Caring for Your Data

Konrad Hinsen

Caring for Your Data

By Konrad Hinsen

In the long run, your data matters more than your code. It's worth investing some effort to keep your data in good shape for years to come.

Data is at the heart of science. A scientist is expected to be able to back up all published conclusions with data. Data management should thus be a priority in science. Scientists can't afford to lose data, be uncertain of what it means, or not know where it came from. In the experimental sciences, there's a long tradition of writing down all experimental setups, parameters, and results meticulously in a lab notebook. Unfortunately, computational science is much less rigorous about data handling, although there are clear signs of improvement. Here, we'll consider what you can do to better prepare and manage your data.

Overview of the Problem

Most scientific computing follows a simple pattern: the computer runs a program that reads in data, manipulates it in some way, and produces output data. Input and output data are typically stored in files, but programs written for interactive use also take input directly from the user and provide output in a visual form (plots, animations, and so on).

For scientific research, the data matters more than the programs. In processing experimental results, the observations are the input data, and some quantities more directly related to the problem at hand (or simply a more familiar quantity to experts in the field) are the output data. In the computational evaluation of theoretical models, the input data consists of parameters and the output data is the theoretical prediction. It's the input and output data that's shown and discussed in scientific publications.

The program that produces output data from input data is of little direct scientific interest. What matters is the mathematical equations that it applies to do its work. As scientific problems and computational resources evolve, the programs are often adapted for technical reasons such as performance, but they still implement the same mathematical equations. A typical scientific publication discusses only the equations, not the other aspects of the program. There's an increasing awareness in the computational science community of the need to publish the full programs as well, in order to permit verification of the computations by other scientists. However, this doesn't change the fact that the central items in computational science are data and equations.

Despite the importance of data for science, the storage and processing of data has traditionally been given little priority in scientific programming. The typical scientific programmer concentrates on the mathematical equations and on the algorithms that implement them. The in-memory representation of data in the program is usually chosen for convenience of implementation, as is (in many cases) the file format used to store the data more permanently. To make it worse, the file formats of many scientific programs aren't documented by anything other than the program's source code.

An unfortunate consequence of this priority of code over data is that computational scientists spend a significant amount of time converting data between the formats used by different programs. In addition to requiring a lot of effort, file format conversion is an error-prone process, in particular when the file formats aren't properly documented. A less-visible consequence of the absence of data design is that scientific programs sometimes get trapped by a legacy data representation that can no longer be adapted to evolving requirements.

Data Models and File Formats

An important distinction to keep in mind is the one between data models and file formats. A data model defines how a specific collection of information is expressed in terms of basic data items such as integers, floating-point numbers, or characters, and simple aggregate data structures such as lists, arrays, sets, or associative arrays. A file format defines how a data model is represented by a sequence of bytes. A data model covers in-memory storage as well as file storage.

A given data model can be represented by several file formats. This is often useful because different stages of data processing have different requirements. For example, it might be advantageous to have a compact and platform-independent file format for archiving, a file format optimized for I/O performance for high-throughput computations, and a text-based file

format for manual inspection and editing of data. As long as these three file formats implement the same data model, conversion between them is straightforward and lossless.

As a simple example, consider an address book. A simple data model would be as follows:

1. An address book is a list of address book entries.
2. An address book entry consists of a fixed set of fields, whose values are character strings. Let's say the fields are "first name," "last name," "phone number," and "e-mail address."

A simple file format representing this data model can be defined by writing the value of each field in an address book entry as one line of a text file. Each entry would then take up four lines. Because the entries have a fixed length, they can simply follow each other in the file. An alternative file format could use one line per entry, with the fields separated by commas. Conversion between these two formats would be straightforward and not lose any information either way.

In practice, you would want to define a more elaborate data model for an address book. Many people have several phone numbers and e-mail addresses. You could give each of them a label, such as "home" or "work." The value of the "phone number" field is then an associative array whose keys and values are character strings. Another generalization would be to make the whole entry an associative array as well, allowing any number of fields such as "Twitter account," "nickname," or "postal address."

However, this generalization creates a new problem: What if different users of your data model pick different field names or the same information? One user could choose "postal address" and another one just "address." Interoperability between programs would be much more difficult. A partial solution would be a set of mandatory field names. This example illustrates that it's not necessarily a good idea to make a data model overly general.

A more subtle problem with our address book data model is that it defines an address book as a list of entries. A list is a sequential data structure, meaning that the order of address book entries is significant. Two address books with the same entries arranged differently would be considered different. A more appropriate choice for the top-level data structure would be a set. On the other hand, a set is more difficult to implement in software, and you lose the convenience of a list in which each element has a simple and unique handle: its index in the list. Data model design is thus often a matter of choosing the right compromise.

Designing Data Representations

How can we design a good data model? Unfortunately, this question has no simple answer. Data model design, like software design, is an art rather than a science. Nevertheless, the following guidelines are a good starting point.

Look at your mathematical model. Most scientific computing is applied mathematics, and in that case your mathematical model is usually the best starting point for your data model. Write down a precise and detailed definition of it. Of course, input from the application domain is useful as well.

Inspect your algorithms. Write down their inputs and outputs in particular. Group data items that belong together.

Think in terms of plain English. Describe your data as you would explain it to a fellow scientist, not as you would implement it in your programs. If you are a Fortran or C programmer, you might automatically translate "a list of points in 3D space" into "an $N \times 3$ array of floats." However, other programming languages encourage different choices.

Any information that can be deduced from other data shouldn't be part of your data model.

Avoid redundancy. Any information that can be deduced from other data shouldn't be part of your data model. Redundant information opens the door to inconsistent datasets. You of course might want to have redundant data in your implementation, usually for efficiency reasons, but it shouldn't leak out.

Design for extension. It's hardly ever a good idea in software development to design beyond clearly identified needs. Such over-engineering costs time and effort and can actually make future extensions more difficult if they don't turn out to be exactly the ones envisaged initially. However, imagining various possible extensions of your data model can help to make it more future-proof.

Always keep in mind that a data model is defined at a more abstract

level than any concrete implementation. The role of data models in software design is similar to the role of pseudocode. Both should contain the essential features of the data structures and algorithms used in a program, but none of the technical implementation details.

As a concrete example that will also serve to illustrate file format design, I propose a minimal data model for the chemical structure of molecules. The term *molecule* is taken to mean a set of atoms linked by chemical bonds. As a useful generalization of standard chemical terminology, commonly made in computational chemistry, a molecule may contain atoms that don't participate in any bond, and in fact a set of atoms without any bonds is accepted as an extreme case of a molecule. Here are the rules that define the data model:

- A molecule is defined by a name, a set of atoms, and a set of bonds.
- The name of a molecule is a character string.
- Each atom is identified by a unique name (a character string) consisting of its chemical element symbol optionally followed by an integer.
- Each bond is defined by a bond order (an integer) and a set of exactly two distinct atoms, each of which must also be an element of the atom set of the molecule.

A water molecule would thus be described as

name "water"
atoms {"O", "H1", "H2"}
bonds {(1, {"O", "H1"}), (1, {"O", "H2"})}.

Note the use of sets, rather than lists or arrays, for atoms and bonds, which expresses the fact that there's no natural order to these constituents. This is a design choice that favors physical realism over simplicity of implementation. Many data models in actual use in computational chemistry do impose an order on the atom set, because this turns out to be quite convenient in practice. However, such convenience, once it becomes a habit, can lead to bad design decisions, even at the level of mathematical models. For example, in the widely used Amber force field (http://ambermd.org/#ff), the potential energy of a molecule depends on the order of the atoms, even though that order is arbitrary and completely unrelated to any physical property of the molecule. Data models can help to avoid such mistakes.

Designing File Formats

Once you have your data model, you can think about file formats that implement it for archiving and for exchange with other programs. The ideal file format would have the following characteristics:

- *Independence of specific operating systems and compilers*—the specification should make it possible to write portable programs that read and write the file format anywhere.
- *Compactness*—even with today's terabyte-sized disks, wasting space is never a good idea.
- *I/O efficiency*—reading and writing data should be as fast as possible on all common computing platforms.
- *Simplicity for programmers*—if existing I/O libraries can be reused, your file format will be much more popular.
- *Convenient handling by scientists*—looking inside a file, or changing some value, should be doable with simple, powerful, and well-known tools.

In practice it's rarely possible to satisfy all of these conditions. As with other aspects of programming, the criteria of convenience and simplicity on one side and efficiency and compactness on the other often lead to contradicting requirements.

The most fundamental choice to make in designing a file format is the choice between a text-based and a binary format. Text formats ultimately represent all data as a sequence of ASCII or Unicode characters. The main advantage is that any text editor on any computer can be used to inspect and modify the data. On the other hand, text formats are neither compact nor efficient to read and write. Binary formats, which represent data by bit patterns identical or very similar to those used by the computer's CPU, have exactly the opposite characteristics: they're compact and I/O efficient, but require format-specific and sometimes platform-specific tools for any inspection or manipulation. It's often useful to define both a text and a binary format for the same data model.

Because data storage in files is a common need in all areas of computing, several generic file formats have been developed that can be adapted to a wide range of applications. These formats specify the representation of basic data items (numbers and text strings) and common data structures (lists, arrays, tables, associative arrays, and so on), which are exactly the basic units in terms of which data models are defined. Ready-to-use libraries for many popular programming languages make them convenient for the programmer, and the existence of generic tools that deal with these formats makes the users' life easier.

The historically first generic text representation of complex data structures

```
(molecule "water"
    (atoms "O" "H1" "H2")
    (bonds (1 "O" "H1") (1 "O" "H2")))
```
(a)

```
<molecule name="water">
   <atoms>O H1 H2</atoms>
   <bonds>
     <bond atoms="O H1" order=1 />
     <bond atoms="O H2" order=1 />
   </bonds>
</molecule>
```
(b)

```
molecule:
   name: water
   atoms: !!set {O, H1, H2}
   bonds: !!set
    ? order: 1
        atoms: !!set {O, H1}
    ? order: 1
        atoms: !!set {O, H2}
```
(c)

```
{"type": "molecule", "name": "water",
    "atoms": ["O", "H1", "H2"],
 "bonds": [{"order": 1, "atoms": ["O", "H1"]},
             {"order": 1, "atoms": ["O", "H2"]}]}
```
(d)

Figure 1. Possible representations of the chemical structure data model. The representations are shown in (a) s-expression, (b) XML, (c) "YAML Ain't Markup Language" (YAML), and (d) JavaScript Object Notation (JSON) formats.

was created in the 1950s together with the Lisp programming language. Known as *s-expressions* (for symbolic expressions), it's based on the list as the central data structure, with list elements being symbols, numbers, character strings, and other lists. Nested lists that also allow non-list elements are a way of encoding a tree structure, with list-type elements representing nodes and non-list elements representing leaves. A tree structure is a good, flexible choice, because most relevant data structures can easily be mapped to a tree.

Today's best-known generic text format is XML (www.w3.org/TR/rec-xml), defined as a simplified subset of the Standard Generalized Markup Language (SGML) for use in Web technology. XML and SGML were designed for applications where the majority of the data consists of text—basically character strings—but they can be used for different kinds of data at the cost of being rather verbose. The strong point of XML is a rich ecosystem of tools for creating, reading, validating, and transforming XML files.

Similar to s-expressions, XML is based on the concept of a tree structure. The items in the tree are called *elements* in XML terminology. Each element is delimited by a start tag and an end tag, and can have *content* (consisting of text and other elements) and *attributes* (an associative array). The label in the start and end tags identifies the element type. Defining a specific format based on XML comes down to choosing the tags for various element types and specifying constraints on the elements' contents and attributes. Examples of scientific data formats based on XML are Mathematics Markup Language (MathML; www.w3.org/Math) for mathematical formulas, Extensible Scientific Interchange Language (XSIL; http://resolver.caltech.edu/CaltechCACR:CACR-1999-171) for array and table data, Chemistry Markup Language (CML; www.xml-cml.org) for chemistry, and Systems Biology Markup Language (SBML; http://sbml.org) for systems biology.

Although XML has gained a lot of support in recent years, it isn't always the best choice for a text-based format. Its main disadvantage is its often verbose style, which increases file sizes and tends to make the data more difficult to understand to the human reader. XML is also rather complex, meaning that implementing XML I/O can be a significant effort, even when using XML libraries. Two other formats worth looking at are YAML (www.yaml.org), which stands for "YAML Ain't Markup Language," to emphasize its differences from XML, and JavaScript Object Notation (JSON; http://json.org), which despite its name is supported by more languages than just JavaScript. The two have much in common, with JSON scoring on ease of handling in programs and YAML having the edge in human readability.

Figure 1 shows how the aforementioned chemical structure data model can be represented in these four file formats. YAML is the only format that has a way to represent sets. In the

other formats, the sets are replaced by lists. The same choice could be made in YAML to yield more compact files. In the s-expression example, a bond is represented by a three-element list, which is more compact than the attribute-style representation I used in the other formats. Again, other choices can be made as a function of the design priorities.

For binary formats, the two main contenders are the Hierarchical Data Format (HDF5; www.hdfgroup.org/HDF5)[1] and the Network Common Data Form (NetCDF; www.unidata.ucar.edu/software/netcdf). Of the two, HDF5 is more flexible but also more difficult to use. Both HDF5 and NetCDF are based on multidimensional arrays as their basic data structure. They permit the storage of arrays that are too big to fit into the computer's working memory, providing efficient access to subarrays. Both libraries come with a set of generic tools for data management.

Data Models inside Your Programs

Most scientists are easily convinced of the utility of well-defined data models and file formats for archiving data and for exchanging it between programs and research groups. After all, we've all wasted a lot of time in guessing at file formats and writing conversion programs. The advantage of defining data models for the in-memory data structures of scientific software is less evident, but quite important as well. In fact, there are two main advantages.

This first one should be evident: the more similar the in-memory data model of your software is to the data model you use for file storage, the easier it is to read and write files. It's easier to program, but also more efficient for the computer, because there's less conversion to be done.

The second advantage is that well-defined data models make software more modular. What you pass around between functions are of course the data structures that your programming language proposes. But if your software is designed around data models, you'll probably pass around combinations of data structures that you identified in your data model as useful units. Without a data model, you're likely to pass around just the data that every individual function needs. You then have to think about each function's interface separately, whereas with a data model you see at a glance what your function parameters mean. For example, you might pass a point object to a function instead of three coordinates if your data model makes use of geometrical points.

Without a data model, you're likely to pass around just the data that every individual function needs.

Just as implementing a data model for disk storage requires defining a file format, implementing an in-memory representation of a data model requires a translation into the concrete data structures available in your chosen programming language. Because programming languages vary enormously in their data definition features, implementing a data model can take highly different forms. At one end of the spectrum, there are languages such as Fortran 77 that don't allow any user-defined data constructs. The only way to implement a data model is thus to provide a list of variables (scalars and arrays) and describe the relations between them by comments. At the other end, there are modern object-oriented or functional languages in which each data item in the data model can be implemented as one problem-specific data type.

Unfortunately, the art of data modeling in scientific computing is still in its infancy, with few tutorials or textbooks available to serve as guides. A good starting point is to look at existing scientific data models. Try an Internet search with some keywords from your field of research followed by "XML" or "HDF5." You might discover that there are already good data models around that you can simply adopt, or at least use as a starting point for your own development.

Reference

1. M. Poinot, "Five Good Reasons to Use the Hierarchical Data Format," *Computing in Science & Eng.*, vol. 12, no. 5, 2010, pp. 84–90.

Konrad Hinsen is a researcher at the Centre de Biophysique Moléculaire in Orléans, France, and at the Synchrotron Soleil in Saint Aubin. His research interests include protein structure and dynamics and scientific computing. Hinsen has a PhD in theoretical physics from RWTH Aachen University, Germany. Contact him at konrad.hinsen@cnrs-orleans.fr.