Text and data formatting 1

An attractive document contains well structured and sensibly formatted sentences, paragraphs, tables, and other elements. To create a document in a computer, we type words, characters, punctuation and other symbols in an electronic file, and then insert *tags* implementing formatting. In practice, most documents are created using advanced word editing programs (applications). Formatting is typically enforced by selecting (highlighting) text with the computer mouse and then clicking on embellishing buttons to underline, set in bold face, change the font size, or implement a wide range of other features offered in menus. The application automatically generates formatting tags that are implicitly contained in the document but are not displayed on the computer screen, unless optionally requested.

Different applications employ different tagging systems for implementing formatting instructions. A tagging system is also known as a *markup language.* The characterization of a tagging system as a *language* is not entirely appropriate. A computer language conveys instructions that implement deliberate procedures, such as algorithms, whereas a spoken language conveys content (data) and form (markup). Although unformatted data are useful, markup without data comprises an empty form. Nevertheless, the terminology *markup language* is broadly accepted to indicate a tagging system as standard semantics.

Reviewing two popular markup languages, *latex* and *html*, naturally leads us to the concept of the extensible markup language (*xml*), where data are presented *and* described using tags of our choice in a self-contained document. The information contained in an *xml* file can be read by a person or processed by a computer application written in a computer language of choice. The data can be parsed and modified in some desirable fashion, or else imported into a computer code for use in professional, scientific, and engineering applications. In fact, we will see that *xml* provides us with a framework for implementing computer instructions and formatting programming components. The salient features of *xml* and its relevance in scientific computing will be reviewed in this chapter.

1.1 Text formatting with latex and html

Two important markup languages (tagging systems) are the *latex* markup language used for typesetting documents, and the hypertext markup language (*html*) used for displaying documents in an Internet browser.* *Latex* was conceived by computer science Professor Donald Knuth and first released in 1978 with scientific document preparation in mind. *Html* was developed at the dawn of the Internet in the late 1980s with electronic document communication in mind. In spite of differences in their intended usage, the two languages have a similar structure and a parallel design. Contemplating the similarities and differences between these two languages naturally leads us to the concept of *xml* as a generalized framework.

1.1.1 Latex

A *latex* tag is a keyword preceded by the backslash (\). Square and curly brackets following the keyword enclose data and parameters. A perfectly valid complete *latex* file entitled *sample.tex* may contain the following lines:

```
\documentclass[11pt, letter]{article}
\begin{document}

    \textit{This sentence is set in italic}

\end{document}
```

The first line defines the size of the standard font (eleven points) to be used in the document, the dimensions of the paper where the text will be printed (letter-sized paper), and the type of document to be produced (article). If we were writing a book, we would have entered *book* instead of *article* in the first line. Document classes are available for scientific papers and seminar presentations.

The beginning of the *latex* document is declared in the second line of the *sample.tex* file. An empty line was inserted for visual clarity after this declaration. To typeset a sentence in italic, we have enclosed it in curly brackets (`{}`) and appended it to the italicizing tag `\textit` in the fourth line. If we wanted to typeset the same sentence in boldface, we would have used the `\textbf` tag. For typewriter face, we would have used the `\texttt` tag. Finally, we mark the end of the document in the last line of the *latex* file.

Equations, graphics, and other elements

A variety of *latex* tags are available for formatting text and composing tables and equations, as explained in books, manuals, and *web* tutorials. For example,

*The capitalization of the *Internet* indicates the word wide web (*web*), as opposed to an arbitrary network.

to typeset the equation

$$E = \frac{a+b}{c+d} \qquad , \tag{1.1}$$

including an equation number, we use the following lines:

```
\begin{equation}
  E = \frac{a+b}{c+d}
\end{equation}
```

where *frac* stands for fraction and the rest of the tags have obvious meanings. In fact, this book was typeset in *latex* under the *ubuntu* operating system, and the preceding lines were used verbatim in the source file.

Figures and graphics elements, simple tables, colored tables, long tables, floating schematics, and other components can be easily imported or typeset in a *latex* document. A key idea is the concept of a local environment entered by the `\begin{...}` tag and exited by the complementary `\end{...}` tag, where the three dots represent an appropriate keyword. Examples are the equation and figure environments.

Human readable code

An ordinary person can easily extract and understand the information contained between a *latex* markup tag and its closure. This means that a *latex* document is human readable even when it contains involved equations and other advanced elements, such as tables, schematics, and figures. Graduate students typically learn the basic rules of *latex* typesetting in a few days based on a handed-down template. Experienced *latex* users are able to edit a *latex* document while mentally processing and simultaneously interpreting the typesetting tags and visualizing the final product in their minds. Computer programming experience is not necessary for typesetting a *latex* document. Typing skills and basic deductive ability are the only prerequisites.

Latex tag processing (interpretation)

When and how are the *latex* tags interpreted to produce the final typeset document, such as that appearing on the pages of this book? Once a *latex* source file has been composed, it must be supplied to a *latex* processor. The processor is a program (application) that receives the source *latex* file and produces a binary device-independent file (*dvi*) that can be viewed on a computer terminal using a file reader. In turn, the *dvi* file can be converted into a postscript file (*ps*) or portable document format file (*pdf*) and sent to a printer.* Direct conversion of a *latex* source document into a *pdf* file is possible using a suitable application.

*Postscript is a computer language developed for high-quality printing. Postscript interpreters are embedded in postscript printers.

Command-line processing

To process a *latex* file entitled *myfile.tex* and produce the typeset text, we open a terminal (command-line window) and launch the *latex* processor by issuing the statement:

```
$ latex myfile.tex
```

followed by the ENTER keystroke, where the dollar sign ($) is a system prompt. The processor generates a *dvi* file named *myfile.dvi* containing the processed *latex* file. To convert the *dvi* file into a postscript (*ps*) file named *myfile.ps*, we run the *dvips* application by issuing the statement:

```
$ dvips -o myfile.ps myfile.dvi
```

followed by the ENTER keystroke. To convert the postscript file into a portable document format (*pdf*) file named *myfile.pdf*, we may use the *ps2pdf* (*ps* to *pdf*) application by issuing the statement:

```
$ ps2pdf myfile.ps
```

followed by the ENTER keystroke. The application generates a file named *myfile.pdf* in the directory of the source code.

Other methods of producing a *pdf* file from a *latex* file are available. For example, we may use the *pdflatex*, *dvipdf*, or *dvipdfm* applications.

Markup or programming language?

Is *latex* a tagging system (markup language), a programming language, or both? The absence of predefined data structures, the inability to perform arithmetic operations and string manipulations, and the importance of textual content suggest that *latex* is a markup language. As a rule of thumb, a person who is skillful in a *bona fide* computer language can easily learn another *bona fide* computer language. Regrettably or fortunately, depending on the point of view, a person who is proficient in *latex* cannot transition directly to *fortran* or C++. However, typesetting instructions are implemented in a *latex* document and a dedicated processor must be used to implement the *latex* tags and produce the final document. It is fair to say that *latex* is a markup language in some ways, and a lightweight programming language in other ways.

1.1.2 Html

The hypertext markup language (*html*) is used for writing *web* documents that can be processed and displayed in the window of a *web* browser. *Html* tags are keywords enclosed by pointy brackets (<>), also called angle brackets. A perfectly valid complete *html* file may contain the following lines:

```
<html>
    <i>This sentence is set in italic</i>
</html>
```

where `<i>` and `</i>` is a pair *html* italicizing tags enclosing data. The forward slash (`/`), simply called a slash, marks the closure of a previously opened tag.

Interpretation

Html interpreters and viewers are embedded in *web* browsers. The tag `<html>` at the beginning of an *html* file launches the *html* interpreter, and its closure `</html>` marks the end of the data to be processed by the *html* interpreter. To interpret an *html* file and view the processed information contained in the file, we may *open* the file by selecting its name in the drop-down *File* menu of a *web* browser. Alternatively, an *html* file can be accessed from an Internet *web* server. To print the processed *html* file displayed in the window of a *web* browser, we may use the browser's printing services typically offered in the *File* drop-down menu.

A wealth of formatting features

Over one-hundred *html* tags are available at the present time, as explained in *html* books and *web* tutorials. Pictures, graphics, tables, music files and videos can be embedded easily in an *html* document using appropriate tags. Processing instructions (PIS) in other programming languages, such as *java* or *javascript*, can be included for the purpose of manipulating or receiving data. For the reasons discussed previously for *latex*, *html* can be regarded both as a tagging system and a lightweight programming language.

Mathml

Equations can be typeset in an *html* document using the mathematical markup language (*mathml*). The following *mathml* code embedded in an *html* document displays equation (1.1) in the window of a *web* browser:

```
<html>
  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <mi>E=</mi>
    <mfrac>
      <mrow>
        <mi>a</mi>
        <mo>+</mo>
        <mi>b</mi>
      </mrow>
      <mrow>
        <mi>c</mi>
        <mo>+</mo>
        <mi>d</mi>
      </mrow>
    </mfrac>
  </math>
</html>
```

The `<mi>` tag stands for *mathematical identifier* and the `<mo>` tag stands for *mathematical object*. The `<math>` tag in the second line defines an *xml* environment identified by its namespace (*xmlns*), as discussed in Chapter 2. All tags are accompanied by their closure indicated by a forward slash (`/`) placed at appropriate places to ensure proper nesting.

It is clear that typesetting equations in *mathml* is much more cumbersome than in *latex*. This comparison underscores the superiority of *latex* in technical typesetting and explains why *latex* is a standard choice in technical publishing and academe.

1.1.3 Latex compared to html

Latex and *html* share a number of important features. Both languages are human readable, and both languages convey information (content) and implement presentation (appearance). *Latex* is superior to *html* in that it offers an impressive menu of tagging options, including mathematical equations, tables, and an assortment of special characters and symbols. Almost anything that can be done in *html* can be done in *latex*, but not necessarily *vice versa*. The typesetting quality of *latex* is superior to that of *html*.

Latex is unforgiving

There are penalties to be paid. *Latex* is less forgiving than *html* in that, if a tagging error is made, the processing of the source file will be abandoned by the processor and warnings will be issued and recorded in a log file. In contrast, *html* forgives misprints and ignores minor and sometimes major tagging errors. For example, if a new paragraph is forced to open in an *html* document, the preceding paragraph does not have to be closed. Even if the structure of various tags in an *html* document is arbitrary and nonsensical, information will still be processed and displayed to the best of the ability of the *html* interpreter.

Latex and html compilers are free software

A variety of *web* browsers are freely available for standard operating systems. A *latex* document cannot be interpreted by a *web* browser but requires a dedicated processor. Free *latex* processor and authoring applications are available in a variety of operating systems. The complete *latex* compiler, including an assortment of add-on packages that provide additional functionality, is included in most *linux* distributions, including *ubuntu*. Typesetting a document in *latex* guarantees longevity and compatibility with future media technology.

Latex to html and back

A *latex* document can be converted into an *html* document using a suitable translation program (application), such as *latex2html* or *tth.* Mathematical equations and figures are treated in special ways. A scientist, engineer, or technical typesetter may write a *latex* document, convert it into an *html* document, and post it on the Internet for direct viewing. The ability to convert a *latex* document into an *html* document hinges on the consistent use of corresponding formatting tags. Although it is also possible to convert an *html* document into a *latex* document, in most cases the effort is a mere academic exercise.

Exercises

1.1.1 *Latex, html, and mathml*

(*a*) Write a *latex* file that prints your favorite color in boldface. (*b*) Repeat for an *html* file. (*c*) Write an *html* file that prints the equation $E = mc^2$ using *mathml.*

1.1.2 *Latex, html, and mathml tags*

(*a*) Prepare a list of sixteen *latex* tags of your choice. (*b*) Repeat for *html* tags. (*c*) Repeat for *mathml* tags.

1.2 Formatting with xml

In both *latex* and *html*, the menu of available tags is determined by the authors of the respective markup language parser and interpreter.* In the case of *latex*, the interpreter is the *latex* processor. *Html* interpreters are embedded in *web* browsers.

In writing a *latex* or *html* document, we must strictly adhere to standard structures and conventions decided by others. For example, the tag `\textlatin` is meaningless and will produce errors in *latex*, and the tag `<puppy>` is meaningless and will be ignored in *html.* If tags for typesetting equations were not available in *html*, the language would be of limited use to quantitative scientists and engineers.

Computer programmers, document typesetters, and data-entry operators welcome the opportunity of using formatting tags that best suit their practical or creative needs in different applications. The generalized framework implemented by the extensible markup language (*xml*) allows us to employ tag names and structures that best describe the components of a document of interest and the individual pieces of data contained in a database. Of equal importance,

*In computer science, *parsing* describes the process of breaking down a chain of words into elementary pieces (tokens) and applying a set of rules to recognize instructions and extract attributes and parameters.

xml facilitates the unique identification of similar pieces of data so that targeted information can be readily extracted from a database, as the need arises.

To illustrate the concept of data identification, let us assume that the title of a book chapter is set in bold face in an *html* document using the `<b>` tag and its closure `</b>`. Because the `<b>` tag could also be used to emphasize the Isbn number, we are unable to identity the book title by searching through the document for the bold face tag in the absence of further identifying information. In an *xml* document, tags that unambiguously and uniquely define the beginning, end, and title of each chapter are employed.

Xml keywords are arbitrary

Xml allows us to use any desired, but sensible and consistent, tagging system that best suits our needs. Arbitrary and unrelated names regarded as user-defined keywords can be assigned to the individual tags, and optional qualifiers known as attributes can be employed. A typical *xml* document contains sequences of nested tags describing and evaluating a multitude of objects and structures without any constraints, apart from those imposed by basic *xml* grammar, as discussed in Chapter 2.

These features render *xml* a meta-language; a better term would be a flexible language; an even better term would be a flexible tagging system. However, in all honestly, *xml* is not a language, but an *adaptable data formatting system* that can be interpreted by a person or processed in unspecified ways by a machine. *Xml* appears as a language only when compared to a spoken world language that evolves in response to new concepts, terms, and emerging communication needs.

Flowers

To record and describe a flower in an *xml* document, we may introduce the flower and list its properties:

```
<flower>
  <kind>rose</kind>
  <color>red</color>
  <smell>captivating</smell>
</flower>
```

Like *html* tags, *xml* tags are enclosed by pointy brackets (`<>`), also called angle brackets. As in *html*, a closing tag arises by prepending to the name of the corresponding opening tag a forward slash (`/`), simply called a slash.

In our example, the flower is an object, identified as an *xml* element, whose properties are recorded in a nested sequence of tags, identified as children elements with suitable names. The opening tag `<flower>` is accompanied by the corresponding closing tag `</flower>` to indicate that the flower description has ended. All other tags inside the parental flower tag open and close in similar ways. Three pieces of data are provided and simultaneously described in this document: rose, red, and captivating. Other flowers can be added to describe a bouquet in a living room or flower shop.

In an alternative representation, the flower of interest can be described in terms of attributes in a single line as:

```
<flower kind="rose" color="red" smell="captivating" />
```

In this formulation, `kind`, `color`, and `smell` are attributes evaluated by the contents of the double quotes constituting the data. For reasons of scalability and ease of retrieval, the expanded representation in terms of nested tags is preferred over the attribute representation, as discussed in Chapter 3.

Data organization

The flower example illustrates two important features of the *xml* formatting system: a high level of organization, and unique data identification. A *latex* document also exhibits a high level of organization. However, *latex* and *html* are primarily concerned with data presentation in printed or electronic form. In these restricted tagging systems, information retrieval is only an afterthought.

Data formatting with xml does not require programming experience

Programming language skills are not necessary for composing and editing an *xml* document containing data. The document can be written by a person in isolation following basic *xml* grammar, as discussed in Chapter 2, with no reference whatsoever to conventions imposed by others. Common sense, editorial consistency, typing skills, and a general plan on how the data will be organized are the only prerequisites.

Text (ascii) files

To generate an *xml* document, we write a *text* file, also called an *ascii* file, using a word editor of our choice, such as *nano*, *pico*, *emacs*, *vi*, or *notepad*. The names of the formatting tags can be words of the English language or any other spoken or fictitious language. Advanced word processors, such as *LibreOffice Writer*, can be used, but the file must be saved as an unformatted *text* or *ascii* document.

Ascii is an acronym of the American standard code for information interchange. A *text* or *ascii* file contains a sequence of integers, each recorded in 7 binary digits (*bits*) in terms of its binary representation. The integers represent characters, including letters, numbers, and other symbols, encoded according to the *ascii* convention discussed in Appendix A. Characters outside the *ascii* range may also be used according to generalized character encoding systems, as discussed in Section 2.4.

Text or *ascii* files contain long binary strings consisting of 0 and 1 digits describing integers that can be decoded by a person or application with reference to the *ascii* map. An example is the string

```
0100110 0111000 1101100 0101110 ···
```

consisting of a chain of seven bits. In contrast, a *binary file* contains binary strings encoding machine instructions, data formatting, and other information pertinent to a specific application, such as a spreadsheet. A binary file can be opened and processed only by its intended application.

Fasolia and keftedakia

If we want to record a sentence in italic in an *xml* document describing a delicious meal (beans and meatballs), we may write:

```
<set_in_italic> Fasolia and keftedakia </set_in_italic>
```

This line can be part of an *xml* file describing a dinner menu. The underscore (_) is used routinely to connect words into a sentence that enjoys visual continuity as an uninterrupted character string. Who will interpret the italicizing tags is of no interest to the reclusive author of this *xml* document. In contrast, mandatory italicizing tags must be employed in *latex* and *html*documents, as discussed in Section 1.1.

Nuts

Following is a complete *xml* file containing a prologue (first line) and a list of nuts:

```
<?xml version="1.0"?>
<pantry>

  <nut>peanuts</nut>
  <nut>macadamia</nut>
  <nut>hazelnuts</nut>

</pantry>
```

The prologue is an *xml* document declaration inserted in most *xml* files, as discussed in Section 2.4. It is clear that the data contained in this file represent three nuts found in a pantry. The tag `<pantry>` defines the *root element* of this *xml* document, enclosing all other children elements implemented by opened and closed tags. Note that all tags close at expected places. Additional nuts can be easily removed, added, or replenished. Further information on how many nuts of each type are available could have been included as properties represented by children elements or element attributes. The question of what to do with these nuts is outstanding.

Equations

A complete *xml* document describing equation (1.1) may contain the following lines:

```
<?xml version="1.0"?>
<equation>

  <left_hand_side>
    E
  </left_hand_side>

  <right_hand_side>
    <fraction>
      <numerator>
        a+b
      </numerator>
      <denominator>
        c+d
      </denominator>
    </fraction>
  </right_hand_side>

</equation>
```

The tag `<equation>` defines the root element of this *xml* document, enclosing all other children elements. This *xml* document can be converted into an equivalent *latex* or *mathml* document manually or with the help of a suitable computer program (application).

An xml document presents and describes data

An extremely important feature of *xml* is that data, including numbers, objects, items, instructions, and statements of a computer programming language, are not only *presented*, but also *described* in a chosen language of the world, such as English or Portuguese. In fact, the data can be written in one language, and the tags can be written in another language. This duality is intimately related to the desirable property of plurality in a human readable database or code.

Data trees

A conceptual tree can be built expressing precisely and unambiguously a hierarchy of information in an *xml* document. One example is a tree describing all parts of a car arranged in branches identified by tags named `engine`, `wheels`, `cabin`, and other components. Another example is a tree describing the elements employed in a finite-element code. A third example is a tree describing orthogonal polynomials, distinguished by their domain of definition and weighing function. A fourth, less apparent but more intriguing example, is a tree of statements of a computer programming language implementing a numerical algorithm.

Data parsing

We have mentioned that data, instructions, and other information encapsulated in an *xml* document can be read and understood by a person or else inspected (parsed) and processed by a computer program (application) written in a language of our choice, such as *fortran*, C, C++, *java*, *perl* or *python*. *Xml* parsers are available as modules of advanced computer languages, including C++, *java*, *javascript*, *php*, *perl*, and *python*. Relevant procedures for selected languages will be illustrated in Chapter 5.

An xml document must be well-formed

Although the names of the tags employed in an *xml* document can be arbitrary, the document itself must be well-formed.

One requirement is that an opening tag defining an element, such as `<nut>`, be accompanying by the corresponding closing tag `</nut>` to indicate the end of a nut. The closing of a tag is mandatory even when it appears unnecessary, as in the case of a tag forcing a line break in a word document.

For an *xml* element to be well-formed, tags must be properly nested, as discussed in Chapter 2. This means that two nuts may not overlap, that is, one nut may not cross over another nut.

However, these restrictions are mild and reasonable constraints imposed to ensure successful data parsing, prevent confusion, and avoid misinterpretation. Identifying grammatical errors in an *xml* document is straightforward. Only when an *xml* document is exceedingly long is the help of a computer processor (*xml* debugger) necessary. In contrast, identifying bugs in a computer code can be tedious and time consuming. In some cases, it may take a few hours to write a computer code and then weeks to remove fatal or benign errors.

Visualization in a web browser

We can open an *xml* file with an *xml* compliant *web* browser through the *Open File* option of the *File* drop-file menu. In the absence of processing instructions (Pi) embedded in the *xml* file, as discussed in Chapter 2, the *xml* data tree will be visualized with a $\pm$ mark on the left margin. Clicking on this mark with the mouse will reveal or hide the branches of the *xml* tree. A warning will be issued if the *xml* file is not well-formed.

Xhtml

Roughly speaking, an *xhtml* document is a well-formed *html* document where all tags are written in lower case. Only established *html* tags can be used in an *xhtml* document, that is, improvised tags cannot be employed. An *xhtml* document is also an *html* document, but an *html* document is not necessarily an *xhtml* document. An *html* document can be certified as an *xhtml* document by a suitable program called an *xhtml* validator.

Is a latex or xhtml document an xml document?

Although *latex* and *xhtml* documents must be well-formed, the tagging system lacks the necessary flexibility and extensibility that is the hallmark of the *xml* layout. Most important, arbitrary tags and tag attributes cannot be added at will.

However, any *xml* document that is bound to an agreed convention also suffers from inflexibility and inextensibility. This observation raises a concern as to whether *xml* lives up to its advertised quality as a genuine meta-language in practical applications tied to industry standards. This well-founded skepticism will be revisited throughout this book.

Xml authoring tools

Xml editors are word editors (applications) with a graphical user interface (GUI) that highlights with color the tagging tree of an *xml* document. The objective is to help ensure that the document is well-formed by approving or dismissing the tagging structure employed. These authoring tools are helpful but not necessary for composing an *xml* document.

Computer language implementations

We have discussed data formatting and physical or abstract object description in the *xml* framework. In fact, *xml* can be used to implement computer language instructions. Tags with attributes in an *xml* compliant programming language play the role of logical and other constructs. Examples are the *for* loop in C++ and the *Do* loops in *fortran*. Details will be given in Section 1.3.

Constraints

Constraints on the tagging system of an *xml* document arise only when the *xml* data are written to be sent to another person or application. The goal of these constraints is to ensure that sender and receiver agree on the amount and type of information contained in an *xml* document of interest to both. Not surprisingly, *xml* formatting becomes relevant to scientific computer programming only with regard to instruction syntax and formatting of input/output (I/O).

Exercises

1.2.1 *Tools in a shop*

Write an *xml* document that lists the tools in a carpentry shop along with other pertinent tool information.

1.2.2 *Books in a shop*

Write an *xml* document that lists the books in a bookshop along with titles, authors, and year of publication.

1.2.3 *Orthogonal polynomials*

Write an *xml* document that describes three families of orthogonal polynomials of your choice using information and tags of your choice.

1.3 Usage and usefulness of xml files

What can we do with a well-formed *xml* document containing useful data, statements, or computer language instructions implemented by nested pairs of opened and closed tags and optional attributes? A few general but related families of applications are possible.

In reviewing these applications, it is helpful to make a distinction between *xml* data files and *xml* program files. An *xml* data file contains data and possibly processing instructions but no code. A human or machine processor is needed to manipulate the data and display the outcome in some desired way. In contrast, an *xml* program file contains instructions of a suitable computer language, such as the *xsl* language discussed in Chapter 3. Statements of any computer language can be recast in the *xml* format using appropriate tags and attributes.

It should be mentioned at the outset that the usage and usefulness of *xml* files can be understood fully only after *xml* data manipulation has been demonstrated and specific applications have been discussed. Realizing the purpose and utility of *xml* requires patience and a certain degree of hindsight.

1.3.1 Data formatting

The vast majority of *xml* applications are concerned with data formatting. In these applications, the structure of the tagging system in an *xml* document (data tree) is designed carefully to hold desired pieces of information, and at the same time avoid redundancy and repetition while anticipating future needs. A reputable *xml* designer will draw a bicycle that could be extended into an automobile, if the need arises, unrestricted by physical exclusion: wheels and engine will not overlap.

Conversion of an xml data file into another formatted data file

An *xml* data file can be transformed into another formatted data file. For example, an *xml* file can be converted into an *html* file whose content can be viewed on a *web* browser and then printed on paper, as discussed in Section 1.5. In the process of conversion, data can be manipulated and calculations can be performed to produce new data or suppress unwanted data. To transfer information from one application into another, data can be extracted from a source *xml* file, reformatted, and recorded into a new file or embedded into a new application under a different tagging system. *Xml* data documents written with ease-of-conversion in mind are sometimes called document-centric.

Data transmission

An *xml* data file created manually by a person or automatically by an application (program) can be sent to another person or application to be used as input. For example, data generated by a spreadsheet can be stored in an *xml* file under agreed conventions, and then imported into another application using different conventions. In fact, an *xml* document can serve as an interface between applications with different native formats: *pdf* may be converted into *xml* and then imported into a spreadsheet, and *vice versa.*

In scientific computing, data can be extracted from an *xml* file and automatically accommodated into variables or arranged into vectors and matrices (arrays) suitable for numerical computation. For example, an *xml* file may describe the geometrical properties of the elements of a boundary-element simulation along with a computed solution.

Data retrieval

The information contained in a small or large *xml* data file is a database. Data of interest contained in this database can be extracted using a general or special-purpose computer language code. For example, the year of publication can be retrieved from a document containing a list of books or research articles. *Xml* data written with ease-of-retrieval in mind are sometimes called data-centric.

Standardization

Numerous *xml* formatting systems have been proposed for data specific to particular disciplines: from music, to transportation, to computer graphics, to science and engineering applications. A comprehensive list of established tagging systems is available on the Internet.[*] New tagging systems consistent with *xml* conventions are frequently introduced. It is generally accepted that *xml* is the default framework for data formatting and storage applications.

[*] http://en.wikipedia.org/wiki/List_of_XML_markup_languages

1.3.2 Computer code formatting

Computer languages whose instructions are implemented in the *xml* format have been developed. Corresponding language compilers or interpreters, generically called a language processors, are necessary. In the absence of a corresponding language processor, a computer code written in *xml* is useful only as a prototype. Specific examples of *xml* computer language instructions will be discussed in Section 2.12 after the basic rules of *xml* grammar have been outlined.

Chula_vista

As a preview, we consider the following instructions in a fictitious computer language called *chula_vista*:

```
Do_this_for i=1:1:10
  display_on_the_screen i*i
End_of_Do_this_for
```

where the asterisk indicates multiplication. These lines print on the screen the square of all integers from 1 to 10. The same instructions could have be encoded in the *xml* format in terms of judiciously selected tags, as follows:

```
<chula_vista:Do_this_for variable="i" low="1" high="10" increment="1">
  <chula_vista:display_on_the_screen select="square(i)"/>
</chula_vista:Do_this_for>
```

Note that the name of the computer language employed, *chula_vista*, is specified for clarity and completeness in each *xml* tag along with necessary attributes. In standard *xml* nomenclature, the keyword *chula_vista* is a namespace. A self-closing tag is employed in the second line where the function *square* is called with a single argument to evaluate an attribute.

It is striking that the native *chula_vista* code is cleaner than its equivalent *xml* implementation. This observation confirms our suspicion that the *xml* formatting protocol is not without shortcomings.

Beware of exaggerations

The usefulness of *xml* in coding computer language instructions will be questioned by scientific programmers. The main reason is that computer code in any mid- or upper-level *bone fide* language, such as *fortran* or C++, is human readable by design. Low-level assembly code is human readable to a lesser extent, whereas machine code is incomprehensible to the casual computer programmer. It could be argued that an *xml* compliant code may serve as a generic blueprint that can be translated into any other computer language code. However, the same is true of *fortran*, C++, or any other upper-, mid- or low-level programming language equipped with appropriate data structures and language constructs.

Xml is appropriate for multilingual code

The *xml* implementation of a programming language is useful in cases where a computer code contains instructions in different programming languages, or employs functions implemented in different linked libraries. Consider the following *xml* implementation of the *chula_vista* code:

```
<chula_vista:Do_this_for variable="i" low="1" high="10" increment="1">
  <chula_vista:display_on_the_screen select="smolikas:gamma(i)"/>
</chula_vista:Do_this_for>
```

In this case, the `smolikas:gamma()` function belonging to the *smolikas* namespace is employed to compute the Gamma function.

In Chapter 3, we will see that multiple languages are used extensively in the *web* processing of *xml* and *html* files. Scientific programmers are used to writing code in one chosen language with occasional cross-over to other languages by way of language wrappers.

Exercises

1.3.1 *Multiple use of data*

Discuss possible ways of reusing text recorded in a question-and-answer (Q&A) session following a lecture recorded in an *xml* file.

1.3.2 *Eternity*

Write a sensible code of your choice in a fictitious language called *eternity*, implemented according to *xml* conventions.

1.4 Constraints on structure and form

To ensure that a sender and a receiver of an *xml* file interpret the data contained in the *xml* file in the same way, the meaning and structure of the formatting tags and data types employed should be defined in anticipation of present and future needs. In addition, other sensible or desirable constraints should be imposed to comply with industry standards. For example, we may require that each chapter of a book has at least one section, and the year of publication of a cited article is entered in the four-digit format. Ten digits would be required if the *xml* document is expected to survive after our sun implodes.

Data type definitions and schema

To achieve these goals, we introduce a document type definition (*dtd*) or an *xml* schema definition (*xsd*). In practice, we write instructions that define the meaning and prescribe the permissible structure of tags to be employed, and also introduce constraints on data types, as discussed in Section 2.10. These instructions are either embedded into an *xml* data document or placed in an accompanying document. The recommended and most powerful method of implementing an *xml* data type definition is the *xml* schema. Initiatives are under way to develop schemata in various branches of commerce and publishing for the purpose of standardization.

Cml and xdmf

As an example, the *xml* schema definition (*xsd*) of the chemical markup language (*cml*) specifies the following typical structure:

```
<molecule>
  <atom>
    <bond>
      ...
    </bond>
  </atom>
</molecule>
```

where the three dots indicate additional data. Similar structures and conventions must be followed in documents that comply with the extensible data model and format *xml* schema (*xdmf*) designed for high-performance computing.

Validation

Once a *dtd* or *xsd* has been selected and implemented, a complying *xml* document must employ only mandatory or optional tags and structures defined in the *dtd* or *xsd*. An *xml* data file can be validated against a *dtd* or *xsd* using a standalone validation program or a validation program included in an *xml* parser. It is important to remember that an *xml* file that fails to be validated may still be well-formed.

Loss of freedom

By accepting a *dtd* or *xsd*, we abandon our freedom to compose and format a document using tag names and data structures of our choice. To be well-formed is no longer sufficient but only necessary. Our status regresses to that of a *latex* or *html* user who must use predetermined tags implemented in the *latex* processor or *html* interpreter. Sadly, we may no longer write an *xml* document in isolation, but must pay attention to an established set of rules. Although programming experience is still not necessary, the use of a reference manual is mandatory.

Snake oil?

We have reached a crossroads and it appears that we have made a full circle. In light of the potential loss of freedom incurred by adopting a *dtd* or *xsd*, it is fair to question whether *xml* lives up to its reputation as a panacea. The truth is that adopting a standard *dtd* or *xsd* makes sense when the formatting protocol is broadly accepted and well documented, or when the source code of the *dtd* or *xsd* is accessible and freely available for modification. Alternatively, the lone researcher, scientific programmer, research group, or computer enthusiast may build their own private set of rules without any constraints. *Xml* formatting can become a personal way of recording thoughts and keeping notes.

1.4.1 DocBook schema

It is instructive to discuss an example of a document type definition used in a popular application. The *DocBook* schema was designed for writing books and other documents following *xml* conventions.[*] *LibreOffice*[†] is a free authoring application available on a variety of platforms, incorporating the *DocBook* schema. The suite includes a *what you see is what you get* (WYSIWYG) word processor named *writer* that is compliant with the *DocBook* schema.

As a digression, we note that WYSIWYG editors have been criticized for dividing an author's attention into substance and form. The main argument is that an author should be encouraged to write complete, thoughtful, and well-structured sentences undistracted by formatting and spelling considerations, and then format or embellish the presentation. This is precisely what *latex* seeks to accomplish.

The Great Gatsby

As an experiment, we launch (open) the *LibreOffice* word processor and use the available menu to write the following lines from F. Scott Fitzgerald's *The Great Gatsby*:

Chapter 1

In my younger and more vulnerable years my father gave me some advice that I've been turning over in my mind ever since.

"Whenever you feel like criticizing any one," he told me, "just remember that all the people in this world haven't had the advantages that you've had."

Every word, every sentence, every punctuation mark in this passage is outstanding. Nothing can be improved in style, meaning, presentation, or intent. F. Scott Fitzgerland was a brilliant writer. Novelists, journalists, and technical writers will benefit a great deal from reading his books.

Next, we select *Save As* from the *file* drop-down menu of the application, choose *DocBook* as Filetype, and save the file under the name *greatgatsby.xml*.

[*] http://www.docbook.org/whatis
[†] http://www.libreoffice.org

The content of this file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
"http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd">
<article lang="">
  <para>Chapter 1</para>
  <para/>
  <para>In my younger and more vulnerable years my father
  gave me some advice that</para>
  <para>I've been turning over in my mind ever since.</para>
  <para>  ``Whenever you feel like criticizing any one,''
   he told me, ``just remember</para>
   <para>that all the people in this world havent
  had the advantages that youve had.'' </para>
</article>
```

The name of the *xml* root element, `article`, is qualified by an empty (default) language attribute named `lang`. The meaning of other tags implementing children elements can be deduced by mere inspection and sensible interpretation. For example, `para` is an abbreviation of *paragraph.* The second line, continuing to the third line, reading:

```
<!DOCTYPE ··· docbookx.dtd">
```

invokes a document type definition (*dtd*), as discussed in Section 2.10.

The *LibreOffice* native file itself, named *greatgatsby.odt*, is a machine readable binary file that can be interpreted only by the word processor. In contrast, *greatgatsby.xml* is a *text* (*ascii*) file.

1.4.2 LibreOffice Math

The *LibreOffice* suite includes an equation editor application named *math.* As an experiment, we open the application and type in the lower partition of the graphical user interface (Gui) the following text:

```
{a} over {b}
```

representing the fraction a/b. Next, we save the text in a file with a chosen name. The content of this file turns out to be:

```
<?xml version="1.0" encoding="UTF-8"?>
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <semantics>
    <mrow>
    <mfrac>
      <mrow>
        <mi>a</mi>
      </mrow>
```

```
       <mrow>
         <mi>b</mi>
       </mrow>
     </mfrac>
     </mrow>
   <annotation encoding="StarMath 5.0">a over b </annotation>
   </semantics>
 </math>
```

This is an *xml* file with a root element named `math` associated with a namespace specified in the *xmlns* attribute. We observe several nested tags with apparent or nearly obvious meanings. In fact, this file can be opened with a *web* browser to display the fraction.

This example illustrates the portability of *xml* data across different applications running on arbitrary hardware platforms. In the absence of proprietary encoding that obscures the meaning of information, an *xml* document can be created by one application and then imported unchanged or slightly modified into another application. This property is sometimes described as *information reuse* or *multiple data use.*

Exercise

1.4.1 *An experiment*

Repeat the *DocBook* experiment discussed in the text with a document of your choice.

1.5 Xml data processing

We have mentioned on several occasions that data contained in an *xml* file can be read, understood, and interpreted by a person or else processed by an application written in a suitable computer language. Specific examples are given in this section.

1.5.1 Human processing

A busy financier fills out a lunch order on an *xml* order form and faxes it to a restaurant. The restaurant owner visually parses the order and delivers instructions to the cook who follows the instructions and prepares the food. The order is also handed to the cashier who assesses charges and taxes, prepares a bill, and faxes a bill to the financier. In this case, the owner of the restaurant is the parser, the cook and the cashier are two different processors, and the financier has a data-entry job.

1.5.2 Machine processing with xsl

The computer processing of data contained in an *xml* document is best explained in the context of the extensible stylesheet language (*xsl*) discussed in detail in Chapters 3 and 4.

Xsl is a computer language comparable to *fortran* or C. An *xsl* processor, like any other *xml* processor, parses the data contained in a well-formed and possibly validated *xml* document (input), performs calculations and manipulations according to instructions given in a companion *xsl* program file containing code (application), and records or displays the outcome (output). The output can be another *xml* file or an *html* file that can be processed and displayed in a *web* browser. Available *xsl* processors are reviewed in Section 3.1.

We will see that, strange though it may seem, an *xsl* code is written according to *xml* conventions, that is, *xsl* is an *xml* implementation. However, the *xml* compliance of *xsl* is not an essential feature of the *xml*/*xsl* framework.

Sweet and sour

Xsl codes contained in two different *xsl* files may assign different meanings to a tag named, for example, *flavor*, in an *xml* file. One *xsl* file may interpret the tag as sweet, while another *xsl* file may interpret the tag as sour. Thus, depending on the instructions given in the companion *xsl* file, an *xml* data file with the same words may taste differently after processing,

Debt and donation

Assume that an *xml* file contains the names and details of university graduates. These data can be used to send the graduates two letters: a fundraising letter asking for donations, and another letter asking them to pay owed tuition and fees. The primary *xml* file (input) reads:

```
<graduate>
  <name>Eliana Smith</name>
  <major>entomology</major>
  <graduation>2003</graduation>
  <debt>87451.20</debt>
<graduate>
```

After processing, the output file relevant to the fundraising letter may read:

```
<donor>
  <name>Eliana Smith</name>
  <major>entomology</major>
  <years_since_graduation>8</years_since_graduation>
  <recommended_donation>20.0</recommended_donation>
<donor>
```

The output file relevant to the debt-collection letter may read:

```
<bill tone="harsh">
  <name>Eliana Smith</name>
  <you_owe_us_with_interest>999817.99</you_owe_us_with_interest>
  <pay_by>yesterday</pay_by>
<bill>
```

Some calculations were performed in generating the output files. This example illustrates that an *xml* document may serve as an information database in a multitude of applications.

Xsl constituents

The *xsl* processor encapsulates three interacting libraries with complementary tasks:

- *Xslt* (transformations) is a language for transforming an *xml* document (input) into another *xml* document (output), including an *html* document or any other text (*ascii*) document.
- *Xpath* is a language for navigating inside an *xml* document by returning references to *xml* element nodes.
- *Xsl-fo* is a language for formatting an *xml* document.

Xslt allows us to rearrange, suppress, modify, and add to the information contained in the *xml* data file, as discussed in Chapters 3 and 4. *Xslt* uses *xpath* to match and select coherent data blocks. When a match is found, *xslt* applies the requested transformations. An *xsl* procedure or function is sometimes delineated as an *xslt* or *xpath* element or function to accurately describe its implementation.

Since the internal organization of the *xsl* processor is of marginal interest in scientific computing, we will generally refer to *xsl* in place of *xslt* or *xpath* when a distinction is not necessary.

Xsl computing

The *xsl* processing of an *xml* file follows the paradigm of scientific computing in that data and programs are separated into different files. However, significant differences in programming structures and available facilities render the two frameworks sharply distinct. In particular, a number of programming structures shared by common scientific languages, such as *fortran* or C, do not have counterparts in the *xsl*, and *vice versa*. The reason can be traced to the paramount importance of the data in the *xml*/*xsl* framework. We will see that the mere presence of an *xml* tag is sufficient to drive the execution (launching) of an *xsl* code.

Bare bones

Because of severe limitations with regard to numerical procedures and functions, the *xml/xsl* framework is not suitable for advanced scientific computing. Arguments to the contrary lack convincing counter-examples. Nonetheless, the unavailability of extensive resources renders the *xml/xsl* framework attractive for developing programming skills, as discussed in Chapter 4. Metaphorically speaking, the *xml/xsl* framework provides us with a screwdriver and some screws and expects us to build a John Deere tractor.

Execution begins with the data

It is worth remarking that the execution of an *xsl* code begins with the data rather than with the code. Specifically, the name of the file hosting the *xsl* code is defined in a processing instruction in the *xml* data file. This feature is consistent with the notion that the data are more valuable than the program that manipulates the data. In scientific computing, a reliable program that computes the eigenvalues of an arbitrary matrix is extremely valuable. In the *xml* framework, the words of *The Catcher in the Rye* are precious, irrespective of how they appear printed on paper or displayed on a screen.

Xml is a formatting language, xsl is a computer language

Most *xml* texts and *web* tutorials discuss exclusively the *xml/xsl* framework, and this may create the misconception that *xml* is intimately connected to *xsl* or *vice versa*. This is certainly not true in the context of scientific computing where the *xsl* language is hardly known. In its pure and intended form, *xml* is a general and unrestricted data or statement formatting protocol.

Xml to html

An *xsl* code can be written that converts an *xml* file into an *html* file that can be stored in a file or processed and viewed in a *web* browser. In the simplest and most direct method, the *xml* file is opened by a *web* browser through a dropdown menu. A processing instruction (Pi) near the beginning of the *xml* file indicates the name of an accompanying *xsl* file that will be used to process the *xml* data. If this file is not found, an error message is issued.

If the file is found, the *xsl* parser and processor embedded in the *web* browser process the data and display the outcome in the browser's window. To print the processed *html* file, we use the browser's printing services. Since all modern browsers can handle basic *xsl* code, *xml* processing is independent of the computer platform employed. This independence is the cornerstone of the *xml* framework within and beyond the *xsl* framework.

Xml was motivated to a large extent by the desire to generalize the restrictive *html* framework in *web* programming. This motivation is of marginal interest in scientific computing where the ability to generate *html* code from *xml* code is hardly compelling. The main attraction of *xml* in computational science relates to its ability to describe, organize, and identify data in the input or output.

Xml processors are strict

The world wide *web* consortium (W3C) specifies that, if an error is found in an *xml* document, the execution of a program or application processing the data, such as an *xsl* code, should terminate. If the execution does not terminate, the code is not W3C compliant. Thus, like *latex* and computer language compilers, but unlike *html* interpreters, *xml* processors are required to be strict.

Consequently, although programming experience is not necessary for composing and editing an *xml* document, syntactic and structural errors are not allowed.

Exercise

1.5.1 *Point particle trajectory*

A data file contains in four columns the three Cartesian coordinates of a point particle in space at a sequence of times. How would this file appear in the *xml* format?

1.6 Relevance of xml in scientific computing

Our main goal in this book is to discuss the relevance of *xml* in scientific computing, which can be contrasted with *web* and professional applications computing. The *xml* framework is consistent with the standard protocol of scientific computing involving input, processing, and output.

When we write a scientific code in a compiled computer language, such as *fortran*, C, or C++, we follow three basic steps:

1. We write a program containing the language instructions.
2. We compile the program and link all object files and necessary libraries into an executable file, which is a binary file containing standalone machine language instructions.
3. We run the executable.

The input data can be contained in the program, entered manually through the input devices (keyboard and mouse) during execution, or read from input data files named in the program. Interpreted code is handled in similar ways.

Value of the executable

In scientific computing, a great deal of effort is expended toward generating efficient code compiled into an executable. In parametric investigations, the same program is run with different input to generate reliable output that can be analyzed by post-processing, visualization, or animation. The central goal of the discipline of high-performance scientific computing is to develop efficient mathematical, numerical, and memory management algorithms that reduce the demand on hardware and central processing unit (Cpu) time.*

Separating program from data

In scientific computing, we routinely separate the computer program (code) from the input data, placing them in different files. The program is written in a main file accompanied by function or subroutine files recognized by appropriate suffixes, such as .f, .c, .cc, and .sce, for *fortran*, C, C++, and *scilab*, respectively. The input data reside in other files recognized by standard or arbitrary suffixes, such as .dat, .conf, or .inp. The output is printed in data files named in the code and produced during the execution.

If we change the input data or parameters inside a code, we must be recompile the code to generate a new executable. This major inconvenience explains why it is highly desirable for the data to be separated from code. In the basic *html* implementation, formatting instructions and data reside in the same file. This monolithic structure considerably complicates data extraction, manipulation, reformatting, and portability across hardware platforms. One advantage is that only one computer file needs to be edited, processed, or communicated.

Data formatting

Assume that a data file contains in two columns the real and imaginary parts of the eigenvalues of a 2×2 matrix, as follows:

```
 0.134 -0.234
-0.878  0.238
```

An *xml* file might represent these data as:

```
<eigenvalue>
  <real>0.134</real>
  <imaginary>-0.234</imaginary>
</eigenvalue>

<eigenvalue>
  <real>-0.878</real>
```

*Pozrikidis, C. (2008) *Numerical Computation in Science and Engineering.* Second Edition, Oxford University Press.

```
  <imaginary>0.238</imaginary>
</eigenvalue>
```

The *xml* file describes the eigenvalues unambiguously, circumventing the need for legends, explanations, and conventions. This example underlines the notion that *xml* presents *and* describes data. We say that *xml* encapsulates content and form.

Verbosity is a concern

There is an elephant in the room: the use of repetitive tags in an *xml* document is a practical concern with regard to document size. In our example, the text file containing the matrix of eigenvalues is much smaller than the *xml* file where repetitive tags are employed. Inflated storage can be tolerated in some, but not all, scientific applications. To address this concern, a binary characterization of *xml* data (*xbc*) has been proposed.

Let's be honest

In fact, *xml* is not the only method of presenting and describing data. Other methods based on high-level computer languages are available, as discussed in the remainder of this section. An important requirement is computer programming experience. An important concern is that the principle of separating code from data is likely to be violated. It is not surprising that the uninvested scientific programmer will linger between spending time and effort in developing *xml* compliant schemes or staying in the mainstream.

1.6.1 Matrices

In scientific computing, we routinely deal with matrices defined as square, rectangular, or slender arrays of numbers. An example is the 3×4 matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \end{bmatrix}, \tag{1.2}$$

containing 3 rows and 4 columns. In *fortran*, the elements of this matrix are denoted as

```
A(i,j)
```

where `i` = $1, 2, 3$ and `j` = $1, 2, 3, 4$. For example, $\mathrm{A}(3, 2) = 8$. In C++, the elements of this matrix are denoted as

```
A[i][j]
```

For example, $\mathrm{A}[3][2] = 8$.

In *fortran*, the indices of matrix elements can have any positive, zero, or negative values. This flexibility considerably simplifies the implementation of algorithms in science and engineering applications. In C and C++, the indices of matrix elements can have positive or zero, but not negative, values. If we reserve a 10×9 matrix $\mathbf{A}$ in a C++ code, we will be allowed to use the matrix elements `A[i][j]`, where `i = ` $0, 1, \ldots, 9$ and `j = ` $0, 1, \ldots, 8$. The zero lower limit ensures the efficient use of all available memory space associated with the binary representation. In *Matlab*, the indices of a matrix element must be positive integers.

Limitations

Two important limitations of the matrix (array) representation of the data shown in (1.2) can be identified:

1. The meaning of the twelve numbers encapsulated in the matrix is not revealed in the matrix itself, but must be separately specified.
2. The elements of the matrix must be either all integers or all real numbers, that is, they must all be of the same data type.

The second difficulty can be resolved by transforming the matrix into an object described by properties and attributes with different data types.

1.6.2 Objects

Consider the following generalized matrix containing numbers and text, regarded as an inhomogeneous object:

$$\mathbf{B} = \left[\begin{array}{ccccccccccc} 1 & \text{triangle} & 3 & 0.0 & 0.0 & 0.0 & 1.0 & 1.0 & 0.0 & & \\ 2 & \text{square} & 4 & 0.0 & 0.0 & 1.0 & 0.0 & 1.0 & 1.0 & 0.0 & 1.0 \end{array} \right]. \qquad (1.3)$$

Although it appears safe to assume that this matrix holds information on polygons whose vertex coordinates are provided as consecutive xy pairs, our intuition could be wrong. The first column appears to host a counter, while the third column states most likely, but not assuredly, the number of polygon vertices.

Xml representation

Xml surpasses the object model by completely and relentlessly describing the information contained in the generalized matrix (1.3) as follows:

```
<polygon id="1" shape="triangle" vertices="3">
  <vertex1><x>0.0</x><y>0.0</y></vertex1>
  <vertex2><x>0.0</x><y>1.0</y></vertex2>
  <vertex3><x>1.0</x><y>0.0</y></vertex3>
</polygon>
```

```
<polygon id="2" shape="square" vertices="4">
  <vertex1><x>0.0</x><y>0.0</y></vertex1>
  <vertex2><x>1.0</x><y>0.0</y></vertex2>
  <vertex3><x>1.0</x><y>1.0</y></vertex3>
  <vertex4><x>0.0</x><y>1.0</y></vertex4>
</polygon>
```

It is fair to admit that the high density of this *xml* document is overwhelming. The generalized matrix (1.3) can be constructed from the *xml* data manually or automatically, but not *vice versa*.

Dom and Sax

Conversely, data encapsulated in an *xml* file can be imported and transformed into objects that can be manipulated or queried using an application written in a scientific language of our choice, such as *fortran* or C++. This methodology is the cornerstone of the document object model (Dom) where an *xml* file is mapped directly into a generalized object. The methods used to access an object are contained in the application programming interface (Api).

The document object model (Dom) is useful for data sets with small or moderate size, but inappropriate for data contained in a large database. Direct query of the database using, for example, the simple Api for *xml* (Sax) incorporating event handling callbacks is preferred in the second case.

Heterogeneous arrays and objects

If the data contained in a conceptual object are homogeneous (of the same data type), the object can be accommodated in a matrix (array) in *fortran* or C, subject to agreed conventions regarding the meaning of the columns or rows. Heterogeneous arrays are available in *perl* and other system programming languages. Arrays containing mixed data types can be stored as objects in object-oriented languages, such as C++.

1.6.3 Data points on a graph as xml elements or C++ objects

To illustrate the concept of objects, we consider data points in the xy plane representing a function to be plotted with colored symbols in a graph. Each point is defined by its x and y coordinates, color, and symbol type, such as a circle, square, or asterisk.

Xml elements

The data points can be conveniently recorded in the following complete *xml* file, including the *xml* declaration in the first line:

```
<?xml version="1.0"?>
<melomakarono>
```

```
  <datapoint>
    <x>0.0</x>
    <y>0.0</y>
    <color>black</color>
    <symbol>circle</symbol>
  </datapoint>

  <datapoint>
    <x>0.1</x>
    <y>0.2</y>
    <color>red</color>
    <symbol>asterisk</symbol>
  </datapoint>

</melomakarono>
```

The name of the root element is `melomakarono`. Two data points are defined in this file. Additional data points can be added following the chosen *xml* data tree.

C++ objects

The data points are now regarded as objects (members) of a class named *datapoint*. The following self-contained C++ code residing in a file named *datapoint.cc* defines the class of data points:

```
#include <iostream>
using namespace std;

/* ------ datapoint class definition ------ */

class datapoint
{
public:
  datapoint(); // default constructor of an object
  datapoint(float, float, string, string); // parametered constructor
  void print() const;
private:
  float x;
  float y;
  string color;
  string symbol;
};

/* ------ datapoint class implementation -------*/

datapoint::datapoint()
{
```

```
x = 0.0;
y = 0.0;
color = "black";
symbol = "circle";
}

datapoint::datapoint(float px, float py,
  string pcolor, string psymbol)
{
x = px;
y = py;
color = pcolor;
symbol = psymbol;
}

void datapoint::print() const
{
cout << x << " " << y << " " << color << " " << symbol << endl;
}

/* ------ main program -------*/

int main()
{
  datapoint A = datapoint();
  A.print();
  datapoint B = datapoint(0.1, 0.2, "red", "asterisk");
  B.print();
  return 0;
}
```

Readers who are not familiar with the C++ programming language can refer to Table 1.1 for miscellaneous explanations.

The code initially defines the class *datapoint* and declares three public interface member functions: a default constructor, a parametered constructor, and a print function of a member's attributes. Four private member attributes undisclosed to the main program are then declared. The *datapoint* class implementation follows the class definition.

The last part of the C++ code consists of the main program. For illustration, the main program defines datapoint A using the default constructor and datapoint B using the parametered constructor. The attributes of the first point are printed by the statement:

```
A.print()
```

The dot operation is commonplace in object-oriented programming.

#include <iostream>	Instructs the C++ preprocessor to attach a header file containing the definition, but not the implementation, of functions in the input/output stream library.
using namespace std;	The names of the functions defined in the standard `std` system library are adopted in the code.
public: private:	Member attributes are declared as *public* if available to the main program and functions of a different class, and *private* otherwise. Similarly, interface functions are declared as *public* if they can be called by the main program and functions of a different class, and *private* otherwise.
cout:	Internal library function for printing.

Table 1.1 Explanation of various lines in the C++ program *datapoint.cc* listed in the text containing information on data points.

Generating an executable

C++ compilers are included in standard *unix* distributions and are freely available on Windows.[*] To compile the *datapoints.cc* code and create an executable binary file named *datapoints*, we open a terminal (command-line window) and issue the statement:

```
c++ -o datapoints datapoints.cc
```

To run the executable, we type its name and press the Enter key,

```
./datapoints
```

To ensure that the path of executables includes the current directory, we have inserted the dot-slash pair (./) in front of the name of the executable. The dot represents the current directory (folder) and the slash is a delimiter of the directory path. Running the executable prints on the screen:

```
0 0 black circle
0.1 0.2 red asterisk
```

It is clear that a C++ code is able to hold information on objects described as heterogeneous arrays. Descendant objects can be constructed as offsprings of parental objects using the concept of inheritance in object-oriented programming. These impressible features explain the phenomenal success of C++ and other object-oriented languages in applications programming.

[*] http://sourceforge.net/projects/mingw

Two practical questions naturally arise: how can we get a C++ code to print *xml* output in a way that both presents and describes the data? how can we import *xml* data into an C++ code? The answer to the first question is relatively straightforward. The answer to the second question is less straightforward, as discussed in Chapter 5.

1.6.4 Perl associative arrays

Perl is a powerful interpreted system programming language. The qualifier *system* emphasizes that the language is used mostly for retrieving and manipulating existing information, and to a lesser extent for generating new information. An outline of the basic language features is given in Appendix B. It is not necessary to compile a *perl* program, typically called a script, into an executable. The instructions contained in the script are executed as they are parsed by the *perl* interpreter.

Scalars and arrays

Perl allows us to use scalar variables, homogeneous arrays with uniform data types, and heterogeneous arrays with different data types, including integers, real number, and character strings. In this light, a *perl* array appears as an object described by numerical and narrative attributes. The value of a *perl* scalar variable is defined or extracted by prepending the dollar sign (`$`) to the variable name. The contents of a *perl* array are defined or extracted by prepending the *at* symbol (`@`) to the array name.

Hashes

A *perl* hash is a *perl* array endowed with references linking variable names (keys) to values that can be numbers or character strings. Thanks to the keys, a *perl* hash defines and describes in simple terms the data it encapsulates. A *perl* hash can be regarded as a map reminiscent of a dictionary. Accordingly, a *perl* hash is also called an associative array. To define or extract the contents of a hash, we prepend the percent symbol (`%`) to the hash name.

Data points

Each of the two data points defined in Section 1.6.3 can be accommodated into a *perl* hash, as shown in the following self-contained *perl* script residing in a file entitled *datapoints.pl*:

```
#!/usr/bin/perl

%datapoint1 = ( x => 0.0,
                y => 0.0,
                color => "black",
                symbol => "circle"
                );
```

```
%datapoint2 = ( x => 0.1,
                y => 0.2,
                color => "red",
                symbol => "asterisk"
                );

print "$datapoint1{color} \n";
print "$datapoint2{symbol} \n";
```

The first line identifies the directory where the *perl* interpreter resides in our *unix* system. One named *perl* hash is defined and evaluated for each data point. Note that each *perl* statement terminates with a semi-colon (`;`). The *perl* hashes defined in this script contain human-readable information for each data point.

The color of the first point is extracted as a scalar value (`$`) in the penultimate line of the script, and the symbol of the first point is extracted as another scalar value (`$`) in the last line of the script. A hash index analogous to a vector subscript is implemented by a pair of curly brackets (`{}`). The extracted variables are printed by two print statements in the last two lines. The character referenced by the `\n` pair forces a new line in the output at the end of each `print` statement.

Interpretation

Perl interpreters are included in standard *unix* distributions and can be obtained freely in other operating systems.* To execute a *perl* script, we open a terminal (command-line window) and type the name of the script followed by the ENTER keystroke:

```
./datapoints.pl
```

To ensure that the path of executables includes the current directory, we have inserted the dot-slash pair (`./`) in front of *perl* file name. Running the script produces the display:

```
black
asterisk
```

We see that a *perl* hash provides us with an attractive method of describing and defining simple objects.

Data points as a named array

The two data points under discussion, or any number of data points, can be ar-

*http://www.perl.org/get.html

ranged in a named array of anonymous *perl* hashes, called `datapoints`, defined as:

```
@datapoints = (
                {
                    x => "0.0",
                    y => "0.0",
                    color => "black",
                    symbol => "circle"
                },
                {
                    x => "0.0",
                    y => "0.1",
                    color => "red",
                    symbol => "asterisk"
                }
                );
```

Note that the array symbol (`@`) has been prepended to the array name to indicate array evaluation. Each component of this array is an anonymous hash enclosed by pairs of curly brackets (`{}`), accessible by an array index.

Conceptually, the data contained in this array can be accommodated in the rows of a generalized matrix,

$$\text{datapoints} = \begin{bmatrix} 0.0 & 0.0 & \text{black} & \text{circle} \\ 0.0 & 0.1 & \text{red} & \text{asterisk} \end{bmatrix},$$

where the first row receives the index 0 and the second row receives the index 1. In *perl*, as in C++, index counting begins at 0 so that all available bits of the integer counter are exploited, including a string of binary zeros.

To extract and print the properties of the first datapoint indexed 0, we use the lines:

```
print $datapoints[0]{x};
print $datapoints[0]{y};
print $datapoints[0]{color};
print $datapoints[0]{symbol};
```

Recall that the dollar sign (`$`) indicates a scalar. The screen display is:

```
0.0 0.0 black circle
```

To access and print the coordinates and properties of the second data point indexed 1, we replace `[0]` by `[1]` in the print statements. The complete code resides in the file *datapoints.pl* accompanying this book.

To illustrate the flexibility of *perl*, now we arrange the data into an anonymous array of anonymous hashes:

```
$tirith = [
             {
                 x => "0.0",
                 y => "0.2",
                 color => "black",
                     symbol => "circle"
             },
             {
                 x => "0.3",
                 y => "0.1",
                 color => "maroon",
                 symbol => "diamond"
             }
           ];
```

The variable `tirith` is a scalar reference to an anonymous array enclosed by the square brackets (`[]`). The contents of the anonymous array are the same as those of the named array discussed previously.

To extract and print the properties of the first datapoint indexed 0, we use the lines:

```
print $tirith->[0]{x};
print $tirith->[0]{y};
print $tirith->[0]{color};
print $tirith->[0]{symbol};
```

The screen display is:

```
0.0 0.2 black circle
```

Note that the reference `tirith` is dereferenced by the *ascii* arrow consisting of two characters (`->`), as discussed in Appendix B.

A graph

Information on a complete graph of data points can be accommodated into an anonymous hash represented by a reference containing data points and other relevant information, defined as:

```
$graph = {
             datapoint => [
                          {
                              x => "1.0"
                              ,y => "3.0"
```

```
                                 ,color => "black"
                                 ,symbol => "circle"
                             },
                             {
                                 x => "3.0"
                                 ,y => "3.1"
                                 ,color => "red"
                                 ,symbol => "asterisk"
                             },
                                 ]
                ,xlabel => "distance"
                ,ylabel => "temperature"
                ,title => "temperature distribution"
          };
```

The scalar variable `graph` is a reference to the outermost anonymous hash enclosed by the outermost curly brackets (`{}`). The scalar key `datapoint` inside the outer hash represents an anonymous array, indicated by the square brackets (`[]`), containing as elements anonymous hashes enclosed by the inner curly brackets (`{}`). The outermost anonymous hash contains three more keys defining the axes labels (`xlabel` and `ylabel`) and the graph title.

Appending to this script the lines:

```
print $graph->{datapoint}[1]{x};
print $graph->{datapoint}[1]{y};
print $graph->{datapoint}[1]{color};
print $graph->{datapoint}[1]{symbol};
print $graph->{xlabel};
print $graph->{ylabel};
print $graph->{title};
```

produces the screen display:

```
3.0 3.1 red asterisk distance temperature temperature distribution
```

The *ascii* arrow consisting of two characters (`->`) leads us from the reference to the content of the outermost anonymous hash.

Why not perl?

We have seen that a *perl* array of hashes can be used to *store and describe* data with inhomogeneous content, with the added advantage that the data can be manipulated using *perl* language instructions. It is clear that *perl*, or any other comparable language, such as *python*, is a viable alternative to *xml*.

Three main concerns in using *perl* and other similar system programming languages for data representation are: (*a*) the principle of code from data separation is likely to be violated, (*b*) difficulties in accommodating data with advanced structure may be encountered, and (*c*) computer programming experience is necessary. With regard to the third concern, we emphasize that an *xml* document can be written and edited by a person who is unfamiliar with any computer language. In practice, *perl* and similar high-level languages are used for *xml* data manipulation, as discussed in Chapter 5.

1.6.5 Computing environments

We have seen that the information encapsulated in an *xml* file can be arranged into homogeneous or inhomogeneous data structures of advanced programming languages, such as *perl*. Proprietary computing environments, such as *Matlab* and *Mathematica*, have made pertinent accommodations. For example, a *Matlab* structure can be defined using the statements:

```
student.name = 'Kathryne Marple';
student.gpa = '4.0';
```

An array of structures can be built into arrays and accessed by indices, as discussed previously in this section for *perl*. The *Mathematica* environment makes analogous accommodations.

1.6.6 Summary

Three main features of the *xml* framework are: (*a*) separation of data from code, (*b*) ability to collect, record, and retrieve data with a generic application in mind, and (*c*) lack of the requirement for computer programming skills. Specific data contained in an *xml* database can be extracted, mined or retrieved, imported, and manipulated by a person or program (application) with a particular goal in mind.

The two salient questions posed earlier in this chapter must be addressed: how can we get a computer code to generate and record *xml* output in a file? how can *xml* data be read efficiently from a code? An overview of available options will be given in this book.

Exercises

1.6.1 *Size of symbols*

Endow the data points defined in the C++ code discussed in the text with one additional attribute concerning the symbol size.

1.6.2 *Perl*

Write a *perl* script that describes two objects of your choice. Each object should be defined by a few alphanumerical properties (attributes).