

## CHAPTER 1: GETTIG STARTED WITH MATLAB® OR PYTHON

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 beginners 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® or Python - we use the word "or" because only one or the other is needed - as the book's software environments, for several reasons, some having to do with their designs and others more practical. The most persuasive design reason is that both fully support linear algebra, which is needed by almost every inverse theory method. Furthermore, both 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: they are long-lived and stable platforms, available for several decades; implementations are available for most commonly used types of computers; and they are widely used in university settings.

In both MATLAB's and Python's scripting languages, 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.

## PART B. PYTHON AS A TOOL FOR LEARNING INVERSE THEORY

### 1B.1 GETTNG STARTED WITH PYTHON

The Python environment that we will be using is assembled from the following freely-available software elements:

python, itself;  
jupyter notebook (or jupyter lab), a "research notebook" environment for developing and running Python scripts and viewing and storing their results;  
numpy, (pronounced *num-pie*), the Python Numbers module, which extends Python by adding vectors, matrices and basic linear algebra;  
scipy, (pronounced *sci-pie*), the Scientific module, which extends Python by adding advanced linear algebraic functions;  
matplotlib, the plotting module, which expands Python by adding graphics;  
gdapy, a folder of Jupyter Notebooks containing exemplary Python scripts referenced by this book.

We will also be using the Anaconda Powershell environment to install the Python packages, and (in some cases) to launch Jupyter Notebook (Or Jupyter Lab). Finally, Jupyter Notebook (and Jupyter Lab) uses a web-browser, such as Chrome or Firefox, so you will need to have one of those installed on your computer. Almost everyone already has one of them installed, anyway.

You first should copy the `gdapy` folder to some convenient place on your computer's file system. It contains all files needed to perform this book's Python examples. Procedures for installing the other packages vary from computer to computer and quickly become outdated. We do not discuss them in this text, but instead provide a document, `InstallationHelp.pdf`, within the `gdapy` folder that provides a step-by-step installation procedure.

Launch the Jupyter Notebook (or Jupyter Lab) and open the file, `gdapy_01.ipynb`. It contains a collection of Python scripts related to this chapter. As you scroll through the Notebook, you will observe that it is organized in sections, called *cells*, and that sometimes text and graphics appear between the cells. Each cell contains a Python script, and the text and graphics following a cell are the results (or "output") associated with it.

In this book's notebooks, the first cell performs all required initializations. It always must be run *first*, immediately after the Notebook is opened, and before running any of the other cells. Lines starting with the `#` character, such as: `# gdapy_01_00`, are comment lines; their only function is to provide information about the script; they do not influence its function. Our convention is that the first line of a cell contains a name-tag, in this case `gdapy_01_01`, which is used in the text to reference the script. The line:

```
%reset -f
```

[gdapy01\_00]

directs the notebook to completely clear its memory, thus deleting any variables that may have been defined previously. The practice of starting with a fresh slate every time one opens a Notebook is a good one, since it eliminates the possibility that old (and possibly incorrect) results will be comingled with new ones. The next section of the first cell "imports" various modules that expand the rather bare-bones functionality of Python. The command:

```
import numpy as np
```

[gdapy01\_00]

imports the complete Python Numbers module, `numpy`, and abbreviates its name as `np`. This module adds vectors and matrices and methods for manipulating them. Alternatively, just part of a module can be imported. For instance, the command:

```
from matplotlib import pyplot as plt
```

[gdapy01\_00]

imports just the Python Plotting module, `pyplot`, from the larger `matplotlib` module, and abbreviates it as `plt`. Finally, functions defined in a module can be imported individually. For instance, the command:

```
from math import exp, pi, sin, sqrt
```

[gdapy01\_00]

imports the exponential function, `exp()`, the sine function, `sin()`, the square root function, `sqrt()` and the number,  $\pi$  from Python's `math` module. This is a very frugal style of importing, but has the advantage that you can never accidentally use any function not in your list. For instance, a slip of the typing-finger that adds an extraneous “h” can never accidentally turn the sine function, `sin()`, into the hyperbolic-sine function, `sinh()`. On the other hand, should a modification of the script introduce the cosine function, `cos()`, then that function needs to be added to the import command.

Incidentally, when you read Python documentation, you will often encounter the word *method* to describe a component of a module. Methods are similar to, but not quite the same as, functions, but we will not draw any distinction between them right now.

Most of the functions and methods used in this book are not part of Python, itself, but instead are part of modules. They must be imported before they can be used. This point brings out several problems encountered by all of us who write Python scripts: how to decide what modules to import; how to discover whether a module that adds sought-after functionality exists; and how to learn to use an unfamiliar module. This book will introduce you to a wide selection of useful modules—enough for you to perform fairly complicated data analysis. On the other hand, your work might be expedited by specialized modules not discussed here. Fortunately, because Python is so popular, web-searches will often turn up documentation and tutorials that offer helpful advice.

Notwithstanding that most of the functions and methods we will be using are not part of Python itself, but rather of some module that extends Python, we will use colloquial and inexact language and call everything “Python” - unless we want to draw attention to some specific module.

The rest of the first cell defines several functions that are not part of any module, but rather were written by us. For instance, the line:

```
def gda_cvec(v):
```

[gdapy01\_00]

(and the many lines of code that follow it) define a function, `gda_cvec()`, that expands Python's ability to manipulate vectors. Its use will be discussed later in this chapter. You should run the first cell once you have opened `edapy_01.ipynb` in the Jupyter Notebook. You should check that it successfully executed by checking for error messages, which would be written immediately following the end of the cell. Hopefully, there are none!

The second cell in `gdapy_01.ipynb` contains the short script that prints today's date:

```
# gdapy01_01 print the date
thedata = date.today(),
print(thedata);
```

[gdapy01\_01]

The first line, starting with the character, `#`, is a *comment* that includes our name of the script, `gdapy01_01` (which stands for Geophysical Data Analysis, Python, Chapter 1, Script 1). The second line gets today's date and puts it into a variable `thedata`. The notation `date.today()` means the method `today()` from the `date` module. (The method takes no arguments, so nothing is enclosed by the parentheses). Once the second line is executed, the variable `thedata` contains the date. The third line prints it at the end of the cell as a character string. For example:

2022-11-01

The `print()` function is extremely useful when debugging scripts, for it can print more-or-less anything and easily can be inserted at strategic points in scripts to print the values of variables. However, often its output is not very aesthetic

## 1B.2 EFFECTIVE USE OF FOLDERS

Files proliferate at an astonishing rate, even in the most trivial data analysis project. Data, notes, Notebooks, intermediate results and final results 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 Python-related files associated with this book are in a folder/subfolder/filename structure modeled on the format of the book itself (Fig. 1B.1). The main folder is named `gdapy`. It contains several subfolders, `Notebooks` (containing Jupyter Notebooks), `Data` (containing data), `TestFolder` (for test purposes) and `Html` (for hypertext). The Jupyter Notebooks are named `gdapy01.ipynb`, `gdapy02.ipynb`, etc., one file for each of the book's chapters. We have chosen to use leading zeros in the naming scheme (e.g., `01`) so that filenames appear in the correct order when they are sorted alphabetically (as when listing the contents of a folder). A given Notebook file contains all the scripts for the corresponding chapter, one per cell.

Comments at the top of each cell are used to identify each script, with name-tags of the form `gdapyNN_MM.mlx`, where the chapter number, NN, and the script number, MM, are sequential integers.

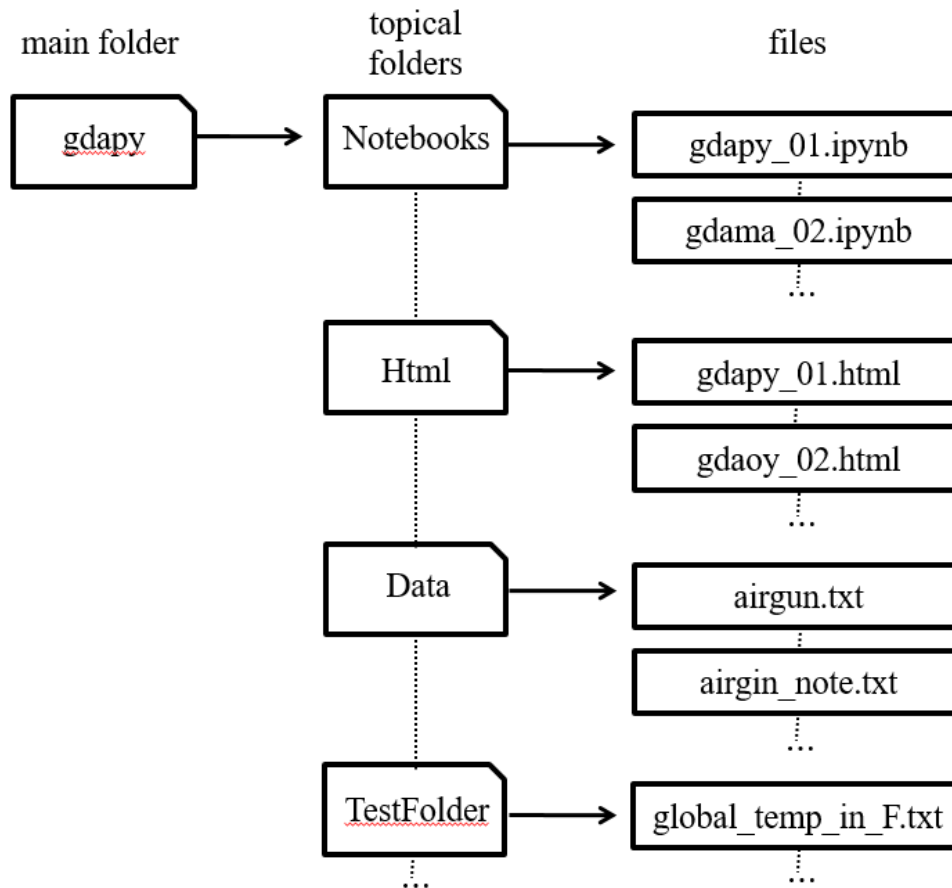


Fig. 1B.1 Folder (directory) structure used for the Python files accompanying this book.

Python supports a number of commands that enable you to navigate from folder to folder, list the contents of folders, etc. For example, the commands:

```
mydir=os.getcwd();
print(mydir);
```

[gdapy01\_02]

get and print the name of the current folder (or current working directory). Initially, this should be the Notebook directory, which on the author's computer is:

```
c:\bill\GDA-5thEd\gdapy\Notebooks
```

Changing the working directory that is one level above NoteBooks is accomplished by the command:

```
os.chdir("../");
```

[gdapy01\_02]

The `..` signifies one folder up (which is the `gdapy` folder). Note that the pathname is quoted to signify that it is a character string (discussed further below). Changing the working directory to one that is a level below `gdapy`, such as `Notebooks`, is accomplished by

```
os.chdir("Notebooks");
```

[gdapy01\_02]

Finally, a listing of the files and subfolders in the current working directory can be created using the commands:

```
mydirlist=os.listdir();  
print(mydirlist);
```

[gdapy01\_02]

As we will discuss in more detail later in this book, these commands can be used to facilitate processing data files located in multiple folders.

### 1B.3 SIMPLE ARITHMETIC

The Python 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;  
print(c);
```

[gdapy01\_03]

evaluates the formula  $c = a + b$  for the case  $a = 4.5$  and  $b = 5.1$  to obtain  $c = 9.6$ . A semicolon at the end of the formula serves to indicate the end of the command. The final command, `print(c)`, causes Python to display the final result,  $c$ .

Note that Python variables are *static*, meaning that they persist until you explicitly delete them or exit the program. Variables created in one cell can be used by subsequent cells. At any time, the value of a variable can be examined, by printing or plotting it. The persistence of variables can sometimes lead to scripting errors, such as when the definition of a variable in a cell is inadvertently omitted, but Python uses the value defined in a previously executed cell. Although the command

```
%reset -f
```

[gdapy01\_00]

deletes all previously defined variables, it also deletes all imported modules, which makes its use cumbersome, at best.

A somewhat more complicated formula is:

$$c = \sqrt{a^2 + b^2} \text{ with } a = 6 \text{ and } b = 8 \quad (1B.1)$$

The corresponding script is:

```
a=6.0;  
b=8.0;  
c = sqrt(a**2 + b**2);  
disp(c);
```

[gdapy01\_04]

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

A final example is:

$$c = \sin\left(\frac{n\pi(x-x_0)}{L}\right) \text{ with } n = 3, x = 4, x_0 = 1 \text{ and } L = 6 \quad (1B.2)$$

The corresponding script is:

```
n=3.0; x=4.0; x0=1.0; L=6.0;  
c = sin(n*pi*(x-x0)/L);  
print(c);
```

[gdapy01\_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). Python's `math` module defines a variety of predefined mathematical constants, including `pi` (the usual mathematical constant,  $\pi$ ).

Many Python manuals, guides, and tutorials are available, both in the printed form (e.g., [Menke, 2022](#); Sundnes, 2020; VanderPlas, 2016) and on the Web (e.g., at <https://docs.python.org/3/tutorial/>, <https://numpy.org/>). The reader may find that they complement this book by providing more detailed information about Python's functionality and script writing in general.

## 1B.4 LISTS, TUPPLES, 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. Similarly, a *matrix* is just a table of numbers that is treated as unit and given a symbolic name. Both are essential to data analysis, because they allow large amounts of data to be processed as units. Most programming languages have vector and matrix data types – but not Python! That functionality is added to it by the Numpy module.

Python does, however, have two types of list-like variables that are somewhat similar to vectors. One is called the *list* and the other the *tuple*. A list of the numbers, 1.0, 7.0, 2.0 and 6.0, can be created with the command

```
mylist = [1.0, 7.0, 2.0, 6.0];
```

[gdapy01\_06]

and an individual element within the list can be accessed by *indexing* it. For example

```
b=mylist[2];
```

[gdapy01\_06]

sets `b` to the third element of the list, which is 2.0. Similarly, a tuple of the same numbers can be created with the command

```
mytuple = (1.0, 7.0, 2.0, 6.0);
```

[gdapy01\_06]

A one element tuple is specified with the notation, `(1.0, )`. An individual element within the tuple can be accessed by indexing it. For example

```
b=mytuple[2];
```

[gdapy01\_06]

sets `b` to the third element of the tuple, which is 2.0. Note that indexing starts at 0, not 1, in Python. Notwithstanding this example, lists and tuples are of limited use for representing numerical data. However, they are often used to organize character data (discussed further below). Furthermore, they are commonly used to pass information to and from methods. Because they do not have exactly the same functionality, lists and tuples are not interchangeable; many methods that expect tuples will not accept lists and vice versa.

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 (i.e., the algebra of vectors and matrices). When a vector is organized horizontally, as a row, it is called a *row-vector* and when it is 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  $1 \times 3$  row-vector,  $\mathbf{r}$ , and a  $3 \times 1$  column-vector,  $\mathbf{c}$ , are

$$\mathbf{r} = [2, 4, 6] \quad \text{and} \quad \mathbf{c} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \quad (1B.3)$$

In Python, a row-vector and a column-vector can be created with the commands

```
r = np.array([[2, 4, 6]]);
c = np.array([[1], [3], [5]]);
```

[gdapy01\_06]

This syntax is most effective when the lengths of the vectors are short. We will discuss a variety of more effective means of creating long vectors later in the text, but mention two very simple, yet extremely useful, methods here. Length- $N$  row- and column-vectors of zeros are formed by

```
r = np.zeros((1, N));
c = np.zeros((N, 1));
```

[gdapy01\_06]

Note that the quantity,  $(1, N)$ , is a tuple. Similarly, length- $N$  row- and column-vectors of ones are formed by

```
r = np.ones((1, N));
c = np.ones((N, 1));
```

[gdapy01\_06]

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

$$\mathbf{r}^T = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} \quad \text{and} \quad \mathbf{c}^T = [1, 3, 5] \quad (1B.4)$$

In Python, the transpose is computed by applying the `T` method to the vector; that is `r.T` is a column-vector and `c.T` is 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.

A column-vector has size  $N \times 1$  row-vector has size  $1 \times N$ . They are specified by the tuples,  $(N, 1)$  and  $(1, N)$ , respectively. Python also permits vectors of size  $N \times 0$ , specified by the tuple  $(N, )$ . Insofar as is possible, we do not use them in this book, for we feel that they create even more opportunity for error. However, a few important methods require them. In these cases, we convert a column-vector into a  $N \times 0$  vector using the `ravel()` method

```
c0 = c.ravel();
```

[gdapy01\_06]

Finally, we provide the function, `gda_cvec()`, that converts ordinary numbers,  $N \times 0$  vectors, row-vectors, lists and tuples into a column-vector. For example:

```
c = gda_cvec(1.0, 3.0, 5.0);
```

[gdapy01\_06]

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, 0, in the leftmost element of the row-vector and the topmost element of the column-vector. Thus,  $r_1 = 4$  and  $c_2 = 5$  in the example above. In Python the index is written inside square brackets, as in

```
a=r[0,1];
b=c[2,0];
```

[gdapy01\_06]

Note that we always use two indices, with the 0 indicating row zero of the row-vector and column zero of the column-vector.

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 integers (whole numbers). We will use bold uppercase names to denote matrices, as in

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad (1B.5)$$

In this 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

$$\mathbf{A}^T = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} \quad (1B.6)$$

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

```
A = np.array([[1., 4., 7.], [2., 5., 8.], [3., 6., 9.]]) ;
```

This syntax is most effective when the dimensions of the array are small. We will discuss a variety of more effective means of creating larger matrices later in the text, but mention two very simple, yet extremely useful, methods here. An  $N \times M$  matrices,  $\mathbf{A}$  of zeros and  $\mathbf{B}$  of ones are formed, respectively, by

```
B = np.zeros ( (N,M) ) ;  
C = np.ones ( (N,M) ) ;
```

[gdapy01\_06]

The individual elements of a matrix are denoted with two integer indices, the first indicating the row and the second the column, starting with zeros in the upper left. Thus, in the earlier example,  $A_{20} = 7$ . Note that transposition swaps indices; that is,  $M_{ji}$  is the transpose of  $M_{ij}$ . In Python, the indices are written inside square brackets, as in

```
s = M[0,2] ;
```

[gdapy01\_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

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix} \text{ and } \mathbf{B} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 2 & 0 \\ 1 & 0 & 3 \end{bmatrix}$$

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

(1B.7)

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

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

(1B.8)

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

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

[gdapy01\_06]

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,  $\mathbf{A}$ , is  $N \times K$  and the matrix  $\mathbf{B}$  is  $K \times M$ , the product  $\mathbf{P} = \mathbf{AB}$  is an  $N \times M$  matrix defined according to the rule (Fig. I.2):

$$P_{ij} = \sum_{k=0}^{K-1} A_{ik} B_{kj} \quad (1B.9)$$

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

$$Q_{ij} = \sum_{k=0}^{K-1} A_{ki} B_{kj} \quad (1B.10)$$

and so the equation corresponds to  $\mathbf{Q} = \mathbf{A}^T \mathbf{B}$  and not  $\mathbf{Q} = \mathbf{AB}$ . Matrix multiplication is not commutative (i.e.,  $\mathbf{AB} \neq \mathbf{BA}$ ) but is associative (i.e.,  $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$  and distributive (i.e.,  $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$ ). An important rule involving the matrix transpose is  $(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$  (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  $s = \mathbf{a}^T \mathbf{b}$  is a scalar number,  $s$ , and is called the *dot product* (or sometimes, *inner product*) of the vectors. It obeys the rule  $\mathbf{a}^T \mathbf{b} = \mathbf{b}^T \mathbf{a}$ . The dot product of a vector with itself is the square of its Euclidian length; that is,  $\mathbf{a}^T \mathbf{a}$  is the sum of its squared elements of  $\mathbf{a}$ . The vector,  $\mathbf{a}$ , is said to be a *unit vector* when  $\mathbf{a}^T \mathbf{a} = 1$ . The combination  $\mathbf{ab}^T$  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  $\mathbf{c} = \mathbf{Ba}$ . One interpretation of this relationship is that the matrix,  $\mathbf{B}$ , “turns one vector into another.” Note that the vectors,  $\mathbf{a}$  and  $\mathbf{c}$ , can be of different length. An  $M \times N$  matrix,  $\mathbf{B}$ , turns the length- $N$  vector  $\mathbf{a}$  into a length- $M$  vector,  $\mathbf{c}$ . The combination  $s = \mathbf{a}^T \mathbf{Ba}$  is a scalar and is called a *quadratic form*, as it contains terms quadratic in the elements of  $\mathbf{a}$ . Matrix multiplication,  $\mathbf{P} = \mathbf{AB}$ , has a useful interpretation in terms of dot products:  $P_{ij}$  is the dot product of the  $i$ th row of  $\mathbf{A}$  with the  $j$ th column of  $\mathbf{B}$ .

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

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (1B.11)$$

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

$$A_{ij} = \sum_{k=0}^{N-1} \delta_{ik} A_{kj} \quad (1B.12)$$

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 within 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  $k$ s are replaced by  $i$ s in this case). In Python an  $N \times N$  identity matrix can be created with the command

```
I = np.identity(N);
```

[gdapy01\_07]

Several different syntaxes can be used to multiply vectors and matrices in Python – but in our opinion, all of them are flawed because they have attributes that can lead to coding errors. We have chosen to use a "function call" syntax, because we believe it is the alternative least prone to error. We illustrate matrix multiplication by considering the column-vectors,  $\mathbf{a}$  and  $\mathbf{b}$ , and matrices,  $\mathbf{A}$  and  $\mathbf{B}$

$$\mathbf{a} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} \quad \text{and} \quad \mathbf{A} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 2 & 0 \\ -1 & 0 & 3 \end{bmatrix} \quad (1B.13)$$

Then, the products:

$$s = \mathbf{a}^T \mathbf{b} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}^T \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} = [1 \quad 3 \quad 5] \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} = 1 \times 2 + 3 \times 4 + 5 \times 6 = 44$$

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

$$\mathbf{c} = \mathbf{A}\mathbf{b} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} = \begin{bmatrix} 1 \times 2 + 0 \times 4 + 2 \times 6 \\ 0 \times 2 + 1 \times 4 + 0 \times 6 \\ 2 \times 2 + 0 \times 4 + 1 \times 6 \end{bmatrix} = \begin{bmatrix} 14 \\ 4 \\ 10 \end{bmatrix}$$

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

(1B.14)

correspond to

```
s = np.matmul(a.T,b);
T = np.matmul(a,b.T);
c = np.matmul(A,b);
P = np.matmul(A,B);
```

[gdapy01\_07]

In our preferred syntax, the `np.matmul(A,B)` method always is provided with the two quantities, *A* and *B*, that are to be multiplied. However, although we have chosen this syntax advisedly, readers should feel free to use one of the other supported syntaxes if they feel it works better for them.

Cases occur where the rules of matrix multiplication need to be overridden and the matrices multiplied *element-wise* (e.g., create a column-vector, **d**, with elements,  $d_i = a_i b_i$ ). Python such a functionality:

```
d = np.multiply(a,b);
```

[gdapy01\_07]

Individual elements of matrices can be accessed by specifying the relevant row and column indices, in square brackets, e.g., and `A[1,2]` is the second row, third column element of the matrix, *A*.

Ranges of rows and columns can be specified by *slicing* an array using the `:` (colon) operator – but its syntax is tricky! The range `i:j` mean the range from *i* to *j*-1 (e.g., `0:3` is from 0 to 2). For the  $3 \times 3$  matrix, **B**, the second column is `B[0:3,1:2]`, the second row is `M[1:2,0:3]` and the  $2 \times 2$  submatrix in the lower right-hand corner is `B[1:3,1:3]`. These operations are further illustrated for

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \text{and} \quad \mathbf{B} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad (1B.15)$$

$$\mathbf{x} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \text{and} \quad \mathbf{y} = \begin{bmatrix} B_{12} \\ B_{22} \\ B_{32} \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 8 \end{bmatrix}$$

$$\text{and} \quad \mathbf{z} = [B_{21} \quad B_{22} \quad B_{23}] = [4 \quad 5 \quad 6] \quad \text{and} \quad \mathbf{T} = \begin{bmatrix} B_{22} & B_{32} \\ B_{32} & B_{33} \end{bmatrix} = \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} \quad (1B.16)$$

The corresponding command are

```
x = a[0:2,0:1];
y = B[0:3,1:2];
c = B[1:2,0:3];
T = B[1:3,1:3];
```

[gdapy01\_08]

In Python, the variables,  $\mathbf{x}$ ,  $\mathbf{y}$ ,  $\mathbf{c}$ , and  $\mathbf{T}$  are not independent, but rather *views* into the larger variables,  $\mathbf{a}$  and  $\mathbf{M}$ . Consequently, if  $\mathbf{a}$  and  $\mathbf{M}$  are modified,  $\mathbf{x}$ ,  $\mathbf{y}$ ,  $\mathbf{c}$ , and  $\mathbf{T}$ , become modified, too. Views must be copied in order to become independent of the original variable

```
x = np.copy( a[0:2,0:1] );
y = np.copy( B[0:3,1:2] );
c = np.copy( B[1:2,0:3] );
T = np.copy( B[1:3,1:3] );
```

[gdapy01\_08]

Two colons can be used in sequence to indicate the spacing of elements in the resulting row-vector. For example, the expression,  $1:2:10$  is the sequence 1, 3, 5, 7, 9 and that the expression  $10:5:-1$  is the sequence 10, 9, 8, 7, 6.

Although Python's slicing syntax is very rich, we use only a very restricted subset in this book. In particular, we always specify both the start and end of the slice of every index, and we only equate views that have exactly the same shape – because our sense is that these restrictions lead to fewer errors, especially among novice script-writers,

A very useful length- $N$  column-vector, say  $\mathbf{t}$ , has elements that increment by a constant amount,  $\Delta t$ , from zero to  $(N - 1)\Delta t$ ; that is,  $t_i = i\Delta t$ . It can be formed as

```
Dt = 2.0;
t = gda_cvec(np.linspace(0,Dt*(N-1),N));
```

[gdapy01\_09]

Here, the increment,  $\Delta t$ , has been set to 2. The method, `np.linspace()` return an  $N \times 0$  vector, so its result is converted into a column-vector using `gda_cvec()`. One use of such a vector is for a *time axis*; that is, a vector that represents time increasing by regular increments.

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

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I} \quad (1B.17)$$

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  $2 \times 2$  case

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \text{and} \quad \mathbf{A}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \quad (1B.18)$$

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  $\mathbf{A}$ , denoted  $\det(\mathbf{A})$  (or sometimes  $|\mathbf{A}|$ ) is not equal to zero. The determinant of the  $2 \times 2$  matrix,  $\mathbf{A}$ , is  $\det(\mathbf{A}) = ad - bc$ . It consists of the sum of products of two elements of the matrix. The determinant of a square  $N \times N$  matrix,  $\mathbf{A}$ , consists of the sum of products of  $N$  elements of the matrix, and is given by

$$\det(\mathbf{A}) = \sum_{n_0=0}^{N-1} \sum_{n_1=0}^{N-1} \sum_{n_2=0}^{N-1} \cdots \sum_{n_{N-1}=0}^{N-1} \varepsilon_{n_0, n_1, n_2, \dots, n_{N-1}} A_{0n_0} A_{1n_1} A_{2n_2} \cdots A_{N-1, n_{N-1}} \quad (1B.19)$$

Here the quantity,  $\varepsilon_{n_0, n_1, n_2, \dots, n_{N-1}}$ , is  $+1$  when  $(n_0, n_1, n_2, \dots, n_{N-1})$  is an even permutation of  $(0, 1, 2, \dots, N-1)$ ,  $-1$  when it is an odd permutation, and zero otherwise.

In Python, the matrix inverse and determinant of a square matrix  $\mathbf{A}$  are computed as

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

[gdapy01\_10]

Here `la` is an abbreviation for the `scipy.linalg` module. In many of the formulas of inverse theory, the matrix inverse either premultiplies or postmultiplies other quantities, for instance:

$$\mathbf{c} = \mathbf{A}^{-1}\mathbf{b} \quad \text{and} \quad \mathbf{C} = \mathbf{A}^{-1}\mathbf{B} \quad \text{and} \quad \mathbf{D} = \mathbf{B}\mathbf{A}^{-1} \quad (1B.20)$$

These cases do not actually require the explicit calculation of  $\mathbf{A}^{-1}$ , just the combinations  $\mathbf{A}^{-1}\mathbf{b}$ ,  $\mathbf{A}^{-1}\mathbf{B}$  and  $\mathbf{B}\mathbf{A}^{-1}$ , which are computationally simpler. They can be computed as



```

c = la.solve(A,b) ;
C = la.solve(A,B) ;
D = la.solve(A.T,B.T) .T;

```

[gdapy01\_10]

The third equation follows from transposing  $\mathbf{D} = \mathbf{B}\mathbf{A}^{-1}$  to  $\mathbf{D}^T = [\mathbf{A}^T]^{-1}\mathbf{B}^T$ .

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  $\mathbf{A}$  is an  $N \times N$  square matrix and that it multiplies an *input* column-vector,  $\mathbf{v}$ , producing an *output* column-vector,  $\mathbf{w} = \mathbf{A}\mathbf{v}$ . 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?  
(1B.21)

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

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \quad \text{or} \quad (\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0 \quad (1B.22)$$

The trivial solution,  $\mathbf{v} = (\mathbf{A} - \lambda\mathbf{I})^{-1}\mathbf{0} = \mathbf{0}$ , is not very interesting. A nontrivial solution is only possible when the matrix inverse,  $(\mathbf{A} - \lambda\mathbf{I})^{-1}$ , does not exist. This is the case where the parameter,  $\lambda$ , is specifically chosen to make the determinant,  $\det(\mathbf{A} - \lambda\mathbf{I})$ , exactly zero, because 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  $\lambda$ , the product will contain powers of  $\lambda$  up to  $\lambda^N$ . Thus, the equation,  $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$ , is an  $N$ th order polynomial equation for  $\lambda$ . An  $N$ th order polynomial equation has  $N$  solutions, so we conclude that there must be  $N$  different proportionality factors, say  $\lambda_i$ , and  $N$  corresponding column vectors, say  $\mathbf{v}^{(i)}$ , that solve  $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ . The column vectors,  $\mathbf{v}^{(i)}$ , are called the *characteristic vectors* (or eigenvectors) of the matrix,  $\mathbf{A}$ , and the proportionality factors,  $\lambda_i$ , 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 the matrix,  $\mathbf{A}$  is symmetric, it can be shown that the eigenvalues,  $\lambda_i$ , are real and the eigenvectors are mutually perpendicular,  $\mathbf{v}^{(i)T}\mathbf{v}^{(j)} = 0$  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  $\delta_{ij}$  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  $\mathbf{V}\mathbf{V}^T = \mathbf{V}^T\mathbf{V} = \mathbf{I}$ . The eigenvalue equation  $\mathbf{A}\mathbf{v}^{(i)} = \lambda_i\mathbf{v}^{(i)}$  can then be succinctly written as  $\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{\Lambda}$ . Postmultiplying by  $\mathbf{V}^T$ , and applying  $\mathbf{V}\mathbf{V}^T = \mathbf{I}$  yields:

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$$

(1B.23)

That is, a symmetric matrix,  $\mathbf{A}$ , is completely specified by its eigenvalue and eigenvectors.

We now note an interesting property of a matrix that can be written as  $\mathbf{A} = \mathbf{B}^T \mathbf{B}$ , where  $\mathbf{B}$  is some matrix. By pre-multiplying the eigenvalue equation by  $\mathbf{v}^{(i)T}$ , and noting that  $\mathbf{v}^{(i)T} \mathbf{v}^{(i)} = 1$ , we obtain  $\lambda_i = \mathbf{v}^{(i)T} \mathbf{A} \mathbf{v}^{(i)}$ . Substituting  $\mathbf{B}^T \mathbf{B}$  for  $\mathbf{A}$ , we have  $\lambda = \mathbf{v}^{(i)T} \mathbf{B}^T \mathbf{B} \mathbf{v}^{(i)} = [\mathbf{B} \mathbf{v}^{(i)}]^T \mathbf{B} \mathbf{v}^{(i)}$ . Consequently,  $\lambda_i$  is the squared length of the vector,  $\mathbf{B} \mathbf{v}^{(i)}$ , and cannot be negative. In this case, the matrix,  $\mathbf{A}$ , is said to be *non-negative definite*.

In Python, the diagonal matrix of eigenvalues,  $\mathbf{\Lambda}$ , and matrix of eigenvectors,  $\mathbf{V}$ , of a symmetric matrix,  $\mathbf{A}$ , are computed as

```
LAMBDA, V = la.eigh(A);
```

[gdapy01\_10]

Here  $\mathbf{\Lambda}$  is an  $N \times 0$  vector of eigenvalues.

## 1B.5 MATRIX DIFFERENTIATION

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,  $d\mathbf{v}/dx$ . Such a derivative represents the fact that the vector changes from  $\mathbf{v}$  to  $\mathbf{v} + \Delta\mathbf{v}$  as the independent variable changes from  $x$  to  $x + \Delta x$ . Note that the resulting change,  $\Delta\mathbf{v}$ , is itself a vector. Derivatives are performed element-wise; that is,

$$\left[ \frac{d\mathbf{v}}{dx} \right]_i = \lim_{\Delta x \rightarrow 0} \frac{v_i(x + \Delta x) - v_i(x)}{\Delta x} = \frac{dv_i}{dx} \quad (1B.24)$$

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

$$\frac{\partial v_i}{\partial y_j} \quad (1B.25)$$

represents the change in the  $i$ th component of  $\mathbf{v}$  caused by a change in the  $j$ th component of  $\mathbf{y}$ . Frequently, we will need to differentiate the linear function,  $\mathbf{v} = \mathbf{A}\mathbf{y}$ , where  $\mathbf{A}$  is a matrix, with respect to  $\mathbf{y}$ :

$$\frac{\partial v_i}{\partial y_j} = \frac{\partial}{\partial y_j} \sum_{k=0}^{N-1} A_{ik} y_k = \sum_{k=0}^{N-1} A_{ik} \frac{\partial y_k}{\partial y_j} = \sum_{k=0}^{N-1} A_{ik} \delta_{kj} = A_{ij} \quad (1B.26)$$

Since the components of  $\mathbf{y}$  are assumed to be independent, the derivative,  $\partial y_k / \partial y_j$ , is zero except when  $k = j$ , in which case it is unity, which is to say  $\partial y_k / \partial y_j = \delta_{kj}$ , where  $\delta_{kj}$  is the Kronecker delta. Thus, the derivative of the linear function,  $\mathbf{v} = \mathbf{A}\mathbf{y}$ , is the matrix,  $\mathbf{A}$ . This relationship is the vector analog to the scalar case, where the derivative of the linear function  $v = ay$  is the constant,  $a$ .

## 1B.6 CHARACTER STRINGS AND LISTS

The numerical variable,  $x$ , created with the command, `x=2.5`, has a close relationship to variables that are encountered in elementary mathematics. They are very useful to performing the calculations essential to of data analysis. However, they are not very useful during file manipulation, which is usually a vital prerequisite to data analysis. Furthermore, the results of data analysis are most understandable if described in a combination of words and numbers – and not just numbers.

Python has a type of variable, called the *string variable*, the value of which can be set to a *character string* (sequence of alpha-numerical characters). For instance, a string variable with name, `myfile`, and with value, `"mydata.txt"`, is created by

```
myfile="mydata.txt";
```

[gdapy01\_11]

Note that the value of the variable is quoted to prevent Python from trying to interpret it as a command.

Combining numerical variables with character strings can be very useful, both in file manipulation, where a sequence of filenames may contain a number that sequentially increments, like `"myfile_1.txt"`, `"myfile_2.txt"`, etc., or when one wants to display the value of a variable together with some text that explains it; e.g., display the sentence `"position x is 10.40 meters"`.

The `%` string formatting operator is used to create a character string with a value that includes both text and the value of a variable. Thus, for instance,

```
i=1;
myfilename = "myfile_%d.txt" % (i);
```

[gdapy01\_11]

creates a character string, `myfilename`, with value `"myfile_1.txt"`, where the 1 is taken from the variable, `i`. Similarly,

```
x=10.40;
mysentence = "position x is %.2f meters" % (x);
```

[gdapy01\_11]

creates a character string, `mysentence`, with value "position x is 10.40 meters", where the 10.4 is taken from the variable, `x`.

Unfortunately, the `%` operator is fairly inscrutable, and we refer readers to the Python help pages for a detailed description. Briefly, it uses a *format string* (in this case "position x is %.2f meters"), which contains *placeholders* that start with the character, `%`, to indicate where in the character string the value of the variable should be placed. Thus, "myfile\_%d.txt" % (`i`) creates the character string 'myfile\_1.txt' (because the value of `i` is 1). The `%d` is the placeholder for an integer. It is replaced with 1 the value of `i`. When the variable can have fractional, as contrasted to integer, values, the floating-point placeholder, `%f`, is used instead. Thus, "position x is %.2f meters" % (`x`) creates the character string "position x is 10.40 meters" (because the value of `x` is 10.4). The precision to which `x` is to be written is indicated by the `.2` inside the `%.2f`; in this case, two significant digits.

The command, `print(mysentence)`, can be used to display the character string.

Lists of character strings are very useful when automating complicated tasks. For instance,

```
mycolorlist = ["red", "crimson", "chartreuse", "teal"];
[gdapy01_11]
```

Individual elements are accessed via standard indexing; that is, `mycolorlist[3]` is element number three containing the character string "teal".

```
myfavoritecolor = mycolorlist[3];
[gdapy01_11]
```

## 1B.7 LOOPS

Python 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 = np.array( [[1, 2, 3], [4, 5, 6], [7, 8, 9]] );
for i in range(3):
    a[i,0] = M[i,i];
[gdapy01_12]
```

executes the  $a(i) = M(i, i)$  formula three times, each time with a different value of `i`, according to the specifications of the `range()` function. Here, `range(3)` means `i=0`, `i=1` and `i=2`. The net effect is to copy the diagonal elements of the matrix, `M`, to the vector, `a`, that is,  $a_i = M_{ii}$ . Indentation is a crucial part of the `for`-loop syntax, for it indicates which statements are inside the loop. Un-indented 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, 2, 3; 4, 5, 6; 7, 8, 9];
for i in range(3):
    for j in range(3):
        N[i,3-j] = M[i,j];
```

[gdapy01\_13]

copies the elements of the matrix,  $M$ , to the matrix,  $N$ , but reverses the order of the elements in each row, that is,  $N_{i,3-j} = M_{i,j}$ . Loops are especially useful in conjunction with *conditional commands*. For example,

```
a=[1.0, 2.0, 1.0, 4.0, 3.0, 2.0, 6.0, 4.0, 9.0, 2.0, 1.0, 4.0];
for i in range(12):
    if ( a[i,0] >= 6.0 ):
        b[i,0] = 6.0;
    else:
        b[i,0] = a[i,0];
```

[gdapy01\_14]

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

A purist might point out that Python 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 = np.diag(M);
N = np.fliplr(M);
b=np.copy(a); b[a>6]=6;
```

[gdapy01\_15]

The first two formulas are very simple, but rely upon the Python methods `np.diag()` (for “diagonal”) and `np.fliplr()` (for “flip left-right”), whose existence we have not hitherto mentioned. The third formula, which used logical addressing, requires further explanation. The vector,  $a$ , is first copied to  $b$ . Then just those elements of  $b$  that are greater than 6 are reset to 6, using a technique called *logical addressing*. The expression  $a>6$  returns a vector of true’s and false’s, depending upon whether the elements of the column-vector  $a$  satisfy the inequality or not. The command `b[a>6]=6` resets an element of  $b$  to 6 only when the value is true.

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 Python 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 Python. Your repertoire of techniques will grow as you become more practiced.

## 1B.8 LOADING DATA FROM A FILE

Python can read files with a variety of 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 57 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 -0.06
1967 -0.02
... ..
```

The data are read into to Python as follows:

```
D = np.genfromtxt("../Data/global_temp.txt", delimiter="\t");
N,M = np.shape(D);
t = np.copy( D[0:N,0:1] ); # time in column 1
d = np.copy( D[0:N,1:2] ); # temperature in column 2
```

[gdapy01\_16]

The `np.genfromtxt` function reads the data into a 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 Notebooks folder while the data are in the Data folder (which is "up and over" from Notebooks). The filename is surrounded by quotes to indicate that it is a character string. 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.

## 1B.9 WRITING DATA TO A FILE

Python can write files with a variety of formats, but we start here with the simplest and most common, the text file. Suppose that we wanted a file containing global temperature anomaly in degrees Fahrenheit, using the fact that  $1^{\circ}\text{C} = 1.8^{\circ}\text{F}$ . Starting with time,  $t$ , and temperature anomaly,  $d$ , from the previous section, we use the command

```
cf = 1.8;
dF = cf*d;
DF = np.concatenate((t,dF),axis=1);
newfilename="../TestFolder/global_temp_in_F.txt";
np.savetxt(newfilename,DF,delimiter="\t");
```

[gdapy01\_16]

The data is converted to degrees Fahrenheit by multiplying by the conversion factor,  $cf$ . A single matrix,  $DF$ , is formed containing both time and temperature, by size-by-size concatenating the time column-vector,  $t$ , and the Fahrenheit temperature data,  $dF$ . The `np.savetxt()` method writes the matrix,  $DF$ , to a file (in this case, `newfilename`) with a tab (denoted `"\t"`) between the columns of data. Note that the new filename, `'global_temp_in_F.txt'` is chosen to indicate that the units have been changed.

## 1B.10 PLOTTING DATA

Python'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 (Fig. 1B.2). was created with the Pyplot (`plt`) commands:

```
fig1 = plt.figure(1,figsize=(8,4)); # smallish figure
ax1 = plt.subplot(1, 1, 1);         # only one subplot
plt.plot(t,d,"k-");                 # plot d(t) with black line
plt.plot(t,d,"ro");                 # plot d(t) with black line
plt.xlabel("time (years)");          # label time axis
plt.ylabel("temperature (deg C)");   # label data axis
plt.show();                         # display plot
```

[gdapy01\_16]

The `plt.figure()` method create a new figure window, labels it as Figure 1 and sets its size. Although Pyplot allows several subplot within the figure, we specify that there be only in the call to the `plt.subplot()` method. The two `plt.plot(...)` commands plot the time-axis,  $t$ , and temperature data,  $d$ , in two different ways: first as a black line (indicated by the `"k-"`) and the second with red circles (the `"ko"`). Finally, the horizontal and vertical axes are labeled using the `plt.xlabel()` and `plt.ylabel()` commands. The final command, `plt.show()`, indicates that the plot is finished and can be displayed

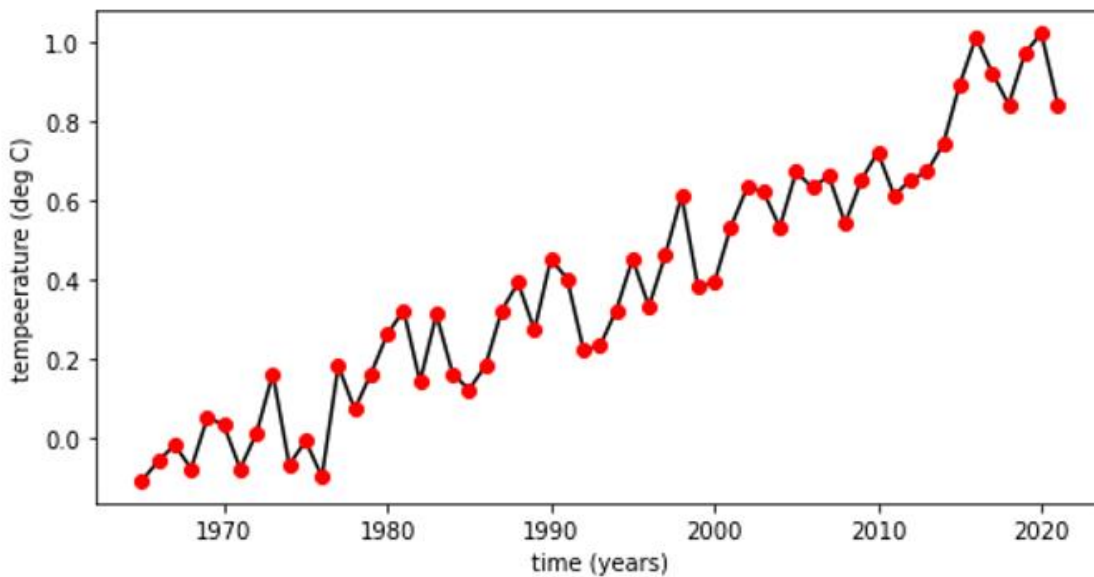


Fig. 1B.2 Python plot of deviations of global temperature from average for the time period, 1965–2021. Data from Hansen, J., Ruedy, R., Sato, M., Lo, K., 2010. Global surface temperature change. *Rev. Geophys.* 48, RG4004. doi:10.1029/2010RG000. Script gdapy01\_16.

## References

- Hansen, J., Ruedy, R., Sato, M., Lo, K., 2010. Global surface temperature change. *Rev. Geophys.* 48RG4004.
- Menke, W., 2022, *Environmental Data Analysis with MATLAB® or Python*, Third Edition, Elsevier, Oxford, 444 pp., ISBN: 978-0-323-95576-8.
- Part-Enander, E., Sjoberg, A., Melin, B., Isaksson, P., 1996. *The MATLAB® Handbook*. Addison-Wesley, New York. 436 pp., ISBN: 0201877570.
- Pratap, R., 2009. *Getting Started with MATLAB®: A Quick Introduction for Scientists and Engineers*. Oxford University Press, Oxford. 256 pp., ISBN 978-0190602062.
- Sundnes, J., 2020, *Introduction to Scientific Programming with Python*, Springer International (Switzerland), 147 pp., ISBN: 978-3-030-50355-0.
- VanderPlas, J., 2016, *Python Data Science Handbook: Essential Tools for Working with Data*, O'Reilly Media, Sebastopol, California (USA), 517 pp., ISBN 978-1-491-91205-8

## Non-Print Items



## Abstract

This chapter helps the reader get started with using the MATLAB® and Python software environments. Two substantially parallel tutorials are provided, one for MATLAB® and the other Python. The tutorial introduces the reader to the process of writing scripts and describes the operations and functions most important to data analysis. Basic linear algebra (the mathematics of vectors and matrices), which is so important in inverse theory methods, also is systematically reviewed. This review includes both the basic topics of the addition, subtraction, and multiplication of vectors and matrices and more advanced topics, such as matrix inverses, eigenvalues and eigenvectors, and derivatives.

Keywords: MATLAB®, Python, Script, Vector, Matrix, Transpose, Inverse, Algebraic eigenvalue problem, Matrix derivative