - a)/n

% define function to integrate
function y = f(x)
  y = exp(x.^2).*cos(x);
end

% create vector of midpoints
m = [a + dx/2 : dx : b - dx/2];

% create vector of function values at midpoints
M = f(m);

% midpoint approximation to the integral
approx = dx*sum(M)
```

This code will give the same results as the traditional looped code, but it executes much faster. □

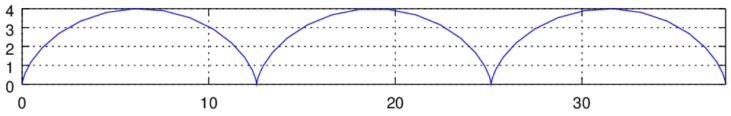


Figure 3.2: Graph of a cycloid

3.3 Parametric and polar plots

Curves defined by parametric and polar equations are usually studied in Calculus II. Such curves can be difficult to graph by hand! The plotting methods we used in Section 1.3 carry over easily to these new settings. For example, parametric equations for a *cycloid* are given by

$$\begin{aligned} x &= r(t - \sin(t)) \\ y &= r(1 - \cos(t)) \end{aligned}$$

Example 3.3.1. Graph three periods of a radius 2 cycloid.

Solution. The functions have period 2π, so we need $0 \leq t \leq 6\pi$ to see three full cycles. We need to define the parameter t as a vector over this range, then we calculate x and y, and plot x vs. y.

```
>> t = linspace(0, 6*pi, 30);
>> r = 2;
>> x = r*(t - sin(t));
>> y = r*(1 - cos(t));
>> plot(x, y)
>> axis('equal')
>> axis([0 12*pi 0 4])
>> grid on
```

The command `axis('equal')` is used to force an equal aspect ratio between the x- and y-axes. The result is shown in Figure 3.2. □

Polar graphs are handled in a similar way. For a function $r = f(\theta)$, we start by defining the independent variable θ, then we calculate r. To plot the graph, we calculate x and y using the standard polar identities $x = r\cos(\theta)$, $y = r\sin(\theta)$, then plot x vs. y.

Example 3.3.2. Plot the *limaçon* $r = 1 - 2\sin(\theta)$.

Solution. The needed commands are shown below and the graph is shown in Figure 3.3.

```
>> theta = linspace(0, 2*pi, 50);
>> r = 1 - 2*sin(theta);
>> x = r.*cos(theta);
>> y = r.*sin(theta);
>> plot(x, y)
```

□

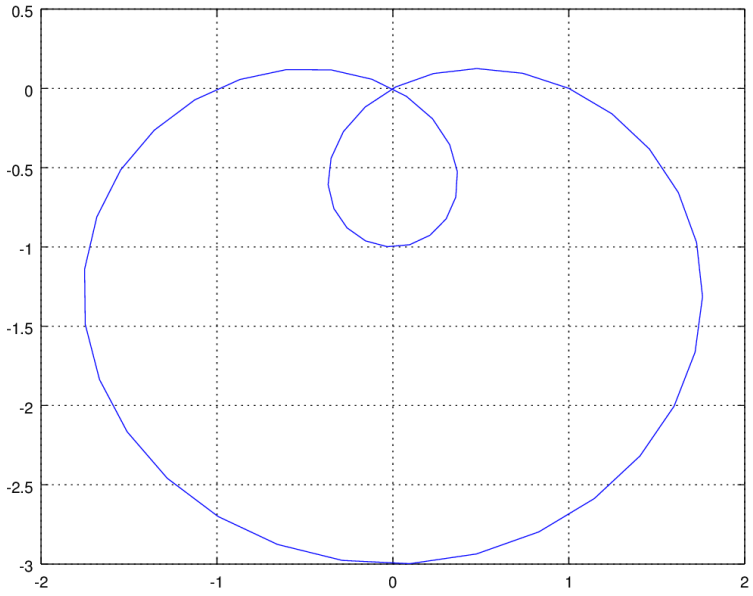


Figure 3.3: Graph of a limaçon

3.4 Special functions

Octave has many common *special functions* available, such as Bessel functions (bessel), the error function (erf), and the gamma function (gamma), to name a few.

For example, the *gamma function* is defined by

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t}\, dt$$

This is an extension of the factorial function, since for positive integers n, the gamma function satisfies

$$\Gamma(n) = (n-1)!$$

Example 3.4.1. Graph $\Gamma(x+1)$ together on the same set of axes with the factorial function $n!$.

Solution. Both the gamma function and factorial function grow quite large very quickly, so we need to take care in selecting the domain. The gamma function is defined for positive and negative real numbers, while the factorial function is of course defined only for nonnegative integers. We will try the graph for $x \in [-5, 5]$ for the gamma function and $n = 0, 1, 2, 3, 4, 5$ for the factorial.

Trial and error shows that a fine increment is needed for a smooth graph of the gamma function.

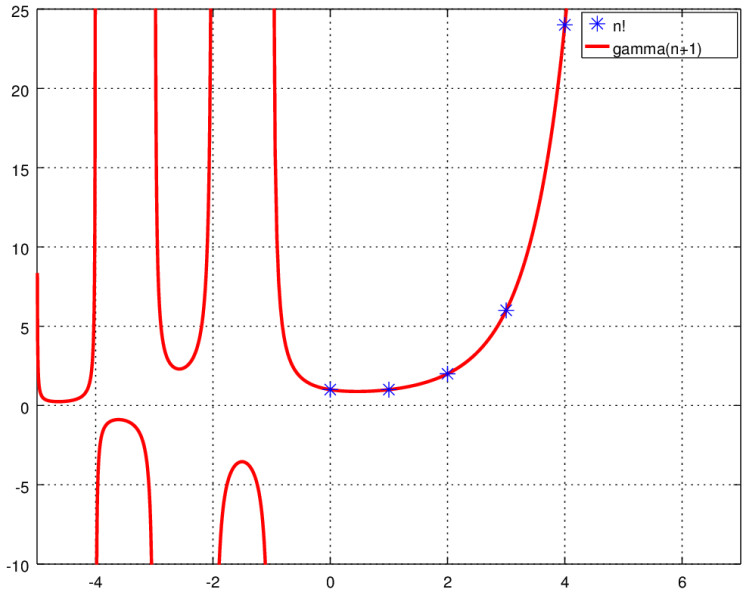


Figure 3.4: Improved graph of gamma function and factorial function

These are the basic commands needed:

```
>> n = [0:5];
>> x = linspace(-5, 5, 500);
>> plot(n, factorial(n), '*', x, gamma(x+1))
>> axis([-5 5 -10 25]);
>> grid on;
>> legend('n!', 'gamma(n+1)')
```

Notice the vertical asymptotes at each negative integer. If you run the plot commands as shown above, you will see vertical line segments that are not a true part of the graph. If we don't want to see these, we can divide the domain into separate intervals with breaks at the discontinuities. This is somewhat tedious, but produces a more accurate graph, as shown in Figure 3.4.

Define the x-values as follows:

```
x1 = linspace(-5, -4, 200);
x2 = linspace(-4, -3, 200);
x3 = linspace(-3, -2, 200);
x4 = linspace(-2, -1, 200);
x5 = linspace(-1, 5, 200);
```

Then, plot x_1 vs. $\Gamma(x_1+1)$, x_2 vs. $\Gamma(x_2+1)$, etc., on the same set of axes. □

Chapter 3 Exercises

1. Show (numerically) that $\lim_{\theta \to 0} \dfrac{\sin \theta}{\theta} = 1$.

2. Let $\sum a_n$ be the series whose nth term is $a_n = \dfrac{1}{2^n} - \dfrac{1}{3^n}, n \geq 1$. Find the first ten terms, the first ten partial sums, and plot the sequence and partial sums. Do you think the series converges? If so, what is the sum?

3. How many terms need to be included in the harmonic series to reach a partial sum that exceeds 10?

4. Write an Octave script based on a for loop to calculate $\displaystyle\int_0^{\pi/2} e^{x^2} \cos(x)\, dx$ using the trapezoid rule with $n = 100$. Compare your answer to the midpoint approximation given above. (Use the command format long to see more decimal places.)

5. Write a vectorized Octave script to calculate $\displaystyle\int_{-2}^{2} \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}\, dx$ using Simpson's rule with $n = 100$. Compare your answer to the midpoint approximation using the script from Example 3.2.3. Which approximation seems to be most accurate, judged against Octave's quad algorithm?

6. Graph each equation.

   (a) $x = t^3, y = t^2$

   (b) $x = \sin(t), y = 1 - \cos(t)$

   (c) $r = \theta$

   (d) $r = \sin(2\theta)$

   (e) $r = \cos(7\theta/3)$

7. Graph the Bessel functions of the first kind $J_0(x)$, $J_1(x)$, and $J_2(x)$ on $[0, 20]$.

8. The gamma function can be used to calculate the "volume" (or "hypervolume") of an n-dimensional sphere. The volume formula is

$$V_n(a) = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2} + 1)} \cdot a^n$$

   where a is the radius, n is the dimension, and $\Gamma(n)$ is the gamma function.

   (a) Write a user-defined Octave function $V_n = f(n, a)$ that gives the volume of an n-dimensional sphere of radius a. Test it by computing the volumes of 2- and 3-dimensional spheres of radius 1. The answers should be π and $4\pi/3$, respectively.

   (b) Use the function to calculate the volume of a 4-dimensional sphere of radius 2 and a 12-dimensional sphere of radius 1/2.

(c) For a fixed radius a, the "volume" is a function of the dimension n. For $n = 1, 2, \ldots, 20$, graph the volume functions for $a = 1$, $a = 1.1$, and $a = 1.2$ on the same axes. Your graph should show only points for integer values of n and should have axis labels and a legend. Use the graph to determine the following limit:

$$\lim_{n \to \infty} V_n$$

Does the answer surprise you?

9. Octave scripts can be used for many problems in numerical analysis. Newton's method for root finding is a good example. Newton's method is an iterative process based on the formula

$$x_{i+1} = x_i + \frac{f(x_i)}{f'(x_i)}$$

Starting from an initial guess of x_1, a sequence of approximations x_i is generated (refer to [1], §2.4 and [6], Vol. 1 §4.9).

(a) The function $f(x) = x^3 + 5x^2 + x - 1$ has one positive root. Write an Octave script to find it using Newton's method.

(b) Compare your answer to the result obtained with Octave's fsolve command.

```
>> fsolve('f', x1)   %solve f(x) = 0 numerically using initial
   guess x1
```

(c) How many iterations of Newton's method were needed to obtain agreement with the fsolve result to five decimal places (using the same initial guess)?

(d) Plot the function and its tangent lines at x_1, x_2, and x_3.

Chapter 4

Eigenvalue problems

4.1 Eigenvalues and eigenvectors

Let $A = \begin{bmatrix} 1 & 2 & -3 \\ 2 & 4 & 0 \\ 1 & 1 & 1 \end{bmatrix}$. We showed in Section 1.2.3 the use of eig(A) to find the eigenvalues of a matrix A. You might be wondering about the eigenvectors for this matrix. To find those, we use the eig command with two output arguments. The correct syntax is [v lambda] = eig(A). The first output will be a matrix whose columns represent the eigenvectors and the second output value will be a diagonal matrix with the eigenvalues on the diagonal.

```
>> [v lambda] = eig(A) % 2-ouput form of eig command
v =

  -0.23995+0.00000i  -0.79195+0.00000i  -0.79195-0.00000i
  -0.91393+0.00000i   0.45225+0.12259i   0.45225-0.12259i
  -0.32733+0.00000i   0.23219+0.31519i   0.23219-0.31519i

lambda =

Diagonal Matrix

   4.52510+0.00000i                   0                   0
                  0   0.73745+0.88437i                   0
                  0                   0   0.73745-0.88437i
```

Notice that the output Λ (lambda) is classified as a diagonal matrix. That means the computer only stores the diagonal entries, which can be an important savings for large matrices.

Perhaps we would like to see a matrix with real eigenvalues. We can make a symmetric matrix (which must have real eigenvalues, as will be explained in Section 4.3.1) by multiplying a matrix and its transpose. For example:

```
>> C = A'*A
C =
```

```
   6   11   -2
  11   21   -5
  -2   -5   10

>> [v lambda] = eig(C)
v =

   0.876137   0.188733  -0.443581
  -0.477715   0.216620  -0.851390
  -0.064597   0.957839   0.279949

lambda =

Diagonal Matrix

    0.14970          0          0
          0    8.47515          0
          0          0   28.37516
```

Here we can see that the diagonal entries of Λ are the eigenvalues and the corresponding columns of V are the associated eigenvectors. We know that each eigenvalue corresponds to an infinite family of eigenvectors. The representative eigenvectors given by Octave are normalized to unit length. Moreover, the collection of eigenvectors given will be linearly independent when possible.

4.2 Markov chains

Consider a sequence of random events, subject to the following conditions.

- A finite number of states are possible.
- At regular intervals an observation is made and the state of the system is recorded.
- For each state, we assign a probability of moving to each of the other states, or staying the same. The essential assumption is that these probabilities depend only on the current state.

Such a system is known as a *Markov chain*. Our problem is to predict the probability of future states of the system.

4.2.1 A random walk

Suppose we walk randomly along a four-block stretch of road in the following manner[1]. At intersections 2, 3, or 4 we either move to the left or to the right at random. Upon reaching the end of the road (intersections 1 or 5) we stop.

[1]The idea for this example comes from [4], which is an excellent open reference for more details about Markov chains and probability.

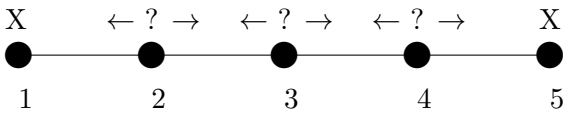


Figure 4.1: Random walk

Our goal is to predict where we will end up. We begin with a *probability vector*. For example, suppose we could start at any point with equal probability. Then the initial vector is $\langle 0.2, 0.2, 0.2, 0.2, 0.2\rangle$. On the other hand, we may know the initial state. Suppose we begin at intersection 3. Then the initial vector is $\langle 0, 0, 1, 0, 0\rangle$. In any case, we want to predict our location after k moves.

This is done by forming a *transition matrix*. Form an $n \times n$ array whose ijth entry is the probability of moving from state i to j. Let T (for transition matrix) be the transpose of this matrix. The matrix product $T\mathbf{x}$ gives the new probability distribution after one time period. Continued left-multiplication by T gives the probabilities for future states. Thus for any initial probability vector $\mathbf{x}$ and any positive integer k, the probability vector after k time periods is $\mathbf{y} = T^k\mathbf{x}$.

Example 4.2.1. For our random walk example, find the probability vector after 5 steps for each of these initial probability vectors:

$\mathbf{a} = \langle 0.2, 0.2, 0.2, 0.2, 0.2\rangle$

$\mathbf{b} = \langle 0.5, 0, 0, 0, 0.5\rangle$

$\mathbf{c} = \langle 0, 1, 0, 0, 0\rangle$

$\mathbf{d} = \langle 0, 0, 1, 0, 0\rangle$

Solution. We form an array that records the probability of moving from each row to column.

		To				
		1	2	3	4	5
From	1	1	0	0	0	0
	2	0.5	0	0.5	0	0
	3	0	0.5	0	0.5	0
	4	0	0	0.5	0	0.5
	5	0	0	0	0	1

The transition matrix is the transpose.

$$T = \begin{bmatrix} 1 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 1 \end{bmatrix}$$

Notice that the sum of each column is 1. Now, the future state probabilities are easily computed as $T^k\mathbf{x}$, where $\mathbf{x}$ is the initial probability vector (as a column vector).

```
>> T = [1 0.5 0 0 0; 0 0 0.5 0 0; 0 0.5 0 0.5 0; 0 0 0.5 0 0; 0 0 0
    0.5 1];
>> a = [0.2; 0.2; 0.2; 0.2; 0.2];
>> b = [0.5; 0; 0; 0; 0.5];
>> c = [0; 1; 0; 0; 0];
>> d = [0; 0; 1; 0; 0];
>> T^5*a
ans =

    0.450000
    0.025000
    0.050000
    0.025000
    0.450000

>> T^5*b
ans =

    0.50000
    0.00000
    0.00000
    0.00000
    0.50000

>> T^5*c
ans =

    0.68750
    0.00000
    0.12500
    0.00000
    0.18750

>> T^5*d
ans =

    0.37500
    0.12500
    0.00000
    0.12500
    0.37500
```

We notice that **b** results in no change from the initial probability. □

A probability vector $\mathbf{x}$ is an *equilibrium vector* if $\mathbf{x} = T\mathbf{x}$ where T is the transition matrix for the Markov chain. An equilibrium vector is one which results in no change moving to future states. Every Markov chain has at least one equilibrium vector and the eigenvalues of the transition matrix are the key to finding it.

Theorem 4.2.2. *Let T be the transition matrix for a Markov chain. Then $\lambda = 1$ is an eigenvalue of T. If $\mathbf{x}$ is an eigenvector for $\lambda = 1$ with nonnegative components that sum to 1, then $\mathbf{x}$ is an equilibrium vector for T.*

Example 4.2.3. Find an equilibrium vector for the Markov chain with transition matrix

$$T = \begin{bmatrix} 0.48 & 0.51 & 0.14 \\ 0.29 & 0.04 & 0.52 \\ 0.23 & 0.45 & 0.34 \end{bmatrix}$$

Solution.

```
>> T = [0.48 0.51 0.14; 0.29 0.04 0.52; 0.23 0.45 0.34]
T =

    0.480000    0.510000    0.140000
    0.290000    0.040000    0.520000
    0.230000    0.450000    0.340000

>> [v lambda] = eig(T)
v =

   -0.64840   -0.80111    0.43249
   -0.50463    0.26394   -0.81601
   -0.57002    0.53717    0.38351

lambda =

Diagonal Matrix

    1.00000          0          0
          0    0.21810          0
          0          0   -0.35810

>> x = v(:, 1)/sum(v(:, 1))
x =

    0.37631
    0.29287
    0.33082
```

Thus $\mathbf{x} = \langle 0.37631, 0.29287, 0.33082 \rangle$ is an equilibrium vector. Let's test it.

```
>> T^10*x
ans =

    0.37631
    0.29287
    0.33082

>> T^50*x
ans =

    0.37631
    0.29287
    0.33082
```

There is no change evident, so it seems to work! □

4.3 Diagonalization

Diagonal matrices have important properties. Some matrices can be transformed into a special diagonal matrix that shares some properties with the original matrix, in particular, its eigenvalues. The diagonalization problem is to find a matrix S such that

$$S^{-1}AS = \Lambda$$

where Λ is a diagonal matrix.

Theorem 4.3.1. *Let A be an $n \times n$ matrix with n linearly independent eigenvectors. Form an $n \times n$ matrix S whose columns are the eigenvectors of A. Then S is invertible and $S^{-1}AS = \Lambda$, where*

$$\Lambda = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{bmatrix}$$

and λ_i is the eigenvalue associated with the ith column of S. It follows that A can be written as $A = S\Lambda S^{-1}$.

Theorem 4.3.1 tells us how to diagonalize a square matrix. Notice that this can be done only for matrices that have enough independent eigenvectors. We need one more result.

Theorem 4.3.2. *If A is an $n \times n$ diagonalizable matrix and $A = S\Lambda S^{-1}$ and k is a positive integer, then*

$$A^k = S\Lambda^k S^{-1} = S \begin{bmatrix} \lambda_1^k & & & \\ & \lambda_2^k & & \\ & & \ddots & \\ & & & \lambda_n^k \end{bmatrix} S^{-1}$$

Theorem 4.3.2 shows how the diagonalized form can be used to simplify certain computational problems, such as raising a matrix to a high power.

Example 4.3.3. Let $A = \begin{bmatrix} 7 & 8 \\ -4 & -5 \end{bmatrix}$. Find A^{100}.

Solution. Octave can solve such a problem easily.

```
>> A = [7  8; -4 -5]
A =

    7    8
   -4   -5

>> A^100
ans =

    1.0308e+048    1.0308e+048
   -5.1538e+047   -5.1538e+047
```

But how does Octave do this? Not by brute force, but by using Theorem 4.3.2. Here's how. First we need to calculate the eigenvalues and associated eigenvectors. Verify that the eigenvalues and eigenvectors are

$$\lambda_1 = 3, \mathbf{v}_1 = \begin{bmatrix} -2 \\ 1 \end{bmatrix}$$

$$\lambda_2 = -1, \mathbf{v}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

Then $\Lambda = \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix}$. We form the matrix S using the eigenvectors:

$$S = \begin{bmatrix} -2 & -1 \\ 1 & 1 \end{bmatrix}$$

Now we need to calculate the inverse matrix. It is

$$S^{-1} = \begin{bmatrix} -1 & -1 \\ 1 & 2 \end{bmatrix}$$

Therefore the diagonalized form is

$$\begin{aligned} A &= S\Lambda S^{-1} \\ &= \begin{bmatrix} -2 & -1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} -1 & -1 \\ 1 & 2 \end{bmatrix} \end{aligned}$$

So,

$$\begin{aligned} A^{100} &= S\Lambda^{100} S^{-1} \\ &= \begin{bmatrix} -2 & -1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix}^{100} \cdot \begin{bmatrix} -1 & -1 \\ 1 & 2 \end{bmatrix} \\ &= \begin{bmatrix} -2 & -1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3^{100} & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & -1 \\ 1 & 2 \end{bmatrix} \\ &= \begin{bmatrix} 2 \cdot 3^{100} - 1 & 2 \cdot 3^{100} - 22 \\ -3^{100} + 1 & -3^{100} + 2 \end{bmatrix} \end{aligned}$$

Compare to the earlier Octave result:

```
>> [2*3^100-1  2*3^100-2; -3^100+1  -3^100+2]
ans =

   1.0308e+048   1.0308e+048
  -5.1538e+047  -5.1538e+047
```

This example shows the power of diagonalization. □

4.3.1 Orthogonal diagonalization

We have already observed that not all square matrices can be diagonalized. However, a certain class of square matrices always has a diagonalization, and this diagonalization has special properties. First, we need to recall a few definitions.

- A *symmetric matrix* is a square matrix A such that $A^T = A$. Recall that a matrix with real entries may have complex eigenvalues. That cannot happen with symmetric matrices. A real symmetric matrix has all real eigenvalues.
- An *orthogonal matrix* is a square matrix whose columns are *orthonormal* (orthogonal and length 1). An important property of orthogonal matrices is that their inverse is equal to their transpose: If A is orthogonal, then $A^{-1} = A^T$.

All symmetric matrices are diagonalizable. Moreover, we can say the following:

Theorem 4.3.4. *Let A be a symmetric matrix. Then A can be diagonalized as*

$$A = Q\Lambda Q^T$$

where Q is an orthogonal matrix whose columns are eigenvectors of A and Λ is a diagonal matrix with the associated eigenvalues on the diagonal.

Example 4.3.5. Find an orthogonal diagonalization for $A = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}$.

Solution. A has eigenvalues 3 and 1. The eigenvectors are $\begin{bmatrix} 1 \\ -1 \end{bmatrix}$ and $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$. Notice that these are orthogonal. They are normalized by dividing by their length (both have length $\sqrt{2}$). Then A can be diagonalized as

$$\begin{aligned} A &= Q\Lambda Q^T \\ &= \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \cdot \begin{bmatrix} 3 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \end{aligned}$$

The eigenvectors in this example were orthogonal since the eigenvalues were distinct. If the matrix A is symmetric, but has repeated eigenvalues, then the problem is a bit more difficult and finding a set of orthonormal eigenvectors requires the Gram-Schmidt process (see Section 4.5.1). We won't show the details here, but note that even in those cases, an orthonormal set of eigenvectors can still be found. □

Now, with these ideas in mind, let's take another look at the output of Octave's eig command.

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

    2   -1
```

```
    -1     2

>> [v lambda] = eig(A)
v =

   -0.70711   -0.70711
   -0.70711    0.70711

lambda =

Diagonal Matrix

    1    0
    0    3
```

While the matrices are arranged slightly differently (the diagonalization is not unique), you should see that results given by Octave are precisely what is needed for the orthogonal diagonalization problem.

Example 4.3.6. Use Octave to diagonalize $A = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}$.

Solution. If an orthogonal diagonalization is possible, Octave will return the output of eig(A) in that format. This explains why Octave chooses normalized vectors that form an orthogonal set, when possible.

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

    2   -1
   -1    2

>> [Q L] = eig(A)
Q =

   -0.70711   -0.70711
   -0.70711    0.70711

L =

Diagonal Matrix

    1    0
    0    3

>> Q*L*Q'    % check the factorization by multiplying
ans =

    2.00000   -1.00000
   -1.00000    2.00000
```

□

4.4 Singular value decomposition

We are now prepared to tackle the singular value decomposition (SVD). This factorization is something of a generalized version of what we just did for symmetric matrices in Section 4.3.1. But, the singular value decomposition exists for any matrix; the matrix need not even be square! The key is to consider the matrices A^TA and AA^T. These are always square symmetric matrices, and so, can be orthogonally diagonalized.

There are many applications associated with the SVD. Netflix recently sponsored a competition with a one million dollar prize to improve their movie recommendation algorithm. The team that won used a method based in part on the SVD[2], which can be used to discover hidden relationships among variables. We will consider applications to least-squares problems and image compression.

Theorem 4.4.1. *Let A be an $m \times n$ matrix. The square roots of the nonzero eigenvalues of A^TA and AA^T (they are the same) are called the* singular values *of A,denoted $\sigma_1, \sigma_2, \ldots, \sigma_r$. Then A can be factored as*

$$A = U\Sigma V^T$$

where the columns of U are eigenvectors of AA^T, the columns of V are eigenvectors of A^TA, and the r singular values of A are on the diagonal of Σ. This factorization is called the singular value decomposition *of A.*

- U is $m \times m$ and orthogonal
- V is $n \times n$ and orthogonal
- Σ is $m \times n$ and diagonal of the special form

$$\Sigma = \begin{bmatrix} \sigma_1 & & & & \vdots \\ & \sigma_2 & & & 0 \\ & & \ddots & & \vdots \\ & & & \sigma_r & \\ & \cdots & 0 & \cdots & 0 \end{bmatrix}$$

If all the eigenvalues of A^TA are distinct, then the associated eigenvectors are "automatically" orthogonal. We only need to make them unit vectors. If there are repeated eigenvalues, it is still possible to choose orthogonal eigenvectors, but more advanced methods are needed (see Section 4.5.1). Our procedure starts with eigenvectors of A^TA, then appropriate orthogonal unit eigenvectors for AA^T are found using a simple formula, rather than by direct computation. The number of singular values (nonzero eigenvalues) corresponds to the rank of the original matrix A. We will only consider examples where the number of singular values r is equal to m, the number of rows of A, otherwise, again, more advanced methods are required. We will consider

[2] http://buzzard.ups.edu/courses/2014spring/420projects/math420-UPS-spring-2014-gower-netflix-SVD.pdf

a simple example using a 2×2 matrix, then see how Octave commands can be used to find the SVD for larger or more difficult matrices.

Here is the simplified procedure we will use:

1. Find $A^T A$.
2. Find the eigenvalues of $A^T A$. The square roots of these are the singular values σ_1, σ_2, ..., σ_r, arranged in decreasing order.
3. Find the corresponding eigenvectors and make them unit vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$.
4. Find the vectors $\mathbf{u}_i$ by computing $\mathbf{u}_i = \dfrac{1}{\sigma_i} A\mathbf{v}_i$.
5. Then $A = U\Sigma V^T$, where $\sigma_1, \sigma_2, \ldots, \sigma_r$ are on the diagonal of Σ and

$$U = \left[\begin{array}{c|c|c|c} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_m \end{array} \right]$$

$$V = \left[\begin{array}{c|c|c|c} \mathbf{v}_1 & \mathbf{v}_2 & \cdots & \mathbf{v}_n \end{array} \right]$$

- Remember to transpose V when you write the factorization.
- Remember to keep the eigenvalues and eigenvectors in their correct order.
- This simplified procedure only works if $A^T A$ has no repeated eigenvalues and $r = m$.

Example 4.4.2. Let $A = \begin{bmatrix} 4 & 4 \\ -3 & 3 \end{bmatrix}$. Find the SVD via the simplified procedure outlined above, then compare to the results obtained using the Octave function svd.

Solution. We can readily verify that rank(A) = 2, so the matrix should have two singular values.

```
>> A = [4  4; -3  3]
A =

   4   4
  -3   3

>> ATA = A'*A
ATA =

   25    7
    7   25

>> [v lambda] = eig(ATA)
v =

  -0.70711   0.70711
   0.70711   0.70711

lambda =
```

```
Diagonal Matrix

    18     0
     0    32
```

Notice that the given eigenvectors are orthogonal unit vectors. However, the eigenvalues are not in decreasing order. So, we need to switch the order of both eigenvectors and singular values (they must be in decreasing order) as we build V and Σ.

```
>> Sigma = zeros(2, 2);
>> Sigma(1, 1) = sqrt(lambda(2, 2))
Sigma =

    5.65685    0.00000
    0.00000    0.00000

>> Sigma(2, 2) = sqrt(lambda(1, 1))
Sigma =

    5.65685    0.00000
    0.00000    4.24264

>> V(:, 1) = v(:, 2)
V =

    0.70711
    0.70711

>> V(:, 2) = v(:, 1)
V =

    0.70711   -0.70711
    0.70711    0.70711
```

Now we build U to complete the factorization.

```
>> U(:, 1) = 1/Sigma(1, 1)*A*V(:, 1)
U =

    1.00000
    0.00000

>> U(:, 2) = 1/Sigma(2, 2)*A*V(:, 2)
U =

    1.00000    0.00000
    0.00000    1.00000
```

Now, let's verify that $U\Sigma V^T = A$.

```
>> U*Sigma*V'
ans =
```

```
    4.0000    4.0000
   -3.0000    3.0000
```

Now that we have a rough sense of how an SVD is determined, let's try the built-in Octave function. The command [U, S, V] = svd(A) computes the SVD of a matrix A and stores the result in matrices U, S, and V. Let's use this command to find the SVD of the matrix A and verify that U*S*V' returns A.

```
>> [U S V] = svd(A) % 3-output format of svd command
U =

   -1    0
    0    1

S =

Diagonal Matrix

    5.6569         0
         0    4.2426

V =

   -0.70711   -0.70711
   -0.70711    0.70711

>> U*S*V'
ans =

    4.0000    4.0000
   -3.0000    3.0000
```

Notice that the factorization returned by svd is slightly different than we obtained above. This is normal: the SVD is not unique due to variations in how representative eigenvectors are chosen. □

4.4.1 Least squares

In Section 2.2, we used the normal equations, $A^TA\mathbf{x} = A^T\mathbf{b}$, to solve least-squares problems. One potential problem with this approach is that the normal equations are typically *ill-conditioned*. This means that a small change in the data can lead to a large change in the numeric solution. This is bad! One way to avoid this computational problem is to use a generalized inverse known as the *pseudoinverse*, based on the SVD.

Theorem 4.4.3. *For an $m \times n$ matrix A with singular value decomposition $A = U\Sigma V^T$, the least-squares solution to the system $A\mathbf{x} = \mathbf{b}$ is given by*

$$\overline{\mathbf{x}} = A^+\mathbf{b}$$

where

$$A^+ = V\Sigma^+U^T$$

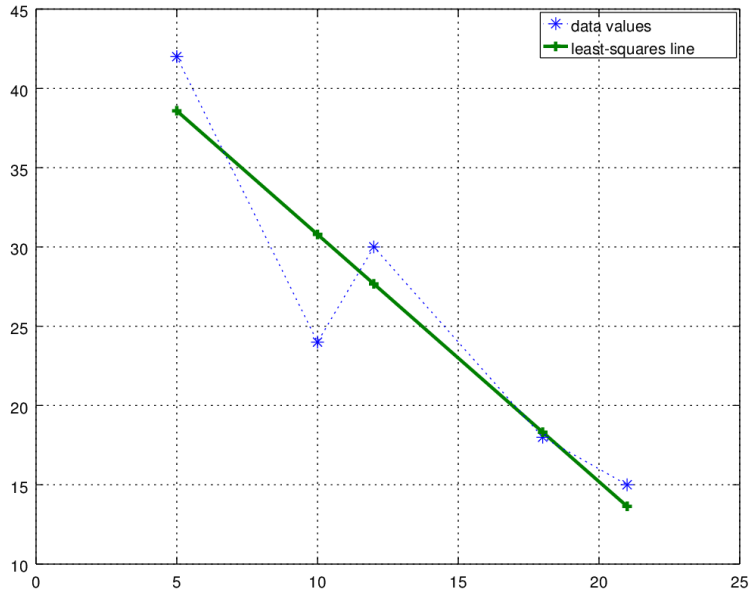


Figure 4.2: Regression line and original data

and Σ^+ *is the* $n \times m$ *matrix found by transposing* Σ *and taking the reciprocals of the singular values:*

$$\Sigma^+ = \begin{bmatrix} 1/\sigma_1 & & & & \vdots \\ & 1/\sigma_2 & & & 0 \\ & & \ddots & & \vdots \\ & & & 1/\sigma_r & \\ & \cdots & 0 & \cdots & 0 \end{bmatrix}$$

The matrix A^+ *is called the* pseudoinverse *or* Moore-Penrose inverse *of* A*.*

Example 4.4.4. Consider the following sample data.

x	5	10	12	18	21
y	42	24	30	18	15

Find a linear equation of the form $y = a + bx$ to model this data.

Solution. The given points give us a system $A\mathbf{x} = \mathbf{b}$, with

$$A = \begin{bmatrix} 1 & 5 \\ 1 & 10 \\ 1 & 12 \\ 1 & 18 \\ 1 & 21 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} a \\ b \end{bmatrix}, \text{ and } \mathbf{b} = \begin{bmatrix} 42 \\ 24 \\ 30 \\ 18 \\ 15 \end{bmatrix}$$

First we need to find the SVD of A. From Octave,

```
>> [U S V] = svd(A)
U =

    0.156839    0.767088   -0.427793   -0.340442   -0.296766
    0.311700    0.407114   -0.027649    0.469589    0.718208
    0.373645    0.263125    0.857417   -0.162313   -0.172178
    0.559478   -0.168843   -0.189306    0.580804   -0.534141
    0.652394   -0.384827   -0.212668   -0.547638    0.284878

S =

Diagonal Matrix

   32.22136         0
          0   0.88546
          0         0
          0         0
          0         0

V =

    0.063748    0.997966
    0.997966   -0.063748
```

Next, we construct Σ^+ by taking the transpose of Σ (called S in our Octave code) and the reciprocal of the nonzero entries:

$$\begin{aligned}\Sigma^+ &= \begin{bmatrix} 1/32.22136 & 0 & 0 & 0 & 0 \\ 0 & 1/0.88546 & 0 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 0.031035 & 0.000000 & 0.000000 & 0.000000 & 0.000000 \\ 0.000000 & 0.885459 & 0.000000 & 0.000000 & 0.000000 \end{bmatrix}\end{aligned}$$

After saving Σ^+ as SP, the pseudoinverse, A^+, can be calculated as:

```
>> Aplus = V*SP*U'
Aplus =

    0.864865    0.459460    0.297297   -0.189189   -0.432432
   -0.050369   -0.019656   -0.007371    0.029484    0.047912
```

Thus the pseudoinverse is:

$$A^+ \approx \begin{bmatrix} 0.864865 & 0.459460 & 0.297297 & -0.189189 & -0.432432 \\ -0.050369 & -0.019656 & -0.007371 & 0.029484 & 0.047912 \end{bmatrix}$$

Finally, we are prepared to solve the original system of equations. The least-squares solution is simply $A^+\mathbf{b}$, easily computed in Octave. The (approximate) solution vector is $\begin{bmatrix} 46.3784 \\ -1.5590 \end{bmatrix}$. So, the correct linear equation is $y = 46.3784 - 1.5590x$. The original data and best-fitting line are shown in Figure 4.2. □

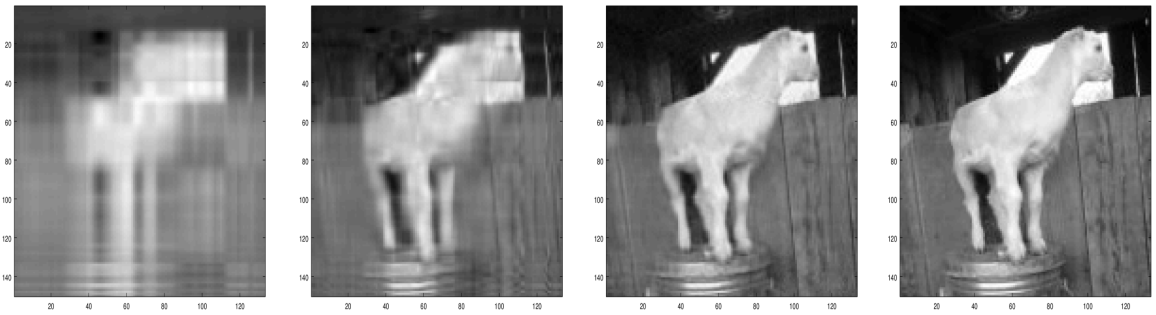


Figure 4.3: SVD approximations

Notice that this method is not a good choice when doing the work by-hand! But, with a computer to do the computations, there are advantages to this approach over the normal equations we used in Section 2.2 (the computations are more numerically stable).

4.4.2 Image compression

How do we reduce large images to manageable file sizes? One approach uses the SVD. A digital image can be represented as a matrix, where each entry represents a pixel and we assign a numeric value to each color. The singular values of the matrix are the key. Typically some of these are large, but many are very small. By keeping only the significant singular values and throwing out the rest, we can significantly reduce the amount data that we need to store.

To illustrate the idea, we have imported a small grayscale image file. It is 133×150, which means the matrix has $19,950$ entries. The SVD is then used to generate several approximations at significantly reduced file sizes. The original and reduced images are shown in Figure 4.3 (the original, exact image is at the far right).

The first approximation uses only 3 singular values. That means we keep three σs, plus three columns of U and three columns of V. We can simply set the other values to 0 (then we don't need to store that data) and multiply the matrices back together to obtain the compressed image. The result is we have only have to store $3 + (3 \times 133) + (3 \times 150) = 852$ data values, compared to the original of total $19,950$! That is only 4% of the original size. Keeping a few more singular values, the second approximation uses 10 singular values and is 14% of the original size. The image quality is not bad, considering how much of the original data we threw away. The third approximation, with 30 singular values, looks almost as good as the original. But, it is only 43% of the original size. For comparison, the exact image is shown on the far right.

Octave supports several image file types. We will use .jpg files, which will be loaded as RGB (red, green, blue) images, represented as a set of three $m \times n$ matrices containing the color values for each pixel. For simplicity, we will convert this to a single $m \times n$ grayscale matrix. Some of the Octave commands needed for basic image processing are listed in the table below.

IMAGE PROCESSING COMMANDS

Syntax	Description
pkg load image	load the image package
im = imread('filename.jpg');	load an image
name = imresize(im, 0.5); .	reduce image size by a specified factor (e.g., 0.5)
name = rgb2gray(im);	convert to grayscale
imshow(im)	display an image
imagesc(im)	display a matrix as a scaled image

In Exercise 6 you will use the SVD method to reduce the size of a digital image.

4.5 Gram-Schmidt and the QR algorithm

4.5.1 The Gram-Schmidt process

Let $\mathbf{u}$ and $\mathbf{v}$ be two linearly independent vectors. Then the vector $\mathbf{u} - \text{proj}_{\mathbf{v}}(\mathbf{u})$ will be orthogonal to $\mathbf{v}$.

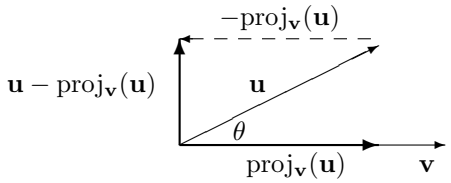


Figure 4.4: Orthogonal projection

Notice that the set $\{\mathbf{v}, \mathbf{u} - \text{proj}_{\mathbf{v}}(\mathbf{u})\}$ is now an orthogonal set which has the same span as the original set $\{\mathbf{v}, \mathbf{u}\}$. This use of orthogonal projections to make a linearly independent set into an orthogonal set is the basis of the famous *Gram-Schmidt process*.

Theorem 4.5.1. THE GRAM-SCHMIDT PROCESS
Let $\{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n\}$ be a linearly independent set. Then the following procedure will produce an orthogonal set $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ with the same span.

$$\begin{aligned}
\mathbf{v}_1 &= \mathbf{u}_1 \\
\mathbf{v}_2 &= \mathbf{u}_2 - \text{proj}_{\mathbf{v}_1}(\mathbf{u}_2) \\
\mathbf{v}_3 &= \mathbf{u}_3 - \text{proj}_{\mathbf{v}_1}(\mathbf{u}_3) - \text{proj}_{\mathbf{v}_2}(\mathbf{u}_3) \\
&\vdots \\
\mathbf{v}_n &= \mathbf{u}_n - \text{proj}_{\mathbf{v}_1}(\mathbf{u}_n) - \text{proj}_{\mathbf{v}_2}(\mathbf{u}_n) - \cdots - \text{proj}_{\mathbf{v}_{n-1}}(\mathbf{u}_n)
\end{aligned}$$

To normalize, set

$$\mathbf{w}_i = \frac{\mathbf{v}_i}{\|\mathbf{v}_i\|}$$

Then the set $\{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_n\}$ is an orthonormal set with the same span as $\{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n\}$ and $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$.

Example 4.5.2. Find an orthonormal set with the same span as

$$\{\langle 10, 9, -3, 0\rangle, \langle -7, 7, -3, 4\rangle, \langle 9, 1, -8, -1\rangle\}$$

Solution. Since we are going to make extensive use of vector projections, it would be a good idea to write a function that handles that part of the computation. This can be entered at the command line, or better yet, it can be saved in a function file 'proj.m' and reused in future problems.

```
>> function vect = proj(u, v)
     vect = dot(u, v)/dot(v, v)*v;
   end
```

As defined, proj(u, v) now computes the projection of **u** onto **v**.

Now, we will enter the original set of vectors as columns in a matrix U.

```
>> U = [10 -7 9; 9 7 1; -3 -3 -8; 0 4 -1]
U =

    10    -7     9
     9     7     1
    -3    -3    -8
     0     4    -1
```

Next, we go through the steps of the Gram-Schmidt process to create a matrix V whose columns are an orthogonal set with the same span as the original set.

```
>> V = zeros(4, 3);
>> V(:, 1) = U(:, 1);
>> V(:, 2) = U(:, 2) - proj(U(:, 2), V(:, 1));
>> V(:, 3) = U(:, 3) - proj(U(:, 3), V(:, 1)) - proj(U(:, 3), V(:, 2))
V =

   10.00000   -7.10526    0.37157
    9.00000    6.90526   -2.73222
   -3.00000   -2.96842   -6.95810
    0.00000    4.00000    0.21304
```

These vectors are orthogonal, but not yet unit vectors, so we normalize. The final output matrix W should have columns that are orthogonal unit vectors with the same span as the original set.

```
>> W = zeros(4, 3);
>> W(:, 1) = V(:, 1)/norm(V(:, 1));
>> W(:, 2) = V(:, 2)/norm(V(:, 2));
>> W(:, 3) = V(:, 3)/norm(V(:, 3))
W =

    0.72548   -0.64071    0.04962
    0.65293    0.62268   -0.36490
   -0.21764   -0.26768   -0.92929
    0.00000    0.36070    0.02845
```

The columns of W are the desired orthonormal set. □

We might want to verify that the process worked. As a spot check, we can look at the dot product of any two columns and we should get 0. Also, each column should have norm 1.

```
>> dot(W(:, 1), W(:, 3))
ans =    2.2204e-016
>> norm(W(:, 3))
ans =  1
```

Notice that the dot product is not quite zero due to rounding error, but the results are more than adequate for our purposes. We should note that the Gram-Schmidt algorithm in its classic form is known to be numerically unstable. Consult a numerical linear algebra book for details about some simple modifications that can reduce the loss of orthogonality due to round-off error[3].

4.5.2 QR decomposition

We have already seen several important matrix factorizations. The Gram-Schmidt process is the key to another, one that turns out to provide a good means for finding eigenvalues numerically. This is known as the *QR decomposition*.

Theorem 4.5.3. *Let A be a nonsingular square matrix. Then there exists an orthogonal matrix Q and an upper triangular matrix R such that $A = QR$.*

Here's how to find Q and R.

1. Apply the Gram-Schmidt process to the columns of A. Use the resulting orthonormal vectors as columns of Q.
2. Let $R = \begin{bmatrix} \mathbf{q}_1 \cdot \mathbf{a}_1 & \mathbf{q}_1 \cdot \mathbf{a}_2 & \mathbf{q}_1 \cdot \mathbf{a}_3 & \cdots & \mathbf{q}_1 \cdot \mathbf{a}_n \\ 0 & \mathbf{q}_2 \cdot \mathbf{a}_2 & \mathbf{q}_2 \cdot \mathbf{a}_3 & \cdots & \mathbf{q}_2 \cdot \mathbf{a}_n \\ 0 & 0 & \mathbf{q}_3 \cdot \mathbf{a}_3 & \cdots & \mathbf{q}_3 \cdot \mathbf{a}_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \mathbf{q}_n \cdot \mathbf{a}_n \end{bmatrix}$, where $\mathbf{q}_i$ is the ith column of Q and $\mathbf{a}_j$ is the jth column of A.

Example 4.5.4. Find the QR decomposition of the matrix $A = \begin{bmatrix} 5 & 7 & 0 \\ 10 & 8 & 0 \\ 5 & 6 & -5 \end{bmatrix}$.

Solution. First we apply the Gram-Schmidt process to A.

```
>> A = [5 7 0; 10 8 0; 5 6 -5]
A =

     5     7     0
    10     8     0
     5     6    -5
```

[3] Or try an internet search for 'numerical instability Gram-Schmidt' and you will find many explanations and modified code examples.

```
>> Q = zeros(3, 3);
>> Q(:, 1) = A(:, 1)/norm(A(:, 1));
>> Q(:, 2) = A(:, 2) - proj(A(:, 2),Q(:, 1));
>> Q(:, 2) = Q(:, 2)/norm(Q(:, 2));
>> Q(:, 3) = A(:, 3) - proj(A(:, 3), Q(:, 1)) - proj(A(:, 3), Q(:, 2))
    ;
>> Q(:, 3) = Q(:, 3)/norm(Q(:, 3))
Q =

    0.40825   0.72900   0.54944
    0.81650  -0.56077   0.13736
    0.40825   0.39254  -0.82416
```

Notice that we normalized each vector as we went through the process to find Q. Now, let's verify that Q is orthogonal. For an orthogonal matrix, $Q^{-1} = Q^T$, so a good way to check for orthogonality is to compute Q^TQ, which should be an identity matrix.

```
>> Q'*Q
ans =

    1.00000  -0.00000   0.00000
   -0.00000   1.00000  -0.00000
    0.00000  -0.00000   1.00000
```

This looks correct (some round-off error can be seen if we check more digits than displayed here). Now, we build R using the appropriate dot products of columns of Q and A.

```
>> R = zeros(3, 3);
>> R(1, 1) = dot(Q(:, 1), A(:, 1));
>> R(1, 2) = dot(Q(:, 1), A(:, 2));
>> R(1, 3) = dot(Q(:, 1), A(:, 3));
>> R(2, 2) = dot(Q(:, 2), A(:, 2));
>> R(2, 3) = dot(Q(:, 2), A(:, 3));
>> R(3, 3) = dot(Q(:, 3), A(:, 3))
R =

   12.24745  11.83920  -2.04124
    0.00000   2.97209  -1.96270
    0.00000   0.00000   4.12082
```

Of course, for a larger problem, we would use loops to compute the entries in R. Finally we check to see that $QR = A$.

```
>> Q*R
ans =

    5.00000   7.00000   0.00000
   10.00000   8.00000   0.00000
    5.00000   6.00000  -5.00000
```

It works as expected. □

4.5.3 The QR algorithm

The QR decomposition is the basis of a numerical method for finding eigenvalues.

Theorem 4.5.5. THE QR ALGORITHM

Let A be an $n \times n$ matrix with n real eigenvalues.

Let $A_1 = A$.

For each $k = 1, 2, 3, \ldots$ do the following:

(i) Find the QR decomposition of A_k, $A_k = Q_k R_k$.

(ii) Set $A_{k+1} = R_k Q_k$.

Repeat steps (i) and (ii).

As k increases, the matrices A_k approach an upper triangular form with the eigenvalues of A on the diagonal.

Example 4.5.6. Apply three iterations of the QR algorithm to the matrix $A = \begin{bmatrix} 5 & 7 & 0 \\ 10 & 8 & 0 \\ 5 & 6 & -5 \end{bmatrix}$.

Solution. We will use the built-in QR-decomposition function, [Q R] = qr(A).

```
>> A1 = A
A1 =

     5     7     0
    10     8     0
     5     6    -5

>> [Q1 R1] = qr(A1);
>> A2 = R1*Q1
A2 =

   13.8333   -1.4881   10.0378
   -1.6254   -2.4371   -2.0258
    1.6823   -1.6176   -3.3962

>> [Q2 R2] = qr(A2);
>> A3 = R2*Q2
A3 =

   15.159187    4.145301   -6.805968
   -0.013431   -4.054621    1.168669
    0.430485    1.750645   -3.104566

>> [Q3 R3] = qr(A3);
>> A4 = R3*Q3
A4 =
```

```
14.959822    6.640881    5.216123
 0.065351   -4.860028   -0.375929
 0.064287   -0.846029   -2.099794
```

It turns out that the correct eigenvalues of A are 15, -5, and -2. These values are already evident on the diagonal after only three iterations. □

It is a simple matter to codify the algorithm into a loop, which allows easily running a large number of iterations. This is left as an exercise for the reader (see Exercise 9).

Chapter 4 Exercises

1. Suppose a hypothetical state is divided into four regions, A, B, C, and D. Each year, a certain number of people will move from one region to another, changing the population distribution. The initial populations are given below:

Region	Population
A	719
B	910
C	772
D	807

The following table records how the population moved in one year.

		To			
		A	B	C	D
From	A	624	79	2	14
	B	79	670	70	91
	C	52	6	623	91
	D	77	20	58	652

For example, we see that A began with $624 + 79 + 2 + 14 = 719$ residents. Of these, 624 stayed in A, 79 moved to B, 2 moved to C, and 14 moved to D. From this empirical data, we can give approximate probabilities for moving from A. Of the 719 residents, 624 stayed in A, so the probability of 'moving' from A to A is $624/719 = 0.8678720$. The probability of moving from A to B is $79/719 = 0.1098748$, and so on.

   (a) Find the transition matrix T for this Markov chain. This is done by converting each entry in the table above to a probability, then transposing.

   (b) Express the initial population distribution as a probability vector $\mathbf{x}$. Remember, the components must add to 1.

   (c) Find the population distribution (expressed as percentages) in 5 years and in 10 years.

   (d) Compute the eigenvalues and eigenvectors for T and use the eigenvector for $\lambda = 1$ to construct an equilibrium vector $\mathbf{q}$ for this Markov chain. This represents a population distribution for which there is no further change from year to year. Verify that the distribution is in equilibrium by computing several future states, such as $T^{25}\mathbf{q}$ and $T^{50}\mathbf{q}$. Is there any change in the distribution?

2. Diagonalize the matrix $A = \begin{bmatrix} 1 & 4 \\ 1 & -2 \end{bmatrix}$ as $A = S\Lambda S^{-1}$ and use this to calculate A^{50}. Show all the steps needed to find the eigenvalues, eigenvectors, etc.

3. Orthogonally diagonalize the symmetric matrix $A = \begin{bmatrix} 1 & 1 & 3 \\ 1 & 1 & 3 \\ 3 & 3 & 9 \end{bmatrix}$. Verify that $A = Q\Lambda Q^T$ and show by direct calculation that Q is an orthogonal matrix.

4. Find the SVD of the matrix $\begin{bmatrix} 2 & 3 \\ 0 & 2 \end{bmatrix}$ without using the svd command. Show all the steps needed to find the eigenvalues, eigenvectors, etc. Verify that $A = U\Sigma V^T$.

5. Use the pseudoinverse to find the least-squares line $y = a + bx$ through the given set of points.

$$\{(-1,5),(1,4),(2,2.5),(3,0)\}$$

You may use the svd command, but show all the rest of the details, including construction of the pseudoinverse. Include a plot of the data values and the least-squares line.

6. For this problem, you will use the SVD to produce a compressed image using n singular values. You choose n (something between 5 and 50 would be suitable). To begin, you will need a digital photo in .JPG format.

   (a) Load the image in Octave as 'imcolor,' then convert to a grayscale image. Check the size of your grayscale image and if it is larger than approximately 320×280, determine an appropriate reduction factor and reduce it. Name the reduced, grayscale format image 'im.' This is the image that will be compressed via the SVD method. Display the reduced grayscale image using imagesc and verify that it still looks like the original. Include a copy of this grayscale image with your problem solutions and state its size.

   Here is a summary of the commands needed for the initial image processing:

```
>> imcolor = imread('photo.jpg'); % load image
>> imgray = rgb2gray(imcolor); % make grayscale
>> size(imgray)                % check image size
>> im = imresize(imgray, factor) % resize image
>> imagesc(im)                 % diplay image matrix
```

   (b) Find the SVD of the matrix representation 'im.' Calculate an approximation using n singular values. That means you should only keep the first n columns of U, the n largest values of Σ, and the first n columns of V. Set the other values to 0, then compute $U\Sigma V^T$ to recover an approximation of the original image. Save it as 'im2' and display it using imagesc.

   How many nonzero values are saved in the compressed factorization compared to the original? Include a copy of the reduced image with your problem solution.

7. Write an Octave script that takes a matrix U with linearly independent columns and outputs a matrix V with orthonormal columns. The core loop could look like this (or use your own formulation):

```
V(:, 1) = U(:, 1)/norm(U(:, 1));
for i = 2:n
  V(:, i) = U(:, i);
  for j = 1:i-1
    V(:, i) = V(:, i) - proj(U(:, i), V(:, j));
  end
  V(:, i) = V(:, i)/norm(V(:, i));
end
```

   You will need to determine m and n and from the dimensions of U and the function proj(u,v) must be defined. Test your code on the vectors from Example 4.5.2.

8. Use your code from Exercise 7 as the starting point of a script file that computes the QR decomposition of an $n \times n$ matrix A. Test your script on a randomly generated 4×4 matrix, A = rand(4,4). Check Q for orthogonality by computing Q^TQ, which should be an identity matrix, and verify that $A = QR$.

9. Using Octave's built-in [Q R] = qr(A) function for the QR decomposition, write a loop (at the command line or in a script file) that will approximate the eigenvalues of the matrix $A = \begin{bmatrix} 1 & -1 & 2 \\ -1 & 1 & -2 \\ 2 & -2 & 0 \end{bmatrix}$. Run your loop through ten iterations. The actual eigenvalues are integers. Were you able to determine the correct values from the QR algorithm?

Chapter 5

Additional topics

5.1 Three dimensional graphs

5.1.1 Space curves

Plotting a curve in 3-dimensions is similar to the 2-dimensional plotting explained in Section 1.3. To plot space curves, we use the command plot3(x, y, z), where x, y, and z correspond to the parametric equations for the function. For example, let's plot a simple helix, with vector equation $\mathbf{r}(t) = \sin(t)\mathbf{i} + \cos(t)\mathbf{j} + t\mathbf{k}$. First we generate a row vector for the parameter t, then we calculate the range for x, y, and z.

```
>> t = linspace(0, 2*pi, 30);
>> x = sin(t);
>> y = cos(t);
>> z = t;
>> plot3(x, y, z)
```

The graph is shown in Figure 5.1.

Now consider a more complicated curve, like

$$x = (4 + \sin 20t)\cos t,\ \ y = (4 + \sin 20t)\sin t,\ \ z = \cos 20t$$

This is a 'toroidal spiral.' We will need to use a much finer increment for t to get a smooth picture.

```
>> t = linspace(0, 2*pi, 500);
>> x = (5 + sin(25*t)).*cos(t);
>> y = (5 + sin(25*t)).*sin(t);
>> z = cos(25*t);
>> plot3(x, y, z)
```

These types of graphs are not easy to draw without a computer! See Figure 5.2.

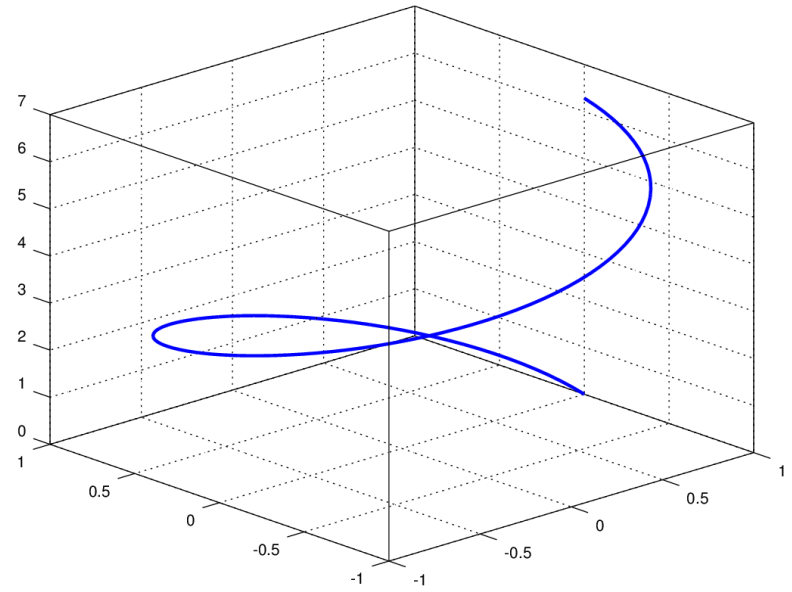


Figure 5.1: Helix

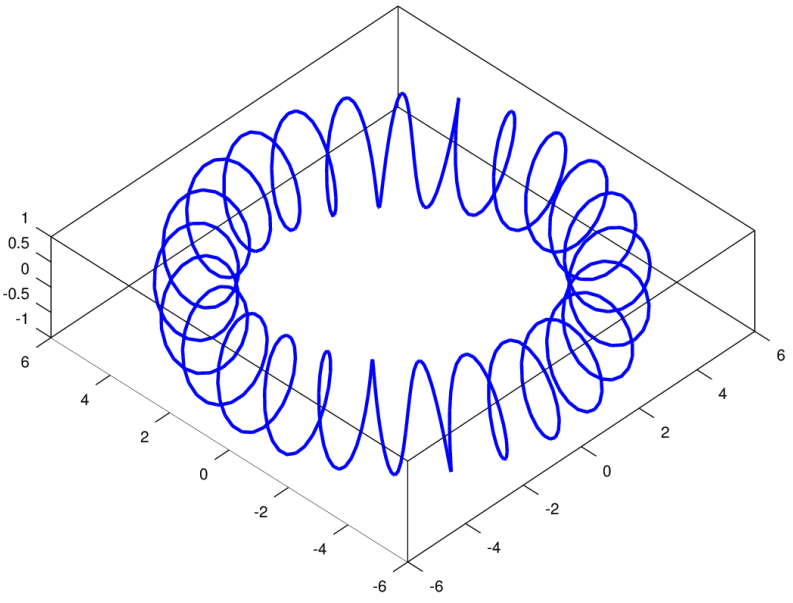


Figure 5.2: Toroidal spiral

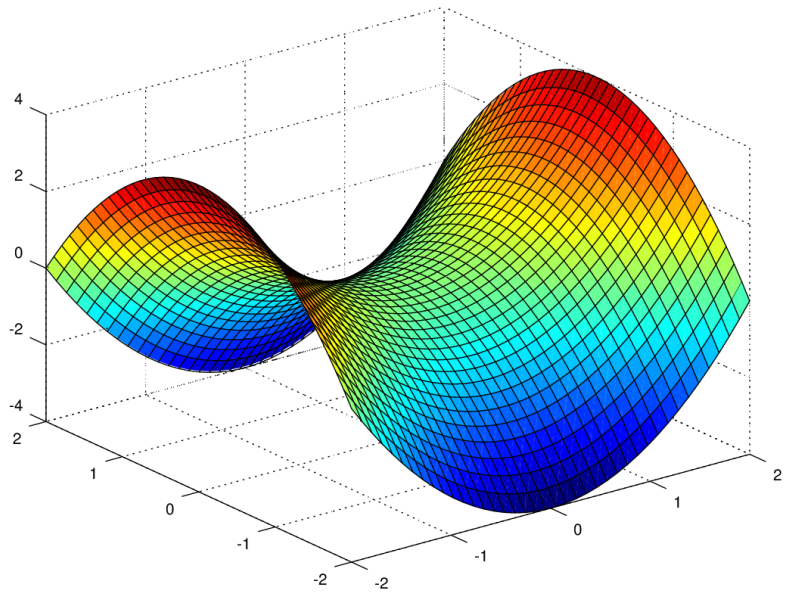


Figure 5.3: Saddle surface

5.1.2 Surfaces

How about plotting surfaces rather than curves? In this case, we use a two-dimensional 'mesh' of input values and calculate the range using a function of two variables. For example, let's graph the familiar saddle surface defined by:

$$f(x, y) = x^2 - y^2$$

First we define the domain.

```
>> x = linspace(-2, 2, 40);
>> y = linspace(-2, 2, 40);
```

Then, we use the meshgrid command to create a mesh of all possible combinations of x and y in the domain.

```
>> [X  Y] = meshgrid(x, y);
```

Now calculate the range using these meshgrid variables.

```
>> Z = X.^2 - Y.^2;
```

Finally, we can plot the surface with the surf command.

```
>> surf(X, Y, Z)
```

Notice that the graph is color coded by elevation. The color map and many other settings can be modified as needed. The graph can be rotated in space by clicking and dragging with the

mouse. To see the graph of the mesh without the surface filled in, use mesh(X, Y, Z). Also try contour(X, Y, Z) to see a contour plot of the surface.

We can use similar steps to plot surfaces defined parametrically, or surfaces defined in terms of polar/cylindrical or spherical coordinates.

Example 5.1.1. The function

$$\rho = 1 + \frac{1}{4}\sin(5\theta)\cos(6\phi), 0 \leq \theta \leq 2\pi, 0 \leq \phi \leq \pi$$

in spherical coordinates is known as a "bumpy sphere." Graph this function.

Solution. We use a (θ, ϕ)-meshgrid to calculate ρ. Then we can calculate X, Y, and Z using the standard spherical to rectangular coordinate identities.

```
>> % define phi (P) and theta (T)
>> theta = linspace(0, 2*pi, 30);
>> phi = linspace(0, pi, 30);
>> [T P] = meshgrid(theta, phi);
>>
>> % calculate rho (R)
>> R = 1 + 1/4*sin(5*P).*cos(6*T);
>>
>> % use spherical identities for X, Y, Z
>> X = R.*sin(P).*cos(T);
>> Y = R.*sin(P).*sin(T);
>> Z = R.*cos(P);
>>
>> % plot the surface
>> surf(X, Y, Z)
```

The graph is shown in Figure 5.4. □

5.1.3 Solids of revolution

Solids of revolution can be graphed as parametrically defined surfaces. For example, parametric equations for the surface formed by rotating the graph of $y = f(x)$ about the x-axis, $a \leq x \leq b$ are:

$$x = x \tag{5.1}$$
$$y = f(x)\cos(t) \tag{5.2}$$
$$z = f(x)\sin(t) \tag{5.3}$$

where $0 \leq t \leq 2\pi$ and $a \leq x \leq b$.

Equations 5.1–5.3 can be modified as needed to produce rotations around the other axes.

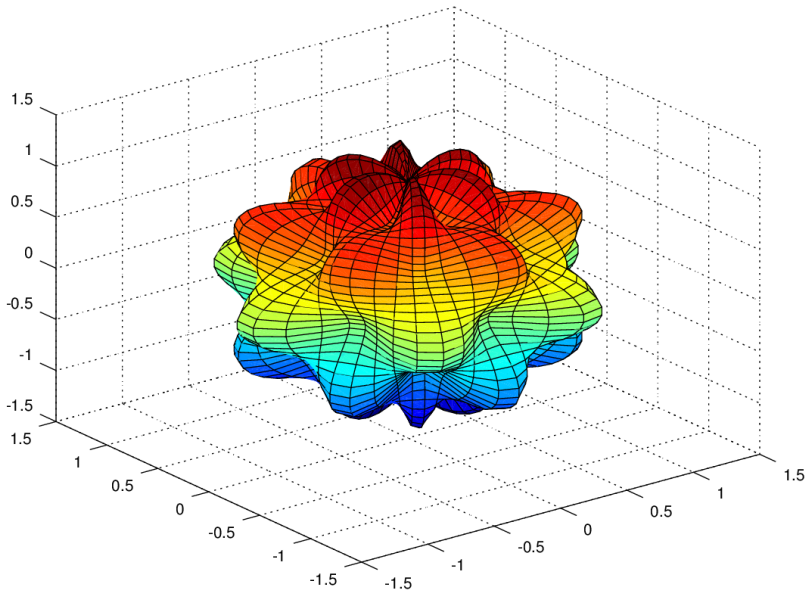


Figure 5.4: Bumpy sphere

Example 5.1.2. Graph the solid obtained by rotating $f(x) = x^2 - 4x + 5$, for $1 \leq x \leq 4$, about the x-axis.

Solution. These commands will graph the surface.

```
>> x = linspace(1, 4, 25);      % define the domain
>> f = @(x) x.^2 - 4*x + 5;     % define the function
>> t = linspace(0, 2*pi, 25);   % define the parameter
>> [X T] = meshgrid(x, t);      % (x,t)-mesh
>> Y = f(X).*cos(T);            % calculate Y
>> Z = f(X).*sin(T);            % calculate Z
>> surf(X, Y, Z)                % graph surface
```

The result is in Figure 5.5. □

5.2 Multiple integrals

We showed methods for evaluating single integrals numerically in Chapter 3. We now consider multiple integrals. The commands dblquad and triplequad can be used to evaluate double and triple integrals over a rectangle or rectangular box.

For example, let's evaluate:

$$\int_{-1}^{3}\int_{0}^{2}(x^2y + 2y)\,dx\,dy$$

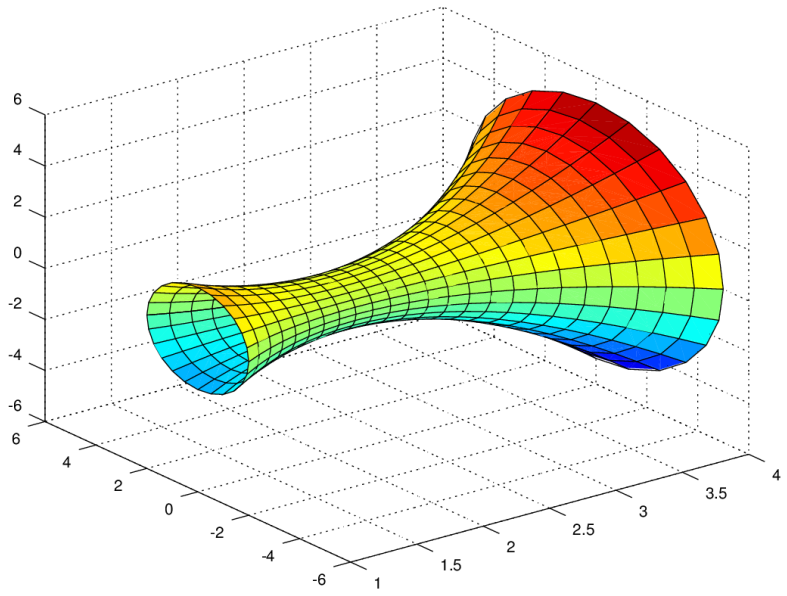


Figure 5.5: Solid of revolution

```
>> % double integral using dblquad
>> function z = f(x, y)
     z = x.^2.*y + 2*y;
   end
>> dblquad('f', 0, 2, -1, 3)
ans =  26.667
```

Evaluating over a nonrectangular domain is a trickier problem. Let's give it a try.

Example 5.2.1. Evaluate

$$\iint_R (x^2y + y^2x)\, dA$$

over the region R bounded by the graphs of $y = x^2$ and $y = \sqrt{x}$.

Solution. An analysis of the region of integration (Figure 5.6) shows that we need to evaluate the following integral:

$$\int_0^1 \int_{x^2}^{\sqrt{x}} (x^2y + y^2x)\, dy\, dx$$

We need to evaluate over only part of the rectangle $[0, 1] \times [0, 1]$. One approach is to define the integrand to be 0 for values outside of the region of integration. We do this using *logical functions*. Logical functions simply test whether a statement is true and return a value of 1 if true or 0 if false. For example $2 + 3 < 4$ returns 0, since the inequality is false. We can also use Boolean operators, like and and or. Our region demands that we meet two conditions:

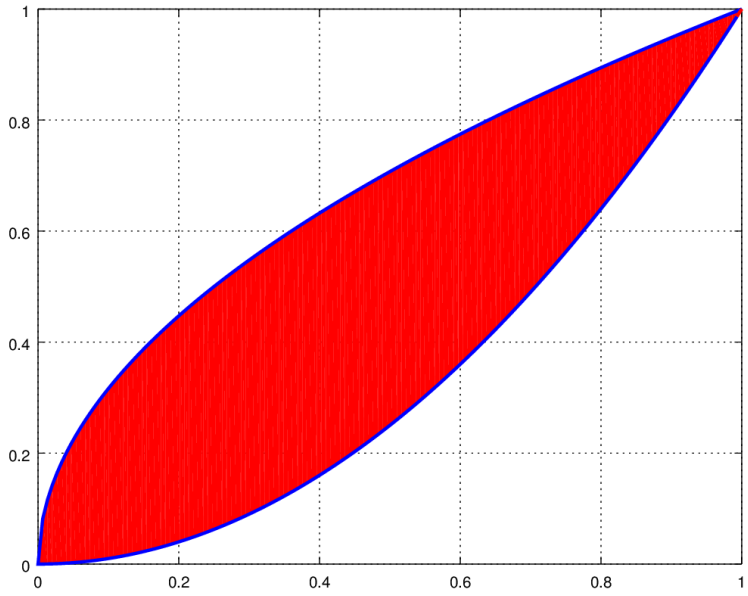


Figure 5.6: Region of integration for Example 5.2.1

$y > x^2$ and $y < \sqrt{x}$, so we use these conditions to define the function. By multiplying the integrand by the correct logical operator, it is set to 0 outside the region of interest.

```
>> % double integral over a nonrectangular domain
>> function z = f(x, y)
     z = (x.^2.*y + y.^2.*x) .* and(y > x.^2, y < sqrt(x));
   end
>> dblquad('f', 0, 1, 0, 1)
ans =  0.10701
```

Thus $\int_{-1}^{3}\int_{0}^{2}(x^2y + 2y)\,dx\,dy \approx 0.10701$. This is reasonably close to the exact value of $3/28$, but not in perfect agreement. The problem is that we have defined f as a discontinuous function (see Figure 5.7), but the quadrature algorithm works best on a smooth integrand. □

The approach in Example 5.2.1 is nice for two reasons: it illustrates the formal definition of a double Riemann integral over a nonrectangular domain (see [6], Vol. 3 §5.2), and it also allows us to plot the surface over the region of interest (Figure 5.7).

```
>> x = linspace(0, 1, 30);
>> y = x;
>> [X Y] = meshgrid(x, y);
>> Z = f(X, Y);
>> surf(X, Y, Z)
```

If we are unsatisfied with the numerical accuracy of this method for the double integral, another

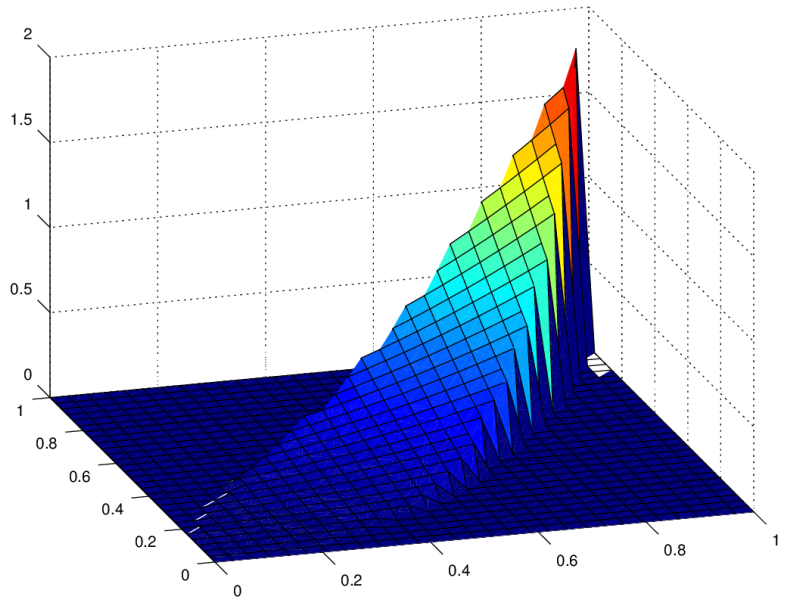


Figure 5.7: Solid volume for Example 5.2.1

approach is to use a change of variables to transform to a rectangular region of integration as follows:

$$y = y_1 + u(y_2 - y_1) \tag{5.4}$$
$$dy = (y_2 - y_1)\, du \tag{5.5}$$

Then as y ranges from y_1 to y_2, u ranges from 0 to 1 and the integrand becomes:

$$\int_{x_1}^{x_2} \int_{y_1}^{y_2} f(x,y)\, dy\, dx = \int_{x_1}^{x_2} \int_{0}^{1} f(x, y_1 + u(y_2 - y_1))(y_2 - y_1)\, du\, dx \tag{5.6}$$

This is a bit cumbersome to enter in Octave. We'll use a series of *anonymous functions* (see page 35) to define the integrand, then try dblquad again.

Example 5.2.2. Use the change of variable formulas in Equations 5.4–5.6 to evaluate

$$\int_0^1 \int_{x^2}^{\sqrt{x}} (x^2 y + y^2 x)\, dy\, dx$$

Solution.

```
>> f1 = @(x, y) x.^2.*y + y.^2.*x;
>> y1 = @(x) x.^2;
>> y2 = @(x) sqrt(x);
>> f2 = @(x, u) f1(x, y1(x) + u.*(y2(x) - y1(x))) .* (y2(x) - y1(x));
```

```
>> format long
>> dblquad(f2, 0, 1, 0, 1) % no quotes around anonymous function name
ans =  0.107142857143983
>> 3/28                    % compare result to known exact answer
ans =  0.107142857142857
```

This approach gives a more satisfactory result. □

If one wishes to evaluate many integrals of this form, writing a *function file* to automate the above steps would be a good idea.

Example 5.2.3. Write an Octave function file that computes

$$\iint_R f(x,y)\,dA$$

over the region R bounded by the graphs of $y = y_1(x)$, $y = y_2(x), x = a,$ and $x = b$, using the change of variables in Equations 5.4–5.6.

Solution. A function file is similar to script: it is a plain text .m-file containing a series of Octave commands. To be recognized as a function file, the first line of code (excluding comments and white space) must be function. With the file placed in the load path, it can be run from the command line like any other Octave function. The function name should match the file name. A well written function file will include details like help text and provisions for error checking. Refer to [3]. We will give a minimal example that accomplishes our change of variables procedure.

Octave Script 5.1: Double integral function file

```
% function file 'dblint.m'
% evaluates dblquad(f, x1, x2, y1, y2)
%   where f is an anonymous function of x and y
%   y1 and y2 are anonymous functions of x
%   x1 and x2 are real numbers

function val = dblint(f, x1, x2, y1, y2)
   f2 = @(x, u) f(x, y1(x) + u.*(y2(x) - y1(x))) .* (y2(x) - y1(x));
   dblquad(f2, x1, x2, 0, 1)
end
```

Note that the comment lines at the top of our function file will be displayed if we type help dblint. Thus we should strive to put a good description of the syntax in those lines. Now, to use this function, saved in our working directory as 'dblint.m,' we need to define the integrand, and the functions representing the limits of integration on y. Then we pass these to our function dblint. Let's try it on the integral from Example 5.2.1.

```
>> f = @(x, y) x.^2.*y + y.^2.*x;
>> y1 = @(x) x.^2;
>> y2 = @(x) sqrt(x);
>> dblint(f, 0, 1, y1, y2)
ans =  0.10714
```

It works! □

5.2.1 Double Riemann sums

Suppose we want to write our own algorithms for double integration. It is straightforward to write an Octave script that will estimate a double integral over a rectangle using a double Riemann sum, taking sample points to be the upper right hand corners of the subrectangles in the partition. For our example, we'll use the function $f(x, y) = x + 2y^2$, defined on $R = [0, 2] \times [0, 4]$, using $m = n = 1000$. Use the following code.

Octave Script 5.2: Nested loop double integral

```
% approximates a double integral using upper right hand corners of
%   subrectangles as sample points
% nested loop version

clear

% define function
function z = f(x, y)
  z = x + 2*y.^2;
end

% define region of integration
a = 0;
b = 2;
c = 0;
d = 4;

% define partition
m = 1000
n = 1000

% calculate dA and initialize Riemann sum total
dx = (b - a)/n
dy = (d - c)/m
dA = dx*dy;
rsum = 0;

% compute double Riemann sum
for i = 1:n
  for j = 1:m
  rsum = rsum + dA *f(a + dx*i, c + dy*j);
  end
end

% display result
rsum
```

This gives an estimate of 93.469, reasonably close to the correct value of 93.333. However, the script is very slow, due to the inefficiency of running the calculation via nested loops. Notice that the program needs to compute one million function values in this example! The routine can be sped up dramatically by using vectorized code, which takes advantage of Octave's fast algorithms for executing matrix and vector calculations. The new strategy is to generate a

meshgrid array of the sample points, then that can be used to define an $m \times n$ matrix containing the function values at the sample points. Then the Riemann sum is simply dA times the sum of all entries in the matrix. This runs MUCH faster!

Octave Script 5.3: Vectorized double integral

```
% approximates a double integral using upper right hand corners of
%   subrectangles as sample points
% vectorized version

clear

% define function
function z = f(x, y)
  z = x + 2*y.^2;
end

% define region of integration
a = 0;
b = 2;
c = 0;
d = 4;

% define partition
m = 1000
n = 1000

% calculate dA
dx = (b - a)/m
dy = (d - c)/n
dA = dx*dy;

% calculate x and y values in partition
x = [a + dx : dx : b];
y = [c + dy : dy : d];

% create matrix of function values
[X Y] = meshgrid(x, y);
A = f(X, Y);

% calculate Reimann sum
rsum = dA*sum(sum(A))
```

While this executes significantly faster, it is still not particularly accurate, considering the rather large values for m and n. The problem is that taking the upper right hand corners as sample points generally does not give the best estimate. The code can easily be improved by taking sample points at the midpoints of each rectangle. This minor adjustment is left as an exercise for the reader (see Exercise 7).

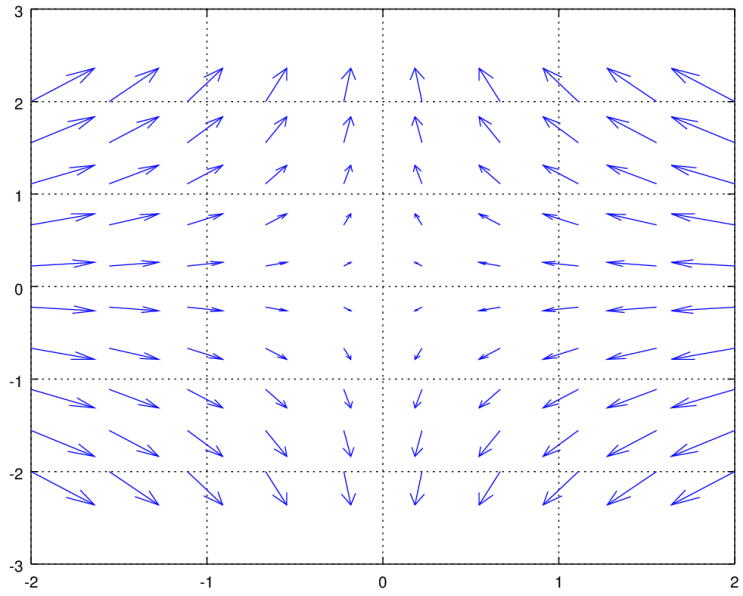


Figure 5.8: Vector field plot

5.3 Vector fields

A *vector field* assigns vectors to points in space. Vector fields are used to describe things like wind speed, fluid flow, electric charge, or gravitational force. A vector field is conveniently visualized by drawing a directed line segment for a series of representative points in the space. As any archer knows, a collection of arrows is called a *quiver*. Thus the command for plotting a vector field is quiver. The simplest form of the command is quiver(X, Y, U, V), where X and Y are meshgrid variables over which the field is plotted and U and V are the x- and y-components, respectively.

Example 5.3.1. Graph the vector field $\mathbf{F}(x, y) = \langle -x, y \rangle$.

Solution.

```
>> x = linspace(-2, 2, 10);
>> y = x;
>> [X Y] = meshgrid(x, y);
>> quiver(X, Y, -X, Y);
>> grid on
```

See Figure 5.8. Some experimentation may be needed to determine the correct grid spacing. Too many points will result in an array of vectors too dense to interpret. □

We can also plot vector fields in three dimensions with quiver3 or add a vector field plot to the contour graph of a surface. We will illustrate these ideas with two more examples.

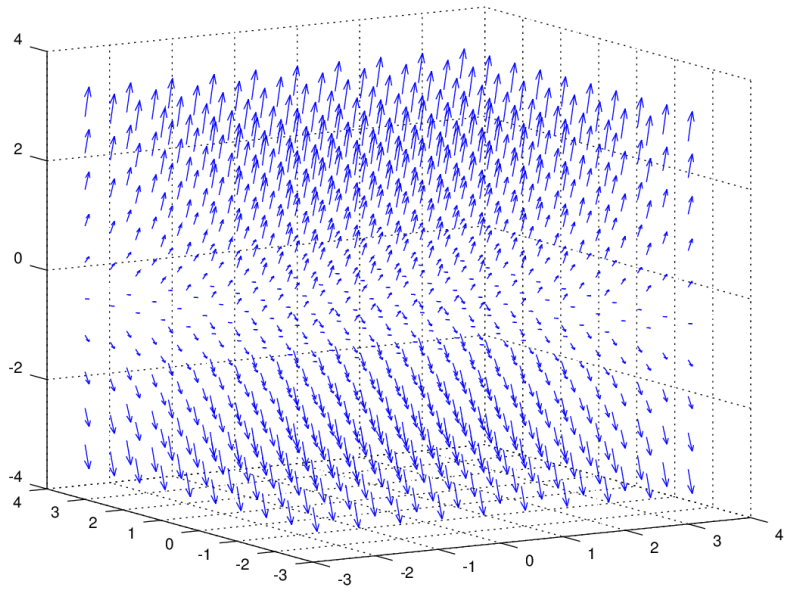


Figure 5.9: Three dimensional vector field

Example 5.3.2. Plot the vector field $\mathbf{F}(x, y, z) = \langle 1, 1, z \rangle$.

Solution.

```
>> x = linspace(-3, 3, 10);
>> y = x;
>> z = x;
>> [X Y Z] = meshgrid(x, y, z);
>> quiver3(X, Y, Z, ones(size(X)), ones(size(Y)), Z)
```

Note the use of the ones command to produce the constant terms. The result is in Figure 5.9. □

Example 5.3.3. Graph a contour plot of the Octave function "peaks" and its gradient field.

Solution. The command peaks plots an example graph of a surface with a number of maximums and minimums. Type help peaks for details, or just peaks to see the graph. It will be instructive to see its contours plotted together with its gradient field. We can use the built-in gradient function.

```
>> [X Y Z] = peaks;
>> [DX DY] = gradient(Z);
>> contour(X, Y, Z)
>> hold on
>> quiver(X, Y, DX, DY)
>> axis([-2 2 -2 2])
>> hold off
```

See the results in Figure 5.10. □

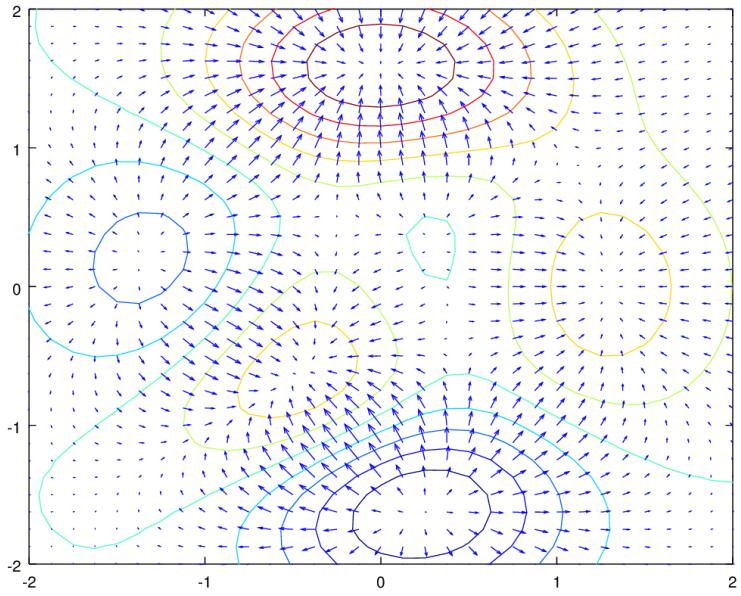


Figure 5.10: Gradient field

5.4 Statistics

Octave has good capabilities for statistical analysis. We'll start with something simple. Let's try rolling a six-sided die:

```
>> floor(6*rand + 1)
ans = 6  % the answer is random - your results will vary!
```

Now let's try repeating the experiment 100 times, storing the results in a column vector. We can analyze the results by looking at the sample mean, variance, and a histogram.

```
>> A = floor(6*rand(100, 1) + 1);
>> mean(A)
ans =  3.4100
>> var(A)
ans =  2.9514
>> hist(A, [1 2 3 4 5 6])
```

Your results will vary, but you should see something that looks close to a uniform distribution, such as in Figure 5.11. The vector [1 2 3 4 5 6] specifies the midpoints of the bins.

Now, let's use a loop to generate a distribution of 100 sample means.

```
>> for i = 1:100
       A = floor(6*rand(100, 1) + 1);
       d(i) = mean(A);
   end
>> hist(d)
```

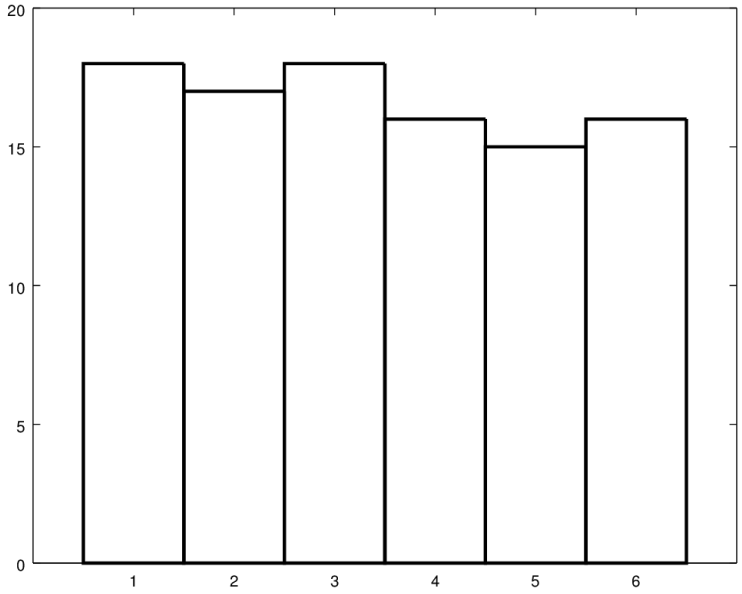


Figure 5.11: Results from 100 6-sided die trials

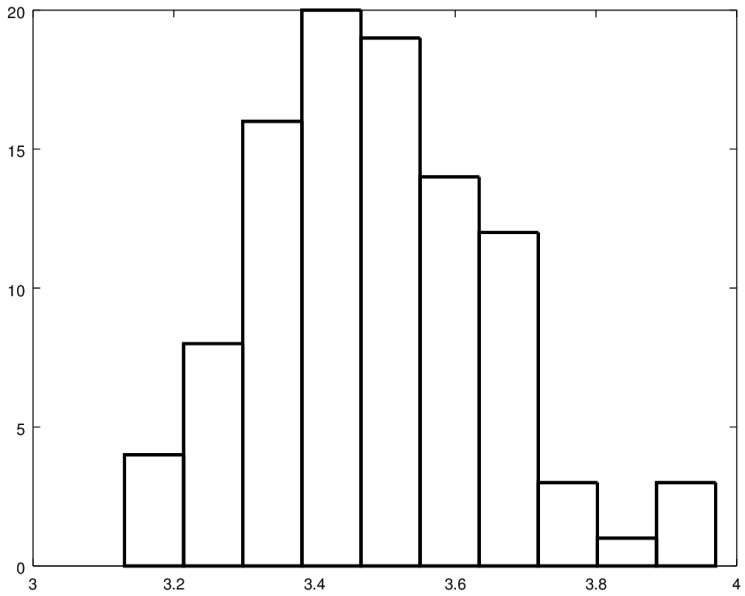


Figure 5.12: Distribution of sample means

Notice that the distribution of the sample means is approximately normal (Figure 5.12), even though the underlying distribution is not. We have just demonstrated the Central Limit Theorem!

We have seen that the rand function corresponds to a uniform distribution. Octave has other distributions built-in. For example, the function randn returns a matrix with normally distributed elements with mean 0 and standard deviation 1.

Example 5.4.1. Create a vector Z of 1000 elements from the standard normal distribution. Use the transformation $X = Z\sigma + \mu$ to generate a vector X of elements from a normal distribution with mean 400 and standard deviation 50. Compare the means and variances of X and Z. Plot histograms of Z and X.

Solution. Here are the commands we need:

```
>> Z = randn(1000, 1);
>> mu = 400;
>> sigma = 50;
>> X = Z*sigma + mu;
>> format free;
>> means = mean([Z X])
means =

  -0.00116119  399.942

>> variances = var([Z X])
variances =

  1.04291  2607.28

>> hist(Z)
>> hist(X)
```

The command format free changes from the default short form scientific notation, which makes it a bit easier to compare the means and variances, in this case. We can see Z has mean and variance near 0 and 1, respectively, while X has a mean near 400 and variance near 2500, as expected. The histograms are identical, except for the scale on the horizontal axis (see, for example, Figure 5.13). □

Octave can handle many other statistical functions. As a final example, we will consider a simple hypothesis test. See [2] for background on the basic theory of statistical tests.

Example 5.4.2. Consider the following set of sample data (from a normally distributed population):

$$\{24.9, 22.8, 16.2, 10.8, 32.0, 19.2\}$$

Test the following hypotheses at significance level $\alpha = 0.05$:

$$\begin{aligned} H_0 &: \ \mu = 30 \\ H_a &: \ \mu < 30 \end{aligned}$$

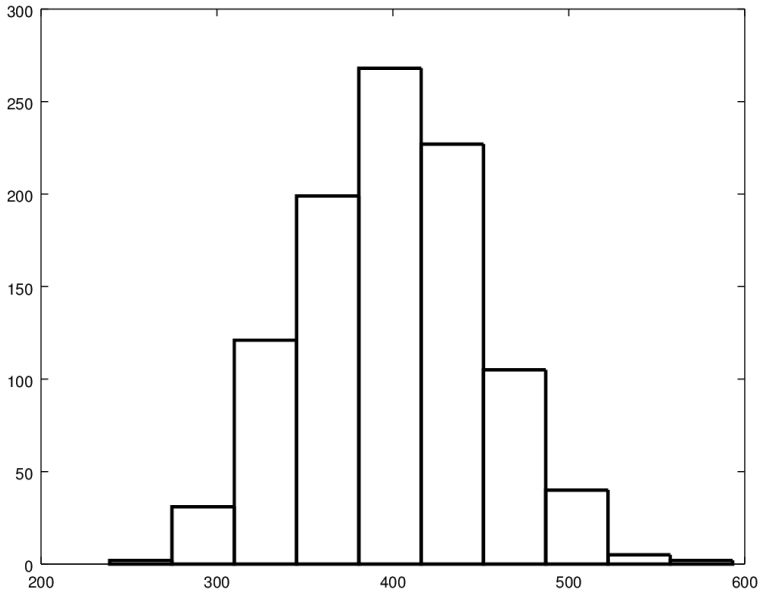


Figure 5.13: X-distribution

Solution. Enter the data and calculate the mean.

```
>> x = [24.9  22.8  16.2  10.8  32.0  19.2]'
x =

   24.900
   22.800
   16.200
   10.800
   32.000
   19.200

>> mean(x)
ans =  20.983
```

Is $\overline{x} = 20.983$ good evidence that $\mu < 30$? We can use the t_test command. We enter the vector **x**, the value of μ from the null hypothesis, and the alternative, which takes the form '<', '>', or '<>' (for $\neq$).

```
>> t_test(x, 30, '<');
   pval: 0.0149318
```

We can see that the P-value is less than α. Based on this, we reject the null hypothesis. □

5.5 Differential equations

5.5.1 Slope fields

The quiver function we used to plot vector fields in Section 5.3 can also be used to plot the slope field of a differential equation. The key is recognizing that a differential equation of the form $dy/dx = f(x, y)$ is a function that gives us slopes, which we can interpret as vectors. This will be illustrated with a simple example.

Example 5.5.1. Plot the slope field along with several solutions of the differential equation

$$\frac{dy}{dx} = x$$

Solution. The solution is $y = \frac{1}{2}x^2 + C$. For differential equations that cannot be solved so easily, plotting the slope field can be used to get a sense of the solutions. In this example, since we know the solution, we can show how the solutions follow the slope field.

We need to define the input range as a meshgrid, define the function, then use the function to calculate slopes. To get a good looking graph, we then scale these slope vectors to a unit length. Finally, we plot some solutions for different values of C.

```
>> % define input values
>> x = linspace(-5, 5, 30);
>> y = x;
>> [X Y] = meshgrid(x, y);
>>
>> % define function
>> f = @(x, y) x;
>>
>> % delta-y, relative to 1 unit delta-x
>> dY = f(X, Y);
>> dX = ones(size(dY));
>>
>> % factor to scale to unit length
>> L = sqrt(1 + dY.^2);
>>
>> % plot the field
>> quiver(X, Y, dX./L, dY./L, 0.5) % scaling factor 0.5
>> axis([-4 4 -4 4])
>> grid on
>> xlabel('x')
>> ylabel('y')
>>
>> % add some particular solutions to graph for comparison
>> hold on
>> for C = -4:3
     plot(x, 0.5*x.^2 + C, 'r', 'linewidth', 2)
   end
```

The results are shown in Figure 5.14. □

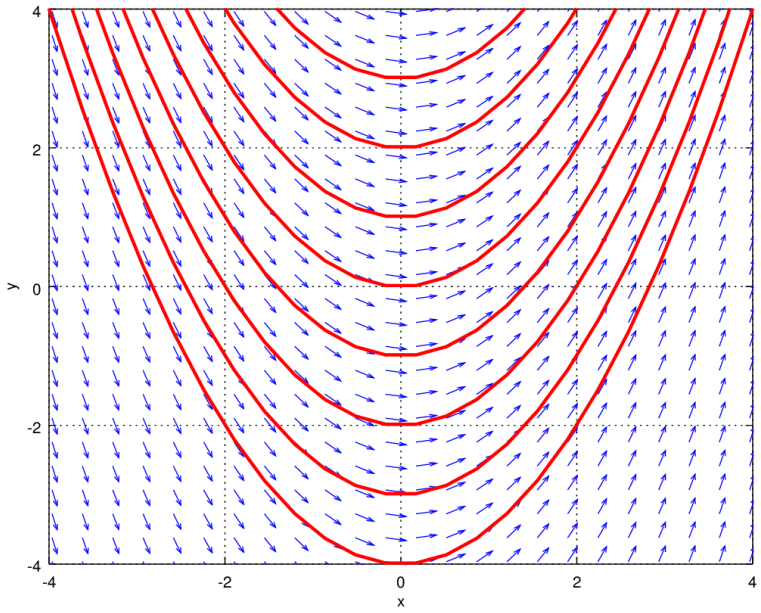


Figure 5.14: Slope field and solutions

5.5.2 Euler's method

Euler's method is probably the simplest numerical technique for solving an ordinary differential equation.

Given a differential equation $y' = f(x, y)$ with initial condition $y(x_0) = y_0$, Euler's method gives approximate solutions:

$$y_{i+1} = y_i + hf(x_i, y_i) \tag{5.7}$$

The value of h is the *step size*. If the interval $[x_0, b]$ is divided into n equally spaced subintervals, then $h = \dfrac{b - x_0}{n}$. To see how this works, let's look at an example.

Example 5.5.2. Solve

$$y' = e^{-3x} - 3y, \quad y(0) = 1$$

on $[0, 3]$ using a step size of 1.

Solution. We will generate a series of approximate y-values at $x = 0, 1, 2, 3$. The value y_0 is given. We compute the remaining values using Equation 5.7. Here is the first step:

$$\begin{aligned} y_1 &= y_0 + hf(x_0, y_0) \\ &= 1 + (1)f(0, 1) \\ &= 1 + (1)(-2) \\ &= -1 \end{aligned}$$

This is then used to compute y_2.

$$\begin{aligned} y_2 &= y_1 + hf(x_1, y_1) \\ &= -1 + (1)f(1, -1) \\ &= 2.0498 \end{aligned}$$

One more step:

$$\begin{aligned} y_3 &= y_2 + hf(x_2, y_2) \\ &= 2.0498 + (1)f(2, 2.0498) \\ &= -4.0971 \end{aligned}$$

Our approximate solutions are summarized in the following table.

x	y
0	1.0000
1	−1.0000
2	2.0498
3	−4.0971

Unfortunately, these solutions are not very accurate. But, we can do much better by decreasing the step size, as shown in the next example. □

These repetitive computations are best implemented in an Octave script. This allows using a smaller step size, which gives a finer range of solution values and also improves the overall accuracy. Refer to [7] for a fuller discussion of the accuracy of Euler's method and a range of more sophisticated algorithms.

Example 5.5.3. Solve

$$y' = e^{-3x} - 3y, \quad y(0) = 1$$

on $[0, 3]$ using a step size of 0.1.

Solution. We will write a fairly general Octave script that can be easily modified for different functions, intervals, and step sizes.

Octave Script 5.4: Euler's method

```
% Euler's method solution for
%   dy/dx = e^(-3x) - 3y, y(0) = 1 on [0, 3]

% define function and initial condition
f = @(x, y) exp(-3*x) - 3*y;
y0 = 1;

% define interval and step size
a = 0;
b = 3;
h = 0.1; % note: step size must divide b-a
n = (b - a)/h;

```

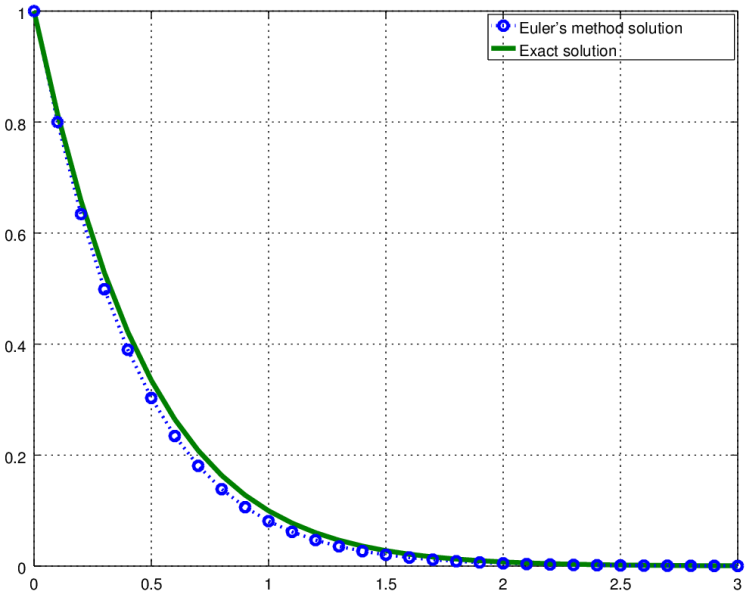


Figure 5.15: Euler's method solution for Example 5.5.3

```
% define x-values
clear x;
x = [a : h : b];

% calculate y-values
clear y;
y(1) = y0;
for i = 1:n
    y(i + 1) = y(i) + h*f(x(i), y(i));
end

% plot solutions
>> plot(x, y, 'o:', 'linewidth', 2)
```

Figure 5.15 shows the approximated solution compared to the exact solution, which is known to be $y = e^{-3x}(x + 1)$. □

5.5.3 The Livermore solver

Octave has a built-in function for solving differential equations numerically, called lsode, which implements the FORTRAN routine of the same name (Livermore solver for ordinary differential equations). The command lsode(f, x_0, t) solves differential equation $dx/dt = f(x, t)$ with initial condition $x(t_0) = x_0$ over the range specified by t. Notice that the initial value x_0 needs to correspond to the first value of the vector **t**. Refer to the documentation for further details.

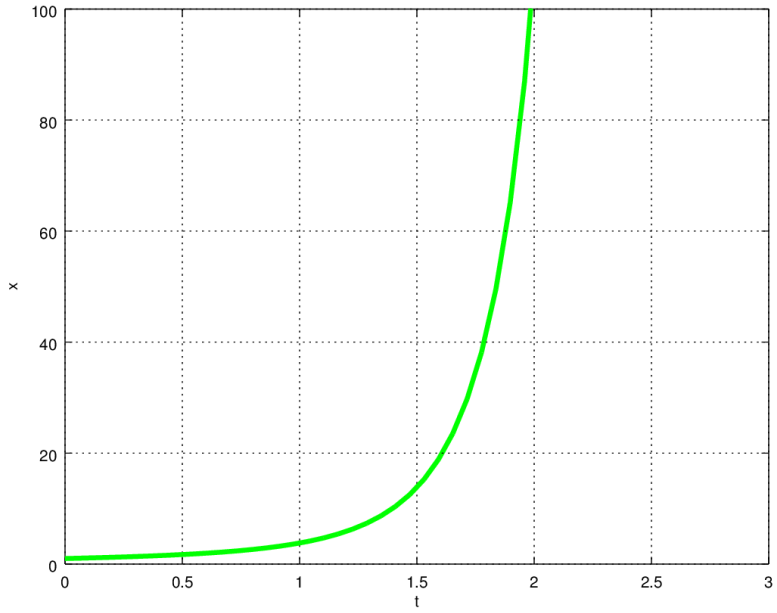


Figure 5.16: lsode numeric solution

Example 5.5.4. Solve the differential equation

$$\frac{dx}{dt} = x(t^2 + 1)$$

with initial condition $x(0) = 1$.

Solution. We use Octave to solve the problem on $0 \leq t \leq 3$.

```
>> % define the function, input values, and initial condition
>> f = @(x, t) x.*(t.^2 + 1);
>> t = linspace(0, 3, 50);
>> x0 = 1;
>>
>> % calculate the solutions
>> sol = lsode(f, x0, t);
>>
>> % plot the solutions
>> plot(t, sol, 'g', 'linewidth', 3)
>> grid on
>> xlabel('t')
>> ylabel('x')
>> axis([0 3 0 100])
```

The solution is shown in Figure 5.16. Note that the exact solution is easily found to be $x = e^{\frac{1}{3}t^2+t}$, which is in good agreement with our graph. □

Chapter 5 Exercises

1. Graph the function $\mathbf{r}(t) = \langle e^{-t}, t, \sin(t) \rangle$.

2. Graph the function $f(x, y) = \sin(xy) + 1$.

3. Find a vector function that represents the curve of intersection of the circular cylinder $x^2 + y^2 = 4$ and the parabolic cylinder $z = x^2$. Graph the two surfaces and the curve of intersection.
   The command [X Y Z] = cylinder([2 2]) can be used to obtain a cylinder of radius 2.

4. Calculate the volume of the bumpy sphere from Example 5.1.1.

5. A cylindrical drill with radius 1 is used to bore a hole through the center of a sphere of radius 5. Graph the ring shaped solid that remains and find its volume.

6. Use dblquad to evaluate the double integral
   $$\iint_D x \cos y \; dA$$
   where D is bounded by $y = 0$, $y = x^2$, and $x = 2$.

7. Let $f(x, y) = x + 2y^2$, defined on $R = [0, 2] \times [0, 4]$. Write a vectorized algorithm to compute a double Riemann sum using midpoints of each subrectangle as the sample points. Use a partition with $m = n = 1000$. Compare your results to the value computed using dblquad('f', 0, 2, 0, 4) and to the estimate using upper right hand corner sample points.

8. Plot the vector field $\mathbf{F}(x, y) = \tan^{-1}\left(\frac{y}{x}\right)\mathbf{i} + \ln(x^2)\mathbf{j}$.

9. Plot the vector field $\mathbf{F}(x, y, z) = \frac{-x}{(x^2+y^2+z^2)^{3/2}}\mathbf{i} + \frac{-y}{(x^2+y^2+z^2)^{3/2}}\mathbf{j} + \frac{-z}{(x^2+y^2+z^2)^{3/2}}\mathbf{k}$.

10. The function binopdf(x, n, p) gives the probability of x successes in n trials of a binomial experiment with a probability of success p on each trial. Plot binomial distributions for $n = 10$, 25, and 50 with $p = 0.8$. This can be done with the command bar(x, B), where x is the vector of possible outcomes and B is the corresponding vector of binomial probabilities. What happens to the shape of the distribution as n increases?

11. Use Euler's method to solve the differential equation
    $$\frac{dx}{dt} = x(t^2 + 1)$$
    with initial condition $x(0) = 1$ on $[0, 3]$ using a step size $h = 0.1$. Compare to the lsode solution from Example 5.5.4.

12. Graph the slope field for the logistic equation $\dfrac{dy}{dx} = y(1 - y)$. Verify that $y = 1/(1 + e^{-x})$ is a solution to the equation. Graph this function over the slope field and set the axes to an appropriate range.

13. Use lsode to solve the differential equation from Exercise 12 if $y(0) = 0.5$. Plot the solution over the slope field. Does the solution agree with the previous solution?

Appendix A

MATLAB compatibility

Octave and MATLAB use similar syntax, but the programs are not identical. MATLAB, especially when extended with its various toolboxes, has many functions not available in Octave. For example, symbolic operations are possible, and many more options exist for solving differential equations beyond Octave's lsode. However, most code in this book will work in MATLAB, so what you've learned here will easily transfer to more advanced MATLAB programming.

Octave allows some flexibility in syntax that MATLAB does not. This book has been written with MATLAB compatibility in mind, so when multiple forms of a command or operation are possible, the MATLAB compatible option has been used. Here is a summary of a few of the potential coding differences.

- Comments in Octave can be preceded by # or %. MATLAB uses %.
- Octave recognizes single quotes and double quotes around strings. MATLAB requires single quotes.
- Blocks in Octave can be ended with end statements based on the initial command (endfor, endfunction, etc). MATLAB uses only end.
- The not-equal comparison can be written as != or ~= in Octave. MATLAB uses ~=.
- Octave allows user-defined functions to be entered at the command line or in scripts. MATLAB requires the use of separate function files.
- The lsode command is not implemented in MATLAB. There are many other options available, such as ode45. To use ode45 and several other MATLAB compatible solvers in Octave, load the package 'odepkg.'

Appendix B

List of Octave commands

The names and basic syntax for many common commands are provided below. Type help NAME at the Octave prompt for more details.

List of Octave commands

Syntax	Description
[a:step:b]	vector from a to b by increment 'step'
A'	transpose of A
A*B	matrix product $A \times B$
A\b	left division, solves system $A\mathbf{x} = \mathbf{b}$
A^n	matrix power
A + B	sum $A + B$
x < y	less than comparison
x > y	greater than comparison
x == y	equality comparison
x ~= y	not equal comparison
x.*y	elementwise product
x./y	elementwise quotient
x.^n	elementwise exponent
abs(x)	absolute value of x
acos(x)	inverse cosine of x in radians
and(a, b)	logical AND
ans	result of last calculation
asin(x)	inverse sine of x in radians
atan(x)	inverse tangent of x in radians
axis([Xmin Xmax Ymin Ymax])	set axis limits
bar(x, y)	bar graph of y vs. x
besselj(n, x)	order n Bessel functions of the first kind
binopdf(x, n, p)	binomial probability of x successes
ceil(x)	$\lceil x \rceil$, the least integer greater than or equal to x
clear var1	clear variable (clear all if no variable listed)
clf	clear plot window
contour(X, Y, Z)	contour plot of surface
corr(x, y)	linear correlation coefficient r
cos(x)	cosine of x (x in radians)
cosh(x)	hyperbolic cosine of x
cross(u, v)	cross product of vectors $\mathbf{u}$ and $\mathbf{v}$
cylinder([r r])	cylinder of radius r
dblquad('f', a, b, c, d)	double integral over rectangle

continued ...

...continued

Syntax	Description
det(A)	determinant of A
dir	list files in current directory
dot(u, v)	dot product of vectors **u** and **v**
[v lambda] = eig(A)	find eigenvalues and eigenvectors
e	the number e
erf(x)	the error function
exp(x)	natural exponential function
eye(n)	$n \times n$ identity matrix
floor(x)	$\lfloor x \rfloor$, the greatest integer less than or equal to x
for i = 1:n ... end	for loop
format opt	decimal format, options = short, long, free, bank, etc
fsolve('f', x1)	solve $f(x) = 0$, initial guess x_1
function y = f(x) ... end	define a function
f = @(x1, x2, ...) rule	anonymous function of $x_1, x_2, \ldots$ given by rule
[DX DY] = gradient(Z)	gradient field of Z
gamma(x)	the gamma function
grid on/off	toggle plot grid
help NAME	get documentation for command 'NAME'
hist(X, B)	histogram of X, optional B specifies bins
hold on/off	add to current plot toggle on/off
I	the imaginary unit
im = imread('name')	load image
imagesc(A)	display matrix as scaled image
imresize(im, factor)	scale image by factor
imshow(im)	display image
inv(A)	inverse of matrix A
legend('plot1', 'plot2', ...)	plot legend
linspace(a, b, n)	vector of n evenly spaced points from a to b
load filename	load saved variables
log(x)	natural logarithm
lsode(f, x_0, t)	solves $dx/dt = f(x,t)$, $x(0) = x_0$
[L U P] = lu(A)	LU decomposition of A, with permutation
max(A)	maximum of vector, or column-wise maximums of a matrix
max(max(A))	maximum of all entries in matrix A
mean(x)	mean of x
mesh(X, Y, Z)	surface plotted as a mesh
[X Y] = meshgrid(x, y)	generate X, Y-meshgrid
min(A)	minimum of vector, or column-wise minimums of a matrix
min(min(A))	minimum of all entries in matrix A
norm(u)	norm (length) of vector **u**
normcdf(x, mu, sigma)	area under normal curve to the left of x
norminv(a, mu, sigma)	inverse normal distribution given area a
ones(m, n)	$m \times n$ matrix of ones

continued ...

. . . continued

Syntax	Description
or(a, b)	logical OR
peaks	example 3d graph of a surface with many local extrema
pi	the number π
pinv(A)	pseudoinverse of A
pkg install −forge NAME	download and install a package from Octave Forge
pkg list	list installed packages
pkg load NAME	load an installed package
plot(x, y)	plot of x vs. y
plot3(x, y, z)	plot space curve
polyfit(x, y, order)	polynomial fit x vs. y of degree 'order'
polyval(P, x)	evaluate polynomial P at x
print −dpng filename.png	save plot as png (substitute jpg, eps, etc)
pwd	list current working directory
[Q R] = qr(A)	QR decomposition of A
quad('f', a, b)	definite integral
quiver(X, Y, U, V)	plot vector field with components U and V
quiver3(X, Y, Z, U, V, W)	plot 3d vector field with components U, V, W
rand(m, n)	$m \times n$ random matrix (uniformly distributed entries)
randn(m, n)	$m \times n$ random matrix (normally distributed entries)
rank(A)	rank of A
rgb2gray(im)	convert image to grayscale
save filename A, B, ...	save variables A, B, ...
sin(x)	sine of x (x in radians)
sinh(x)	hyperbolic sine of x
size(A, opt)	dimensions of A; dimension option: 1=rows, 2=cols
sqrt(x)	principle square root of x
sum(A)	sum of vector components or column-wise sum of matrix A
sum(sum(A))	sum of all entries in matrix A
surf(X, Y, Z)	surface plot
[U S V] = svd(A)	singular value decomposition of A
tcdf(x, n)	Students t-distribution CDF with degrees of freedom n
tinv(a, n)	inverse t-distribution with degrees of freedom n
t_test(X, mu_0, alt)	t-test, alternative hypothesis = '<','>', or '<>'
tan(x)	tangent of x (x in radians)
tanh(x)	hyperbolic tangent of x
title('name')	assign plot title
triplequad('f', a, b, c, d, e, f)	triple integral over a rectangular box
var(x)	variance of x
whos	list variables in current scope
xlabel('name')	horizontal axis label
ylabel('name')	vertical axis label
zeros(m, n)	$m \times n$ matrix of zeros

References

[1] Brin, Leon Q, *Tea Time Numerical Analysis, 2nd edition.* CC-BY-SA, 2016.
http://lqbrin.github.io/tea-time-numerical/

[2] Diez, David M, Christopher D Barr, and Mine Çetinkaya-Rundel, *OpenIntro Statistics, 3rd edition.* CC-BY-SA, 2015.
https://www.openintro.org/stat/textbook.php

[3] Eaton, John W, David Bateman, Søren Hauberg, and Rik Wehbring, *GNU Octave Manual: Edition 4.* GNU FDL, 2017.
https://www.gnu.org/software/octave/octave.pdf

[4] Grinstead, Charles M, and J Laurie Snell, *Introduction to Probability, 2nd Edition.* American Mathematical Society. GNU FDL, 2006.
http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/book.html

[5] Kuttler, Ken, *A First Course in Linear Algebra.* CC-BY, 2017.
http://lyryx.com/textbooks/Kuttler-LinearAlgebra-AFirstCourse-2017A.pdf

[6] Strang, Gilbert, Edwin Herman, et al, *Calculus Volumes 1–3.* OpenStax. CC-BY-NC-SA, 2016.
https://openstax.org/details/books/calculus-volume-1
https://openstax.org/details/books/calculus-volume-2
https://openstax.org/details/books/calculus-volume-3

[7] Trench, William F, *Elementary Differential Equations.* Trinity University, Digital Commons. CC-BY-NC-SA, 2013.
http://digitalcommons.trinity.edu/mono/8/

Index

Cover image: *Mandelbrot Set*, created with GNU Octave by the author.

These notes are intended to provide a brief, noncomprehensive introduction to GNU Octave, a free open source alternative to MatLab. The basic syntax and usage is explained through concrete examples from the mathematics courses a math, computer science, or engineering major encounters in the first two years of college: linear algebra, calculus, and differential equations.