Introduction

I.1 FORWARD AND INVERSE THEORIES

Inverse theory is an organized set of mathematical techniques for reducing data to obtain *knowledge* about the physical world on the basis of inferences drawn from observations. Inverse theory, as we shall consider it in this book, is limited to observations and questions that can be represented numerically. The observations of the world will consist of a tabulation of measurements, or *data*. The questions we want to answer will be stated in terms of the numerical values (and statistics) of specific (but not necessarily directly measurable) properties of the world. These properties will be called model *parameters* for reasons that will become apparent. We shall assume that there is some specific method (usually a mathematical theory or model) for relating the model parameters to the data.

The question, what causes the motion of the planets? for example, is not one to which inverse theory can be applied. Even though it is perfectly scientific and historically important, its answer is not numerical in nature. On the other hand, inverse theory can be applied to the question, assuming that Newtonian mechanics applies, determine the number and orbits of the planets on the basis of the observed orbit of Halley's comet. The number of planets and their orbital ephemerides are numerical in nature. Another important difference between these two problems is that the first asks us to determine the reason for the orbital motions, and the second presupposes the reason and asks us only to determine certain details. Inverse theory rarely supplies the kind of insight demanded by the first question; it always requires that the physical model or theory be specified beforehand.

The term *inverse theory* is used in contrast to *forward theory*, which is defined as the process of predicting the results of measurements (predicting data) on the basis of some general principle or model and a set of specific conditions relevant to the problem at hand. Inverse theory, roughly speaking, addresses the reverse problem: starting with data and a general principle, theory, or quantitative model, it determines estimates of the model parameters. In the above example, predicting the orbit of Halley's comet from the presumably well-known orbital ephemerides of the planets is a problem for forward theory.

Another comparison of forward and inverse problems is provided by the phenomenon of temperature variation as a function of depth beneath the earth's surface. Let us assume that the temperature increases linearly with depth in the earth, that is, temperature T is related to depth z by the rule $T(z) = az + b$, where a and b are numerical constants that we will refer to as *model parameters*. If one knows that $a = 0.1$ and $b = 25$, then one can solve the forward problem simply by evaluating the formula for any desired depth. The inverse problem would be to determine a and b on the basis of a suite of temperature measurements made at different depths in, say, a bore hole. One may recognize that this is the problem of fitting a straight line to data, which is a substantially harder problem than the forward problem of evaluating a first-degree polynomial. This brings out a property of most inverse problems: that they are substantially harder to solve than their corresponding forward problems.

Forward problem:

estimates of model parameters → quantitative model → predictions of data

Inverse problem:

observations of data → quantitative model → estimates of model parameters

Note that the role of inverse theory is to provide information about unknown numerical parameters that go into the model, not to provide the model itself. Nevertheless, inverse theory can often provide a means for assessing the correctness of a given model or of discriminating between several possible models.

The model parameters one encounters in inverse theory vary from discrete numerical quantities to continuous functions of one or more variables. The intercept and slope of the straight line mentioned above are examples of discrete parameters. Temperature, which varies continuously with position, is an example of a continuous function. This book deals mainly with discrete inverse theory, in which the model parameters are represented as a set of a finite number of numerical values. This limitation does not, in practice, exclude the study of continuous functions, since they can usually be adequately approximated by a finite number of discrete parameters. Temperature, for example, might be represented by its value at a finite number of closely spaced points or by a set of splines with a finite number of coefficients. This approach does, however, limit the rigor with which continuous functions can be studied. Parameterizations of continuous functions are always both approximate and, to some degree, arbitrary properties, which cast a certain amount of imprecision into the theory. Nevertheless, discrete inverse theory is a good starting place for the study of inverse theory, in general, since it relies mainly on the theory of vectors and matrices rather than on the somewhat more complicated theory of continuous functions and operators. Furthermore, careful application of discrete inverse theory can often yield considerable insight, even when applied to problems involving continuous parameters.

Although the main purpose of inverse theory is to provide estimates of model parameters, the theory has a considerably larger scope. Even in cases in which the model parameters are the only desired results, there is a plethora of related information that can be extracted to help determine the "goodness" of the solution to the inverse problem. The actual values of the model parameters are indeed irrelevant in cases when we are mainly interested in using inverse theory as a tool in experimental design or in summarizing the data. Some of the questions inverse theory can help answer are the following:

(a) What are the underlying similarities among inverse problems?
(b) How are estimates of model parameters made?
(c) How much of the error in the measurements shows up as error in the estimates of the model parameters?
(d) Given a particular experimental design, can a certain set of model parameters really be determined?

These questions emphasize that there are many different kinds of answers to inverse problems and many different criteria by which the goodness of those answers can be judged. Much of the subject of inverse theory is concerned with recognizing when certain criteria are more applicable than others, as well as detecting and avoiding (if possible) the various pitfalls that can arise.

Inverse problems arise in many branches of the physical sciences. An incomplete list might include such entries as

(a) medical and seismic tomography,
(b) image enhancement,
(c) curve fitting,
(d) earthquake location,
(e) oceanographic and meteorological data assimilation,
(f) factor analysis,
(g) determination of earth structure from geophysical data,
(h) satellite navigation,
(i) mapping of celestial radio sources with interferometry, and
(j) analysis of molecular structure by X-ray diffraction.

Inverse theory was developed by scientists and mathematicians having various backgrounds and goals. Thus, although the resulting versions of the theory possess strong and fundamental similarities, they have tended to look, superficially, very different. One of the goals of this book is to present the various aspects of discrete inverse theory in such a way that both the individual viewpoints and the "big picture" can be clearly understood.

There are perhaps three major viewpoints from which inverse theory can be approached. The first and oldest sprang from probability theory—a natural starting place for such "noisy" quantities as observations of the real world. In this version of inverse theory, the data and model parameters are treated as random variables, and a great deal of emphasis is placed on determining the probability density functions that they follow. This viewpoint leads very naturally to the analysis of error and to tests of the significance of answers.

The second viewpoint developed from that part of the physical sciences that retains a deterministic stance and avoids the explicit use of probability theory. This approach has tended to deal only with estimates of model parameters (and perhaps with their error bars) rather than with probability density functions per se. Yet what one means by an estimate is often nothing more than the expected value of a probability density function; the difference is only one of emphasis.

The third viewpoint arose from a consideration of model parameters that are inherently continuous functions. Whereas the other two viewpoints handled this problem by approximating continuous functions with a finite number of discrete parameters, the third developed methods for handling continuous functions explicitly. Although continuous inverse theory is not the primary focus of this book, many of the concepts originally developed for it have application to discrete inverse theory, especially when it is used with discretized continuous functions.

I.2 *MATLAB* AS A TOOL FOR LEARNING INVERSE THEORY

The practice of inverse theory requires computer-based computation. A person can learn many of the *concepts* of inverse theory by working through short pencil-and-paper examples and by examining precomputed figures and graphs. But he or she cannot become proficient in the *practice* of inverse theory that way because it requires skills that can only be obtained through the experience of working with large data sets. Three goals are paramount: to develop the judgment needed to select the best solution method among many alternatives; to build confidence that the solution can be obtained even though it requires many steps; and to strengthen the critical faculties needed to assess the quality of the results. This book devotes considerable space to case studies and homework problems that provide the practical problem-solving experience needed to gain proficiency in inverse theory.

Computer-based computation requires software. Many different software environments are available for the type of scientific computation that underpins data analysis. Some are more applicable and others less applicable to inverse theory problems, but among the applicable ones, none has a decisive advantage. Nevertheless, we have chosen *MatLab*, a commercial software product of *The MathWorks, Inc.* as the book's software environment for several reasons, some having to do with its designs and other more practical. The most persuasive design reason is that its syntax fully supports linear algebra, which is needed by almost every inverse theory method. Furthermore, it supports *scripts*, that is, sequences of data analysis commands that are communicated in written form and which serve to document the data analysis process. Practical considerations include the following: it is a long-lived and stable product, available since the mid-1980s; implementations are available for most commonly used types of computers; its price, especially for students, is fairly modest; and it is widely used, at least, in university settings.

In *MatLab*'s scripting language, data are presented as one or more *named variables* (in the same sense that c and d in the formula, $c = \pi d$, are named variables). Data are manipulated by typing formula that create new variables from old ones and by running *scripts*, that is, sequences of formulas stored in a file. Much of inverse theory is simply the application of well-known formulas to novel data, so the great advantage of this approach is that the formulas that are typed usually have a strong similarity to those printed in a textbook. Furthermore, scripts provide both a way of documenting the sequence of a formula used to analyze a particular data set and a way to transfer the overall data analysis procedure from one data set to another. However, one disadvantage is that the parallel between the syntax of the scripting language and the syntax of standard mathematical notation is nowhere near perfect. A person needs to learn to translate one into the other.

I.3 A VERY QUICK *MATLAB* TUTORIAL

Unfortunately, this book must avoid discussion of the installation of *MatLab* and the appearance of *MatLab* on your computer screen, for procedures and appearances vary from computer to computer and quickly become outdated, anyway. We will assume that you have successfully installed it and that you can identify the Command Window, the place where *MatLab* formula and commands are typed. Once you have identified the Command Window, try typing:

```
date
```

MatLab should respond by displaying today's date. All the *MatLab* commands that are used in this book are in freely available *MatLab* scripts. This one is named `gda01_01` and is in a *MatLab* script file (*m-file*, for short) named `gda01_01.m` (conventionally, m-files have filenames that end with "`.m`"). In this case, the script is very short, since it just contains this one command, `date`, together with a couple of comment lines (which start with the character "`%`"):

```
% gda00_01
% displays the current date
date
```

(*MatLab* script gda00_01)

All the scripts are in a folder named `gda`. You should copy it to a convenient and easy-to-remember place in your computer's file system. The *MatLab* command window supports a number of commands that enable you to navigate from folder to folder, list the contents of folders, etc. For example, when you type:

```
pwd
```

(for "print working directory") in the Command Window, *MatLab* responds by displaying the name of the current folder. Initially, this is almost invariably the wrong folder, so you will need to `cd` (for "change directory") to the folder where you want to be—the `ch00` folder in this case. The pathname will, of course, depend upon where you copied the `gda` folder but will end in `gda/ch00`. On the author's computer, typing:

```
cd c:/menke/docs/gda/ch00
```

does the trick. If you have spaces in your pathname, just surround it with single quotes:

```
cd 'c:/menke/my docs/gda/ch00'
```

You can check if you are in the right folder by typing `pwd` again. Once in the `ch00` folder, typing:

```
gda00_01
```

will run the `gda00_01` m-script, which displays the current date. You can move to the folder above the current one by typing:

```
cd ..
```

and to one below it by giving just the folder name. For example, if you are in the `gda` folder you can move to the `ch00` folder by typing:

```
cd ch00
```

Finally, the command `dir` (for "directory") lists the files and subfolders in the current directory.

```
dir
```

(*MatLab* script gda00_02)

The *MatLab* commands for simple arithmetic and algebra closely parallel standard mathematical notation. For instance, the command sequence

```
a=4.5;
b=5.1;
c=a+b;
c
```

(*MatLab* script gda00_03)

evaluates the formula $c = a + b$ for the case $a = 4.5$ and $b = 5.1$ to obtain $c = 9.6$. Only the semicolons require explanation. By default, *MatLab* displays the value of every formula typed into the Command Window. A semicolon at the end of the formula suppresses the display. Hence, the first three lines, which end with semicolons, are evaluated but not displayed. Only the final line, which lacks the semicolon, causes *MatLab* to print the final result, `c`.

Note that *MatLab* variables are *static*, meaning that they persist in *MatLab*'s *Workspace* until you explicitly delete them or exit the program. Variables created by one script can be used by subsequent scripts. At any time, the value of a variable can be examined, either by displaying it in the Command Window (as we have done above) or by using the spreadsheet-like display tools available through *MatLab*'s Workspace Window. The persistence of *MatLab* variables can sometimes lead to scripting errors, such as when the definition of a variable in a script is inadvertently omitted, but *MatLab* used the value defined in a previous script. The command `clear all` deletes all previously defined variables in the Workspace. Placing this command at the beginning of a script causes it to delete any previously defined variables every time that it is run, ensuring that it cannot be affected by them.

The four commands discussed above can be run as a unit by typing `gda00_03`. Now open the m-file `gda01_03` in *MatLab*, using the File/Open menu. *MatLab* will bring up a text-editor type window. First, save it as a new file, say, `mygda01_03`; edit it in some simple way, say, by changing the `3.5` to `4.5`; save the edited file; and run it by typing `mygda01_03` in the Command Window. The value of `c` that is displayed will have changed appropriately.

A somewhat more complicated formula is

$$c = \sqrt{a^2 + b^2} \quad \text{with} \quad a = 6 \quad \text{and} \quad b = 8$$

```
a=6;
b=8;
c = sqrt(a^2 + b^2);
c
```

(*MatLab* gda00_04)

Note that the *MatLab* syntax for a^2 is `a^2` and that the square root is computed using the function, `sqrt()`. This is an example of *MatLab*'s syntax differing from standard mathematical notation.

A final example is

$$c = \sin\frac{n\pi(x - x_0)}{L} \quad \text{with} \quad n = 3, x = 4, x_0 = 1, L = 6$$

```
n=3; x=4; x0=1; L=6;
c = sin(n*pi*(x-x0)/L);
c
```

(*MatLab* gda00_05)

Note that several formulas, separated by semicolons, can be typed on the same line. Variables, such as `x0` and `pi`, above, can have names consisting of more than one character and can contain numerals as well as letters (though they must start with a letter). *MatLab* has a variety of predefined mathematical constants, including `pi`, which is the usual mathematical constant, π.

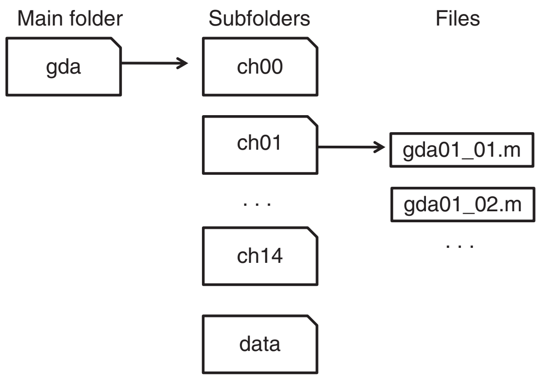

FIGURE I.1 Folder (directory) structure used for the files accompanying this book.

Files proliferate at an astonishing rate, even in the most trivial data analysis project. Data, notes, m-scripts, intermediate and final results, and graphics will all be contained in files, and their numbers will grow during the project. These files need to be organized through a system of folders (directories), subfolders (subdirectories), and filenames that are sufficiently systematic that files can be located easily and so that they are not confused with one another. Predictability both in the pattern of filenames and in the arrangement of folders and subfolders is an extremely important part of the design.

The files associated with this book are in a two-tiered folder/subfolder structure modeled on the chapter format of the book itself (Figure I.1). Folder and filenames are systematic. The chapter folder names are always of the form `chNN`, where `NN` is the chapter number. The m-script filenames are always of the form `gdaNN_MM.m`, where `NN` is the chapter number and `MM` are sequential integers. We have chosen to use leading zeros in the naming scheme (for example, `ch00`) so that filenames appear in the correct order when they are sorted alphabetically (as when listing the contents of a folder).

Many *MatLab* manuals, guides, and tutorials are available, both in printed form (e.g., Menke and Menke, 2011; Part-Enander *et al*., 1996; Pratap, 2009) and on the Web (e.g., at www.mathworks.com). The reader may find that they complement this book by providing more detailed information about *MatLab* as a scientific computing environment.

I.4 REVIEW OF VECTORS AND MATRICES AND THEIR REPRESENTATION IN *MATLAB*

Vectors and matrices are fundamental to inverse theory both because they provide a convenient way to organize data and because many important operations on data can be very succinctly expressed using *linear algebra* (that is, the algebra of vectors and matrices).

In the simplest interpretation, a *vector* is just a list of numbers that is treated as unit and given a symbolic name. The list can be organized horizontally, as a row, in which case it is called a *row vector*. Alternately, the list can be organized vertically, as a column, in which case it is called a *column vector*. We will use lower-case bold letters to represent both kinds of vector. An exemplary 1×3 row vector $\mathbf{r}$ and a 3×1 column vector $\mathbf{c}$ are

$$\mathbf{r} = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix} \quad \text{and} \quad \mathbf{c} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}$$

A row vector can be turned into a column vector, and vice versa, by the *transpose* operation, denoted by the superscript "T." Thus,

$$\mathbf{r}^{\mathrm{T}} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} \quad \text{and} \quad \mathbf{c}^{\mathrm{T}} = \begin{bmatrix} 1 & 3 & 5 \end{bmatrix}$$

In *MatLab*, row vector and column vectors are defined by

```
r = [2, 4, 6];
c = [1, 3, 5]';
```

(*MatLab* gda00_06)

In *MatLab*, the transpose is denoted by the single quote, so the column vector `c` in the script above is created by transposing a row vector. Although both column vectors and row vectors are useful, our experience is that defining both in the same script creates serious opportunities for error. A formula that requires a column vector will usually yield incorrect results if a row vector is substituted into it, and vice versa. Consequently, we will adhere to a protocol where all vectors defined in this book are column vectors. Row vectors will be created when needed—and as close as possible to where they are used in the script—by transposing the equivalent column vector.

An individual number within a vector is called an *element* (or, sometimes, *component*) and is denoted with an integer index, written as a subscript, with the index 1 in the left of the row vector and top of the column vector. Thus, $r_2 = 4$ and $c_3 = 5$ in the example above. In *MatLab*, the index is written inside parentheses, as in

```
a=r(2);
b=c(3);
```

(*MatLab* gda00_06)

Sometimes, we will wish to indicate a generic element of the vector, in which case we will give it a variable index, as in r_i and c_j, with the understanding that i and j are integer variables.

In the simplest interpretation, a *matrix* is just a rectangular table of numbers. We will use bold uppercase names to denote matrices, as in

$$\mathbf{M} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

In the above example, the number of rows and number of columns are equal, but this property is not required; matrices can also be rectangular. Thus, row vectors and column vectors are just special cases of rectangular matrices. The transposition operation can also be performed on a matrix, in which case its rows and columns are interchanged.

$$\mathbf{M}^{\mathrm{T}} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

A square matrix $\mathbf{M}$ is said to be *symmetric* if $\mathbf{M}=\mathbf{M}^{\mathrm{T}}$. In *MatLab*, a matrix is defined by

```
M=[ [1, 4, 7]', [2, 5, 8]', [3, 6, 9]'];
```

(*MatLab* gda00_06)

or, alternately, by

```
M2 = [1, 2, 3;
      4, 5, 6;
      7, 8, 9];
```

(*MatLab* gda00_06)

The first case, the matrix, $\mathbf{M}$, is constructed from a "row vector of column vectors" and in the second case, as a "column vector of row vectors." The individual elements of a matrix are denoted with two integer indices, the first indicating the row and the second the column, starting in the upper left. Thus, in the example above, $M_{31}=3$. Note that transposition swaps indices; that is, M_{ji} is the transpose of M_{ij}. In *MatLab*, the indices are written inside parentheses, as in

```
c = M(1,3);
```

(*MatLab* gda01_06)

One of the key properties of vectors and matrices is that they can be manipulated symbolically—as entities—according to specific rules that are similar to normal arithmetic. This allows tremendous simplification of data processing formulas, since all the details of what happens to individual elements within those entities are hidden from view and automatically performed.

In order to be added, two matrices (or vectors, viewing them as a special case of a rectangular matrix) must have the same number of rows and columns. Their sum is then just the matrix that results from summing corresponding elements. Thus, if

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{N} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 2 & 0 \\ 1 & 0 & 3 \end{bmatrix} \quad \text{then}$$

$$\mathbf{S} = \mathbf{M} + \mathbf{N} = \begin{bmatrix} 1+1 & 0+0 & 2-1 \\ 0+0 & 1+2 & 0+0 \\ 2-1 & 0+0 & 1+3 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 1 \\ 0 & 3 & 0 \\ 1 & 0 & 4 \end{bmatrix}$$

Subtraction is performed in an analogous manner. In terms of the components, addition and subtraction are written as

$$S_{ij} = M_{ij} + N_{ij} \quad \text{and} \quad D_{ij} = M_{ij} - N_{ij}$$

Note that addition is commutative (that is, $\mathbf{M}+\mathbf{N}=\mathbf{N}+\mathbf{M}$) and associative (that is, $(\mathbf{A}+\mathbf{B})+\mathbf{C}=\mathbf{A}+(\mathbf{B}+\mathbf{C})$). In *MatLab*, addition and subtraction are written as

```
S = M+N;
D = M-N;
```

(*MatLab* gda00_07)

Multiplication of two matrices is a more complicated operation and requires that the number of columns of the left-hand matrix equal the number of rows of the right-hand matrix. Thus, if the matrix $\mathbf{M}$ is $N \times K$ and the matrix $\mathbf{N}$ is $K \times M$, the product $\mathbf{P}=\mathbf{NM}$ is an $N \times M$ matrix defined according to the rule (Figure I.2):

$$P_{ij} = \sum_{k=1}^{K} M_{ik} N_{kj}$$

The order of the indices is important. Matrix multiplication is in its standard form when all summations involve neighboring indices of adjacent quantities. Thus, for instance, the two instances of the summed variable k are *not* neighboring in the equation

$$Q_{ij} = \sum_{k=1}^{K} M_{ki} N_{kj}$$

and so the equation corresponds to $\mathbf{Q}=\mathbf{M}^{\mathrm{T}}\mathbf{N}$ and not $\mathbf{Q}=\mathbf{MN}$. Matrix multiplication is not commutative (that is, $\mathbf{MN} \neq \mathbf{NM}$) but is associative (that is, $(\mathbf{AB})$

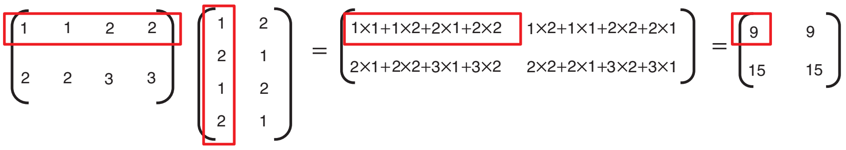

FIGURE I.2 Graphical depiction of matrix multiplication. A row of the first matrix is paired up with a column of the second matrix. The pairs are multiplied together and the results are summed, producing one element of resultant matrix.

$\mathbf{C} = \mathbf{A}(\mathbf{BC})$) and distributive (that is, $\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB}+\mathbf{AC}$). An important rule involving the matrix transpose is $(\mathbf{MN})^{\mathrm{T}} = \mathbf{N}^{\mathrm{T}}\mathbf{M}^{\mathrm{T}}$ (note the reversal of the order).

Several special cases of multiplication involving vectors are noteworthy. Suppose that $\mathbf{a}$ and $\mathbf{b}$ are length-N column vectors. The combination $s = \mathbf{a}^{\mathrm{T}}\mathbf{b}$ is a scalar number s and is called the *dot product* (or sometimes, *inner product*) of the vectors. It obeys the rule $\mathbf{a}^{\mathrm{T}}\mathbf{b} = \mathbf{b}^{\mathrm{T}}\mathbf{a}$. The dot product of a vector with itself is the square of its *Euclidian length*; that is, $\mathbf{a}^{\mathrm{T}}\mathbf{a}$ is the sum of its squared elements of $\mathbf{a}$. The vector $\mathbf{a}$ is said to be a unit vector when $\mathbf{a}^{\mathrm{T}}\mathbf{a} = 1$. The combination $\mathbf{T} = \mathbf{ab}^{\mathrm{T}}$ is an $N \times N$ matrix; it is called the *outer product*. The product of a matrix and a vector is another vector, as in $\mathbf{c} = \mathbf{Ma}$. One interpretation of this relationship is that the matrix $\mathbf{M}$ "turns one vector into another." Note that the vectors $\mathbf{a}$ and $\mathbf{c}$ can be of different length. An $M \times N$ matrix $\mathbf{M}$ turns the length-N vector $\mathbf{a}$ into a length-M vector $\mathbf{c}$. The combination $s = \mathbf{a}^{\mathrm{T}}\mathbf{Ma}$ is a scalar and is called a *quadratic form*, since it contains terms quadratic in the elements of $\mathbf{a}$. Matrix multiplication, $\mathbf{P} = \mathbf{NM}$, has a useful interpretation in terms of dot products: P_{ij} is the dot product of the ith row of $\mathbf{M}$ with the jth column of $\mathbf{N}$.

Any matrix is unchanged when multiplied by the *identity matrix*, conventionally denoted $\mathbf{I}$. Thus, $\mathbf{a} = \mathbf{Ia}$, $\mathbf{M} = \mathbf{IM} = \mathbf{MI}$, etc. This matrix has ones along its main diagonal, and zeroes elsewhere, as in

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The elements of the identity matrix are usually written δ_{ij} and not I_{ij}, and the symbol δ_{ij} is usually called the *Kronecker delta* symbol, not the *elements of the identity matrix* (though that is exactly what it is). The equation $\mathbf{M} = \mathbf{IM}$, for an $N \times N$ matrix $\mathbf{M}$, is written component-wise as

$$M_{ij} = \sum_{k=1}^{N} \delta_{ik} M_{kj}$$

This equation indicates that any summation containing a Kronecker delta symbol can be performed trivially. To obtain the result, one first identifies the variable that is being summed over (k in this case) and the variable that the summed variable is paired with in the Kronecker delta symbol (i in this case). The summation and the Kronecker delta symbol then are deleted from the equation, and all occurrences of the summed variable are replaced with the paired variable (all ks are replaced by is in this case). In *MatLab*, an $N \times N$ identity matrix can be created with the command

```
I = eye(N);
```

(*MatLab* gda00_07)

MatLab performs all multiplicative operations with ease. For example, suppose column vectors, $\mathbf{a}$ and $\mathbf{b}$, and matrices, $\mathbf{M}$ and $\mathbf{N}$, are defined as

$$\mathbf{a} = \begin{bmatrix}1\\3\\5\end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix}2\\4\\6\end{bmatrix} \quad \text{and} \quad \mathbf{M} = \begin{bmatrix}1&0&2\\0&1&0\\2&0&1\end{bmatrix} \quad \text{and} \quad \mathbf{N} = \begin{bmatrix}1&0&-1\\0&2&0\\-1&0&3\end{bmatrix}$$

Then

$$s = \mathbf{a}^{\mathrm{T}}\mathbf{b} = \begin{bmatrix}1\\3\\5\end{bmatrix}^{\mathrm{T}}\begin{bmatrix}2\\4\\6\end{bmatrix} = [\,1\;3\;5\,]\begin{bmatrix}2\\4\\6\end{bmatrix} = 2\times1+3\times4+5\times6 = 44$$

$$\mathbf{T} = \mathbf{a}\mathbf{b}^{\mathrm{T}} = \begin{bmatrix}1\\3\\5\end{bmatrix}\begin{bmatrix}2\\4\\6\end{bmatrix}^{\mathrm{T}} = \begin{bmatrix}2\times1 & 4\times1 & 6\times1\\2\times3 & 4\times3 & 6\times3\\2\times5 & 4\times5 & 6\times5\end{bmatrix} = \begin{bmatrix}2&4&6\\6&12&18\\10&20&30\end{bmatrix}$$

$$\mathbf{c} = \mathbf{M}\mathbf{a} = \begin{bmatrix}1&0&2\\0&1&0\\2&0&1\end{bmatrix}\begin{bmatrix}1\\3\\5\end{bmatrix} = \begin{bmatrix}1\times1 & + & 0\times3 & + & 2\times5\\0\times1 & + & 1\times3 & + & 0\times5\\2\times1 & + & 0\times3 & + & 1\times5\end{bmatrix} = \begin{bmatrix}11\\3\\7\end{bmatrix}$$

$$\mathbf{P} = \mathbf{M}\mathbf{N} = \begin{bmatrix}1&0&2\\0&1&0\\2&0&1\end{bmatrix}\begin{bmatrix}1&0&-1\\0&2&0\\-1&0&3\end{bmatrix} = \begin{bmatrix}-1&0&5\\0&2&0\\1&0&1\end{bmatrix}$$

corresponds to

```
s = a'*b;
T = a*b';
c = M*a;
P = M*N;
```

(*MatLab* gda00_07)

In *MatLab*, matrix multiplication is signified using the multiplications sign, `*` (the asterisk). There are cases, however, where one needs to violate the rules and multiply the quantities element-wise (for example, create a vector, **d**, with elements $d_i = a_i b_i$). *MatLab* provides a special element-wise version of the multiplication sign, denoted `.*` (a period followed by an asterisk)

```
d = a.*b;
```

(*MatLab* gda00_07)

As described above, individual elements of vectors and matrices can be accessed by specifying the relevant row and column indices, in parentheses, e.g., `a(2)` is the second element of the column vector **a**, and `M(2,3)` is the second row, third column element of the matrix, **M**. Ranges of rows and columns can be specified using the `:` (colon) operator, e.g., `M(:,2)` is the second column of matrix, **M**; `M(2,:)` is the second row of matrix, **M**; and `M(2:3,2:3)` is the 2×2 submatrix in the lower right-hand corner of the 3×3 matrix, **M** (the expression, `M(2:end,2:end)`, would work as well). These operations are further illustrated below:

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \text{and} \quad \mathbf{M} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$s = a_2 = 2 \quad \text{and} \quad t = M_{23} = 6 \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} M_{12} \\ M_{22} \\ M_{32} \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 8 \end{bmatrix}$$

$$\mathbf{c} = [M_{21} \quad M_{22} \quad M_{23}]^{\mathrm{T}} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \quad \text{and} \quad \mathbf{T} = \begin{bmatrix} M_{22} & M_{23} \\ M_{32} & M_{33} \end{bmatrix} = \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}$$

correspond to

```
s = a(2);
t = M(2,3);
b = M(:,2);
c = M(2,:)';
T = M(2:3,2:3);
```

(*MatLab* gda00_08)

The colon notation can be used in other contexts in *MatLab* as well. For instance, `[1:4]` is the row vector [1, 2, 3, 4]. The syntax, `1:4`, which omits the square brackets, works fine in *MatLab*. However, we will usually use square brackets, since they draw attention to the presence of a vector. Finally, we note that two colons can be used in sequence to indicate the spacing of elements in the resulting vector. For example, the expression `[1:2:9]` is the row vector [1, 3, 5, 7, 9] and that the expression `[10:-1:1]` is a row vector whose elements are in the reverse order from `[1:10]`.

Matrix division is defined in analogy to reciprocals. If s is a scalar number, then multiplication by the reciprocal s^{-1} is equivalent to division by s. Here, the reciprocal obeys $s^{-1}s = ss^{-1} = 1$. The matrix analog to the reciprocal is called the *matrix inverse* and obeys

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$$

It is defined only for square matrices. The calculation of the inverse of a matrix is complicated, and we will not describe it here, except to mention the 2×2 case

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

Just as the reciprocal s^{-1} is defined only when $s \neq 0$, the matrix inverse $\mathbf{A}^{-1}$ is defined only when a quantity called the *determinant* of **A**, denoted det(**A**), is not equal to zero. The determinant of a square $N \times N$ matrix **M** is defined as

$$\det(\mathbf{M}) = \sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N}\cdots\sum_{q=1}^{N}\varepsilon^{ijk\ldots q}M_{1i}M_{2j}M_{3k}\ldots M_{Nq}$$

Here the quantity $\varepsilon^{ijk\ldots q}$ is $+1$ when $(i, j, k, \ldots, q)$ is an even permutation of $(1, 2, 3, \ldots, N)$, -1 when it is an odd permutation and zero otherwise. Note that the determinant of an $N \times N$ is the sum of products of N elements of the matrix. In the case of a 2×2 matrix, the determinant contains products of two elements and is given by

$$\det\begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

Note that the reciprocal of the 2×2 determinant appears in the formula for the 2×2 matrix inverse, implying that this matrix inverse does not exist when the determinant is zero. This is a general property of matrix inverses; they exist only when the matrix has nonzero determinant. In *MatLab*, the matrix inverse and determinant of a square matrix $\mathbf{A}$ are computed as

```
B = inv(A);
d = det(A);
```

(*MatLab* gda00_09)

In many of the formulas of inverse theory, the matrix inverse either premultiplies or postmultiplies other quantities, for instance:

$$\mathbf{c} = \mathbf{A}^{-1}\mathbf{b} \quad \text{and} \quad \mathbf{D} = \mathbf{B}\mathbf{A}^{-1}$$

These cases do not actually require the explicit calculation of $\mathbf{A}^{-1}$, just the combinations $\mathbf{A}^{-1}\mathbf{b}$ and $\mathbf{B}\mathbf{A}^{-1}$, which are computationally simpler. *MatLab* provides generalizations of the division operator that implement these two cases:

```
c = A\b;
D = B/A;
```

(*MatLab* gda00_09)

A surprising amount of information on the structure of a matrix can be gained by studying how it affects a column vector that it multiplies. Suppose that $\mathbf{M}$ is an $N \times N$ square matrix and that it multiplies an *input* column vector, $\mathbf{v}$, producing an *output* column vector, $\mathbf{w} = \mathbf{M}\mathbf{v}$. We can examine how the output $\mathbf{w}$ compares to the input $\mathbf{v}$ as $\mathbf{v}$ is varied. One question of particular importance is

When is the output parallel to the input?

This question is called the *algebraic eigenvalue problem*. If $\mathbf{w}$ is parallel to $\mathbf{v}$, then $\mathbf{w} = \lambda\mathbf{v}$, where λ is a scalar proportionality factor. The parallel vectors satisfy the following equation:

$$\mathbf{M}\mathbf{v} = \lambda\mathbf{v} \quad \text{or} \quad (\mathbf{M} - \lambda\mathbf{I})\mathbf{v} = 0$$

The trivial solution $\mathrm{v} = (\mathrm{M} - \lambda\mathrm{I})^{-1}0 = 0$ is not very interesting. A nontrivial solution is only possible when the matrix inverse $(\mathrm{M} - \lambda\mathrm{I})^{-1}$ does not exist. This is the case where the parameter λ is specifically chosen to make the determinant $\det(\mathrm{M} - \lambda\mathrm{I})$ exactly zero, since a matrix with zero determinant has no inverse. The determinant is calculated by adding together terms, each of which contains the product of N elements of the matrix. Since each element of the matrix contains, at most, one instance of λ, the product will contain powers of λ up to λ^N. Thus, the equation, $\det(\mathrm{M} - \lambda\mathrm{I}) = 0$, is an Nth order polynomial equation for λ. An Nth order polynomial equation has N roots, so we conclude that there must be N different proportionality factors, say λ_i, and N corresponding column vectors, say $\mathbf{v}^{(i)}$, that solve $\mathbf{M}\mathbf{v}^{(i)} = \lambda_i \mathbf{v}^{(i)}$. The column vectors, $\mathbf{v}^{(i)}$, are called the *characteristic vectors* (or *eigenvectors*) of the matrix, $\mathbf{M}$, and the proportionality factors, λ_i, are called the *characteristic values* (or *eigenvalues*). Eigenvectors are determined only up to an arbitrary multiplicative factor s, since if $\mathbf{v}^{(i)}$ is an eigenvector, so is $s\mathbf{v}^{(i)}$. Consequently, they are conventionally chosen to be unit vectors.

In the special case where $\mathbf{M}$ is symmetric, it can be shown that the eigenvalues, λ_i, are real and the eigenvectors are mutually perpendicular, $\mathbf{v}^{(i)\mathrm{T}}\mathbf{v}^{(j)} = 0$ for $i \neq j$. The N eigenvalues can be arranged into a diagonal matrix, $\mathbf{\Lambda}$, whose elements are $[\mathbf{\Lambda}]_{ij} = \lambda_i \delta_{ij}$, where δ_{ij} is the Kronecker delta. The corresponding N eigenvectors $\mathbf{v}^{(i)}$ can be arranged as the columns of an $N \times N$ matrix $\mathbf{V}$, which satisfies, $\mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{V}\mathbf{V}^{\mathrm{T}} = \mathbf{I}$. The eigenvalue equation $\mathrm{Mv} = \lambda\mathrm{v}$ can then be succinctly written as

$$\mathbf{M}\mathbf{V} = \mathbf{V}\mathbf{\Lambda} \quad \text{or} \quad \mathbf{M} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{\mathrm{T}}$$

(The second equation is derived from the first by postmultiplying it by $\mathbf{V}^{\mathrm{T}}$.) Thus, the matrix $\mathbf{M}$ can be reconstructed from its eigenvalues and eigenvectors. In *MatLab*, the matrix of eigenvalues $\mathbf{\Lambda}$ and matrix of eigenvectors $\mathbf{V}$ of a matrix $\mathbf{M}$ are computed as

```
[V,LAMBDA] = eig(M);
```

(*MatLab* gda00_09)

Here the eigenvector matrix is called LAMBDA.

Many of the derivations of inverse theory require that a column vector $\mathbf{v}$ be considered a function of an independent variable, say x, and then differentiated with respect to that variable to yield the derivative $\mathrm{d}\mathbf{v}/\mathrm{d}x$. Such a derivative represents the fact that the vector changes from $\mathbf{v}$ to $\mathbf{v} + \mathrm{d}\mathbf{v}$ as the independent variable changes from x to $x + \mathrm{d}x$. Note that the resulting change $\mathrm{d}\mathbf{v}$ is itself a vector. Derivatives are performed element-wise; that is,

$$\left[\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}x}\right]_i = \frac{\mathrm{d}v_i}{\mathrm{d}x}$$

A somewhat more complicated situation is where the column vector $\mathbf{v}$ is a function of another column vector, say $\mathbf{y}$. The partial derivative

$$\frac{\partial v_i}{\partial y_j}$$

represents the change in the *i*th component of **v** caused by a change in the *j*th component of **y**. Frequently, we will need to differentiate the linear function, **v** = **My**, where **M** is a matrix, with respect to **y**:

$$\frac{\partial v_i}{\partial y_j} = \frac{\partial}{\partial y_j}\left[\sum_k M_{ik} y_k\right] = \sum_k M_{ik} \frac{\partial y_k}{\partial y_j}$$

Since the components of **y** are assumed to be independent, the derivative $\frac{\partial y_k}{\partial y_j}$ is zero except when $j = k$, in which case it is unity, which is to say $\frac{\partial y_k}{\partial y_j} = \delta_{kj}$. The expression for the derivative then simplifies to

$$\frac{\partial v_i}{\partial y_j} = \frac{\partial}{\partial y_j}\left[\sum_k M_{ik} y_k\right] = \sum_k M_{ik} \frac{\partial y_k}{\partial y_j} = \sum_k M_{ik}\delta_{kj} = M_{ij}$$

Thus the derivative of the linear function **v** = **My** is the matrix **M**. This relationship is the vector analogue to the scalar case, where the linear function $v = my$ has the derivative $\frac{\mathrm{d}v}{\mathrm{d}y} = m$.

I.5 USEFUL *MATLAB* OPERATIONS

I.5.1 Loops

MatLab provides a looping mechanism, the `for` command, which can be useful when the need arises to sequentially access the elements of vectors and matrices. Thus, for example,

```
M = [ [1, 4, 7]', [2, 5, 8]', [3, 6, 9]' ];
for i = [1:3]
    a(i) = M(i,i);
end
```

(*MatLab* gda00_10)

executes the `a(i)=M(i,i)` formula three times, each time with a different value of `i` (in this case, $i = 1$, $i = 2$, and $i = 3$). The net effect is to copy the diagonal elements of the matrix **M** to the vector, **a**, that is, $a_i = M_{ii}$. Note that the `end` statement indicates the position of the bottom of the loop. Subsequent commands are not part of the loop and are executed only once.

Loops can be nested; that is, one loop can be inside another. Such an arrangement is necessary for accessing all the elements of a matrix in sequence. For example,

```
M = [ [1, 4, 7]', [2, 5, 8]', [3, 6, 9]'];
for i = [1:3]
for j = [1:3]
    N(i,4-j) = M(i,j);
end
end
```

(*MatLab* gda00_11)

copies the elements of the matrix, **M**, to the matrix, **N**, but reverses the order of the elements in each row, that is, $N_{i,4-j}=M_{i,j}$. Loops are especially useful in conjunction with *conditional* commands. For example,

```
a = [ 1, 2, 1, 4, 3, 2, 6, 4, 9, 2, 1, 4 ]';
for i = [1:12]
    if ( a(i) >= 6 )
       b(i) = 6;
    else
       b(i) = a(i);
    end
end
```

(*MatLab* gda00_12)

sets $b_i = a_i$ if $a_i < 6$ and sets $b_i = 6$, otherwise (a process called *clipping* a vector, for it lops off parts of the vector that are larger than 6).

A purist might point out that *MatLab* syntax is so flexible that `for` loops are almost never really necessary. In fact, all three examples, above, can be computed with one-line formulas that omit `for` loops:

```
a = diag(M);
N = fliplr(M);
b=a; b(find(a>6))=6;
```

(*MatLab* gda00_13)

The first two formulas are quite simple, but rely upon the *MatLab* functions `diag()` (for "diagonal") and `fliplr()` (for "flip left-right"), whose existence we have not hitherto mentioned. The third formula, which used the `find()` function, requires further explanation. The first part just copies the column vector **a** to **b**. In the second part, the `(a>6)` operation returns a vector of zeros and ones, depending upon whether the elements of the column vector **a** satisfy the inequality or not. The `find()` function uses this result and returns a list of the *indices* of the ones, that is, of the indices of the column vector **a** that match the condition. This list is then used to reset just those elements of **b** to 6, leaving the other elements unchanged.

One of the problems of a script-based environment is that learning the complete syntax of the scripting language can be pretty daunting. Writing a long script, such as one containing a `for` loop, will often be faster than searching through *MatLab* help files for a predefined function that implements the desired functionality in a single line of the script. When deciding between alternative ways of implementing a given functionality, you should always choose the one which *you* find clearest. Scripts that are terse or even computationally efficient are not necessarily a virtue, especially if they are difficult to debug. You should avoid creating formulas that are so inscrutable that you are not sure whether they will function correctly. Of course, the degree of inscrutability of any given formula will depend upon your level of familiarity with *MatLab*. Your repertoire of techniques will grow as you become more practiced.

I.5.2 Loading Data from a File

MatLab can read and write files with a variety for formats, but we start here with the simplest and most common, the text file. As an example, we load a global temperature dataset compiled by the National Aeronautics and Space Administration. The author's recommendation is that you always keep a file of notes about any data set that you work with, and that these notes include information on where you obtained the data set and any modifications that you subsequently made to it:

> *The text file global_temp.txt contains global temperature change data from NASA's web site http://data.giss.nasa.gov/gistemp. It has two columns of data, time (in calendar years) and temperature anomaly (in degrees C) and is 46 lines long. Information about the data is in the file global_temp_notes.txt. The citation for this data is Hansen* et al. *(2010).*

We reproduce the first few lines of `global_temp.txt`, here:

```
1965  -0.11
1966  -0.03
1967  -0.01
 ...    ...
```

The data are read into to *MatLab* as follows:

```
D = load('../data/global_temp.txt');
t = D(:,1);
d = D(:,2);
```

(*MatLab* gda00_14)

The `load()` function reads the data into a 46×2 matrix, **D**. Note that the filename is given as `../data/global_temp.txt`, as contrasted to just `global_temp.txt`, since the script is run from the `ch00` folder while the data are in the `data` folder. The filename is surrounded by single quotes to indicate that it is a *character string* and not a variable name. The subsequent two lines break out **D** into two separate column vectors, **t** of time and **d** of temperature data. This step is not strictly speaking necessary, but fewer mistakes will be made if the different variables in the dataset have each their own name.

The `figure(1)`command directs *MatLab* to create a figure window and to label it Figure 1. The `clf` (for clear figure) command erases any previous plots in the figure window, ensuring that it is blank. The default line width of plot axes in *MatLab* is one point, too thin to show up clearly in printed documents. We increase it to three points using the `set()` command. The `hold on` command instructs *MatLab* to overlay multiple plots in the same window, overriding the default that a second plot erases the first. The `axis( [1965, 2010, -0.5, 1.0] )` command manually sets the axes to an appropriate range. If this command is omitted, *MatLab* chooses the scaling of the axes based on the range of the data. The two `plot(...)` commands plot the time `t` and temperature `d` data in two different ways: first as a red line (indicated by the `'r-')` and the second with black circles (the `'ko'`). We also increase the width of the lines with the `'LineWidth', 3` directive. Finally, the horizontal and vertical axes are labeled using the `xlabel()` and `ylabel()` commands, and the plot is titled with the `title()` command. Note that the text is surrounded with single quotes, to indicate that it is a character string.

I.5.4 Creating Character Strings Containing the Values of Variables

The results of computations are most understandable if described in a combination of words and numbers. The `sprintf()`function (for "string print formatted") creates a character string that includes both text and the value of a variable. The string can then be used as a plot label, a filename, as text displayed in the Command Window, or for a wide variety of similar purposes. Thus, for instance,

```
    title(sprintf('temperature data from %d to %d',t(1),t(N)));
```

(*MatLab* gda00_15)

creates a plot title `'temperature data from 1965 to 2010.'` The `sprintf()` command can be used in any function that expects a character string, including `title()`, `xlabel()`, `ylabel()`, `load()`, and the function `disp()`, not mentioned until now, which writes a character string to the Command Window. The `sprintf()` function is fairly inscrutable, and we refer readers to the *MatLab* help pages for a detailed description. Briefly, the function uses *placeholders* that start with the character % to indicate where in the character string the value of the variable should be placed. Thus,

```
    disp(sprintf('Number of data: %d', N));
```

(*MatLab* gda00_15)

writes the character string '`Number of data: 46'` (since the value of `N` was `46`) to the Command Window. The `%d` is the *placeholder for an integer*. It is replaced

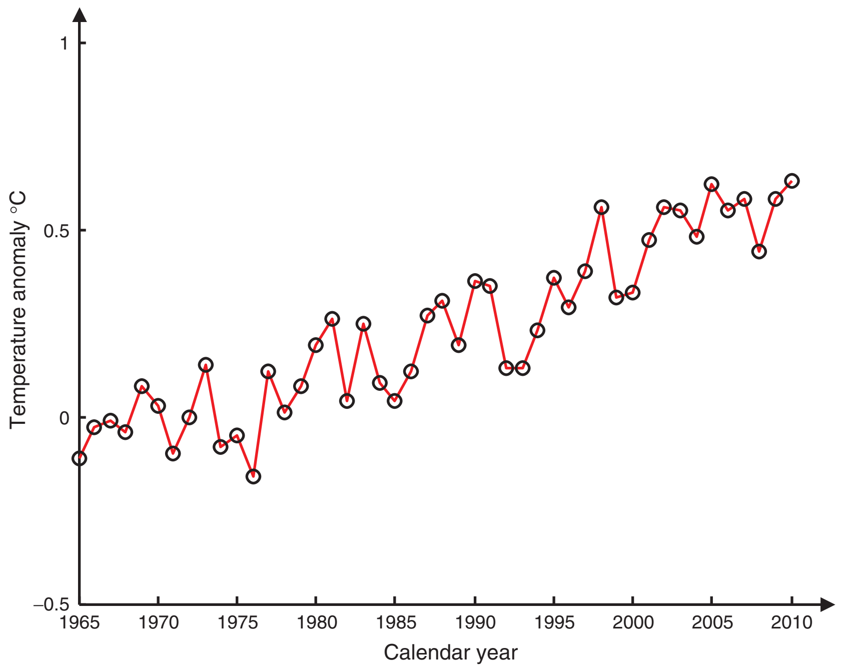

FIGURE I.3 Global temperature data for the time period 1965–2010. See text for further discussion. *MatLab* script gda00_14.

I.5.3 Plotting Data

MatLab's plotting commands are very powerful, but they are also very complicated. We present here a set of commands for making a simple x–y plot that is intermediate between a very crude, unlabeled plot, and an extremely artistic one. The reader may wish to adopt either a simpler or a more complicated version of this set, depending upon need and personal preference. The plot of the global temperature data shown in Figure I.3 was created with the commands:

```
figure(1);
clf;
set(gca,'LineWidth',3);
hold on;
axis( [1965, 2010, -0.5, 1.0] );
plot(t,d,'r-','LineWidth',3);
plot(t,d,'ko','LineWidth',3);
xlabel('calendar year');
ylabel('temperature anomaly, deg C');
ylabel('temperature anomaly, deg C');
title('global temperature data');
```

(*MatLab* gda00_14)

with `'46,'` the value of `N`. If the variable is fractional, as contrasted to integer, the *floating-point placeholder*, `%f`, is used instead. For example,

```
% display first few lines
disp('first few lines of data');
for i=[1:4]
    disp(sprintf('%f %f',t(i),d(i)));
end
```

(*MatLab* gda00_15)

writes the first four lines of data to the Command Window. Note that two placeholders have been used in the same *format string*. When displaying text to the Command Window, an alternative function is available that combines the functionality of `sprintf()` and `disp()`

```
fprintf('%f %f\n',t(i),d(i));
```

which is equivalent to

```
disp(sprintf('%f %f',t(i),d(i)));
```

The `\n` (for *newline*) at the end of the format string `'a=%f\n'` indicates that subsequent characters should be written to a new line in the command window, rather than being appended onto the end of the current line.

REFERENCES

Hansen, J., Ruedy, R., Sato, Mki., Lo, K., 2010. Global surface temperature change. Rev. Geophys. 48, RG4004.

Menke, W., Menke, J., 2011. Environmental Data Analysis with MatLab. Academic Press, Elsevier Inc, Oxford, UK, 263pp.

Part-Enander, E., Sjoberg, A., Melin, B., Isaksson, P., 1996. The Matlab Handbook. Addison-Wesley, New York, 436pp.

Pratap, R., 2009. Getting Started with MATLAB: A Quick Introduction for Scientists and Engineers. Oxford University Press, Oxford, 256pp.