LINEAR ALGEBRA



AND PROBABILITY

FOR COMPUTER SCIENCE

APPLICATIONS

ERNEST DAVIS

Linear Algebra and Probability

for Computer Science Applications

##### This page intentionally left blank

Linear Algebra and Probability

for Computer Science Applications

Ernest Davis





MATLAB® is a trademark of The MathWorks, Inc. and is used with permission. The MathWorks does not warrant the accuracy of the text or exercises in this book. This book’s use or discussion of MATLAB® software or related products does not constitute endorsement or sponsorship by The MathWorks of a particular pedagogical approach or particular use of the MATLAB® software.

CRC Press

Taylor & Francis Group

6000 Broken Sound Parkway NW, Suite 300 Boca Raton, FL 33487-2742

© 2012 by Taylor & Francis Group, LLC

CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works Version Date: 20120330

International Standard Book Number-13: 978-1-4665-0159-1 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access [www.copyright.com](http://www.copyright.com/) (<http://www.copyright.com/)> or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

**Trademark Notice:** Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

**Visit the Taylor & Francis Web site at** [**http://www.taylorandfrancis.com**](http://www.taylorandfrancis.com/)

**and the CRC Press Web site at** [**http://www.crcpress.com**](http://www.crcpress.com/)

#### For my beloved father,

Philip J. Davis,

who has had a lifelong love of *matrices*,

a profound respect for the workings of *chance,*

and a healthy distrust of the *applications of statistical inference*.

Consider the recent flight to Mars that put a “laboratory vehicle” on that planet.

... From first to last, the Mars shot would have been impossible without a tre- mendous underlay of mathematics built into chips and software. It would defy the most knowledgeable historian of mathematics to discover and describe all the mathematics involved.

—Philip Davis, “A Letter to Christina of Denmark,”

*Newsletter of the European Mathematical Society* 51 (March 2004), 21–24

# Contents

Preface xiii

1. MATLAB 1
   1. Desk Calculator Operations 1
   2. Booleans 2
   3. Nonstandard Numbers 3
   4. Loops and Conditionals 4
   5. Script File 6
   6. Functions 7
   7. Variable Scope and Parameter Passing 9

I Linear Algebra 15

1. Vectors 17
   1. Definition of Vectors 17
   2. Applications of Vectors 17
      1. General Comments about Applications 20
   3. Basic Operations on Vectors 21
      1. Algebraic Properties of the Operations 21
      2. Applications of Basic Operations 22
   4. Dot Product 23
      1. Algebraic Properties of the Dot Product 23
      2. Application of the Dot Product: Weighted Sum 23
      3. Geometric Properties of the Dot Product 24
      4. Metacomment: How to Read Formula Manipulations 27
      5. Application of the Dot Product: Similarity of

Two Vectors 28

* + 1. Dot Product and Linear Transformations 30

vii

* 1. [Vectors in MATLAB: Basic Operations 31](#_TOC_250002)
     1. [Creating a Vector and Indexing 31](#_TOC_250001)
     2. [Creating a Vector with Elements in Arithmetic Sequence 31](#_TOC_250000)
     3. Basic Operations 33
     4. Element-by-Element Operations 33
     5. Useful Vector Functions 34
     6. Random Vectors 35
     7. Strings: Arrays of Characters 36
     8. Sparse Vectors 36
  2. Plotting Vectors in MATLAB 37
  3. Vectors in Other Programming Languages 41

1. Matrices 47
   1. Definition of Matrices 47
   2. Applications of Matrices 47
   3. Simple Operations on Matrices 49
   4. Multiplying a Matrix Times a Vector 49
      1. Applications of Multiplying a Matrix Times a Vector 51
   5. Linear Transformation 55
   6. Systems of Linear Equations 56
      1. Applications of Systems of Linear Equations 57
   7. Matrix Multiplication 62
   8. Vectors as Matrices 65
   9. Algebraic Properties of Matrix Multiplication 66
      1. Matrix Exponentiation 67
   10. Matrices in MATLAB 68
       1. Inputting Matrices 68
       2. Extracting Submatrices 70
       3. Operations on Matrices . 71
       4. Sparse Matrices 73
       5. Cell Arrays 75
2. Vector Spaces 81
   1. Fundamentals of Vector Spaces 81
      1. Subspaces 82
      2. Coordinates, Bases, Linear Independence 84
      3. Orthogonal and Orthonormal Basis 87
      4. Operations on Vector Spaces 88
      5. Null Space, Image Space, and Rank 89
      6. Systems of Linear Equations 91
      7. Inverses 93
      8. Null Space and Rank in MATLAB 93
   2. Proofs and Other Abstract Mathematics (Optional) 94
      1. Vector Spaces 94
      2. Linear Independence and Bases 95
      3. Sum of Vector Spaces 98
      4. Orthogonality 99
      5. Functions 102
      6. Linear Transformations 104
      7. Inverses 106
      8. Systems of Linear Equations 107
   3. Vector Spaces in General (Very Optional) 108
      1. The General Definition of Vector Spaces 108
3. Algorithms 115
   1. Gaussian Elimination: Examples 115
   2. Gaussian Elimination: Discussion 117
      1. Gaussian Elimination on Matrices 120
      2. Maximum Element Row Interchange . 121
      3. Testing on Zero 122
   3. Computing a Matrix Inverse 123
   4. Inverse and Systems of Equations in MATLAB 127
   5. Ill-Conditioned Matrices 131
   6. Computational Complexity 135
      1. Viewpoints on Numerical Computation 135
      2. Running Times 136
4. Geometry 143
   1. Arrows 143
   2. Coordinate Systems 145
   3. Simple Geometric Calculations 146
      1. Distance and Angle 146
      2. Direction 147
      3. Lines in Two-Dimensional Space 148
      4. Lines and Planes in Three-Dimensional Space 150
      5. Identity, Incidence, Parallelism, and Intersection 152
      6. Projections 154
   4. Geometric Transformations 155
      1. Translations 156
      2. Rotation around the Origin 157
      3. Rigid Motions and the Homogeneous Representation 161
      4. Similarity Transformations 165
      5. Affine Transformations 167
      6. Image of a Distant Object 169
      7. Determinants 172
      8. Coordinate Transformation on Image Arrays 174
5. Change of Basis, DFT, and SVD 181
   1. Change of Coordinate System 181
      1. Affine Coordinate Systems 182
      2. Duality of Transformation and Coordinate Change; Handedness 184 7.1.3 Application: Robotic Arm 185
   2. Color Graphics 192

|  |  |  |
| --- | --- | --- |
| 7.2 | The Formula for Basis Change | . . . . . . . . . . . . . . . . . . . . . . . . 189 |
| 7.3 | Confusion and How to Avoid It | . . . . . . . . . . . . . . . . . . . . . . . . 191 |
| 7.4 | Nongeometric Change of Basis | . . . . . . . . . . . . . . . . . . . . . . . . 192 |

* 1. Discrete Fourier Transform (Optional) 193
     1. Other Applications of the Fourier Transform 198
     2. The Complex Fourier Transform 199
  2. Singular Value Decomposition 200
     1. Matrix Decomposition 202
     2. Proof of Theorem 7.4 (Optional) 203
  3. Further Properties of the SVD 205
     1. Eigenvalues of a Symmetric Matrix 208
  4. Applications of the SVD 209
     1. Condition Number 209
     2. Computing Rank in the Presence of Roundoff 210
     3. Lossy Compression 211
  5. MATLAB 213
     1. The SVD in MATLAB 213
     2. The DFT in MATLAB 213
  6. The Interpretations of Probability Theory 223

|  |  |  |
| --- | --- | --- |
| II | Probability | 221 |
| 8 | Probability | 223 |

* 1. Finite Sample Spaces 224
  2. Basic Combinatorial Formulas 226
     1. Exponential 226
     2. Permutations of *n* Items 226
     3. Permutations of *k* Items out of *n* 227
     4. Combinations of *k* Items out of *n* 227
     5. Partition into Sets 227
     6. Approximation of Central Binomial 228
     7. Examples of Combinatorics 228
  3. The Axioms of Probability Theory 230
  4. Conditional Probability 231
  5. The Likelihood Interpretation 232
  6. Relation between Likelihood and Sample Space Probability 235
  7. Bayes’ Law 236
  8. Independence 238
     1. Independent Evidence 239
     2. Application: Secret Sharing in Cryptography 242
  9. Random Variables 243
  10. Application: Naive Bayes Classification 245

1. Numerical Random Variables 257
   1. Marginal Distribution 259
   2. Expected Value 260
   3. Decision Theory 262
      1. Sequence of Actions: Decision Trees 263
      2. Decision Theory and the Value of Information 266
   4. Variance and Standard Deviation 269
   5. Random Variables over Infinite Sets of Integers 271
   6. Three Important Discrete Distributions 273
      1. The Bernoulli Distribution 273
      2. The Binomial Distribution 273
      3. The Zipf Distribution 274
   7. Continuous Random Variables 279
   8. Two Important Continuous Distributions 287
      1. The Continuous Uniform Distribution 287
      2. The Gaussian Distribution 287
   9. MATLAB 291
2. Markov Models 299
   1. Stationary Probability Distribution 301
      1. Computing the Stationary Distribution 303
   2. PageRank and Link Analysis 304
      1. The Markov Model 306
      2. Pages with No Outlinks 308
      3. Nonuniform Variants 309
   3. Hidden Markov Models and the K-Gram Model 309
      1. The Probabilistic Model 310
      2. Hidden Markov Models 312
      3. The Viterbi Algorithm 314
      4. Part of Speech Tagging 315
      5. The Sparse Data Problem and Smoothing 318
3. Confi nce Intervals 323
   1. The Basic Formula for Confidence Intervals 323
   2. Application: Evaluating a Classifier 325
   3. Bayesian Statistical Inference (Optional) 328
   4. Confidence Intervals in the Frequentist Viewpoint (Optional) 330
   5. Hypothesis Testing and Statistical Significance 333
   6. Statistical Inference and ESP 336
4. Monte Carlo Methods 339
   1. Finding Area 339
   2. Generating Distributions 341
   3. Counting 342
   4. Counting Solutions to a DNF Formula (Optional) 343
   5. Sums, Expected Values, and Integrals 345
   6. Probabilistic Problems 347
   7. Resampling 348
   8. Pseudorandom Numbers 350
   9. Other Probabilistic Algorithms 351
   10. MATLAB 351
5. Information and Entropy 359
   1. Information 359
   2. Entropy 361
   3. Conditional Entropy and Mutual Information 362
   4. Coding 364
      1. Huffman Coding 374
   5. Entropy of Numeric and Continuous Random Variables 375
   6. The Principle of Maximum Entropy 376
      1. The Principle of Maximum Entropy 377
      2. Consequences of the Maximum Entropy Principle 377
   7. Statistical Inference 379
6. Maximum Likelihood Estimation 383
   1. Sampling 384
   2. Uniform Distribution 385
   3. Gaussian Distribution: Known Variance 385
   4. Gaussian Distribution: Unknown Variance 386
   5. Least Squares Estimates 387
      1. Least Squares in MATLAB 390
   6. Principal Component Analysis 391
   7. Applications of Principal Component Analysis 393
      1. Visualization 393
      2. Data Analysis 393
      3. Bounding Box 394
      4. Surface Normals 395
      5. Latent Semantic Analysis 396

References 401

Notation 405

# Preface

Since computer science (CS) first became an academic discipline almost 50 years ago, a central question in defining the computer science curriculum has always been, “How much, and what kind of, college-level mathematics does a computer scientist need to know?” As with all curricular questions, the correct answer, of course, is that everyone should know everything about everything. Indeed, if you raise the question over lunch with computer science professors, you will soon hear the familiar lament that the students these days don’t know Desargues’ theorem and have never heard of a p-adic number. However, in view of the limited time available for a degree program, every additional math course that a CS student takes is essentially one fewer CS course that he/she has time to take. The conventional wisdom of the field has, for the most part, therefore converged on the decision that, beyond first-semester calculus, what CS students really need is a one-semester course in “discrete math,” a pleasant smorgasbord of logic, set theory, graph theory, and combinatorics. More math than that can be left as electives.

I do not wish to depart from that conventional wisdom; I think it is accu- rate to say that a discrete math course indeed provides sufficient mathematical background for a computer scientist working in the “core” areas of the field, such as databases, compilers, operating systems, architecture, and networks. Many computer scientists have had very successful careers knowing little or no more math. Other mathematical issues no doubt arise even in these areas, but peripherally and unsystematically; the computer scientist can learn the math needed as it arises.

However, other areas of computer science, including artificial intelligence, graphics, machine learning, optimization, data mining, computational finance, computational biology, bioinformatics, computer vision, information retrieval, and web search, require a different mathematical background. The importance of these areas within the field of computer science has steadily increased over the last 20 years. These fields and their subfields vary widely, of course, in terms of exactly what areas of math they require and at what depth. Three

xiii

mathematical subjects stand out, however, as particularly important in all or most of these fields: linear algebra, probability, and multivariable calculus.

An undergraduate degree typically involves about 32 semester courses; it may not be unreasonable to suggest or even to require that CS majors take the undergraduate courses in linear algebra, multivariable calculus, and probabil- ity given by the math department. However, the need for many CS students to learn these subjects is particularly difficult to address in a master’s degree pro- gram, which typically requires only 10 or 12 courses. Many students entering a master’s program have weak mathematical backgrounds, but they are eager to get started on CS courses that have a substantial mathematical prerequisite. One can hardly ask them to delay their computer science work until they have completed three or four mathematics courses. Moreover, they cannot get grad- uate credit for undergraduate math courses, and if the math department does offer graduate courses in these areas, the courses are almost certainly too diffi- cult for the CS master’s student.

To fill this gap, I have created a new course entitled, vaguely, “Mathemati- cal Techniques for Computer Science Applications,” for the master’s program at New York University (NYU), which gives an intensive introduction to linear algebra and probability, and is particularly addressed to students with weak mathematical backgrounds. This course has been offered once a year, in the fall semester, every year starting in 2009. I wrote this textbook specifically for use in this course.

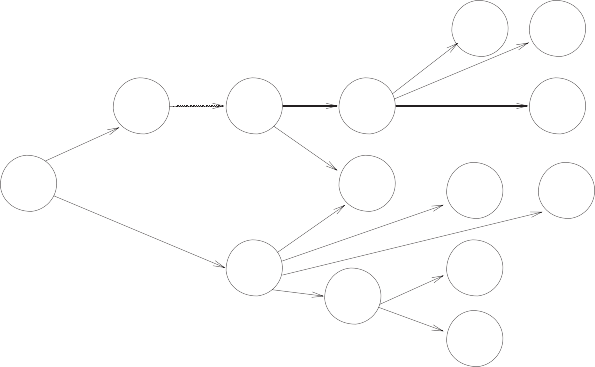
Master’s courses in the computer science department at NYU meet 14 times in a semester, once a week, in two-hour sessions. In my class in 2010, Chapters 1 and 2 were covered together in a single lecture; Chapters 6 and 9 required two lectures each; Chapters 3, 4, 5, 7, 8, 10, 11, 12, and 13 were each covered in a single lecture; and Chapter 14 was omitted. About halfway through the class, I decided it was necessary to add a recitation section for an additional hour a week; I certainly recommend that.

Multivariable calculus remains a gap that I have not found any practical way of closing. Obviously, it would not be possible to squeeze all three top- ics into a single semester, but neither, probably, is it practical to suggest that master’s students take two semesters of “mathematical techniques.”

The course as I teach it involves extensive programming in MATLAB. Corre- spondingly, the book contains an introductory chapter on MATLAB, discussions in each chapter of the relevant MATLAB functions and features, and many MAT- LAB assignments.

Figure 1 illustrates the strong dependencies among chapters. (Weaker de- pendencies also exist, in which one section of a later chapter depends on ma- terial from an earlier chapter; these are not shown.)

There are, of course, plenty of textbooks for probability and a huge number of textbooks for linear algebra, so why write a new one? For one thing, I wanted



5

6

2

3

4

7

1

10

11

12

8

13

9

14

**Figure 1.** Chapter dependencies.

to have both subjects in a single book; the only other book I have seen that does this is *The Mathematics of Digital Images* by Stuart Hoggar (2006). More important, this book is distinguished by being addressed to the computer sci- entist, rather than the mathematician, physical scientist or engineer. This fo- cus affects the *background assumed,* the *choice of topics*, the *examples*, and the *presentation*.

#### Background

The textbook assumes as little mathematical background as I could manage. Most of the book assumes only high-school mathematics. Complex numbers are nowhere used. In the linear algebra section of the book, calculus is entirely avoided, except in one optional short discussion of Jacobians. In the probabil- ity section of the book, this is less feasible, but I have for the most part segre- gated the sections of the text that do require calculus from those that do not. A basic understanding of integrals is an unavoidable prerequisite for under- standing continuous probability densities, and understanding multiple inte- grals is a prerequisite for understanding the joint distribution of multiple con- tinuous variables.

The issue of mathematical proof in a course of this kind is a difficult one. The book includes the proofs of most, though not all, of the theorems that are stated, including a few somewhat lengthy proofs. In Chapter 4 on vector spaces, I have in fact split the chapter into two parts: the first contains the minimal material needed for subsequent chapters with almost no proofs, and the second presents more abstract material and more proofs. My own prac- tice in teaching is that in lectures I present some of the proofs that I feel to be enlightening. I do try to keep in mind, however, that—whatever the mathe- matically trained instructor may imagine—a proof is not the same as an

explanation, even for students who are mathematically inclined, and for stu- dents who are math-averse, a proof bears no relation to an explanation. This textbook includes a number of problems that ask the students to write proofs. My own practice in teaching, however, is that I do not assign problems that require proofs. My experience is that the “proofs” produced by students with weak mathematical backgrounds tend to be random sequences of sentences, only some of which are true. Unless an instructor is willing to invest substan- tial effort into teaching what is and is not a valid proof and how one constructs a valid proof, assigning proofs as homework or exam problems is merely frus- trating and wearisome for both the student and the grader.

I have, however, assumed some familiarity with the basic concepts of com- puter science. I assume throughout that the student is comfortable writing pro- grams, and I discuss issues such as computational complexity, round-off error, and programming language design (in connection with MATLAB).

#### Choice of Topics

In both parts of the course, the topics included are intended to be those ar- eas that are most important to the computer scientist. These topics are some- what different from the usual material in the corresponding introductory math classes, where the intended audience usually comprises math and science stu- dents. I have been guided here both by own impression and by discussions with colleagues.

In the linear algebra section, I have restricted the discussion to finite- dimensional vectors and matrices over the reals. Determinants are mentioned only briefly as a measure of volume change and handedness change in geo- metric transformations. I have almost entirely omitted discussion of eigenval- ues and eigenvectors, both because they seem to be more important in physics and engineering than in CS applications, and because the theory of eigenvalues really cannot be reasonably presented without using complex eigenvalues. In- stead, I have included a discussion of the singular value decomposition, which has more CS applications and involves only real values. I have also included a more extensive discussion of geometric applications and of issues of floating- point computation than is found in many standard linear algebra textbooks.

In the probability section, I have included only a minimal discussion of combinatorics, such as counting combinations and permutations. I have also omitted a number of well-known distributions such as the Poisson distribu- tion. I have, however, included the inverse power-law “Zipf” distribution, which arises often in CS applications but is not often discussed in probability text- books. I have also included discussions of the likelihood interpretation ver- sus the sample space interpretation of probability, and of the basic elements of information theory as well as a very rudimentary introduction to basic tech- niques of statistics.

#### Examples and Presentation

The examples and the programming assignments focus on computer science applications. The applications discussed here are drawn from a wide range of areas of computer science, including computer graphics, computer vision, robotics, natural language processing, web search, machine learning, statisti- cal analysis, game playing, graph theory, scientifi computing, decision theory, coding, cryptography, network analysis, data compression, and signal process- ing. There is, no doubt, a bias toward artificial intelligence, particularly natural language processing, and toward geometric problems, partly because of my own interests, and partly because these areas lend themselves to simple pro- grams that do interesting things. Likewise, the presentation is geared toward problems that arise in programming and computer science.

Homework problems are provided at the end of each chapter. These are divided into three categories. *Exercises* are problems that involve a single cal- culation; some of these can be done by hand, and some require MATLAB. Most exercises are short, but a few are quite demanding, such as Exercise 10.2, which asks students to compute the Markov model and stationary distribution for the game of Monopoly. *Programming Assignments* require the student to write a MATLAB function with parameters. These vary considerably in difficulty; a few are as short as one line of MATLAB, whereas others require some hundreds of lines. I have not included any assignments that would qualify for a semester project. *Problems* include everything else; generally, they are “thought prob- lems,” particularly proofs.

#### Course Website

The website for course materials is

<http://www.cs.nyu.edu/faculty/davise/MathTechniques/> In particular, MATLAB code discussed in this text can be found here.

Errors, queries, and suggestions for improvements should be emailed to [davise@cs.nyu.edu.](mailto:davise@cs.nyu.edu)

#### Acknowledgments

I am very grateful to my colleagues for their encouragement and suggestions in the development of this course and the writing of this book. I am especially grateful to Michael Overton, who was deeply involved in the design of the linear algebra section of the course. He read two drafts of that section of the text and made many suggestions that greatly enriched it. I have also received valuable suggestions and information from Marsha Berger, Zvi Kedem, Gregory Lawler, Dennis Shasha, Alan Siegel, and Olga Sorkine.

I owe a special debt of gratitude to the students in “Mathematical Tech- niques for Computer Science Applications,” who suffered through early drafts of this text as well as many of the assignments, and who gave invaluable feed- back. My thanks to Klaus Peters, Charlotte Henderson, Sarah Chow, and Sandy Rush for all their careful and patient work in preparing this book for publica- tion.

During part of the writing of this book, I had support from the National Science Foundation on grant #IIS-0534809.

My own introduction to linear algebra and to probability theory came from courses that I took with Bruno Harris in spring 1974 and Gian-Carlo Rota in fall 1975. I hope that this book is a credit to their teaching. Finally, as always, thanks to Bianca for everything.

Chapter 1

MATLAB

MATLAB (short for MATrix LABoratory) is a programming language, together with a programming environment, designed to facilitate mathematical calcu- lations and rapid prototyping of mathematical programs. It was created in the late 1970s by Cleve Moler, and it has become very popular in the mathematical, scientific, and engineering communities.

There are many fine handbooks for MATLAB, including those by Driscoll (2009) and Gilat (2008). The online documentation is also good.

A number of freeware clones of MATLAB are available, including Octave and Scilab. These should certainly be adequate for the programs discussed and as- signed in this book. All the MATLAB examples in this book were generated by using MATLAB 7.8.0 (R2009a).

MATLAB creates a collection of windows. These may be “docked”—that is, all placed together within a single window—or “undocked.” The most impor- tant is the Command window. The user types MATLAB commands into the Command window. The MATLAB interpreter executes the command and prints the value in the Command window. To suppress a printout (useful with a large object), the user can type a semicolon ( ) at the end of the command. The user prompt in the Command window is . Comments begin with a percent sign ( ) and continue to the end of the line.

This chapter presents some basic features of the MATLAB language. More advanced features, including operations on vectors and matrices, are discussed in the chapters where the associated math is presented.

### Desk Calculator Operations

The basic arithmetic operations in MATLAB use a standard format. The com- mand window can be used as a convenient interactive desk calculator.

1

*! " # $ $*

*%& '()*

*(*

*\* & )*

*+ +*

*!*

*+ + (,- ( (. ,*

*& '()*

*. --.*

*" $ //*

*& ' ) & ',)*

*'(,*

*\* & )*

*( (' ,*

### Booleans

MATLAB uses 1 for true and 0 for false. Strictly speaking, these are not the same as the integers 1 and 0, but MATLAB automatically casts from one to the other as needed, so the distinction only occasionally makes a difference. (We will see an example in Section 2.5.)

*, ,*

*# , -*

*#*

*.*

*0#*

*.*

*1#*

*2*

*.*

*! 3 !*

*& 4 )0 ! ! 3*

*4*

### Nonstandard Numbers

MATLAB conforms to the IEEE standard for floating-point arithmetic (see Over- ton, 2001), which mandates that a system of floating-point arithmetic support the three nonstandard values (positive infinity), (negative infinity), and (not a number). These values are considered as numbers in MATLAB. The infinite values can generally be used numerically in any context where an infinite value makes sense; some examples are shown next. is used for values that are completely undefined, such as or . Any computa- tion involving gives , and any comparison involving is considered false.

*'.*

*4(*

*4*

*%*

*, 5*

*.'.*

*6 6*

*, 5 6 6*

*.*

*, 6 6*

*.*

*& )*

*6 6*

*& )*

*, .*

### Loops and Conditionals

MATLAB has the usual conditional and looping constructs to build complex statements out of simple ones. Note that

* + - Loops and conditionals end with the key word . Therefore, there is no need for “begin ... end” or “{ ... }” blocks.
    - Atomic statements are separated by line breaks. Once a compound state- ment has been entered, the interpreter continues to read input until the compound statement is ended.
* One can continue an atomic statement past the line break by typing “...” at the end of the line.
* The value of each atomic statement is printed out as it is executed. This is helpful in debugging; we can trace the execution of a statement just by deleting the semicolon at the end. To suppress printout, put a semicolon at the end.

*7,*

*" 8 "*

*(*

*-*

*.*

*,*

*7(*

*9 6*

*#*

*# #*

*"*

*#*

*#*

*(*

*(*

*#*

*,*

*6 7 : ; ! ! ! ! " #*

*# ;*

*9*

*< 5 ,.*

*"*

*+*

*-*

*(*

*-+*

*= "*

*(9*

*(*

*< & 2 )*

*& " & ) )*

*(*

*'*

*"*

*"*

*.*

*,*

*-*

*+*

### Script File

A script file is a plain text file with MATLAB commands. The file has the exten- sion .m. To execute a script file that is in the working directory, just enter the name of the file (omitting the “.m”) in the command window. The working di- rectory can be changed in the “directory” window. An example is the file p35.m:

*(, %(, # " \* ! < " #*

*%*

*+ %+*

*+ + %*

*- % -*

*( - - %(*

*(, ( %(,*

And this is how to execute it in the command window:

*(,*

*+*

*+*

*-*

*-*

*,-*

*-,,(-*

*(*

*+ + - -*

*(,*

*(+(, ( (-*

*> # " " # "*

*+*

*+*

*-*

*"*

*(,*

*(, %(, # " \* ! < " #*

### Functions

The MATLAB code for the function named foo is in the file foo.m. To call foo in the command window, just type “foo(...)"; this both loads and executes the code.

The main differences between functions and scripts is that variables in a function, including the input and output parameters, are local to the function. (It is also possible to define and access global variables.) The function declara- tion has one of two forms:

*5 # 5 & # )*

or

*? 5 # ℄ 5 & # )*

For instance, for the file fib.m:

*A " B # #*

*# & )*

*& .) 9*

*& ) 9*

*# & 4 ) # & 4 )9*

*"*

*"*

And its use in MATLAB is:

*# &,)*

*#*

*A " B # #*

Functions may return multiple values. For example, for quadform.m:

*\* " & # ) < \* " \**

*%C D # .*

*? ℄ \* " & # )*

*\* &#% 4 + )9*

*& 4 # 4 )'& )9*

*& 4 # )'& )9*

*"*

*? \* ℄ \* " & . 4 )*

*4*

*\**

Function files can have subfunctions. These are placed in the file after the main function (the one with the same name as the file). Subfunctions can be called only by functions within this same file.

*Important note:* and are used by MATLAB as predefined symbols for the square root of −1. However, MATLAB does not prevent us from using these as

variables and reassigning them. If we aren’t using complex numbers in our code (none of the assignments in this book require complex numbers), this does not usually lead to trouble but occasionally can cause confusion. The most common case is that if our code uses or without initializing them, we do not get an error, as with other variables.

*< E " / / / /*

*FFF : " " # / /*

*.... ....*

### Variable Scope and Parameter Passing

By default a variable used in a function is local to the function call; that is, if a function calls itself recursively, then the variables in the two instances of the functions are different. Variables declared in the command window are local to the command window.

If a variable is declared global in a number of functions, then it is shared among all the calls of those functions. If it is also declared global in the com- mand window, then it is also shared with the command window environment.

Parameters are *always* passed call-by-value. That is, the formal parameters of a function are local variables; the value of the actual parameter is copied into the formal parameter when the function is called, but not copied back when the function returns.

Here is a toy example. Suppose that we have the two functions and

defined in the file t1.m as follows:

*& ) " ! # " #*

*& )*

*! # ;*

*9*

*9*

*; 9*

*(9*

*& )*

*? ; ℄*

*"*

*& E)*

*! # ;*

*9*

*; 9*

*+9*

*E -9*

*"*

The variable is a global variable shared between and . The vari- able in is a global variable that will be shared with the command

window; it is not shared with . We now execute the following in the command window:

*! #*

*; ..*

*;*

*..*

*..*

*..*

*(..*

*(..*

*+..*

*+..*

*& )*

*+.. (*

*= 7*

*? ; ℄*

*6 ; # ! " # # <*

*" " ! # # < " 9 ! " #*

*6 < # ; " < " <*

*+.. (*

*? ; ℄*

*(.. +.. .. (*

*6 # ! " # # <*

*" " ! # # < " < " < " =*

*! "*

The use of call-by-value in MATLAB means that if function has parameter

and function executes the statement , then the value of in must be copied into the variable in before execution of can begin. If is a large ma- trix, this can involve a significant overhead. Therefore, the programming style that is encouraged in LISP and other similar languages—using large numbers of small functions and using recursion for implementing loops—is ill-suited to MATLAB. Loops should generally be implemented, if possible, by using MATLAB operators or library functions, or else by using iteration.

Copying also occurs at any assignment statement; if is a large matrix, then executing the statement involves copying into . If we are executing a long loop involving large arrays and are concerned with efficiency, we may need to give some thought to reducing the number of unnecessary copies.

The truth is that if our program is largely executing our own MATLAB code (rather than the built-in MATLAB functions, which are implemented efficiently), if it manipulates large quantities of data, and if efficiency is a concern, then we should probably be using some other programming language. Or at least we should write the critical sections of the program in another language; MATLAB has facilities for interfacing with code written in C or C++.

### Problem

Problem 1.1. Can you write a function “swap(A,B)” in MATLAB that swaps the values of its arguments? That is, the function should have the following behav- ior:

*9*

*G , 9*

*; . 9*

*< & G) 9*

*,*

*G*

*G*

*< &G ;)* *9*

*G*

*G*

*.*

*;*

*;*

Explain your answer.

### Programming Assignments

*Note:* We have not yet defined any data structures, so the programming assign- ments for this chapter are necessarily very numerical in flavor.

Assignment 1.1. A pair of *twin primes* is a pair of prime numbers that differ by two. For example, 〈3, 5〉, 〈5, 7〉, 〈11, 13〉, 〈17, 19〉, and 〈29, 31〉 are the first

five pairs of twin primes. It has been conjectured that for large *N* , the number of twin prime pairs less than *N* is approximately the function *f* (*N* ) = 1.3203 · *N* /(log*e N* )2.

Write a MATLAB function *!" #* that counts the number of

twin prime pairs less than and compares the result to the above estimate. That is, your function should return two values:

* *C* , the number of twin-prime pairs less than *N*
* |(*C* − *f* (*N* ))/ *f* (*N* )|, where *f* (*N* ) is the expression defined above. You may use the built-in MATLAB function *#$" %* .

Assignment 1.2. Goldbach’s conjecture asserts that every even number greater than 2 is the sum of two primes. For instance, 4 = 2 + 2, 6 = 3 + 3, 8 = 3 + 5,

132 = 23 + 109, and so on.

Write a function *& ' ( \** which takes as argument an even number

and returns a pair of primes *!+,* such that *!-,*.

Assignment 1.3. The *four squares* theorem states that every positive integer can be written as the sum of four square integers. For example,

26 = 52 + 12 + 02 + 02 ,

56 = 62 + 42 + 22 + 02 ,

71 = 72 + 32 + 32 + 22 .

Write a MATLAB function *. "/0 " #* that returns four integers *1+2+ +3*

such that N = A2 + B2 + C2 + D2.

Assignment 1.4. Write a MATLAB function *" ' 1" 1+2+* that computes the area of a triangle with sides of lengths *1+2+* . For instance,

*" ' 1" 4+5+6* should return 6;

*" ' 1" + +* should return 0.4330 (=

*" ' 1" + +#0"* should return 0.5.

3/4);

*Hint:* Look up “Heron’s formula” in Wikipedia or your favorite search engine.

Assignment 1.5. The recurrence equation *xn*+1 = 2*xn* − 1 exhibits *chaotic* be- havior for *x* between −1 and 1. That is, if you compute the series starting from a

2

particular starting point *y*0 and then recompute the series starting from a start- ing point *z*0 = *y*0 + *ǫ*, which is very close to *y*0, the two series grow apart very

quickly, and soon the values in one are entirely unrelated to the values in the other.

For example, the two series starting at *y*0 = 0.75 and *z*0 = 0.76 are

*y*0 = 0.75, *y*1 = 0.125, *y*2 = −0.9688, *y*3 = 0.8770, *y*4 = 0.5381, *y*5 = −0.4209, *y*6 = −0.6457;

*z*0 = 0.76, *z*1 = 0.1552, *z*2 = −0.9518, *z*3 = 0.8119, *z*4 = 0.3185, *z*5 = −0.7971, *z*6 = 0.2707.

1. Write a MATLAB function *$ " \* 7 +8* that takes as arguments two starting values *y*0 and *z*0 and returns the minimum value of *n* for

which |*yn* − *zn* |> 0.5. For instance *$ " \* 9:6+ 9:;* should

return 6, since in the above series |*y*6 − *z*6|= 0.9164 > 0.5.

1. Experiment with a sequence of pairs that are increasingly close together, such as 〈Y0 = .75, Z0 = 0.751〉, 〈Y0 = 0.75, Z0 = 0.7501〉, 〈Y0 = 0.75, Z0 = 0.75001〉, and so on. Formulate a conjecture as to how the value of *$ "* -

*\* 7 +8* increases with the value of 1/|Y0 − Z0|.

1. Double precision numbers are represented with about 16 digits (51 bits) of precision. Suppose that you start with a value of *y*0 in double pre- cision and compute the series *y*0, *y*1, *y*2 ,.... Given your conjecture in (b), how many terms of the series can you compute before the values become completely unrelated to the true value?

I

Linear Algebra

Chapter 2

Vectors

Linear algebra is the study of vectors, discussed in this chapter, and matrices, discussed in Chapter 3.

### Definition of Vectors

An *n-dimensional vector* is a *n*-tuple of numbers. (A more general definition will be discussed in Section 4.3.1.) The indices 1,... , *n* are the *dimensions* of the vector. The values of the tuple are the *components* of the vector.

In this text, we use angle brackets 〈...〉 to delimit vectors. We use a letter

with an arrow over it, such as *v* , to denote a vector. The *i* th component of vector *v* is notated *v* [*i* ]. (In mathematical writings, it is often notated *vi* , but we use the subscript exclusively for naming different vectors.) For example,

*v* = 〈0, 6, −2.5〉 is a three-dimensional vector. *v* [1] = 0. *v* [3] = −2.5.

A *zero vector*, denoted 0, is a vector whose components are all 0. The four- dimensional zero-vector is 〈0, 0, 0, 0〉. (The dimension associated with the no-

tation 0 is determined by context.)

The *unit vector in the i dimension*, denoted *e i* , is the vector with a 1 in the *i* th coordinate and 0s everywhere else. For instance, in four dimensions,

*e* 2 = 〈0, 1, 0, 0〉. The *n*-dimensional one vector, denoted 1, is the vector with 1s in each dimension; for instance, in four dimensions, 1 = 〈1, 1, 1, 1〉.

The set of all *n*-dimensional vectors is the *n-dimensional Euclidean vector space,* denoted **R***n* .

### Applications of Vectors

Vectors can be used in many different kinds of applications. A few typical ex- amples are discussed here; many more examples are encountered in the course of this book.

17

y

**q**

**p**

**o** x

**Figure 2.1.** Points represented as vectors.

Application 2.1 (Geometric points). In two-dimensional geometry, fix a coor-

dinate system C by specifying an origin, an *x*-axis, and a *y* -axis. A point **p** can then represented by using the vector *p* = 〈**p**[*x*], **p**[*y* ]〉, where **p**[*x*], **p**[*y* ] are the

coordinates of **p** in the *x* and *y* dimensions, respectively. The vector *p* is called the *coordinate vector* for point **p** relative to the coordinate system C . For in-

stance, in Figure 2.1, **p** is associated with the vector 〈2, 1〉 and **q** is associated with the vector 〈0, 3〉.

Likewise, in three-dimensional geometry, point **p** can be associated with the vector 〈**p**[*x*], **p**[*y* ], **p**[*z*]〉 of components in the *x*, *y*, *z* dimensions relative to a

particular coordinate system. In *n*-dimensional geometry, point **p** can be asso- ciated with the vector 〈**p**[1],... , **p**[*n*]〉 where **p**[*i* ] is the component along the *i* th

coordinate axis. Geometric coordinates are discussed at length in Chapter 6.

Application 2.2 (Time series). A vector can be used to represent a sequence of

numeric values of some quantity over time. For instance, the daily closing value of the S&P 500 index over a particular week might be 〈900.1, 905.2, 903.7, 904.8, 905.5〉. A patient’s hourly body temperature in degrees Farenheit over a four- hour period might be 〈103.1, 102.8, 102.0, 100.9〉. Here the dimensions corre-

spond to points in time; the components are the values of the quantity.

Application 2.3 (Almanac information). Numerical information about the 50 states could be recorded in 50-dimensional vectors, where the dimension cor- responds to the states in alphabetical order: Alabama, Alaska, Arizona, Arkan-

sas, etc. We could then have a vector *p* =〈4530000, 650000, 5740000, 2750000, . . .〉 representing the populations of the states; or *a* = 〈52400, 663200, 114000, 2750000,...〉 representing the areas of the states in square miles; and so on.

Application 2.4 (Shopping). The dimensions correspond to a sequence of gro- cery products: gallon of milk, loaf of bread, apple, carrot, and so on. Each store has a *price vector*; for example, *s* [*i* ] is the price of the *i* th product at the Stop and Shop, and *g* [*i* ] is the price of the *i* th product at Gristedes. Each customer has a *shopping list vector*; for example, a vector *a*, where *a* [*i* ] is the number

2.2. Applications of Vectors 19

of item *i* on Amy’s shopping list, and *b*, where *b* [*i* ] is the number of item *i* on Bob’s shopping list.

Note that if the dimensions include all products standardly sold in grocery stores, then each shopping list vector has mostly zeroes. A vector that has mostly zeroes is called a *sparse* vector; these are common in many applications. We may well ask, “How should we represent the price of an item that Stop and Shop does not sell?" These are known as null values. In MATLAB, we can use the special value (not a number) to represent null values. In program- ming languages that do not support , we either have to use a value that is known to be not the price of any actual groceries, such as a negative value or a very large value, or we have a second vector of 1s and 0s that indicates whether or not the object is sold. In either case, applications that use these vectors have to be aware of this representation and take it into account. In fact, in some applications it may be necessary to have more than one kind of null value; for instance, to distinguish items that Stop and Shop does not sell from items for which the price is unknown. We ignore all these issues here, however, and as-

sume that every store has a price for every item.

Application 2.5 (Personal database). Consider a database of personal infor- mation for various kinds of applications, such as medical diagnosis, mortgage evaluation, security screening, and so on. Each dimension is the result of a numerical measurement or test. Each person has a corresponding vector; for example, the value *p* [*i* ] is the *i* th feature of person *p*. In medical applications, we might have age, height, weight, body temperature, various kinds of blood tests, and so on. In financial applications, we might have net worth, income, credit score, and so on. (Note that features that are critical in one application may be illegal to use in another.)

Alternatively, we can have a vector for each feature, indexed by person. For instance, *a* could be the vector of ages, where *a* [*i* ] is the age of person *i* .

Application 2.6 (Document analysis). In a library of documents, one can have one dimension corresponding to each word that appears in any document in the collection and a vector for each document in the collection. Then, for doc-

ument and word *w* , vector denotes a document, and [*w* ] is the number

*d d d*

of times word *w* appears in document *d* .

Alternatively, we can have the dimensions correspond to documents and the vectors correspond to words. That is, for each word *w* in the collection there is a vector *w* , and the value of *w* [*d* ] is the number of times word *w* ap- pears in document *d* . Document vectors and word vectors are mostly sparse in most collections.

This use of document vectors was introduced in information retrieval by Gerard Salton (1971). In actual information retrieval systems, the definition of document vector is a little more complex. It gives different weights to differ- ent words, depending on their frequency; more common words are considered

less important. Specifically, let *N* be the number of documents in a given col- lection. For any word *w* , let *mw* be the number of documents in the collection

that contain *w* , and let *iw* = log(*N* /*mw* ), called the *inverse document frequency*.

Note that the fewer the documents that contain *w* , the larger *iw* ; for common words such as “the,” which are found in all documents, *iw* = 0. For any docu-

ment *d* , let *tw*,*d* be the number of occurrences of word *w* in document *d* . Then

we can define the document vector indexed by words, as [*w* ] = *tw*,*d iw* .

*d* , *d*

#### 2.2.1 General Comments about Applications

Broadly speaking, applications of vectors come in three categories: geometric interpretations (Application 2.1), sequential interpretations (Application 2.2), and numeric functions of two entities or features (Applications 2.3–2.6). We will see some other categories, but these three categories include many of the applications of vectors. A couple of general points may be noted.

First, the geometric interpretations depend on an arbitrary choice of coor- dinate system: how a *point* is modeled as a *vector* depends the choice of the ori- gin and *x*- and *y* -axes. An important question, therefore, concerns how differ- ent choices of coordinate system affect the association of vectors with points. Other categories generally have a natural coordinate system, and the only simi- lar arbitrary choice is the choice of unit (e.g., feet rather than meters, US dollars rather than euros, liters instead of gallons). Even in those cases, as we see at length in Chapter 7, it is often important to think about alternative, less natural coordinate systems.

Second, in the first and third categories, the association of dimensions with numerical indices is essentially arbitrary. There is no particular reason that the states should be listed in alphabetical order; they could just as well be listed in backward alphabetical order (Wyoming, West Virginia, Washington, ...); in or- der of admission to the union (Delaware, Pennsylvania, New Jersey, ...); or any other order. Any question for which the answer depends on having a particu- lar order, such as “Are there three consecutive states with populations greater than 7 million?” is almost certain to be meaningless. Likewise, in geometric

applications, there is no particular reason that the *x*, *y* , and *z* directions are enumerated in that order; the order *z*, *x*, *y* is just as good.1 For time series and other sequential vectors, by contrast, the numeric value of the index is signifi- cant; it represents the time of the measurement.

In a programming language such as Ada, which supports enumerated types and arrays that are indexed on enumerated types, the programmer can declare explicitly that, for instance, the vectors in Application 2.3 are indexed on states or that the person vectors in Application 5 are indexed on a specified list of

1There is, however, a significant distinction between left- and right-handed coordinate systems, which is related to the order of the coordinates; see Section 7.1.2.

features. However, MATLAB does not support the type structure needed for this. For the vectors of Application 2.4 indexed by product type, or the vectors of Application 2.6 indexed by document or words, where the class of indices is not predetermined, we would want arrays indexed on *open* types, such as words; I don’t know of any programming language that supports this.

This distinction is not reflected in standard mathematical notations; math- ematicians, by and large, are not interested in issues of types in the program- ming language sense.

### Basic Operations on Vectors

There are two basic2 operations on *n*-dimensional vectors:

* + - *Multiplying a n-dimensional vector by a number.* This is done component- by-component and the result is a new *n*-dimensional vector. That is, if

*v* [*i* ]. So, for example, 4·〈3, 1, 10〉= 〈4·3, 4·1, 4·10〉=

〈12, 4, 40〉.

In linear algebra, a number is often called a *scalar*, so multiplication by a number is called *scalar multiplication.* It is conventional to write *a* · *v*

*v* is written *v* . As usual in mathematical notation, we may omit the multiplicative dot symbol and just write *a v* when that is not confusing.

−

* + - *Adding two vectors of the same dimension.* This is done component-by- component; that is, if *w* = *v* + *u* then *w* [*i* ] = *v* [*i* ] + *u* [*i* ]. So, for example,

〈3, 1, 10〉+ 〈1, 4, −2〉= 〈3 + 1, 1 + 4, 10 − 2〉= 〈4, 5, 8〉.

The difference of two vectors *v* − *u* is defined as *v* + (−*u*). Two vectors of different dimensions may not be added or subtracted.

#### Algebraic Properties of the Operations

The following basic properties of these operators are important. They are all easily proven from the definitions. In all the following properties, *v* , *u*, *w* , and

2“Says who?” you may well ask. What’s so “not basic” about operations such as finding the length of a vector, adding a scalar to a vector, finding the largest element of a vector, or sorting a vector? Certainly these are important, and, of course, they are built into MATLAB. There are, however, two related reasons for emphasizing these two operations. First, as we shall see, linear algebra is essen- tially about linear transformations; the two operations of vector addition and scalar multiplication

are linear transformations, whereas operations such as sorting are not. Second, in the more gen- eral defi ition of vector discussed in Section 4.3.1, these other operations may not be pertinent. For instance, the components of a vector may be entities that are not ordered in terms of “greater” and “less.” In that case, sorting is not meaningful.

0 are *n*-dimensional vectors, and *a* and *b* are numbers:

*v* + *u* = *u* + *v* (commutative) ( *v* + *u* ) + *w* = *v* + (*u* + *w* ) (associative)

*a* · ( *v* + *u* ) = (*a* · *v* ) + (*a* · *u* ) (distributive) (*a* + *b*) · *v* = (*a* · *v* ) + (*b* · *v* ) (distributive) *a* · (*b* · *v* ) = (*ab*) · *v*

0 · *v* = 0

*v* + 0 = *v* (0 is the additive identity)

*v* − *v* = 0 (−*v* is the additive inverse of *v* )

#### Applications of Basic Operations

Geometric. Fix a coordinate system in space with origin **o**. Suppose that *p* and *q* are the coordinate vectors for points **p** and **q**, respectively. Let *a* be a number. Draw arrows from **o** to **p** and from **o** to **q**.

Now make the arrow from **o** to **p** *a* times as long, keeping the tail of the arrow at **o** and the direction the same. Then the coordinate vector of the head of the arrow is *ap*.

Then copy the arrow from **o** to **q**, keeping the direction and length the same, but put its tail at **p**. Then the coordinate vector of the head of the arrow is *p* + *q*. Figure 2.2 illustrates this geometric application with *p* = 〈2, 1〉, *q* = 〈0, 3〉,

*a* = 1.5.

Nongeometric. The nongeometric interpretations of vector addition and scalar multiplication are mostly obvious. We multiply by a scalar when we need to multiply every component by that scalar; we add two vectors when we need to add every component. For example, consider Application 2.4 again, where the dimensions correspond to different kinds of groceries. If *a* is Amy’s shop-

ping list, and is Bob’s, and they decide to shop together, then *a* + is their

*b b*

joint shopping list. If *s* is the price list for Stop and Shop and *g* is the price

y **p+q**

**q**

*a* **p**

**p**

**o** x

**Figure 2.2.** Points represented as vectors.

for Gristede’s then *s* − *g* is the amount they save at Gristede’s for each item. (If ( *s* − *g* )[*i* ] is negative, then item *i* is more expensive at Gristede’s.) If Shop and

Shop announces a 20% off sale on all items in the store, then the new price vec- tor is 0.8 · *s*. To convert a price vector from dollars to euros, we multiply by the

current exchange rate. Other applications work similarly.

### Dot Product

The *dot product* of two *n*-dimensional vectors is computed by multiplying cor- responding components, and then adding all these products. That is,

*v* • *w* = *v* [1] · *w* [1] + ... + *v* [*n*] · *w* [*n*].

For example, 〈3, 1, 10〉• 〈−2, 0, 4〉= (3 · −2) + (1 · 0) + (10 · 4) = −6 + 0 + 40 = 34.

The dot product is also known as the *scalar product* or the *inner product*. We indicate it by a large solid dot •, as above; this is not standard mathematical

notation, but a long sequence of small dots can be confusing.

The dot product is not defined for vectors of different dimensions.

#### Algebraic Properties of the Dot Product

The following basic properties of the dot product are important. They are all easily proven from the above definition. In the following properties, *v* , *u* and *w* are *n*-dimensional vectors, and *a* is a number:

*v* • *u* = *u* • *v* (commutative) (*u* + *v* ) • *w* = (*u* • *w* ) + ( *v* • *w* ) (distributive) (*a* · *u* ) • *v* = *a* · (*u* • *v* ) = *u* • (*a* · *v* )

Another obvious but important property is that the dot product of vector *v* with the *i* th unit vector *e i* is equal to *i* th coordinate *v* [*i* ]. A generalization of this is presented in Section 4.1.3.

#### Application of the Dot Product: Weighted Sum

The simplest application of the dot product is to compute weighted sums. A few examples follow.

In the grocery shopping application (Application 2.4), *s* is the vector of prices at Stop and Shop, and *a* is Amy’s shopping list. If Amy goes shopping at Stop

and Shop, she will pay *a* [*i* ] · *s* [*i* ] for item *i* , and therefore her total bill will be

*i a* [*i* ] · *s* [*i* ] = *a* • *s*.

The sum of the elements of an *n*-dimensional vector *v* is 1 · *v* . The average value of a *n*-dimensional vector *v* is the sum divided by *n*; thus, (1/*n*) 1 • *v* .

·

In the almanac application (Application 2.3), *p* is the population of each state. Let *q* be the average income in each state. Then *p* [*i* ] · *q* [*i* ] is the total income of all people in state *i* , so *p* • *q* is the total income of everyone in the

country. The average income across the country is the total income divided by the population, (*p* • *q*)/(*p* • 1). Note that we are not allowed to “cancel out” the

*p* in the numerator and denominator; dot products don’t work that way.

Application 2.7 (Linear classifi rs). An important category of applications in- volves the *classification* problem. Suppose we are given a description of an entity in terms of a vector *v* of some kind, and we want to know whether the entity belongs to some specific category. As examples, a bank has collected in- formation about a loan applicant and wants to decide whether the applicant is a reasonable risk; a doctor has a collection of information (symptoms, medical history, test results, and so forth) about a patient and wants to know whether the patient suffers from a particular disease; or a spam filter has the text of an email message, encoded as a document vector (see Application 2.6) and wants to know whether the message is spam.

A *classifier* for the category is an algorithm that takes the vector of features as input and tries to calculate whether the entity is an instance of the category. One of the simplest and most widely used kinds of classifier are *linear classi- fi rs*, which consist of a vector of weights *w* and a threshhold *t* . If the weighted

sum of the feature vector, *w* • *v* > *t* , then the classifier predicts that the entity

is in the category; otherwise, the classifier predicts that it is not.

Most machine learning programs work by constructing a classifier for a cat- egory, based on a corpus of examples. One simple algorithm to do this, the *Naive Bayes* algorithm, is discussed in Section 8.11.

#### Geometric Properties of the Dot Product

Geometrical analysis yields further interesting properties of the dot product operation that can then be used in nongeometric applications. This takes a little work.

Consider a fixed two-dimensional coordinate system with origin **o**, an *x*- axis, and a *y* -axis. Let **p** and **q** be points and let *p* and *q* be the associated coordinate vectors.

First, note that, by the Pythagorean theorem, the distance from **o** to **p** (in the units of the coordinate system), which we denote as *d* (**o**, **p**) is

*p* [*x*]2 + *p* [*y* ]2.

But *p* [*x*]2 + *p* [*y* ]2 = *p* • *p* . So *d* (**o**, **p**) = *p* • *p*. This quantity, *p* • *p*, is called the *length* of vector *p* and is denoted |*p*|. Similarly,

*d* (**p**, **q**) = *q* [*x*] − *p* [*x*])2 + ( *q* [*y* ] − *p* [*y* ])2.

(

So

Therefore,

*d* (**p**, **q**)2 = ( *q* [*x*] − *p* [*x*])2 + ( *q* [*y* ] − *p* [*y* ])2

= ( *q* − *p* ) • ( *q* − *p*)

= *q* • *q* − 2*p* • *q* + *p* • *p*

= *d* (**o**, **q**)2 − 2*p* • *q* + *d* (**o**, **p**)2.

*d* (**o**, **p**)2 + *d* (**o**, **q**)2 − *d* (**p**, **q**)2

*p* • *q* =

. (2.1)

2

This proof is for two-dimensional points, but in fact the same proof works in Euclidean space of any dimension.

Proceeding from the formula for *d* (**p**, **q**), a number of important conclu- sions can be deduced. First, by the triangle inequality,

|*d* (**o**, **p**) − *d* (**o**, **q**)|≤ *d* (**p**, **q**) ≤ *d* (**o**, **p**) + *d* (**o**, **q**).

Since all these terms are nonnegative, we may square all parts of the inequality, giving

*d* (**o**, **p**)2 − 2*d* (**o**, **p**)*d* (**o**, **q**) + *d* (**o**, **q**)2 ≤ *d* (**p**, **q**)2

≤ *d* (**o**, **p**)2 + 2*d* (**o**, **p**)*d* (**o**, **q**) + *d* (**o**, **q**)2.

Therefore,

−2*d* (**o**, **p**)*d* (**o**, **q**) ≤ *d* (**o**, **p**)2 + *d* (**o**, **q**)2 − *d* (**p**, **q**)2 ≤ 2*d* (**o**, **p**)*d* (**o**, **q**).

But, by Equation (2.1), the middle term here is just 2*p* • *q*. Substituting and dividing through by 2 gives

−*d* (**o**, **p**)*d* (**o**, **q**) ≤ *p* • *q* ≤ *d* (**o**, **p**)*d* (**o**, **q**).

Using the facts that *d* (**o**, **p**) =|

*q*|, and dividing through gives

*p*| and *d* (**o**, **q**) =|

*p* • *q*

−1 ≤

|·|

|*p q* |

≤ 1. (2.2)

Equation (2.2) is known as the *Cauchy-Schwarz inequality*.3

3This may seem a little suspicious. How did we derive this nontrivial algebraic inequality from this simple geometric argument? If it does seem suspicious, then you have good instincts; what we’ve pushed under the rug here is proving that the triangle inequality holds for Euclidean distance in *n*-dimensions.

**w**

**q**

**p**

**Figure 2.3.** Geometric interpretation of the dot product.

Next, consider the case where **p**, **o**, **q** form a right angle; that is, the arrow

from **o** to **q** is at right angles to the arrow from **o** to **p**. Then, by the Pythagorean theorem, *d* (**p**, **q**)2 = *d* (**o**, **p**)2 + *d* (**o**, **q**)2. Using Equation (2.1), it follows that *p* • *q* = 0. In this case, we say that *p* and *q* are *orthogonal*.

Equation (2.2) can be made more precise. Let *θ* be the angle between the arrow from **o** to **q** and the arrow from **o** to **p**. Recalling the law of cosines from trigonometry class in the distant past and applying it to the triangle **opq** yields

*d* (**p**, **q**)2 = *d* (**o**, **p**)2 + *d* (**o**, **q**)2 − 2*d* (**o**, **p**)*d* (**o**, **q**)cos(*θ*),

so

*d* (**o**, **p**)2 + *d* (**o**, **q**)2 − *d* (**p**, **q**)2

*p* • *q*

cos(*θ*) =

2*d* (**o**, **p**)*d* (**o**, **q**) = *p*

*q* .

| |·| |

Thus, the angle *θ* between the two vectors *p* and *q* can be calculated as

*q*|).

Another way to write this formula is by using *unit vectors*. A unit vector is a vector of length 1. For any vector *v* = 0, the unit vector in the same direction as

*v* |; this is often written *v*ˆ. So we can rewrite the formula for cos(*θ*) as

*p*

cos(*θ*) = •

*p*

| |

*q*

*q*|

|

= *p*ˆ • *q*ˆ. (2.3)

We have used this geometric argumentation to derive important proper- ties of the dot product, but we have not said what, in general, the dot product actually *means*, geometrically. Sadly, there is no very intuitive or interesting explanation. (Equations (2.1) and (2.3) are geometrical definitions but they do not give a clear intuitive sense for the dot product.) The best, perhaps, is this:

Given *p* and *q*, let *w* be the vector that is perpendicular to *q*, in the same plane as *p* and *q*, and the same length as *q*. Then |*p* • *q*| is the area of the parallel-

ogram with sides *p* and *w* (Figure 2.3). This is not a very helpful definition, however; first, because the distributivity rule is not geometrically obvious; and second, because the sign of the dot product is not easy to define in this way.

#### Metacomment: How to Read Formula Manipulations

The derivation of the Cauchy-Schwarz inequality in Section 2.4.3 is a sterling example of what people dislike about math and math books: “〈Dull long formula〉 so 〈series of even duller, even longer formulas〉 so 〈pretty dull but short formula〉. How wonderful!”

I can’t eliminate formula manipulation from this book because that is an important part of how math works. I can’t always replace formula manipula- tion proofs with “insightful” proofs because a lot of the time there aren’t any in- sightful proofs. Moreover, proofs that avoid formula manipulation are not nec- essarily more insightful. Sometimes they rely on a trick that is actually largely beside the point. Sometimes they don’t generalize as well.

The one thing that I can do is to tell you how I read this kind of thing, and I suggest doing likewise. Trying to read a proof the way one would read the newspaper, or even the way one would read the earlier sections of this chapter, is (for me) a complete waste of time. (Of course, people are all different, and there may be someone somewhere who gets something out of that; but no one I’ve ever met.) I find that there are three possible approaches to reading a proof:

1. The best is to glance through it, get kind of an idea how it works, close the book, and try to write out the proof without looking at the book. Unless the proof is quite short, you should write it out rather than trying to think it through in your head. Once you’ve done this, you really understand how the proof works and what it means.
2. If you’re dealing with a long proof or if you’re short on time, approach (1) will probably be too difficult. In that case, the second best approach is to go through the proof *slowly.* Check the simple steps of the proof by eye, making sure that you keep track of where all the terms go and that you understand why the author brought in another formula from before at this point. If you can’t follow the step by eye, then copy it down and work it out on paper. The limitation of this approach, as opposed to (1), is that you can end up understanding each of the steps but not the overall progress of the proof; you hear the notes but not the overall tune.

In either (1) or (2), if the book provides a picture or example, then you should carefully look at it and make sure you understand the connection be- tween the picture or example and the proof. If the book doesn’t provide one, then you should draw a picture if the proof is geometric, or you should work through some numeric examples if the proof is numeric. Manipulating the numbers often gives a clearer sense of what is going on than looking at the algebraic symbols.

Approaches (1) and (2) are both difficult and time-consuming, but there is no way around that if you want to learn the math—“There is no royal road to geometry” (Euclid). That leaves

1. Skip the manipulation and just learn the conclusion.

Unless you are responsible for the manipulation—you are reviewing the proof for a scientific journal, or your instructor has told you that you are required to know the proof for the exam—this is probably safe to do. I regularly read books and articles with proofs in them; I work through perhaps 5% of them, probably less. Your time is valuable, and there are other things to do in life. But even though you can do this sometimes, you can’t always take approach (3) if you want to learn how to do math.

#### Application of the Dot Product: Similarity of Two Vectors

The geometric discussion of the previous section can be applied to nongeo- metric vectors to give two measures of how close or similar two vectors are.

The obvious measure of the similarity between two vectors *p* and *q* is just the distance between them,

*q* − *p* |= *q* − *p*) • ( *q* − *p* ).

| (

For instance, if we have a set of pricing vectors from different stores, as in Ap- plication 2.4, and we want to determine which stores have most similar pricing policies (perhaps with a view to detecting price fixing), this might be a reason- able measure to use. Note that if we are just comparing greater and lesser dis-

tances, there is no need to extract the square root; we can just use the distance squared, which is equal to ( *q* − *p* ) • ( *q* − *p*).

However, it is often more pertinent to consider a *scale invariant* measure of similarity of two vectors, which concerns the difference in the direction of the two vectors independent of their magnitude. In that case, a natural measure is the angle between the two vectors, which can be calculated in terms of the

*q*|). Again, if we are just comparing greater and smaller

angles, there is no need to calculate the inverse cosine function to get the actual angle; it suffices just to calculate the cosine of the angle by using this formula.

For example, suppose we want to evaluate the similarity of two documents in terms of the words they use—for example, to suggest “related documents” in a search engine. We can then use the document model described in Applica- tion 2.6. If we just use the distance between the document vectors, then long documents will tend to be close to other long documents and far from short documents, which is not at all what we want. Rather, we want to base the simi- larity judgment on the relative frequency of words in the documents. This can be done by using the angle cosine between the document vectors.

Application 2.8 (Recommender systems). A company wants to recommend specific products to specifi users. The company has available a large database of purchases by customers.

One way to determine the recommendations, in principle,4 is the following: For each customer in the data base, we construct a vector indexed by product. That is, corresponding to customer *c*, we construct an *n*-dimensional vector *c*, where *n* is the number of different products and *c* [*i* ] is equal to the quantity of product *i* that customer *c* has bought. Of course, *c* [*i* ] is 0 for most *i* ; thus, this is a sparse vector. Now, in making recommendations for customer *d* , we find the *k* customer vectors that are most similar to *d* in terms of the above measure and recommend products that have been popular among these customers.

There is an alternative approach: for each product *p*, we construct an *m*- dimensional vector *p*, where *m* is the number of different customers and *p* [ *j* ] is the quantity of product *p* that customer *j* has bought. To find customers who might like a specific product *q*, we look for the *k* product vectors most similar to *q* and recommend *q* to customers who have bought some of these products.

Application 2.9 (Pattern comparison). Suppose we want to compare the pat- tern of the stock market crash of fall 1929 with the stock market crash of fall 2008. Since the Dow Jones average was around 700 at the start of September 1929 and around 12,000 at the start of September 2008, there is no point in us- ing the distance between the corresponding vectors. The angle cosine between the two vectors might be a more reasonable measure.

Application 2.10 (Statistical correlation).5 A course instructor wishes to de- termine how well the results on his final exam correspond to the results on the

problem sets. Suppose that there are six students in the class; the vector of average problem set scores was *p* = 〈9.1, 6.2, 7.2, 9.9, 8.3, 8.6〉 and the vector of final exam scores was *x* = 〈85, 73, 68, 95, 77, 100〉.

The first step is to shift each vector so that they are both zero-centered; oth- erwise, the comparison will mostly just reflect the fact that students as a whole did fairly well on both. We are interested in the performances of individual stu- dents relative to the average, so we subtract the average value from each vector.

The average problem set score *p*¯ = 8.22 and the average exam score *e*¯ = 83, so the shifted vectors are *p* ′ = *p* − *p*¯ 1 = 〈0.8833, −2.0167, −1.0167, 1.6833, 0.0833,

·

0.3833 and *x x e*¯ 1 2, 15, 12, 6, 17 . The correlation is the angle cosine between these, *p* ′ • *x* ′/|*p* ′||*x* ′|= 0.7536.

〉 ′ = − · = 〈 −10, − − 〉

In general, the correlation between two vectors *p* and *q* is defined by the algorithm

*p* ′ ← *p* − mean(*p*) 1;

·

*q* ′ ← *q* − mean( *q*) 1;

·

*q* ′|.

4Getting this method to work efficiently for a huge database is not easy.

5This example is adapted from Steven Leon, *Linear Algebra with Applications*.

#### Dot Product and Linear Transformations

The fundamental significance of a dot product is that it is a linear transforma- tion of vectors. This is stated in Theorem 2.2, which, though easily proven, is very profound. First we need a definition.

Defi 2.1. Let *f* ( *v*) be a function from *n*-dimensional vectors to num- bers. Function *f* is a *linear transformation* if it satisfies the following two prop- erties:

* + - * For any vector *v* and scalar *a*, *f* (*a* · *v* ) = *a* · *f* ( *v*).
      * For any vectors *v* , *u*, *f* ( *v* + *u*) = *f* ( *v* ) + *f* (*u*).

The algebraic properties of the dot product listed in Section 2.4.1 show the following: For any vector *w* , the function *f* ( *v* ) = *w* • *v* is a linear transforma-

tion. Theorem 2.2 establishes the converse: Any linear transformation *f* ( *v*) corresponds to the dot product with a weight vector *w* .

Theorem 2.2. *Let f be a linear transformation from n-dimensional vectors to numbers. Then there exists a unique vector w such that, for all v, f* ( *v*) = *w* • *v.*

For example, imagine that Stop and Shop does not post the price of indi- vidual items; the checkout clerk just tells you the price of an entire basket. Let *f* be the function that maps a given shopping basket to the total price of that basket. The function *f* is linear, assuming that Stop and Shop has no “two for the price of one" offers, “maximum of one per customer” restrictions, or other stipulations. Specifically,

1. If you multiply the number of each kind of item in a basket by *a*, then

*b*

the total price of the basket increases by a factor of *a*.

1. If you combine two baskets *b* and *c* into a single basket *b* + *c*, then the price of the combined basket is just the sum of the prices of the individual baskets.

Then Theorem 2.2 states that there is a price vector *s* such that *f* (*b*) = *s* • *b* for any basket *b*. How do we find the vector *s*? Couldn’t be simpler. Put a single unit of a single item *i* into a basket, and ask Stop and Shop the price of that basket. We will call that the “price of item *i* ,” and we set *s* [*i* ] to be that price. By property (1), a basket that contains *b* [*i* ] units of item *i* and nothing else costs

*s* [*i* ]. By property (2), a basket that contains [1] units of item 1, [2] units

*b* [*i* ] · *b b*

of item 2, ..., and *b* [*n*] units of item *n* costs *b* [1] · *s* [1] + *b* [2] · *s* [2] + ... + *b* [*n*] ·

*s* [*n*] = *s* • .

*b*

The general proof of Theorem 2.2 is exactly the same as the argument we have given above for Stop and Shop prices.6

6The theorem does not hold, however, for infinite-dimensional vector spaces.

### Vectors in MATLAB: Basic Operations

In this section, we illustrate the basic vector operations and functions by exam- ple. For the most part, these examples are self-explanatory; in a few cases, we provide a brief explanation.

Strictly speaking, MATLAB does not have a separate category of vectors as such; vectors are either 1 × *n* matrices (row vectors) or *n* × 1 matrices (column

vectors). Most functions that take a vector argument, such as , *"* , and

*$'* , do the same thing with either kind of vector. Functions that return a vec- tor value, such as *# <* and *" $ "* , have to choose, of course; these mostly return row vectors, presumably because they print out more compactly.

#### Creating a Vector and Indexing

*? ( , ℄*

*( ,*

*&+)*

*,*

*&+)*

*(*

*& ) .*

*( . . .*

*" # " ! H #*

*" "*

*& .)*

*FFF I " & .)9 " # " # & ) ,*

*3 # " !*

*& 4 ) (*

*FFF I " & 4 )9 " # ! !*

*" "*

#### Creating a Vector with Elements in Arithmetic Sequence

*7 .*

*( + , - .*