

Chapter 6

Basic Data and Computations

This chapter surveys a variety of topics dealing with different kinds of data and the computations provided for them. The topics are “basic” in two senses: they are among those most often covered in introductions to R or S-Plus; and most of them go back to fairly early stages in the long evolution of the S language.

On the data side, we begin with the various ways of organizing data that have evolved in R. Then object types (Section 6.2, page 141), which characterize data internally; vectors and vector structures (6.3, 143); and data frames (6.5, 166). Matrices and their computations are discussed together in Section 6.8, page 200. Other computational topics are: arithmetic and other operators (6.6, 184); general numeric computations (6.7, 191); statistical models (6.9, 218); random generators and simulation (6.10, 221); and the special techniques known as “vectorizing” (6.4, 157).

Many of the topics deserve a whole chapter to themselves, if not a separate book, given their importance to data analysis. The present chapter focuses on some concepts and techniques of importance for integrating the data and computations into programming with R, particularly viewed from our principles of effective exploration and trustworthy software. Further background on the topics is found in many of the introductions to R, as well as in the online R documentation and in some more specific references provided in the individual sections of the chapter.

6.1 The Evolution of Data in the S Language

Since its beginnings in 1976, the S language has gone through an evolution of concepts and techniques for representing data and organizing computations on data structures. Four main epochs can be identified, all of which are still with us, and all of which need to be understood to some extent to make use of existing software, and sometimes for new programming as well. Labeled by the names used for the corresponding mechanisms in R, the main epochs are:

1. *Object types*, a set of internal types defined in the C implementation, and originally called *modes* in S;
2. *Vector structures*, defined by the concept of vectors (indexable objects) with added structure defined by *attributes*;
3. *S3 classes*, that is, objects with class attributes and corresponding one-argument method dispatch, but without class definitions;
4. Formal *classes* with class definitions, and corresponding generic functions and general methods, usually called *S4 classes* and methods in R.

This section summarizes the relevance of each approach, with pointers to further details in this chapter and elsewhere. The main recommendation is to use formal classes and methods when developing new ideas in data structure, while using the other approaches for specialized areas of computing.

Object types: All implementations of the S language have started from an enumeration of object types or modes, implied by the very first design documents (such as the figure on page 476). In R, this takes the form of a field in the internal C structure, and the corresponding function `typeof()`. You need to deal with object types for some C extensions and when defining classes that extend a specific type. Section 6.2 gives details.

Vectors and vector structures: The concept of objects as dynamically indexable by integer, logical and perhaps character expressions also goes back to the early days. The S3 version of the language around 1988 added the notion of vector structures defined by named attributes, seen as complementary to the vector indexing. Section 6.3, page 143, discusses these concepts, which remain important for computing effectively with the language. The term *vectorizing* has evolved for computations with vectors that avoid indexing, by expressing computations in “whole object” terms. In favorable applications, efficiency and/or clarity benefits; see Section 6.4, page 157.

S3 classes: As part of the software for statistical models, developed around 1990 and after, a `class` attribute was used to dispatch single-argument methods. The attribute contained one or more character strings, providing a form of inheritance. Otherwise, the change to data organization was minimal; in particular, the content of objects with a particular class attribute was not formally defined. S3 classes are needed today to deal with software written for them (for example, the statistical model software (Section 6.9, page 218) and also for incorporating such data into modern classes and methods (see Section 9.6, page 362 for programming with S3 classes).

Formal (S4) classes: The S3 classes and methods gave a useful return on a small investment in changes to the language, but were limited in flexibility (single-argument dispatch) and especially in supporting trustworthy software. Classes with explicit definitions and methods formally incorporated into generic functions have been developed since the late 1990s to provide better support. That is the programming style recommended here for new software—chapters 9 and 10, for classes and methods respectively.

6.2 Object Types

For most purposes, `class(x)` is the way to determine what kind of thing object `x` really is. Classes are intended to be the official, public view, with as clear and consistent a conceptual base as possible. Deep down, though, objects in R are implemented via data structures in C. By definition, the *object type* corresponds to the set of possible types encoded in those structures. For a complete list of the internal types at the C level, see the *R Internals* manual in the R documentation or at the CRAN Web site. The function `typeof()` returns a character string corresponding to the internal object type of an object.

Table 6.1 lists the object types commonly encountered in R programming. The first column gives the class name for simple objects of the object type named in the second column. The expressions in the third column will evaluate to an object of the corresponding object type.

The classes in the rows down to the first line in the table are the basic vector classes; these correspond to a object type of the same name, except for type "double", indicating the specific C declaration for numeric data. For more discussion of these, see section 6.3. The classes in the second group of rows are the basic classes for dealing with the language itself. The first three object types correspond to function objects in R: "closure" for ordinary functions, "builtin" and "special" for primitive functions. (For details

| Class(es) | Object Type(s) | Examples |
|---------------------|----------------|---------------------------|
| "logical" | "logical" | TRUE; FALSE |
| "numeric" | "double" | 1; 0.5; 1e3 |
| "integer" | "integer" | as.integer(1) |
| "character" | "character" | "Carpe \n Diem" |
| "list" | "list" | list(a=1,b=plot) |
| "complex" | "complex" | 1 + .5i |
| "raw" | "raw" | as.raw(c(1,4,15)) |
| "expression" | "expression" | expression(a,1) |
| "function" | "closure" | function(x)x+1 |
| | "builtin" | `sin` |
| | "special" | `if` |
| "call" | "language" | quote(x+1) |
| "{", etc. (many) | "S4" | quote({}) new("track") |
| "name" | "symbol" | quote(x) |
| "environment" | "environment" | .GlobalEnv |

Table 6.1: Object Types in R. The types in the first group are vectors, the types in the first and second behave as non-reference objects. See the text for details, and for types generated from C.

on primitive functions, see Section 13.4, page 463.) Primitive functions are an R implementation extension, not part of the S language definition; for this reason, objects of all three object types have class "function". Conversely, one object type, "language", corresponds to essentially all the unevaluated expressions other than constants or names. Function calls, braced subexpressions, assignments, and other control structures have specific classes as objects, but all are in fact implemented by one object type. In effect, R organizes all "language" objects as if they were function calls. The last row in the second group, "S4", is used for objects generated from general S4 classes.

All the classes down to the second line in the table behave normally as arguments in calls, and can be used in class definitions. Classes can be defined to extend these classes, an important ability in programming with R. We might want a new class of data with all the properties of character vectors, but with some additional features as well. Similarly, programming techniques using the language might need to define objects that can behave

as functions but again have extra features. Examples of such classes are shown in Chapter 9, on pages 370 and 356. Objects from such classes retain the corresponding basic type, so that legacy code for that type works as it should. If `x` has a class extending "character" or "function", then the value of `typeof(x)` will be "character" or "function" correspondingly. Objects from classes that do *not* extend one of the basic object types have type "S4".

In contrast to the types in the first two groups of the table, the object types in the third group are essentially references. Passing `.GlobalEnv` as an argument to a function does not create a local version of the environment. For this reason, you should not attach attributes to such objects or use them in the `contains=` part of a class definition, although they can be the classes for slots.

Besides the object types in the table, there are a number of others that are unlikely to arise except in very specialized programming, and in the internal C code for R. These include "pairlist", "promise", "externalptr", and "weakref". Except for the first of these, all are reference types. For a complete table of types, see Chapter 2 of the *R Language Definition* manual.

6.3 Vectors and Vector Structures

The earliest classes of objects in the S language, and the most thoroughly "built-in" are vectors of various object types. Essentially, a vector object is defined by the ability to index its elements by position, to either extract or replace a subset of the data. If `x` is a vector, then

```
x[i]
```

is a vector with the same type of data, whenever `i` defines a set of indices (in the simplest case, positive integer values). If `y` is also a vector (in the simplest case, with the same type of data as `x`), then after evaluating

```
x[i] <- y
```

the object `x` will be a vector of the same type of data. The range of possibilities for `i` and `y` is much more general than the simple cases, but the simple cases define the essence of the `vector` class, and the general cases can be understood in terms of the simplest case, as discussed on page 146.

An early concept for organizing data in the S language was the *vector structure*, which attached *attributes* to vectors in order to imply structure, such as that of a multi-way array. Vector structures were a step on the way

to classes of objects, and usually can be subsumed into class definitions. However, there are some objects and computations in R that still work directly on attributes, so an understanding of vector structures is included, starting on page 154.

Basic classes of vectors

Table 6.2 shows the basic classes of vectors built into R. Identically named functions (`numeric()`, `logical()`, etc.) generate vectors from the corresponding classes.

| Class | Data Contained |
|--------------|------------------------------|
| "logical" | Logicals: (TRUE, FALSE). |
| "numeric" | Numeric values. |
| "character" | Character strings. |
| "list" | Other R objects. |
| "complex" | Complex numbers. |
| "raw" | Uninterpreted bytes. |
| "integer" | Integer numeric values. |
| "single" | <i>For C or Fortran only</i> |
| "expression" | Unevaluated expressions. |

Table 6.2: The vector classes in R.

The basic vector classes are the essential bottom layer of data: indexable collections of values. Single individual values do not have a special character in R. There are no scalar objects, either in the sense of separate classes or as an elementary, “sub-class” layer, in contrast to other languages such as C or Java. Computations in the language occasionally may make sense only for single values; for example, an `if()` test can only use one logical value. But these are requirements for the result of a particular computation, not a definition of a different kind of data (for computations that need a single value, see page 152).

For the basic vectors, except for "list" and "expression", the individual elements can only be described in terms of the implementation, in C. In terms of that language, the data in each vector class corresponds to an array of some C type. Only when writing code in C to be used with R are you likely to need the explicit type, and even then the best approach is to hide the details in C macros (see Section 11.3, page 420).

Numeric data occasionally involves considering two other classes, "integer"

and "single." The type for data of class "numeric", as returned by `typeof()`, is "double", indicating that the internal data is double-precision floating point. There is also a separate class and type for "integer" data. You can force data into integer (via function `as.integer()`) and a few functions do return integer results (function `seq()` and operator ``:``), but because R does not do separate integer computations very often, trying to force integer representation explicitly can be tricky and is usually not needed.

Users sometimes try to force integer representation in order to get "exact" numerical results. In fact, the trick required is not integer representation, but integral (i.e., whole number) values. These are exactly represented by type "double", as long as the number is not too large, so arithmetic will give exact results. Page 192 shows an example to generate "exact" values for a numeric sequence.

The "single" class is still more specialized. Essentially, it exists to notify the interface to C or Fortran to convert the data to single precision when passing the vector as an argument to a routine in those languages. R does not deal with single-precision numeric data itself, so the class has no other useful purpose.

The S language includes a built-in vector type representing points in the complex plane, class "complex".¹ See `?complex` for generating complex vectors and for manipulating their various representations. The class has its own methods for arithmetic, trigonometric, and other numerical computations, notably Fourier transforms (see `?fft`). Most functions for numerical computations do accept complex vectors, but check the documentation before assuming they are allowed. Complex data is also suitable for passing to subroutines in Fortran or C. Fortran has a corresponding built-in type, which can be used via the `.Fortran()` interface function. There is a special C structure in R for calls to `.C()`, which also matches the complex type built into modern C compilers on "most" platforms. Section 11.2, page 415 discusses the interfaces to C and Fortran; for the latest details, see section 5.2 of the *Writing R Extensions* manual.

Vectors of type "raw" contain, as the name suggests, raw bytes not assumed to represent any specific numeric or other structure. Although such data can be manipulated using `x[i]`-style expressions, its essential advantage is what will *not* be done to it. Aside from explicitly defined computations, raw vectors will not likely be changed, and so can represent information

¹Statistics research at Bell Labs in the 1970s and before included important work in spectral analysis and related areas, relying on computations with complex data. Complex vectors became a built-in object type with S3.

outside standard R types, exactly and without accidental changes. Neither of these properties applies, for example, if you try to use numeric data to represent exact values. Even if the initial data is exactly as expected, numeric computations can easily introduce imprecision. On the other hand, you can generally count on "raw" data remaining exactly as created, unless explicitly manipulated. For this reason, objects of class "raw" may be used to pass data from arbitrary C structures through R, for example.

Vectors of type "expression" contain unevaluated expressions in the language. The main advantages of such an object over having a "list" object with the same contents are the explicit indication that all elements should be treated as language expressions and the generating function, `expression()`, which implicitly quotes all its arguments:

```
> transforms <- expression(sin(x), cos(x), tan(x))
```

You can mix such literal definitions with computed expressions by replacing elements of the vector:

```
> transforms[[4]] <- substitute(f(x), list(f=as.name(fname)))
```

Indexing into vectors

The most important data manipulation with vectors is done by extracting or replacing those elements specified by an index expression, using the function in R represented by a pair of single square brackets:

```
x[i]
x[i] <- y
```

These are the fundamental *extraction* and *replacement* expressions for vectors.

When `i` is a vector of positive values, these index the data in a vector, from 1 for the first element to `length(x)` for the last. Indexing expressions of type "logical" are also basic, with the obvious interpretation of selecting those elements for which the index is `TRUE`. If you have used R or the S language at all, you have likely used such expressions. It's valuable, however, to approach them from the general concept involved, and to relate various possibilities for the objects `x`, `i`, and `y` in the example to the general concept.

In contrast to programming languages of the C/Java family, expressions like `x[i]` are not special, but are evaluated by calling a reasonably normal R function, with the name `[`. As with any functional computation in the S language, the value is a new object, defined by the arguments, `x` and `i`. The

expression does not “index into” the vector object in the sense of a reference as used in other languages. Instead, evaluating `x[i]` creates a new object containing the elements of `x` implied by the values in `i`. As an example, let’s use the sequence function, `seq()`, to generate a vector, and index it with some positive values.

```
> x <- seq(from=1.1, to=1.7, by=.1)
> x
[1] 1.1 1.2 1.3 1.4 1.5 1.6 1.7
> length(x)
[1] 7
> x[c(1,3,1,5,1,7)]
[1] 1.1 1.3 1.1 1.5 1.1 1.7
```

Repeated values in the positive index are entirely within the standard definition, returning the values selected in the order of the index. Positive index values are silently truncated to integers: `x[1.9]`, `x[1.01]`, and `x[1]` all return the same subset.

When the index is a logical vector, it arises most naturally as the value of a test applied to `x` itself and/or to another vector indexed like `x`.

```
> x[x>1.45]
[1] 1.5 1.6 1.7
```

Logical indexes are applied to the whole vector; in particular, if `i` has length less than that of `x`, it is interpreted as if it were replicated enough times to be the same length as `x`. The logical index `c(TRUE, FALSE)` extracts the odd-numbered elements of any vector:

```
> x[c(TRUE, FALSE)]
[1] 1.1 1.3 1.5 1.7
```

Note the second special rule below, however, for the case that `i` is longer than `x`.

The behavior of the replacement expression, `x[i] <- y`, is to create and assign a new version of `x` in the current evaluation environment. In the new version, the values in `x` indexed by `i` have been replaced by the corresponding elements of `y`. Replacement expressions are evaluated by calling the corresponding *replacement function*, as discussed in Section 5.2, page 117. In this case the replacement function is ``[<-``, and the expression is equivalent to:

```
x <- `[<-`(x, i, y)
```

The behavior of replacement functions is still within a simple functional model: A replacement function computes and returns a new object, which is then assigned in the current environment, with the same name as before. In this case the new object is a copy of the old, with the indexed values replaced.

Unlike extraction, replacement can change the type of x . There is an implied ordering of basic vector types from less information to more. Logicals have only two values, numerics many, strings can represent numbers, and lists can hold anything. If the type of y is more general than that of x , the replacement will convert x to the type of y . For example:

```
> x[2] <- "XXX"
> x
[1] "1.1" "XXX" "1.3" "1.4" "1.5" "1.6" "1.7"
```

The numeric vector is converted to a character vector, on the reasoning that this would preserve all the information in x and in y . More details on conversions between basic types are given on page 149.

For simple positive or logical indexing expressions, the interpretation follows naturally from the concept of a vector. There are in addition a number of extensions to the indexing argument in the actual implementation. These can be convenient and you need to be aware of the rules. For your own computing, however, I would discourage taking too much advantage of them, at least in their more esoteric forms. They can easily lead to errors or at least to obscure code. With that in mind, here are some extensions, roughly ordered from the innocuous to the seriously confusing.

1. The index can be a vector of negative values. In this case, the interpretation is that the value should be all the elements of x except the elements corresponding to $-i$.

```
> x[-c(1,7)]
[1] 1.2 1.3 1.4 1.5 1.6
```

With this interpretation, the order of the negative values is ignored, and so are repeated values. You cannot mix positive and negative values in a single index.

2. An index for extraction or replacement can be longer than the current length of the vector. The interpretation is that the length of x is set to the largest index (implicitly) and the expression is applied to the

“stretched” version. Replacements can change the length of a vector by assigning beyond its current length.

Increasing the length of a vector is interpreted as concatenating the appropriate number of NA items, with NA being interpreted as an undefined value suitable for the type of the vector.

```
> length(x)
[1] 7
> x[c(1,9)]
[1] 1.1 NA
> x[c(1,9)] <- -1
> x
[1] -1.0 1.2 1.3 1.4 1.5 1.6 1.7 NA -1.0
```

Logical, numeric, character and complex object types have a built-in NA form; lists and expressions use NULL for undefined elements; type "raw" uses a zero byte.

3. Integer index arguments can contain 0 values mixed in with either all positive or all negative indices. These are ignored, as if all the 0 values were removed from the index.
4. When the index contains an undefined value, NA, the interpretation for extraction is to insert a suitable NA or undefined value in the corresponding element of the result, with the interpretation of undefined as above for the various types. In replacements, however, NA elements in the index are ignored.

R also has single-element extraction and replacement expressions of the form `x[[i]]`. The index must be a single positive value. A logical or negative index will generate an error in R, even if it is equivalent to a single element of the vector.

Conversion of vector types

The basic vector types have some partial orderings from more to less “simple”, in the sense that one type can represent a simpler type without losing information. One ordering, including the various numeric types, can be written:

```
"logical", "integer", "numeric", "complex", "character", "list"
```

If a simpler vector type (to the left in the ordering) is supplied where a less simple vector type is wanted, an automatic conversion will take place for numeric and comparison operators (see Section 6.6, page 186). The conversion rules are implemented in the internal code and are not part of the inheritance relations used when methods are dispatched. Defining a method corresponding to an argument of class "numeric", for example, does not result in that method being used when the argument is of class "logical" or "integer", even though those classes are "simpler" in terms of the listing above. That implementation decision could be argued, but it's perhaps best just to realize that the two parts of the language—basic code for operators and formal class relations—were written at very different times. In the early coding, there was a tendency to make as many cases "work" as possible. In the later, more formal, stages the conclusion was that converting richer types to simpler automatically in all situations would lead to confusing, and therefore untrustworthy, results.

The rules of conversion are basically as follows.

- Logical values are converted to numbers by treating FALSE as 0 and TRUE as 1.
- All simpler types are converted to "character" by converting each element individually (as, for example, in a call to `cat()` or `paste()`).
- All simpler types are converted to "list" by making each element into a vector of length 1.
- Numeric values are converted to "complex" by taking them to be the real part of the complex number.

Class "raw" is not included in the ordering; generally, your best approach is to assume it is not automatically converted to other types. Vectors of type "raw" are not numeric, and attempts to use them in numeric expressions will cause an error. They are allowed with comparison operators, however, with other vectors of any of the basic types except "complex". The implementation of the comparisons with types "logical", "integer", and "numeric" uses roughly the following logic. Interpret each of the elements (single bytes, remember) in the "raw" vector as a corresponding integer value on the range 0 to 255 ($2^8 - 1$), and then use that conversion in the comparison. This should be equivalent to applying the comparison to `as.numeric(x)` where `x` is the vector of type "raw". Watch out for comparisons with "character" vectors however. The rule, natural in itself, is that the comparison should be done as if with `as.character(x)`. But `as.character()` converts "raw" vectors by

replacing each element by the two hexadecimal characters that represent it, basically because this is how "raw" vectors are printed. As a result, the comparison is not at all the same as if the "raw" vector had first been converted to the numeric vector of its byte code. On the whole, avoid comparisons of "raw" vectors with "character" vectors, because they are only sensible if the character elements are each the print version of byte codes (and in this case they probably should have been converted to "raw" anyway). And just to make things worse, there is another conversion, `rawToChar()`, which interprets the bytes as character codes, entirely different from `as.character()`. The situation is further complicated by the existence in modern R of multiple character encodings to deal with international character sets. Read the documentation carefully and proceed with caution.

Besides automatic conversions, explicit coercion can be performed between essentially any of the basic vector classes, using `as()`. For the general behavior of `as()`, see Section 9.3, page 348; in the case of basic vector classes the methods used are identical to the corresponding class-specific functions, `as.integer()`, `as.character()`, etc. Some additional general rules for coercion include:

- Numeric values are coerced to logical by treating all non-zero values as `TRUE`.
- General numeric values are converted to integer by truncation towards zero.
- Complex values are converted to numeric by taking their real part.
- Character data is coerced to simpler types roughly as if the individual values were being read, say by `scan()`, as the simpler type. On elements for which `scan` would fail, the result is `NA`, and a warning is issued (but not an error as `scan()` would produce).
- Lists are converted to simpler types only if each element of the list is a vector of length one, in which case the coercion works one element at a time. (If an element is itself a list of length 1, that produces an `NA`, perhaps accidentally.)
- Conversion from "raw" to all numeric types generally treats each byte as an integer value; conversion to "raw" generally converts numeric values to integer, uses values that fit into one byte and sets all others to 00 (which is generally used instead of `NA` with type "raw").

Conversion from "raw" to "character" produces the hexadecimal codes, from "00" to "ff". Unfortunately, conversion from "character" to "raw" first converts to integer, not likely to be what you want. The inverse of the conversion to "character" is `scan(x, raw())`.

As will perhaps be clear, the wise approach is to look for ambiguous conversions and either deal with them as makes sense for your own application or else generate an error. The rules are pretty reasonable for most cases but should not be taken as universally appropriate.

Single values when you need them

Vectors in the S language play a particularly important role in that there are no scalar object types underlying them, and more fundamentally there is no lower layer beneath the general model for objects in the language. Contrast this with Java, for example. Java has a general model of classes, objects and methods that forms the analogous programming level to programming with R. The implementation of a Java method, however, can contain scalar variables of certain basic types, which are not objects, as well as arrays, which are objects (sort of) but not from a class definition. The situation in R is simpler: everything is an object and anything that looks like a single value of type numeric, logical or character is in fact a vector. The lower layer is provided instead by the inter-system interface to C, as discussed in Chapter 11.

However, some computations really do need single values. To ensure that you get those reliably and that the values make sense for the context may require some extra care.

By far the most common need for single values comes in tests, either conditional computations or iterations.

```
if(min(sdev) > eps)
  Wt <- 1/sdev
```

The condition in the `if` expression only makes sense if `min(sdev) > eps` evaluates to a single value, and that value must be unambiguously interpretable as `TRUE` or `FALSE`. Similarly, the condition in a `while` loop must provide a single `TRUE` or `FALSE` each time the loop is tested.

So, what's the problem? Often no problem, particularly for early stages of programming. If we know that `eps` was supplied as a single, positive numeric value and that `sdev` is a non-empty vector of numbers (none of them missing values and, most likely, none of them negative), then `min(sdev)` is a

single numeric value and the comparison evaluates to a single TRUE or FALSE. The test will either pass or not, but in any case will be computable.

Problems can arise when such a computation occurs inside a function with the objects `eps` and `sdev` passed in or computed from arguments. Now we are making assertions about the way in which the function is called. As time goes by, and the function is used in a variety of contexts, these assertions become more likely to fail. For the sake of the *Prime Directive* and trustworthy software, tests of the arguments should be made that ensure validity of the conditional expression. The tests are best if they are made initially, with informative error messages.

As your functions venture forth to be used in unknown circumstances, try to add some tests on entry that verify key requirements, assuming you can do so easily. Don't rely on conditional expressions failing gracefully deep down in the computations. Failure of assumptions may not generate an error, and if it does the error message may be difficult to relate to the assumptions.

Consider two failures of assumptions in our example: first, that `sdev` was of length zero; second, that it contained NAs. For trustworthy computation we might reasonably want either to be reported as an error to the user. As it happens, the second failure does generate an error, with a reasonable message:

```
> if(min(sdev) > eps) Wt <- 1/sdev
Error in if (min(sdev) > eps) Wt <- 1/sdev :
  missing value where TRUE/FALSE needed
```

With a zero length vector, however, `min()` returns infinity, the test succeeds and `Wt` is set to a vector of length zero. (At least there is a warning.)

If the test is computationally simple, we can anticipate the obvious failures. For more elaborate computations, the test may misbehave in unknown ways. Having verified all the obvious requirements, we may still feel nervous about obscure failures. A strategy in such situations is to guarantee that the computation completes and then examine the result for validity.

Evaluating the expression as an argument to the function `try()` guarantees completion. The `try()` function, as its name suggests, attempts to evaluate an expression. If an error occurs during the evaluation, the function catches the error and returns an object of class "try-error". See `?try` for details and Section 3.7, page 74, for a related programming technique.

Here is an ultra-cautious approach for this example:

```
testSd <- try(min(sdev) > eps)
if(identical(testSd, TRUE))
```

```

Wt <- 1/sdev
else if(!identical(testSd, FALSE))
  if(is(testSd, "try-error"))
    stop("Encountered error in testing sdev ",
         testSd, "\n")
  else
    stop("Testing sdev produced an invalid result: ",
         summaryString(testSd))

```

The only legitimate results of the test are `TRUE` and `FALSE`. We check for either of these, identically. If neither is the result, then either there was an error, caught by `try()`, or some other value was computed (for example, `NA` if there were any missing values in `sdev`). With `try()`, we can re-issue an error message identifying it as a problem in the current expression. For more complicated expressions than this one, the message from the actual error may be obscure, so our extra information may be helpful.

In the case of an invalid result but no error, one would like to describe the actual result. In the example, the function `summaryString()` might include the class and length of the object and, if it is not too large, its actual value, pasted into a string. Writing a suitable `summaryString()` is left as an exercise. A reasonable choice depends on what you are willing to assume about the possible failures; in the actual example, there are in fact not very many possibilities.

Some situations require single values other than logicals for testing; for example, computing the size of an object to be created. Similar guard computations to those above are possible, with perhaps additional tests for being close enough to a set of permitted values (positive or non-negative integers, in the case of an object's size, for example).

Overall, trustworthy computations to produce single values remain a challenge, with the appropriate techniques dependent on the application. Being aware of the issues is the important step.

Vector structures

The concept of the vector structure is one of the oldest in the S language, and one of the most productive. It predates explicit notions of classes of objects, but is best described using those notions. In this section we describe the general `"structure"` class, and the behavior you can expect when computing with objects from one of the classes that extend `"structure"`, such as `"matrix"`, `"array"`, or `"ts"`. You should keep the same expectations in

mind when writing software for structure classes, either methods for existing classes or the definition of new classes. We use the name of the corresponding formal class, “structure”, to mean “vector structure” throughout this section, as is common in discussions of R.

A class of objects can be considered a structure class if it has two properties:

1. Objects from the class contain a data part that can be any type of basic vector.
2. In addition to the data part, the class defines some organizational structure that describes the layout of the data, but is not itself dependent on the individual values or the type of the data part.

For example, a matrix contains some data corresponding to a rectangular two-way layout, defined by the number of rows and columns, and optionally by names for those rows and columns. A time-series object, of class “ts”, contains some data corresponding to an equally-spaced sequence of “times”.

Matrices and time-series are regular layouts, where the structure information does not grow with the total size of the object, but such regularity is not part of the requirement. An irregular time series, with individual times for the observations, would still satisfy the structure properties.

The importance of the “structure” class comes in large part from its implications for methods. Methods for a number of very heavily used functions can be defined for class “structure” and then inherited for specific structure classes. In practice, most of these functions are primitives in R, and the base code contains some of the structure concept, by interpreting certain vector objects with attributes as a vector structure. The base code does not always follow the structure model exactly, so the properties described in this section can only be guaranteed for a formal class that contains “structure”.

The two properties of vector structures imply consequences for a number of important R functions. For functions that transform vectors element-by-element, such as the `Math()` group of functions (trigonometric and logarithmic functions, `abs()`, etc.), the independence of data and structure implies that the result should be a structure with the data transformed by the function, but with the other slots unchanged. Thus, if `x` is a matrix, `log(x)` and `floor(x)` are also matrices.

Most of the functions of this form work on numeric data and return numeric data, but this is not required. For example, `format(x)` encodes vectors as strings, element-by-element, so that the data returned is of type “character”. If `x` is a vector structure, the properties imply that `format(x)`

should be a structure with the same slots as `x`; for example, if `x` is a matrix, then `format(x)` should be a character matrix of the same dimensions.

Binary operators for arithmetic, comparisons, and logical computations are intrinsically more complicated. For vectors themselves, the rules need to consider operands of different lengths or different types. Section 6.6, page 186, gives a summary of the R behavior. What if one or both of the operands is a vector structure? If only one operand is a structure, and the result would have the same length as the structure, the result is a structure with the same slots. If both operands are structures, then in general there will be no rational way to merge the two sets of properties. The current method for binary operators (function `Ops()`) returns just the vector result. In principle, the structure might be retained if the two arguments were identical other than in their data part, but testing this generally is potentially more expensive than the basic computation. Particular structure classes such as `"matrix"` may have methods that check more simply (comparing the dimensions in the `"matrix"` case).

The `base` package implementation has rules for matrices, arrays, and time-series. If one argument is one of these objects and the other is a vector with or without attributes, the result will have the matrix, array, or time-series structure unless it would have length greater than that of the structure, in which case the computation fails. The rule applies to both arithmetic and comparisons. Operations mixing arrays and time-series or arrays with different dimensions produce an error. See Section 6.8, page 200, for more discussion of computations with matrix arguments.

For vectors with arbitrary attributes, the current base code in R for operators and for element-by-element functions is not consistent with treating these as a vector structure. Numeric element-by-element functions usually retain attributes; others, such as `format()` drop them. For arithmetic operators, if one argument has attributes, these are copied to the result. If both arguments have attributes, then if one argument is longer than the other, arithmetic operators use its attributes; if the arguments are of equal length, the result combines all the attributes from either argument, with the left-hand value winning for attributes appearing in both arguments. Comparison operators drop all attributes, except for the `names` attribute.

The overall message is clear: For consistent vector structure behavior, you should create an explicit class, with `"structure"` as a superclass.

To create a vector structure class formally, call `setClass()` with the `contains=` argument specifying either class `"structure"` or some other S4 vector structure class. Class `"structure"` is a virtual class that extends class `"vector"`, which in turn extends all the basic vector object types in R.

For example, here is a class "irregTS" for an irregular time-series structure, with an explicit time slot.

```
setClass("irregTS", representation(time = "DateTime"),
        contains = "structure")
```

Objects from this class will inherit the structure methods, providing much of the desired behavior automatically. Methods then need to be added for the particular behavior of the class (at the least, a `show()` method and some methods for ``[`.`)

One can program methods for various functions with class "structure" in the signature. The methods will be inherited by specific vector structure classes such as "irregTS". In addition, methods are supplied in R itself for the formal "structure" class that implement the vector structure view described in this section. For a list of those currently defined:

```
showMethods(classes = "structure")
```

This will list the corresponding signatures; another `showMethods()` call for a particular function with `includeDefs = TRUE` will show the definitions.

An important limitation arises because the informal vector structures such as matrices, arrays, time-series, and S3 classes will not inherit formal methods for class "structure", at least not with the current version of R. Nor does it generally work to have such informal vector structures in the `contains=` argument of a formal class definition, largely for the same reason. So formal and informal treatment of vector structures don't currently benefit each other as much as one would like.

6.4 Vectorizing Computations

Over the history of R and of S, there has been much discussion of what is variously called "avoiding loops", "vectorizing computations", or "whole-object computations", in order to improve the efficiency of computations. The discussion must appear rather weird to outsiders, involving unintuitive tricks and obscure techniques. The importance of vectorizing is sometimes exaggerated, and the gains may depend subtly on the circumstances, but there are examples where computations can be made dramatically faster. Besides, re-thinking computations in these terms can be fun, and occasionally revealing.

The original idea, and the name "vectorizing", come from the contrast between a single expression applied to one or more R vectors, compared to

a loop that computes corresponding single values. Simple vector objects in R consist of n elements, typically numbers. The value of n is often the number of observed values in some data, or a similar parameter describing the size of our application. Very important practical problems involve large applications; n may of necessity be large, and in any case we would like our computations to be reasonably open to large-data applications. A computation of interest that takes one or more such vectors and produces a new vector nearly always takes computing time proportional to n , when n is large. (At least proportional: for the moment let's think of computations that are linear in the size of the problem. The interest in vectorizing will only be stronger if the time taken grows faster than linearly with n .)

Vectorizing remains interesting when the parameter n is not the size of a vector, but some other parameter of the problem that is considered large, such as the length of loops over one or more dimensions of a multiway array; then n is the product of the dimensions in the loops. In other examples, the loop is an iteration over some intrinsic aspect of the computation, so that n is not a measure of the size of the data but may still be large enough to worry about. In an example below, n is the number of bits of precision in numeric data, not a variable number but still fairly large.

We're considering linear computations, where elapsed time can be modeled reasonably well, for large n , by $a + bn$, for some values of a and b , based on the assertion that some multiple of n calls to R functions is required. The goal of vectorizing is to find a form for the computation that reduces the proportionality, b . The usual technique is to replace all or part of the looping by a single expression, possibly operating on an expanded version of the data, and consisting of one or more function calls. For the change to be useful, these functions will have to handle the larger expanded version of the data reasonably efficiently. (It won't help to replace a loop by a call to a function that does a similar loop internally.) The usual assumption is that "efficiently" implies a call to functions implemented in C. Notice that the C code will presumably do calculations proportional to n . This is not quantum computing! The hope is that the time taken per data item in C will be small compared to the overhead of n function calls in R.

The fundamental heuristic guideline is then:

Try to replace loops of a length proportional to n with a smaller number of function calls producing the same result, usually calls not requiring a loop in R of order n in length.

Functions likely to help include the following types.

1. Functions that operate efficiently on whole objects to produce other whole objects, usually of the same size and structure; examples include the arithmetic and other binary operators, numerical transformation, sorting and ordering computations, and some specialized filtering functions, such as `ifelse()`.
2. Operations that extract or replace subsets of objects, using expressions of the form `x[i]`, provided that the indexing is done on a significantly sizable part of `x`.
3. Functions that efficiently transform whole objects by combining individual elements in systematic ways, such as `diff()` and `cumsum()`.
4. Functions to transform vectors into multi-way arrays, and vice versa, such as `outer()` and certain matrix operations;
5. Functions defined for matrix and array computations, such as matrix multiplication, transposition, and subsetting (these are used not just in their standard roles, but as a way to vectorize other computations, as the example below shows).
6. New functions to do specialized computations, implemented specially in C or by using some other non-R tools.

A different approach uses functions that directly replace loops with sequences of computations. These are the `apply()` family of functions. They don't precisely reduce the number of function calls, but have some other advantages in vectorizing. The `apply()` functions are discussed in Section 6.8, page 212.

The craft in designing vectorized computations comes in finding equivalent expressions combining such functions. For instance, it's fairly common to find that a logical operation working on single values can be related to one or more equivalent numerical computations that can apply to multiple values in one call. Other clues may come from observing that a related, vectorized computation contains all the information needed, and can then be trimmed down to only the information needed. None of this is purely mechanical; most applications require some reflection and insight, but this can be part of the fun.

First, a simple example to fix the idea itself. Suppose we want to trim elements of a vector that match some string, `text`, starting from the end of the vector but not trimming the vector shorter than length `nMin` (the

computation arises in summarizing available methods by signature). The obvious way in most programming languages would be something such as:

```
n <- length(x)
while(n > nMin && x[[n]] == text)
  n <- n-1
length(x) <- n
```

Quite aside from vectorizing, the use of `==` in tests is a bad idea; it can return `NA` and break the computation. Instead, use an expression that will always produce `TRUE` or `FALSE`.

Back to vectorizing. First, let's think object rather than single number. We're either looking for the new object replacing `x`, or perhaps the condition for the subset of `x` we want (the condition is often more flexible). The key in this example is to realize that we're asking for one of two logical conditions to be true. Can we express these in vector form, and eliminate the loop? If you'd like an exercise, stop reading here, go off and think about the example.

The idea is to compute a logical vector, call it `ok`, with `TRUE` in the first `n` elements and `FALSE` in the remainder. The elements will be `TRUE` if either they come in the first `nMin` positions or the element of `x` does not match `text`. The two conditions together are:

```
seq(along = x) <= nMin # c(1,2,...,n) <= nMin;
| is.na(match(x, text))
```

The use of `seq()` here handles the extreme case of zero-length `x`; the function `match()` returns integer indices or `NA` if there is no match. See the documentation for either of these if you're not familiar with them.

The vectorized form of the computation is then:

```
ok <- seq(along = x) <= nMin | is.na(match(x, text))
```

Chances are that either `ok` or `x[ok]` does what we want, but if we still wanted the single number `n`, we can use a common trick for counting all the `TRUE` values:

```
n <- sum(ok)
```

The computations are now a combination of a few basic functions, with no loops and so, we hope, reasonably efficient. Examining `is.na()`, `match()`, and `sum()` shows that all of them go off to C code fairly quickly, so our hopes are reasonable.

If that first example didn't put you off, stay with us. A more extended example will help suggest the process in a more typical situation.

Example: Binary representation of numeric data

Our goal is to generate the internal representation for a vector of numeric data. Numeric data in R is usually floating point, that is, numbers stored internally as a combination of a binary fraction and an integer exponent to approximate a number on the real line. As emphasized in discussing numeric computations (Section 6.7, page 191), it's occasionally important to remember that such numbers are only an approximation and in particular that most numbers displayed with (decimal) fractional parts will not be represented exactly.

Suppose we wanted to look at the binary fractions corresponding to some numbers. How would we program this computation in R?

To simplify the discussion, let's assume the numbers have already been scaled so $.5 \leq x < 1.0$ (this is just the computation to remove the sign and the exponent of the numbers; we'll include it in the final form on page 165). Then all the numbers are represented by fractions of the form:

$$b_1 2^{-1} + b_2 2^{-2} + \dots + b_m 2^{-m}$$

where m is the size of the fractional part of the numerical representation, and b_i are the bits (0 or 1) in the fraction. It's the vector `b` of those bits we want to compute. (Actually, we know the first bit is 1 so we only need the rest, but let's ignore that for simplicity.)

This is the sort of computation done by a fairly obvious iteration: replace x by $2x$; if $x \geq 1$ the current bit is 1 (and we subtract 1 from x); otherwise the current bit is zero. Repeat this operation m times. In a gloriously C-like or Perl-like R computation:

```
b <- logical(m)
for(j in 1:m) {
  x <- x * 2
  b[[j]] <- (x >= 1)
  if(b[[j]])
    x <- x - 1
}
```

We will vectorize this computation in two ways. First, the computation as written only works for `x` of length 1, because the conditional computation depends on `x >= 1` being just one value. We would like `x` to be a numeric vector of arbitrary length; otherwise we will end up embedding this computation in another loop of length n . Second, we would like to eliminate the loop of length m as well. Admittedly, m is a fixed value. But it can be large

enough to be a bother, particularly if we are using 64-bit numbers. The situation of having two parameters, either or both of which can be large, is a common one (think of the number of variables and number of observations).

Eliminating the loop over `1:m` can be done by a conversion rather typical of vectorizing computations. Notice that the iterated multiplication of `x` by 2 could be vectorized as multiplying `x` by a vector of powers of 2:

```
pwr<- 2^(1:m)
xpwr<- x*pwr
```

Getting from here to the individual bits requires, excuse the expression, a bit of imaginative insight. Multiplying (or dividing) by powers of 2 is like shifting left (or right) in low-level languages. The first clue is that the i -th element of `xpwr` has had the first i bits of the representation shifted left of the decimal point. If we truncate `xpwr` to integers and shift it back, say as

```
xrep<- trunc(xpwr)/pwr
```

the i -th element is the first i bits of the representation: each element of `xrep` has one more bit (0 or 1) of the representation than the previous element. Let's look at an example, with `x <- .54321`:

```
> xrep
 [1] 0.5000000 0.5000000 0.5000000 0.5000000 0.5312500 0.5312500
 [7] 0.5390625 0.5429688 0.5429688 0.5429688 0.5429688 0.5429688
[13] 0.5430908 0.5431519 0.5431824 0.5431976 0.5432053 0.5432091
[19] 0.5432091 0.5432091 0.5432096 0.5432098 0.5432099 0.5432100
```

Next, we isolate those individual bits, as powers of two ($b_j 2^{-j}$): the first bit is `xrep[[1]]`, and every other bit j is `xrep[[j]] - xrep[[j-1]]`. The difference between successive elements of a vector is a common computation, done by a call to the function `diff()`. Using that function:

```
bits<- c(xrep[[1]], diff(xrep))
```

We would then need to verify that the method used for `diff()` is reasonably efficient. An alternative is to realize that the $m - 1$ differences are the result of subtracting `xrep` without the first element from `xrep` without the last element: `xrep[-1] - xrep[-m]`, using only primitive functions. When we account for multiple values in `x`, however, we will need a third, more general computation.

Assuming we want the individual bits to print as 0 or 1, they have to be shifted back left, which is done just by multiplying by `pwr`. Now we have a complete vectorization for a single value. With the same value of `x`:


```

> pwrs <- 2^(1:m)
> xpwrs <- x*pwrs
> xrep <- trunc(xpwrs)/pwrs
> bits <- c(xrep[[1]], diff(xrep))*pwrs
> bits
[1] 1 0 0 0 1 0 1 1 0 0 0 0 1 1 1 1 1 1 0 0 1 1 1 1

```

The computations all operate on vectors of length m ; in particular, in the second line, the single value in x is replicated in the multiplication.

The remaining task is to generalize the computation to n values in x . We need n instances of each of the length- m computations. As is often the case, we can get the desired result by expanding the objects into matrices, here with n rows and m columns. To begin, we replicate the x into m columns and define `pwrs` as a matrix with n identical rows. All the computations down to defining `xrep` expand to compute $n*m$ values at once:

```

n <- length(x)
x <- matrix(x, n, m)
pwrs <- matrix(2^(1:m), n, m, byrow = TRUE)
xpwrs <- x*pwrs
xrep <- trunc(xpwrs)/pwrs

```

What about `c(xrep[[1]], diff(xrep))`? Now we want this computation to apply to each row of the matrix, indexing on the columns. Remember that `diff()` was just a function to subtract `x[[2]]` from `x[[1]]`, etc. We could introduce a loop over rows, or use the `apply()` function to do the same loop for us.

But in fact such patterned row-and-column combinations can usually be done in one function call. Here we do need a “trick”, but fortunately the trick applies very widely in manipulating vectors, so it’s worth learning. Differences of columns are simple versions of linear combinations of columns, and all linear combinations of columns can be written as a matrix multiplication.

```
bits <- xrep %*% A
```

with A some chosen m by m matrix. The definition of matrix multiplication is that, in each row of `bits`, the first element is the linear combination of that row of `xrep` with the first column of A , and so on for each element.

This trick is useful in many computations, not just in examples similar to the current one. For example, if one wanted to sum the columns of a matrix, rather than doing any looping, one simply multiplies by a matrix with a single column of 1s:

```
x %*% rep(1, ncol(x))
```

The vector on the right of the operator will be coerced into a 1-column matrix.

To see the form of A , consider what we want in elements 1, 2, ... of each row—that determines what columns 1, 2, ... of A should contain. The first element of the answer is the first element of the same row of $xrep$, meaning that the first column of A has 1 in the first element and 0 afterwards. The second element must be the second element minus the first, equivalent to the second column being -1, 1, 0, 0, The form required for A then has just an initial 1 in the first column, and every other column j has 1 in row j and -1 in row $j - 1$:

| | [,1] | [,2] | [,3] | [,4] | [,5] | [,6] | [,7] | [,8] | [,9] | [,10] |
|------|------|------|------|------|------|------|------|------|------|-------|
| [1,] | 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [2,] | 0 | 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| [3,] | 0 | 0 | 1 | -1 | 0 | 0 | 0 | 0 | 0 | 0 |
| ... | | | | | | | | | | |

Constructing this sort of matrix is the other part of the trick. We take account of the way R stores a matrix; namely, by columns. If we lay out all the columns one after the other, the data in A starts off with 1 followed by $m-1$ zeroes, followed by -1, and then followed by the same pattern again. By repeating a pattern of $m+1$ values, we shift the pattern down by one row each time in the resulting matrix. We can create the matrix by the expression:

```
A <- matrix(c(1, rep(0, m-1), -1), m, m)
```

The general thought process here is very typical for computing patterned combinations of elements in matrices. The example in Section 6.8, page 206, constructs a general function to produce similarly patterned matrices.

This is clearly a computation with enough content to deserve being a function. Here is a functional version. We have also added the promised code to turn an arbitrary x into a sign, an exponent, and a binary fraction on the range $.5 \leq x < 1.0$. The complete representation is then made up of three parts: sign, exponent, and the bits corresponding to the fraction.

We express this result in the form of a new class of objects. The three parts of the answer are totally interdependent, and we would be inviting errors on the part of users if we encouraged arbitrary manipulations on them. Having a special class allows us to be specific about what the objects mean and what computations they should support. Some extra work is required to create the class, but the results of the computations will be easier to

understand and to work with when their essential structure is captured in a class definition. Our users will benefit in both insightful exploration (the *Mission*) and trustworthy software (the *Prime Directive*) from our extra effort. (The details are explored in Section 9.2, page 343; for now, just consider them part of the background.)

```
binaryRep <-
function(data, m = .Machine$double.digits) {
  x <- data
  n <- length(x)

  xSign <- sign(x)
  x <- xSign * x
  exponent <- ifelse(x > 0, floor(1+log(x, 2)), 0)
  x <- x/2 ^ exponent
  pwrs <- binaryRepPowers(n, m)
  x <- matrix(x, n, m)
  xpwrs <- x * pwrs
  xrep <- trunc(xpwrs)/pwrs
  bits <- (xrep %*% binaryRepA(m)) *pwrs

  bits[] <- as.integer(bits[])
  new("binaryRep", original = data,
      sign = as.integer(xSign),
      exponent = as.integer(exponent),
      bits = binaryRepBits(bits))
}
```

The body of the function is in three parts, separated by empty lines. It's the middle part that we are concerned with here, the rest has to do with the class definition and is discussed on page 343.

The function `sign()` returns ± 1 , and multiplying by the sign shifts `x` to positive only. The next line computes the exponent, specifically the largest integer power of 2 not larger than `x` (play with the expression `floor(1+log(x, 2))` to convince yourself). Dividing by 2 to this power shifts `x` to be larger than .5 but not larger than 1. The rest of the computation is just as we outlined it before.

This has been a long example, but in the process we have touched on most of the mechanisms listed on page 158. We have also worked through heuristic thinking typical of that needed in many vectorizing computations.

6.5 Statistical Data: Data Frames

Through most of this book, we use the term *data* in the computational sense, meaning numerical or other information in some form that can be stored and processed. But in statistics, and more generally in science, the term has an older meaning, from the Latin “datum” for a “given”, and so for an observed value.² We examine in this section one concept central to data in this sense, the *data frame*, and its implementation in a variety of computational contexts.

The topic is large and important, so while this section is not short, we can only cover a few aspects. The plan is as follows. We start with some reflections on the concept (as usual, you can skip this to get on to techniques). Next, we examine how data frames as a concept can be used in several languages and systems: in R (page 168), in Excel and other spreadsheets (page 173), and in relational database systems (page 178). Each of these discussions focuses on acquiring and using data corresponding to data frames in the corresponding system. Finally, on page 181, we consider transferring data frames between systems, mostly meaning between R and other systems.

The data frame concept

The concept of a data frame lies at the very heart of science. Gradually, very long ago, and in more than one place, people began to act on the belief that things could be meaningfully observed, and that taking these observations as given could lead to true, or at least useful, predictions about the future. This is in fact the central notion for our computational discussion: that there are things that can be observed (in data analysis called *variables*), and that it’s meaningful to make multiple *observations* of those variables. The computational version of the concept is the *data frame*. This section deals mainly with practical computations that implement the data frame concept.

If we look for early evidence of the underlying concept, we must go back long before science as such existed. Consider, for example, the structures known as “calendar stones” and the like. These are structures created to behave in a particular way at certain times of the year (typically the summer or winter solstice). Stonehenge in England, built some five thousand years ago, is designed so that the rising sun on the winter solstice appears in a

²The *American Heritage Dictionary* [13] has a nice definition including both senses. It also disposes of the pedantry that “data” is always plural: in modern usage it can be a “singular mass entity like information”.

particular arch of the monument. Some modern interpretations suggest that the monument is designed to match several patterns in the sun/earth/moon system (for example, the book by John North [20]). Similar devices existed in the ancient Near East and Central America.

Think of the process of designing such a calendar stone. Someone must observe the positions of the sun as it rises each day. At Stonehenge, this position will appear to move farther south each day as winter approaches, until at the solstice the sun “stands still”, and then begins to move back north. If the site is designed also to correspond to changes in the appearance and position of the moon, corresponding observations for its changes had to be made.

The builders of Stonehenge had no written language, so they probably did not record such data numerically. But they must have made systematic observations and then drew inferences from them. From the inferences they designed a huge structure whose orientation came from a fundamentally scientific belief that observing data (in particular, observing variables such as length of day and sunrise position) would lead to a useful prediction. Where the sun stood still last year predicts where it will stand still in the years to come.

We seem to have digressed a long way indeed from software for data analysis, but not really. It can't be stressed too emphatically how fundamental the data frame concept is for scientific thinking or even for more informal empirical behavior. We select observable things, variables, and then make observations on them in the expectation that doing so will lead to understanding and to useful models and prediction.

Two consequences for our needs arise from the fundamental role of data frame concepts. First, the concepts have influenced many areas of computing, scientific and other. Software ranging from spreadsheets to database management to statistical and numerical systems are all, in effect, realizing versions of the data frame concept, different in terminology and organization, but sharing ideas. We will benefit from being able to make use of many such systems to capture and organize data for statistical computation. Second and related, in order to exploit these diverse systems, we need some central framework of our own, some statement of what data frames mean for us. Given that, we can then hope to express our computations once, but have them apply to different realizations of the data frame ideas. From the perspective of R, it is the class definition mechanism that gives us the essential tools for a central description of data frames. Section 9.8, page 375 outlines one such framework.

The "data.frame" class in R

The S language has included the specific "data.frame" class since the introduction of statistical modeling software (as described in the *Statistical Models in S* book [6]). This is an informal ("S3") class, without an explicit definition, but it is very widely used, so it's well worth describing it here and considering its strengths and limitations. Section 9.8, page 375, discusses formal class definitions that might represent "data.frame" objects.

A "data.frame" object is essentially a named list, with the elements of the list representing variables, in the sense we're using the term in this section. Therefore, each element of the list should represent the same set of observations. It's also the intent that the object can be thought of as a two-way array, with columns corresponding to variables. The objects print in this form, and S3 methods for operators such as ``[`` manipulate the data as if it were a two-way array. The objects have some additional attributes to support this view, for example to define labels for "rows" and "columns". Methods allow functions such as `dim()` and `dimnames()` to work as if the object were a matrix. Other computations treat the objects in terms of the actual implementation, as a named list with attributes. The expression

```
w$Time
```

is a legal way to refer to the variable `Time` in data frame `w`, and less typing than

```
w[, "Time"]
```

However, using replacement functions to alter variables as components of lists would be dangerous, because it could invalidate the data frame by assigning a component that is not a suitable variable. In practice, a large number of S3 methods for data frames prevent most invalid replacements.

Because of the focus on software for statistical models, the variables allowed originally for "data.frame" objects were required to be from one of the classes that the models software could handle: numerical vectors, numerical matrices, or categorical data ("factor" objects). Nothing exactly enforced the restriction, but other classes for variables were difficult to insert and liable to cause strange behavior. R has relaxed the original restrictions, in particular by providing a mechanism to read in other classes (see the argument `colClasses` in the documentation `?read.table` and in the example below).

As a first example, let's read in some data from a weather-reporting system as a data frame, and then apply some computations to it in R.

Software for a weather station provides for data export in comma-separated value form. Here are the first 10 lines of an exported file:

```

Time, TemperatureF, DewpointF, PressureIn, WindDirection, WindDirectionDegrees, \
WindSpeedMPH, WindSpeedGustMPH, Humidity, HourlyPrecipIn, Conditions, Clouds, \
dailyrainin, SoftwareType
2005-06-28 00:05:22, 72.7, 70.6, 30.13, ESE, 110, 3, 6, 93, 0.00, , -RA, , VWS V12.07
2005-06-28 00:15:46, 72.7, 70.6, 30.12, ESE, 105, 2, 5, 93, 0.00, , -RA, , VWS V12.07
2005-06-28 00:35:28, 72.7, 70.3, 30.12, East, 100, 3, 6, 92, 0.00, , OVC024, , VWS V12.07
2005-06-28 00:45:40, 72.5, 70.1, 30.12, ESE, 113, 6, 6, 92, 0.00, , OVC024, , VWS V12.07
2005-06-28 01:05:04, 72.5, 70.1, 30.11, ESE, 110, 0, 7, 92, 0.00, , OVC100, , VWS V12.07
2005-06-28 01:15:34, 72.5, 70.1, 30.10, East, 91, 1, 2, 92, 0.00, , OVC100, , VWS V12.07
2005-06-28 01:35:09, 72.3, 70.2, 30.10, SE, 127, 0, 5, 93, 0.02, , OVC009, 0.02, VWS V12.07
2005-06-28 01:45:33, 72.3, 70.5, 30.09, ESE, 110, 2, 2, 94, 0.04, , OVC009, 0.04, VWS V12.07
2005-06-28 02:05:21, 72.3, 70.5, 30.09, ESE, 110, 1, 6, 94, 0.04, , OVC009, 0.04, VWS V12.07

```

The first line contains all the variable names; to show it here we have broken it into 3, but in the actual data it must be a single line. R has a function, `read.table()`, to read files that represent "data.frame" objects, with one line of text per row of the object, plus an optional first line to give the variable names.

Two file formats are widely used for data that corresponds to data frames: *comma-separated-values files* (as in the example above) and *tab-delimited files*. Two corresponding convenience functions, `read.csv()` and `read.delim()`, correspond to such files. Both functions then call `read.table()`. For the data above:

```
weather1 <- read.csv("weather1.csv")
```

The result has the desired structure of a data frame with the variables named in the first line of the file:

```

> colnames(weather1)
 [1] "Time"           "TemperatureF"
 [3] "DewpointF"     "PressureIn"
 [5] "WindDirection" "WindDirectionDegrees"
 [7] "WindSpeedMPH"  "WindSpeedGustMPH"
 [9] "Humidity"      "HourlyPrecipIn"
[11] "Conditions"    "Clouds"
[13] "dailyrainin"  "SoftwareType"
> dim(weather1)
 [1] 92 14

```

All is not quite well, however. The first column, `Time`, does not fit the originally planned variable classes, not being either numeric or categorical. The

entries for the column contain date-times in the international standard format: 2005-06-28 00:05:22, for example. Some R software does understand time formats but they are not automatically converted in `read.table()`. Because the text is not numeric, the default action is to treat the column as a factor, but because each time is distinct, the factor has as many levels as there are observations.

```
> wTime <- weather1$Time
> class(wTime)
[1] "factor"
> length(levels(wTime))
[1] 92
```

R has an S3 class "POSIXct" that corresponds to time represented numerically. S3 methods exist to convert from character data to this class. The function `read.table()` allows variables from this class, and from any class that can be coerced from a character vector, through an optional argument `colClasses`, in which the user specifies the desired class for columns of the data frame. If told that the `Time` column should have class "POSIXct", `read.table()` will make the correct conversion.

So with a slight extension to the previous call, we can set the `Time` variable to an appropriate class:

```
> weather1 <- read.csv("weather1.csv",
+   colClasses = c(Time = "POSIXct"))
```

Now the variable has a sensible internal form, with the advantage that it can be treated as a numeric variable in models and other computations.

The `colClasses` argument is one of several helpful optional arguments to `read.table()` and its friends:

colClasses: The `colClasses` argument supplies the classes (the names, as character strings) that you want for particular columns in the data. Thus, for example, "character" keeps the column as character strings, where the default is to turn text data into factors, but see `as.is` below. This argument can also be used to skip columns, by supplying "NULL" as the class.

skip: The number of initial lines to omit.

header: Should the first row read in be interpreted as the names for the variables?

`as.is`: Should text be treated as character vectors, rather than the traditional default, which turns them into factors? It can be a per-column vector, but if factors are irrelevant, just supply it as `TRUE`.

There are many other arguments; see `?read.table`. Similar choices arise when importing data into other systems, such as spreadsheet or database programs. The discussion continues on page 182.

Once data frame objects are created, they can be used with a variety of existing R packages, principally for statistical models (see Section 6.9, page 218) and for the trellis/lattice style of plotting (see Section 7.6, page 280). These both use the idea of formula objects to express compactly some intended relation among variables. In the formula, the names of the variables appear without any indication that they belong to a particular data frame (and indeed they don't need to). The association with the data frame is established either by including it as an extra argument to the model-fitting or plotting function, or else by attaching the data frame to make its variables known globally in R expressions. The attachment can be persistent, by using the `attach()` function, or for a single evaluation by using the `with()` function, as shown on page 172.

For example, to plot temperature as a function of time in our example, one could use the `xyplot()` function of the `lattice` package to produce Figure 6.1 on page 172:

```
> xyplot(TemperatureF ~ Time, data = weather1)
```

The labels on the horizontal axis in the plot need some help, but let's concentrate here on the relationship between the data and the computations. The `data` argument to `xyplot()` and to similar plotting and model-fitting functions supplies a context to use in evaluating the relevant expressions inside the call to the function. The details are sometimes important, and are explored in the discussions of model software and `lattice` graphics. The essential concept is that the object in the `data` argument provides references for names in the `formula` argument. Formulas are special in that the explicit operator, `~`, is symbolic (if you evaluate the formula, it essentially returns itself). In the `xyplot()` call, the left and right expressions in the formula are evaluated to get the vertical and horizontal coordinates for the plot. You could verbalize the call to `xyplot()` as:

```
Plot: TemperatureF ~ (as related to) Time
```

The data frame concept then comes in via the essential notion that the variables in the data frame do define meaningful objects, namely the observations made on those named variables.

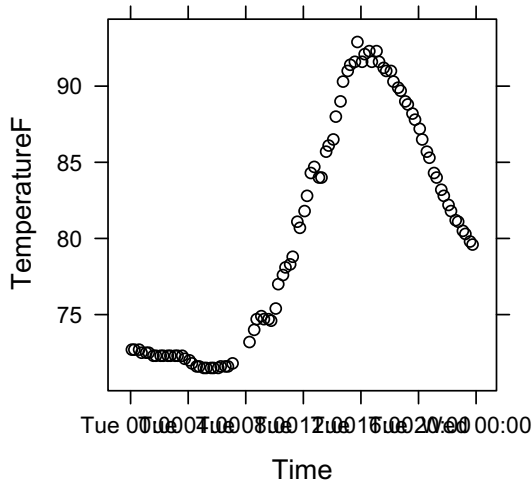


Figure 6.1: Scatter plot of variables from a data frame of weather data.

One can use the same conceptual framework in general computations, either locally by explicitly evaluating an expression using `with()`:

```
> with(weather1, mean(diff(Time)))
Time difference of 15.60678 mins
```

or persistently by attaching the data frame to the session:

```
> attach(weather1)
> mean(diff(Time))
Time difference of 15.60678 mins
```

Using `attach()` has the advantage that you can type an arbitrary expression involving the variables without wrapping the expression in a call to `with()`. But the corresponding disadvantage is that the variable names may hide or be hidden by other objects. R will warn you in some cases, but not in all. For this reason, I recommend using a construction such as `with()` to avoid pitfalls that may seem unlikely, but could be disastrous.

As an example of the dangers, suppose you had earlier been studying a different set of weather data, say `weather2`, and for convenience “copied” some of the variables to the global environment:

```
> Time <- weather2$Time
```

The global `Time` object hides the one in the attached data frame, and if the `attach()` occurred earlier, no warning is issued. You're supposed to remember the explicit assignment. But if you were, say, to evaluate a model formulated in terms of several variables in the `weather1` data, you could easily forget that just one of those happened to have been hidden. Nothing in the expression itself would reveal that you had just computed an incorrect answer, seriously violating the *Prime Directive*.

Therefore, the general advice is always: if the answer is important, make the computations explicit about what data is used, as the `with()` function did above. The data supplied to `with()` will be searched first, so that other sources of data will not override these. There is still a chance to mistakenly use an object that is *not* in the supplied data (perhaps a mistyped name), because R insists on looking for objects in the chain of parent environments of the data object. To be strict about all the available objects in a call to `with()` requires constructing an environment with a suitable parent environment. For example, if `ev` is an environment object:

```
parent.env(ev) <- baseenv()
```

will set the parent environment of `ev` to the environment of the `base` package, essentially the minimum possible. If you're starting with a `"data.frame"` object rather than an environment, the same restriction can be enforced by using the `enclos` argument to `eval()` or `evalq()`. The strict way to evaluate `diff(Time)` as above would be

```
evalq(diff(Time), weather1, baseenv())
```

If the expression requires functions from a package, you need something more generous than `baseenv()`. It's often useful to evaluate an expression using the namespace of a relevant package. For example, to evaluate an expression using the namespace of package `"lattice"`:

```
evalq(xyplot(TemperatureF ~ Time), weather1,
      asNamespace("lattice"))
```

Data frame objects in spreadsheet programs

Because spreadsheets are all about two-way layouts, they have a natural affinity for the data frame concept. In fact, Excel and other spreadsheets are very widely used for computations on data that can be viewed as a data frame. It has been asserted, not always facetiously, that Excel is the world's most widely used statistical software. Spreadsheets include facilities for

summaries and plotting that are applied in many data-intensive activities. They are less likely to have a wide range of modern data analysis built in, making R a natural complement for serious applications.

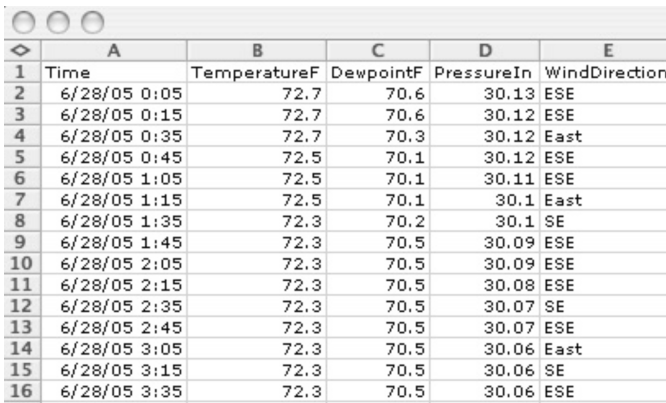
This section will present some techniques for using data frame objects inside spreadsheet programs. Combined with techniques for exporting data from R, these will allow the results of analysis to be brought into the spreadsheet. More sophisticated techniques for interfacing to R from a spreadsheet are possible and desirable, but considerably more challenging to program. See, for example, `RDCOMEvents` and related packages at the `omegahat` Web site; this approach would in principle allow sufficiently intrepid programmers to access R functions and objects from the spreadsheet, at least for Windows applications. The interface in the other direction, for analysis based in R, is discussed in Chapter 12.

Excel is very widely used, but is not the only spreadsheet program. Its competitors include several open-source systems, notably `OpenOffice.org`. Most of the techniques discussed below are found in `OpenOffice.org` and other spreadsheets as well, perhaps with some variation in user interface. In the discussion below, using “spreadsheet” to describe a system means Excel or one of its major competitors.

Let’s begin by importing into Excel the same `csv` file shown on page 169. To import the file, select the menu item for importing external data from a text file. In the version of Excel I’m using, the import menu selection is:

Data > Get External Data > Import Text File...

You then interact with an Excel “Wizard” in a sequence of dialogs to identify the file, choose the delimiter (comma) and specify the format for individual columns (usually “General”, but see below). The text file is then imported as a worksheet, in Excel terminology. Here’s the upper left corner:



| | A | B | C | D | E |
|----|--------------|--------------|-----------|------------|---------------|
| 1 | Time | TemperatureF | DewpointF | PressureIn | WindDirection |
| 2 | 6/28/05 0:05 | 72.7 | 70.6 | 30.13 | ESE |
| 3 | 6/28/05 0:15 | 72.7 | 70.6 | 30.12 | ESE |
| 4 | 6/28/05 0:35 | 72.7 | 70.3 | 30.12 | East |
| 5 | 6/28/05 0:45 | 72.5 | 70.1 | 30.12 | ESE |
| 6 | 6/28/05 1:05 | 72.5 | 70.1 | 30.11 | ESE |
| 7 | 6/28/05 1:15 | 72.5 | 70.1 | 30.1 | East |
| 8 | 6/28/05 1:35 | 72.3 | 70.2 | 30.1 | SE |
| 9 | 6/28/05 1:45 | 72.3 | 70.5 | 30.09 | ESE |
| 10 | 6/28/05 2:05 | 72.3 | 70.5 | 30.09 | ESE |
| 11 | 6/28/05 2:15 | 72.3 | 70.5 | 30.08 | ESE |
| 12 | 6/28/05 2:35 | 72.3 | 70.5 | 30.07 | SE |
| 13 | 6/28/05 2:45 | 72.3 | 70.5 | 30.07 | ESE |
| 14 | 6/28/05 3:05 | 72.3 | 70.5 | 30.06 | East |
| 15 | 6/28/05 3:15 | 72.3 | 70.5 | 30.06 | SE |
| 16 | 6/28/05 3:35 | 72.3 | 70.5 | 30.06 | ESE |

With the spreadsheet, as in the previous section with R, there are a few details to get right, which will also raise some points of general interest.

In an R `"data.frame"` object the variable names are attributes of the data, but in a spreadsheet essentially everything resides in the worksheet itself. Concepts of slots or attributes are uncommon; instead, spreadsheets use individual cells informally to store items other than data values. It's a typical convention to take the first row, or several rows, to include names for the columns and other contextual information. Formally, however, the columns are always "A", "B", "C", etc. and the rows are always "1", "2", "3", etc. The first row happens to contain text items "Time", "TemperatureF", "DewpointF", etc. that we will use to label variables. The remaining rows contain the actual data in the usual sense.

Having a column that contains its name as the first element would pose a problem in R if the data were numeric or any other class than `"character"`. The class of the variable would have to be something able to contain either a string or a number, and computations on the variable would be more complicated. Fortunately, the presence of a name in the first row of every column is less crippling in Excel. For one thing, many computations are defined for a *range* in the worksheet, a rectangular subset of the worksheet defined by its upper-left and lower-right corners cells. So the data-only part of the worksheet starts at the `"A2"` cell, in spreadsheet terminology, meaning the first column and the second row.

Names in the first row can cause problems, however, when specifying the format for importing the `csv` file. The Excel wizard allows you to choose a format (`numeric`, `text`, `date`) for each column. Because the variable names are stored in the first row, you cannot choose `numeric` for `TemperatureF` or `date` for `Time`. Instead, Excel allows and suggests choosing format `"General"`, which means that each cell is formatted according to a heuristic interpretation of its contents. Typically, the first cell in each column is text and the rest will be date, number, or text appropriately. If you do this in the present example, however, you will see some strange cells labeled `"Name?"` in the column for the `Clouds` variable. That's because several cells for this column are recorded in the file as `"-RA"` and the Excel heuristic throws up its hands for this entry: it starts out looking like a number but then turns into text. The fix is to specify this column manually as `"Text"` to the wizard, so as not to lose information. The result then looks as it did in the file:

| | L | |
|---|--------|----|
| ε | Clouds | da |
| | -RA | |
| | -RA | |
| | OVC024 | |
| | OVC024 | |
| | OVC100 | |
| | OVC100 | |
| | OVC009 | |
| | OVC009 | |
| | OVC009 | |
| | OVC009 | |
| | | |

Take a warning from the example, however. Although the spreadsheet may seem to be dealing with objects in a similar sense to R, that is not really the case. The spreadsheet model for computation is quite different, with an emphasis on cells and ranges of cells as basic concepts. Rather than using class definitions to add extra information, spreadsheet programs tend to differentiate cells within the table.

Nevertheless, there is enough commonality between the systems to allow for exporting and importing data. For the example we've been looking at, the only catch is that dates are by default saved in the local format, not the international standard. Inferring the format of dates is an example in the discussion of computing with text in Chapter 8 (see Section 8.6, page 321).

Worksheets can be exported into the same `csv` format used to import the data. To illustrate exporting data frames, let's look at a truly classic data set, the observations taken by Tycho Brahe and his colleagues on the declination of Mars (the angle the planet makes with the celestial equator). There are over 900 observations taken in the late 16th century. This fascinating dataset, which inspired Johannes Kepler to his studies of the orbit of Mars, was made available in an Excel spreadsheet by Wayne Pafko at the Web site pafko.com/tycho. Suppose we want to export the data, to be used as a data frame in R. The data may be read into Excel, or into other spreadsheet programs that accept Excel files. A menu selection, typically *Save As*, will give you an option to save as a comma-separated-values, or ".csv" file. The upper-left corner of the worksheet looks like this:

| Year | Day | Time | Quote | Volume | Page |
|------|-------------------------------|--------------|---|--------|------|
| 1582 | DIE 12 NOUEMBRIS, MANE. | | Declinatio [MS] 23 7 B | 10 | 174 |
| 1582 | DIE 30 DECEMBRIS | | AE: R. [MS] 107o 56' Declin. 26o 3 | 10 | 197 |
| 1582 | DIE 27 DECEMBRIS | | declinatio [MS] 26o 22 1/3 et Akeno | 10 | 200 |
| 1583 | DIE 18 JANUARI, VESPERI | | Declinatio 27 18 minus bona | 10 | 244 |
| 1584 | DIE 13 NOUEMBRIS, A.M. | H.13 26 P.M. | Declinatio [MS] B. 15 54 | 10 | 321 |
| 1584 | DIE 27 NOUEMBRIS | H.2 15' | Declinatio [MS] 14 42 | 10 | 322 |
| 1584 | DIE 20 DECEMBRIS AD VESPERAS. | | Decl. [MS] (erat prope horizont.) 14 24 | 10 | 322 |
| 1584 | DIE 21 DECEMBRIS AD VESPERAS. | | Declinatio [MS] 14 24 | 10 | 322 |

In this example, we use OpenOffice.org rather than Excel to open and then save the spreadsheet. The program provides some useful flexibility, including a simple option to save the data with quoted text fields. The first few lines of the resulting ".csv" file (with lines truncated to fit on the printed page) are:

```
"Tycho Brahe's Mars Observations",,,,,,,,,,,,,,
,"Source: Tychonis Brahe Dani Opera Omnia",,,,,,,,,,,,,,
,"Imput by: Wayne Pafko (March 24, 2000)",,,,,,"Brahe's Declinati
,"[MS] = Mars Symbol (you know...the "male" sign)",,,,,,"(not a
,,,,,,,,,,,,,
,"Year","Day","Time","Quote","Volume","Page",,"Year","Month","Day"
,1582,"DIE 12 NOUEMBRIS, MANE.",,"Declinatio [MS] 23 7 B",10,174
,1582,"DIE 30 DECEMBRIS",,"Afc. R. [MS] 107o 56' Declin. 26o 3
,1582,"DIE 27 DECEMBRIS",,"declinatio [MS] 26o 22 1/3' et Afcenfi
```

The row numbers and the column letters shown in the spreadsheet are not part of the actual data and are not saved.

Then the `read.csv()` function can import the data into R, as in the example on page 169. The first five lines of the file are comments, which we'd like to skip. Also, the text data is just text, so we need to suppress the default computations to turn text into factor variables, by supplying the option `as.is = TRUE`. With two optional arguments then, we can read in the data and examine it:

```
> mars <- read.csv("mars.csv", skip = 5, as.is = TRUE)
> dim(mars)
[1] 923 21
> sapply(mars, class)
      X          Year          Day          Time
"logical"  "integer"  "character" "character"
  Quote          Volume          Page          X.1
"character" "integer"  "integer"  "logical"
  Year.1          Month          Day.1          Day..adj.
"integer"  "integer"  "integer"  "integer"
  Hour          Min Days.since.1.AD          Date
"integer"  "numeric"  "numeric"  "numeric"
  X.2          Dec..deg.          Dec..min.          Dec..sec.
"numeric"  "integer"  "integer"  "integer"
Declination
"numeric"
```

The call to `sapply()` computes and displays the class of each of the variables in the data frame, exploiting its implementation as a list.

In this example, we might have chosen to preprocess the text data to remove uninteresting columns, such as two empty fields, `x` and `x.1`. However, these do no particular harm. Cleaning up after reading into R is probably easier in this case. In case you're wondering what the 21 variables are doing: The spreadsheet contains a number of intermediate variables used to arrive at numeric estimates of time and declination, starting from the highly irregular journal entries (in variable `Quote`). The computation is nontrivial and the data would be fascinating to examine as an example of text analysis.

Data frames in a relational database

From the perspective of statistical computing, the most essential feature of relational databases is that their model for data centers on the *table*, a two-way organization of columns (our variables) by rows (our observations), providing a thoroughly natural analog to a data frame. The analogy is very useful. In this section, we concentrate on using the tools of relational database software directly to store and access data from the data frame perspective. In addition, R packages provide inter-system interfaces to such software, so that SQL queries, such as those illustrated here, can be invoked from R. A discussion of the intersystem interface aspect is provided in Section 12.7, page 446. In addition, database systems usually allow exporting of tables as comma-separated-values files, so that the techniques discussed on page 181 can be used.

Relational databases intersect statistical computing most frequently because such databases are sources of data for analysis. The databases have often been organized and collected for purposes such as business transactions, commercial record keeping, or government information, as well as for scientific data collection. In this context, the data analysis is usually only concerned with extracting data (*queries* in database terminology), possibly also applying some summary calculation at the same time. Frequently these databases are very large, relative to the size of data objects used directly for analysis. In fact, relational databases may also be worth considering for data that is constructed or modified during the analysis (*data manipulation* or *transactions* in database terminology), either because of the software's ability to handle large amounts of data or to interface both read-only and modifiable portions of the data.

Access to data in relational databases other than for whole tables uses queries expressed in SQL, the *Structured Query Language*. SQL is common to essentially all relational database systems, and is in fact an international standard language. In spite of its name, the language includes data manip-

ulation and transaction commands as well as queries.

It's fortunate that SQL is supported by essentially all relational database systems, because there are many of these, and when the database was created for purposes other than data analysis, the analyst usually must use whatever particular system was chosen for this database. Of the many such systems, three will usefully stand in for the range of options.

SQLite: An open-source system implemented as a C library and therefore embeddable in other applications, including R.

MySQL[®]: Also an open-source system, emphasizing competitive capability for large applications.

Oracle[®]: One of the most successful commercial systems.

SQLite is usually the easiest to interface to R and to install on a particular platform. All three systems are implemented on a wide range of platforms, however. The other two are more likely to be competitive for large applications, and they do explicitly compete for such applications. In the following discussion we use DBMS to stand for one of these three or some other reasonably compatible relational database system.

SQL's design dates back to the 1970s, when computer languages for business applications often tried to look like human languages, specifically English. "Wouldn't it be nice if you could talk to your computer in English?" Perhaps, and modern technology does make that possible for some purposes, where simple tasks and voice recognition get us close to being able to talk to the machine in more-or-less ordinary English. Unfortunately, that was not what SQL and similar languages provided; instead, they introduced computer languages whose grammar was expressed in terms of English-language keywords and a syntax that combined the keywords into a fixed range of "phrases". English input not matching the grammar would be unintelligible to the parser, however natural to the programmer. In addition, the grammars of such languages tended to enumerate the expected requirements rather than starting with higher-level concepts such as objects, classes, and functional computing. The absence of such concepts makes the use of these languages less convenient for our purposes.

Most of these languages have faded from view, but SQL is definitely still with us. Fortunately, its English-like grammar is fairly simple, particularly if we're concerned largely with extracting data from tables.

Queries are performed by the **SELECT** command in SQL. (We follow a common convention by showing all command names and other reserved words

in SQL in upper case, but beware: SQL is usually case-insensitive, so don't rely on upper case and lower case to distinguish names.)

The `SELECT` command plays the role of expressions using the operator `[`]` in R for extracting subsets from two-way tables. The command takes three “arguments”, modifiers that correspond to the table object, the column subset, and the row subset. (There are also a number of optional modifiers.) In the terminology of English or other natural languages, the column subset is given by the direct object of the `SELECT` verb, the table by a `FROM` phrase, and the row subset by a `WHERE` clause. Suppose `weather` is a table in a DBMS database, with columns including `Date`, `TemperatureF`, and `Conditions`. Then a query that selects `TemperatureF` and `Conditions` from the rows for a particular date could be written:

```
SELECT TemperatureF, Conditions
FROM weather
WHERE Date == 2005-06-28 ;
```

Columns are specified by a comma-separated list of names or by `*` to select all columns. The table modifier is usually just a name, but it can also construct a table by combining information from several existing tables (the `JOIN` operation in SQL).

The `WHERE` clause is a logical expression involving columns of the table. The rows for which the expression is `TRUE` will be in the selected data. Simple expressions look familiar to users of R or other C-style languages, as above. To combine simple comparisons, use the `AND` and `OR` infix operators.

Notice that only expressions involving data in the columns can be used as row subsets: there are no intrinsic row numbers; unlike `data.frame` objects in R or a worksheet in spreadsheet, tables in SQL are not stored in a specific order by rows. This design decision was made partly for practical reasons, so that storage and updating could be done in a flexible, efficient way. But it makes considerable sense intrinsically as well. If we think of data frames in a general sense, the assumption that the “row numbers” of observations are meaningful is often not correct, and can lead to misleading results. One sometimes thinks of rows as representing the time when the observations are made, but those times may be unknown or meaningless (if observations were made at several recording sites, for example). Better in most cases to require time, place and/or other ordering variables to be included explicitly when they make sense.

Queries in SQL can have additional modifiers to add various features: to order the output, either by grouping together rows having the same value(s) on some column(s) or by sorting according to some column(s) (the `GROUP` and

ORDER modifiers); to filter the selected data on further criteria (the HAVING modifier); to direct the output to a file (the INTO modifier); and several others.

From a modern language perspective, it's easy to deplore the ad hoc nature of SQL, but its wide availability and efficiency in handling large amounts of data compensate for programming ugliness. In any case, if your applications are large, you're unlikely to avoid programming with relational databases forever. Do be careful, however, if you want to create SQL software that is portable among database systems. Nearly all major systems extend the standard SQL language definition, in sometimes inconsistent ways. A good SQL manual for a particular system should clarify the nonstandard parts, but don't count on it.

External files for data frames

A "data.frame" object in R or a table in a spreadsheet or relational database implements a version of the data frame concept. In many applications, you will need to add such data directly or communicate it between systems. There are several approaches, but two that work for most applications are either to enter data from a file of text in a standard format or to use an inter-system interface in one system to access data managed by another. Inter-system interfaces are described in Chapter 12. For large or otherwise computationally intensive applications, they have advantages of efficiency and flexibility over using files. They do require some initial setup and possibly customization, so it's reasonable to start with external files for less demanding applications. Files are also needed for getting data into a system or for communicating where an inter-system interface is not available. We consider here some questions that arise when preparing files to contain data-frame-style data, and techniques for dealing with the questions.

Once again, the basic data frame concept of observations and variables is the key: files are simplest when organized as lines of text corresponding to observations in the data frame, with each line containing values for the variables, organized as fields by some convention. R, spreadsheets, and most DBMS can import data from a text file laid out as lines of fields, with the fields separated by a specified character (with the tab and the comma the usual choices). Many other software systems also either export or import data in one of these forms. There are differences in detail among all the systems, so expect to do some cleaning of the data, particularly if you're exporting it from a more specialized system.

Text files using tabs or commas are often called "tab delimited files" or

“comma-separated-values files” respectively. They work roughly the same way, again up to the inevitable details. Here again is the beginning of the weather-station data introduced on page 169 and used to illustrate input of such data into R:

```

Time, TemperatureF, DewpointF, PressureIn, WindDirection, WindDirectionDegrees, \
WindSpeedMPH, WindSpeedGustMPH, Humidity, HourlyPrecipIn, Conditions, Clouds, \
dailyrainin, SoftwareType
2005-06-28 00:05:22,72.7,70.6,30.13,ESE,110,3,6,93,0.00,, -RA, ,VWS V12.07
2005-06-28 00:15:46,72.7,70.6,30.12,ESE,105,2,5,93,0.00,, -RA, ,VWS V12.07
2005-06-28 00:35:28,72.7,70.3,30.12,East,100,3,6,92,0.00,,OVC024, ,VWS V12.07
2005-06-28 00:45:40,72.5,70.1,30.12,ESE,113,6,6,92,0.00,,OVC024, ,VWS V12.07
2005-06-28 01:05:04,72.5,70.1,30.11,ESE,110,0,7,92,0.00,,OVC100, ,VWS V12.07
2005-06-28 01:15:34,72.5,70.1,30.10,East,91,1,2,92,0.00,,OVC100, ,VWS V12.07
2005-06-28 01:35:09,72.3,70.2,30.10,SE,127,0,5,93,0.02,,OVC009,0.02,VWS V12.07
2005-06-28 01:45:33,72.3,70.5,30.09,ESE,110,2,2,94,0.04,,OVC009,0.04,VWS V12.07
2005-06-28 02:05:21,72.3,70.5,30.09,ESE,110,1,6,94,0.04,,OVC009,0.04,VWS V12.07

```

The basic idea seems trivial, just values separated by a chosen character. Triviality here is a good thing, because the concept may then apply to a wide variety of data sources.

Here is a checklist of some questions you may need to consider in practice.

1. The first line: Variable names or not?
2. One line per observation or free-format values?
3. What are the field types (classes)?
4. What about special values in the fields?

For each question, you need to understand the requirements of the system that will import the data. Options in the system may let you adapt to the details of the file at hand, but expect to do some data cleaning in many cases. Data cleaning in this context requires computing with text, the subject of Chapter 8; that chapter presents some additional techniques related to the questions above.

First, variable names: Are they included in the data; does the target system want them; and do you want them? The answer to the first part depends on where the data came from. Many specialized systems that support data export in one of the data-frame-like formats do generate an initial line of column names. The names will be meaningful to the originating system, so they may not be in the natural vocabulary of the data analysis, but it’s a good default to leave them alone, to reduce the chance of confusion when you look back at the documentation of the originating system to understand

what some variable means. As for the target system, R's `read.table()` function allows them optionally, spreadsheets have no concept of variable names but an initial row of labels is a common convention, and for a relational database, you can't usually provide variable names in the imported table (and in any case you will have to have defined names and types for the variables before doing the import). Once you have decided what you need to do, removing/adding/changing the first line should be easy, but you may want to check that the line does really look like variable names if it should (and doesn't if it shouldn't).

For the free-format question, we usually need to ensure that lines in the file correspond to rows in the data frame. All three systems we've discussed really believe in importing lines of text. A line with p fields has $p-1$ delimiter characters, as shown in our example where $p == 14$. If the exporting system takes "comma separated values" literally, however, it may include a trailing delimiter at the end of each line or, worse, believe the input can be in free format ignoring the one-line/one-row correspondence. Excel does not mind the trailing comma, but the other systems do; and none of them will accept input in free format.

Turning free form input into regular lines is an exercise in computing with text, and can be handled either in R or in a language such as Perl or Python. The comparison is a good example of the tradeoffs in many applications: R is simpler, partly because it absorbs all the data in free form, and then just recasts it in a fixed number of fields per row. Perl has to do some more complicated book-keeping to read free-form lines and write fixed-form when enough fields have been read. But the extra logic means that the Perl code deals more efficiently with data that is large enough to strain memory limits in R. The details are presented as an example of computing with text in Section 8.6, page 325.

The third question, specifying the types or classes, may require attention to ensure that the contents of each variable conform to what the receiving system expects in that variable. All three systems need some specification of the "class" of data expected, although in both R and spreadsheets the variables can contain arbitrary character strings, so specification is only needed when something more specialized is expected. Standard SQL on the other hand is thoroughly old-fashioned in requiring declarations for the columns when the table is created (prior to actually filling the table with any data). The declarations even require widths (maximum number of characters) for text fields. A declaration for the table in our example might look something like:

```
CREATE TABLE weather (Time DATETIME, TemperatureF FLOAT, DewpointF FLOAT,
    PressureIn FLOAT, WindDirection VARCHAR(10), WindDirectionDegrees FLOAT,
    WindSpeedMPH FLOAT, WindSpeedGustMPH FLOAT, Humidity FLOAT,
    HourlyPrecipIn FLOAT, Conditions VARCHAR(20), Clouds VARCHAR(20),
    dailyrainin FLOAT, SoftwareType VARCHAR(20)
);
```

The weather data has an initial field giving the date and time, and then a variety of fields containing either numeric values or character strings, the strings usually coding information in a more-or-less standardized way. For all the systems, date/time and numeric fields are common occurrences, but each system has its own view of such data, and the data in the input needs to be checked for conformity.

The fourth question, special values, arises because transferring data from one system to another may require attention to conventions in the two systems about how values are represented. There are many potential issues: different conventions according to locale for dates and even for decimal numbers, techniques for quoting strings and escaping characters, and conventions for missing values. Techniques may be available in the receiving system to adjust for some such questions. Otherwise, we are again into text computations. For example, Section 8.6, page 321, has computations to resolve multiple formats for dates.

6.6 Operators: Arithmetic, Comparison, Logic

The S language has the look of other languages in the C/Java family, including a familiar set of operators for arithmetic, comparisons and logical operations. Table 6.3 lists them. Operators in the S language are more integrated, less specialized, and open to a wider role in programming than in many languages. In C, Java, and similar languages, the operators are nearly always built in. They translate into low-level specialized computations, and often assume that arguments are simple data. In languages that support OOP-style classes and methods, methods are not natural for operators, because method invocation is itself an operator (usually ".").

In R, each operator is a function, with the rights and generality of other functions, for the most part. Operator expressions are evaluated as function calls; all that is fundamentally different is that one can write these function calls in the familiar operator form. In fact

```
x+1
```

could legally be written as ``+`(x, 1)`, and would return the same value.

An operator in R is in fact anything the parser is willing to treat as one. Binary and unary operators (sometimes called infix and prefix operators) are any pattern *op* for which the parser will interpret the forms

```
e1 op e2 # or
op e1
```

as equivalent to the function calls

```
`op`(e1, e2) # or
`op`(e1)
```

Here *e1* and *e2* stand for arbitrary expressions. The "`" quotes mean that the string will be treated as a name, to be looked up as the name of a function object.

It is true that many built-in operators are *primitive* functions, which does occasionally make them special for programming (for the way primitives work, see Section 13.4, page 463). However, this only becomes relevant after finding the particular object corresponding to the operator's name, and does not differ fundamentally from the treatment of any primitive function.

Table 6.3 on page 186 shows operators found in the `base` package of R. This section examines the implementations, particularly of the arithmetic and comparison operators. The implementations handle arguments from commonly occurring classes, such as vectors of the various basic R object types, matrices, arrays and time-series. For more information on the other operators, see the function index of the book and/or the online documentation for the operators.

From a programming view, operators in R are part of its extensible functional model. They can be extended to apply to new classes of objects, by defining methods to use when the arguments come from such classes. Operator methods are discussed in Section 10.3, page 389. If a group name is shown in a row of Table 6.3, methods for all the operators in the group may be provided by defining a method for the corresponding group generic (see Section 10.5, page 404).

New operators can be introduced using a lexical convention that `%text%` can appear as a binary operator. Just define a function of two arguments, assign it a name matching this pattern, and users can insert it as an operator. R has added operators `%in%` and `%x%` to the language, for example, to carry out matching and compute Kronecker products respectively. The grammar only recognizes such operators in binary form; you cannot define a new unary operator.

| Operator | Group | Comment |
|---|-----------|---|
| <code>~+`, ~-`, ~*`, ~/`, ~^`, ~%`, ~%/`</code> | Arith() | Operations for numerical arithmetic, the last two being modulus and truncated division. The arguments are, or are coerced to, "numeric" or "complex". |
| <code>~>`, ~<`, ~>=`, ~<=`, ~==`, ~!=`</code> | Compare() | Comparison operations, defined for arguments that are, or are coerced to, "numeric", "character", or "complex". |
| <code>~&`, ~ `, ~!`</code> | Logic() | Logical operations "and", "or", and "not". |
| <code>~&&`, ~ `</code> | | Control operations, only valid with single logical values as arguments. |
| <code>~%`,`, ~%in`,`, ~%o`,`, ~%*`,`, ~%x`,`</code> | | Binary operators using the general convention that <code>~%text~`</code> is an operator name. |
| <code>~\$`,`, ~?` ~@`,`, ~~`,`, ~:`,`, ~::`,`, ~:::`,`</code> | | Other binary operators. |

Table 6.3: Binary and unary operators defined in the `base` package of R.

Table 6.3 is not quite complete. The various assignment operators are all treated as binary operators (with limited programming allowed; for example, methods cannot be defined for them). Among the miscellaneous operators, `~?~`` and `~~`` can also be used as unary operators.

Rules for operator expressions with vector arguments

There are general rules for the arithmetic and comparison operators, specifying the objects returned from calls to the corresponding functions, depending on the class and length of the arguments. The rules date back to the early design of the S language. Restricting our attention to the basic vector object types, the following summarizes the current rules in the `base` package implementation.

1. If the arguments are of the same type and length, the operation is unambiguous. The value returned will be of that length and either of the same type or "logical", depending on the operator.

Arithmetic operators work on "logical", "numeric", and "complex"

data, but not on "raw", "character", or "list". Comparison operators work on all types, but not if both arguments are lists.

2. Given two arguments of different type, one will be coerced to the type of the other argument before the operation takes place. Arithmetic operations are limited to the four types mentioned in item 1, but for comparisons nearly anything goes, other than a few comparisons with "complex" data. Conversions are made to the less simple type, according to the rules discussed in Section 6.3, page 149.
3. If one argument is shorter than the other, it will be implicitly replicated to the length of the other argument before the operation takes place, except that a zero-length operand always produces zero-length results. A warning is given if the longer length is not an exact multiple of the shorter.

For more details on conversions see Section 6.3, page 149.

As mentioned, most of these rules date back to early in the evolution of the S language, and are unlikely to change. With hindsight though, I'm inclined to think the original design was too eager to produce an answer, for any arguments. Some of the rules, such as those for replicating shorter objects, were heuristics designed to work silently when operating on a matrix and one of its rows or columns.

For important computations, stricter rules that only allow unambiguous mixing of types and lengths would be more trustworthy. The function `withStrictOps()` in the `SoDA` package allows you to evaluate any expression applying such rules. A call to `withStrictOps()` either returns the value of the expression or generates an error with an explanation of the ambiguities. Mixtures of types are allowed only for numeric types, including complex (no logical/numeric conversion, for example). Unequal lengths are allowed only if one operand is a "scalar", of length 1. I would recommend running examples whose validity is important with these rules in place; in other words, when the *Prime Directive* weighs heavily on you, it pays to check for ambiguous code.

Arithmetic and comparison operators deal with vector structures as well as simple vectors, both in general and specifically for matrices, multi-way arrays, and time-series. Classes can be formally defined to be vector structures (see Section 6.3, page 154), in which case they inherit methods from class "structure". As noted on page 156, the current `base` package rules do not treat vectors with arbitrary attributes as a vector structure. You should use a formal class definition that extends "structure" rather than

relying on the base package behavior to obtain trustworthy results for vector structures.

Arithmetic operators

Arithmetic operations with numeric arguments go back to the early history of computing with “floating-point numbers”. Down to the 1970s, software for these computations was complicated by a variety of representations and word lengths. An essential step forward was the adoption of the IEEE floating point standard, which mandated aspects of both the representation and the way computations should behave. The standard included a model for the numbers, with parameters expressing the range of numbers and the accuracy. The model also included both `Inf`, to stand for numbers too large to be represented, and `NaN` for the result of a computation that was *Not a Number*; for example,

```
> 0/0
[1] NaN
```

The S language adopted the model, and R includes it. The object `.Machine` has components for the parameters in the model; see its documentation. We look at some details of numeric computations in Section 6.7, page 191.

For most numeric computations in R, numeric means double-precision, type `"double"`. This is certainly true for arithmetic. The operators will take arguments of type logical or integer as well as numeric, but the results will nearly always be of type `"double"`, aside from a few computations that preserve integers. Logicals are interpreted numerically by the C convention that `TRUE` is 1 and `FALSE` is 0. Complex arithmetic accepts `"numeric"` arguments with the usual convention that these represent the real part of the complex values. Raw and character data are not allowed for arithmetic.

Arithmetic operators allow operands of different lengths, according to the rules described on page 186, calling for replicating the shorter operand. R warns if the two lengths are not exact multiples. The following examples illustrate the rules.

```
> 1:5 + 1
[1] 2 3 4 5 6
> 1:6 + c(-1, 1)
[1] 0 3 2 5 4 7
> 1:5 + c(-1, 1)
[1] 0 3 2 5 4
Warning message:
```

```
longer object length
  is not a multiple of shorter object length in: 1:5 + c(-1, 1)
```

The second and third examples would be errors if the strict rules implemented by the function `withStrictOps()` were applied.

Comparison and logical operators.

The comparison operators (`>`, `>=`, `<`, `<=`, `==`, and `!=`), when given two vectors as arguments return a logical vector with elements `TRUE`, `FALSE`, or `NA` reflecting the result of the element-by-element comparisons. Arguments will be implicitly replicated to equal length and coerced to a common type, according to the ordering of types shown in Section 6.3, page 149. So, for example, comparing numeric and character data implicitly converts the numbers to character strings first.

Comparison expressions in R look much like those in languages such as C, Java, or even Perl. But they are designed for different purposes, and need to be understood on their own. The purpose of a comparison in R is to produce a vector of logical values, which can then be used in many other computations. One typical use is to select data. The logical vector can be used to select those elements of any other object for which the comparison is `TRUE`.

```
y[ y > 0 ]; trafficData[ day != "Sunday", ]
```

Comparisons are often combined using the logical operators:

```
weekend <- trafficData[ day == "Sunday" | day == "Saturday", ]
```

Combinations of comparisons and logical operators work in R similarly to conditions in database selection. If you're familiar with database software queries, using SQL for example, then consider the comparison and logical operators as a way to obtain similar data selection in R.

One consequence of the general operator rules on page 186 needs to be emphasized: The comparison operators are not guaranteed to produce a single logical value, and if they do, that value can be `NA`. For really trustworthy programming, try to follow a rule such as this:

Don't use comparison operators naked for control, as in `if()` expressions, unless you are really sure the result will be a single `TRUE` or `FALSE`. Clothe them with an expression guaranteed to produce such a value, such as a call to `identical()`.

Violations of this rule abound, even in core R software. You may get away with it for a long time, but often that is the bad news. When the expression finally causes an error, the original programming experience may be long gone, possibly along with the programmer responsible.

The most commonly occurring unsafe practice is to use the "==" operator for control:

```
if(parameters$method == "linear") # Don't do this!
  value <- lm(data)
```

What can go wrong? Presumably `parameters` is meant to be an object with components corresponding to various names, usually an R list with names. If one of the names matches "method" and the corresponding component is a single character string or something similar, the computation will proceed. Someday, though, the assumption may fail. Perhaps the computations that produced `parameters` changed, and now do not set the "method" component. Then the unfortunate user will see:

```
Error in if (parameters$method == "linear") ... :
  argument is of length zero
```

The `if` expression may have looked reasonable, particularly to someone used to C-style languages, but it did not actually say what it meant. What we meant to say was:

```
if(the object parameters$method is identical to
  the object "linear" )
```

A function defined to implement this definition would return `TRUE` if the condition held and `FALSE` in all other circumstances. Not surprisingly, there is such a function, `identical()`. The condition should have been written:

```
if(identical(parameters$method, "linear"))
  value <- lm(data)
```

For more on techniques to produce such single values, see Section 6.3, page 152.

The two operators `&&` and `||`, however, are specifically designed for control computations. They differ from `&` and `|` in that they are expected to produce a single logical value and they will not evaluate the second argument if that argument cannot change the result. For example, in the expression

```
if(is(x, "vector") && length(x)>0) x[] <- NA
```

the expression `length(x)>0` will not be evaluated if `is(x, "vector")` evaluates to `FALSE`. Here too, however, the evaluation rules of R can be dangerously generous. Arguments to these operators can be of length different from 1. Only the first element will be used, but no warning is issued, and if one argument is of length zero, the result is silently `NA`. Therefore, the arguments to `&&` and `||` themselves need to follow the guidelines for computing single logical values.

6.7 Computations on Numeric Data

Numeric data in R can in principle be either `"double"` or `"integer"`, that is, either floating-point or fixed-point in older terminology. In practice, numeric computations nearly always produce `"double"` results, and that's what we mean by the term `"numeric"` in this discussion. Serious modern numeric computation assumes the floating-point standard usually known as "IEEE 574" and R now requires this. The standard enforces a number of important rules about how numeric data is represented and how computations on it behave. Most of those don't need to concern us explicitly, but a brief review of floating-point computation will be useful here.

Standard floating-point numbers have the form $\pm b2^k$ where b , known as the *significand*,³ is a binary fraction represented by a field of m bits:

$$b_12^{-1} + b_22^{-2} + \dots + b_m2^{-m}$$

and k , the *exponent* is an integer field of fixed width. Given the size of the floating-point representation itself (say, 32 or 64 bits) the standard specifies the width of the fraction and exponent fields. Floating-point numbers are usually *normalized* by choosing the exponent so that the leading bit, b_1 , is 1, and does not need to be stored explicitly. The fraction is then stored as a bit field $b_2 \dots b_m$. The \pm part is conceptually a single sign bit. The exponent represents both positive and negative powers, but not by an internal sign bit; rather, an unsigned integer is interpreted as if a specific number was subtracted from it. All that matters in practice is that the exponent behaves as if it has a finite range $-k_u < k \leq k_u$. In the standard, only a few choices are allowed, mainly single and double precision. However, the model is general, and future revisions may add further types.

In addition to the general patterns for floating-point numbers, the standard defines some special patterns that are important for its operations.

³In my youth, this was called the *mantissa*, but the standard deprecates this term because it has a conflicting usage for logarithms.

Zero, first off, because that cannot be represented as a normalized number; in fact, the standard includes both ± 0 . Then a pattern to represent “infinity”; that is, numbers too large in absolute value for the representation. This we will write as $\pm \text{Inf}$. Finally, a pattern called `NaN` (*Not a Number*) indicates a result that is undefined as a number. This is related to the long-standing value `NA` in the S language standing for a *Not Available*, or missing value. The latter is more general and should be used in most R programming; see page 194.

The standard also set down requirements for arithmetic operations and for rounding. If you want to understand more about floating-point computations and the standard, there is a huge literature. One of the best items is an unpublished technical report by one of numerical analysis’ great figures and eccentrics, W. Kahan [18].

Details of numerical representation are usually far from your thoughts, and so they should be. A few important consequences of floating-point representation are worth noting, however. The finite set of floating-point numbers represented by the standard, for a given word size, are essentially a model for the mathematical notion of *real* numbers. The standard models all real numbers, even though only a finite subset of real numbers correspond exactly to a particular floating-point representation. The general rule of thumb is that integer values can be represented exactly, unless they are very large. Numbers expressed with *decimal fractions* can be represented exactly only if they happen to be equal to an integer multiplied by a negative power of 2. Otherwise, the stored value is an approximation. The approximation is usually hidden from the R user, because numbers are approximated from decimal input (either parsed or read from a file), and printed results usually round the numbers to look as if they were decimal fractions.

The approximate nature of the floating-point representation sometimes shows up when computations are done to produce numbers that we think should be exactly zero. For example, suppose we want the numbers $(-.3, -.15, 0., .15, .3)$. The function `seq()` computes this, taking the first element, the last, and the step size as arguments. But not quite:

```
> seq(-.45, .45, .15)
[1] -4.500000e-01 -3.000000e-01 -1.500000e-01 -5.551115e-17
[5]  1.500000e-01  3.000000e-01  4.500000e-01
```

Did something go wrong? (Every so often, someone reports a similar example as a bug to the R mailing lists.)

No, the computation did what it was asked, adding a number to the initial number to produce the intermediate results. Neither number is rep-

resented exactly as a binary fraction of a power of 2, as we can see using the `binaryRep()` function developed as an example in Section 6.4, page 161:

```
> binaryRep(c(-.45, .15))
Object of class "binaryRep"
1: -.11100110011001100110011001100110011001100110011001100110011001101 * 2^-1
   (-0.45)
2: .10011001100110011001100110011001100110011001100110011001100110011 * 2^-2
   (0.15)
```

In fact, we can see what probably happens in the computation:

```
> binaryRep(c(3 * .15))
Object of class "binaryRep"
1: .1110011001100110011001100110011001100110011001100110011001100 * 2^-1
   (0.45)
```

Notice that the last bit is different from the representation of `-0.45`; the difference is in fact, 2^{-54} , or `5.551115e-17`.

There is nothing numerically incorrect in the computed result, but if you prefer to get an exact zero in such sequences, remember that integer values are likely to be exact. Rescaling an integer sequence would give you the expected result:

```
> seq(-3, 3, 1) * .15
[1] -0.45 -0.30 -0.15  0.00  0.15  0.30  0.45
```

Although `seq()` returns a result of type `"integer"`, that is not the essential point here. Integer values will be represented exactly as `"double"` numbers so long as the absolute value of the integer is less than 2^m , the length of the fractional part of the representation (2^{54} for 32-bit machines).

Numbers that are too large positively or negatively cannot be represented even closely; the floating-point standard models these numbers as `Inf` and `-Inf`. Another range of numbers cannot be represented because they are too close to zero (their exponents are too negative for the model). These numbers are modeled by `0`. In terms of arithmetic operations, these two ranges of numbers correspond to *overflow* and *underflow* in older terminology; in R, overflow and underflow just produce the corresponding values in the floating-point standard, with no error or warning.

The standard specifies that arithmetic operations can also produce `NaN`. If either operand is a `NaN`, the result will be also. In addition certain computations, such as `0/0`, will generate a `NaN`.

The NaN pattern will immediately remind R users of the NA pattern, which also represents an undefined value, although not just for floating-point numbers. The two patterns arose separately, but R treats NaN in numeric data as implying NA. Therefore, for most purposes you should use the function `is.na()` to test for undefined values, whether they arose through numerical computations or from other sources. There is also a function `is.nan()` that in principle detects only values generated from floating-point operations. Much more important in thinking about undefined values than the distinction between NA and NaN is to be careful to treat either as a pattern, not as a value. Always use the functions to detect undefined values rather than testing with `identical()`, and certainly never use the ``==`` operator:

```
identical(x, NaN) # Never do this! Use is.nan() instead.
x == NA; x == NaN # Always wrong!
```

The first expression is lethally dangerous. The floating-point standard defines NaN in such a way that there are many distinct NaN patterns. There is no guarantee which pattern an internal computation will produce. Therefore `identical(x, NaN)` may sometimes be TRUE and at other times FALSE on numerical results for which `is.nan(x)` is TRUE. The second and third expressions always evaluate to NA and NaN, and so will always be wrong if you meant to test for the corresponding condition.

Having warned against comparisons using the objects NA and NaN, we now have to point out that they are quite sensible in some other computations. For example, if input data used a special value, say 9999 to indicate missing observations, we could convert those values to the standard NA pattern by assigning with the NA object.

```
x[ x == 9999 ] <- NA
```

Just a moment, however. If you have been reading about the generally inexact numeric representation of decimal numbers, you would be wise to ask whether testing for exact equality is dangerous here. The answer depends on how `x` arose; see below on page 196 for some discussion.

Similar computations can set elements to the floating-point NaN pattern:

```
x[ x < 0 ] <- NaN
```

When does it make sense to use NaN versus NA? Because the NaN pattern is part of the floating-point standard, it's natural to insert it as part of a numeric calculation to indicate that the numeric value of the result is undefined for some range of inputs. Suppose you wanted a function `inv` to be $1/x$, but only for positive values, with the transform being undefined for negative values.


```
invPos <- function(x) {
  ifelse( x<0, NaN, 1/x)
}
```

Regular missing values (NA) will now be distinguished from numerically undefined elements.

```
> xx <- c(NA, -1, 0, 1, 2)
> invPos(xx)
[1] NA NaN Inf 1.0 0.5
```

Similar built-in mathematical functions follow the same pattern (for example, `log()` and `sqrt()`), but with a warning message when NaN is generated. The use of `ifelse()` here makes for simple and clear programming, but keep in mind that the function evaluates all its arguments, and then selects according to the first. That's fine here, but you might need to avoid computing the values that will not be used in the result, either because an error might occur or because the computations would be too slow. If so, we would be thrown back on a computation such as:

```
invPos <- function(x) {
  value <- rep(NaN, length(x))
  OK <- x >= 0
  value[OK] <- 1/x[OK]
}
```

As an exercise: This version fails if `x` has NAs; how would you fix it?

Turning back to general treatment of NA, you may encounter a replacement version of `is.na()`:

```
is.na(x) <- (x == 9999)
```

The right side of the assignment is interpreted as an index into `x` and internal code sets the specified elements to be undefined. The catch with using the replacement function is interpretation: What is it supposed to do with elements that are already missing? Consider:

```
> x <- c(NA, 0, 1)
> is.na(x) <- c(FALSE, FALSE, TRUE)
```

What should the value of `is.na(x)` be now? You could expect it to be the pattern on the right of the assignment, but in fact the replacement does not alter existing NA elements. (What value would it use for those elements?)

```
> is.na(x)
[1] TRUE FALSE TRUE
```

Given the ambiguity, I suggest using the direct assignment.

Numerical comparisons

Numerical comparisons are generally applied in R for one of two purposes: *filtering* or *testing*. In filtering, a computation is to be applied to a portion of the current data, with a numerical comparison determining that portion:

```
x[ x<0 ] <- NaN
```

In testing, a single logical value will control some step in a calculation, maybe the convergence of an iterative computation:

```
if(abs(xNew - xOld) < xeps)
  break
```

Discussions of numerical accuracy often mix up these situations, but the whole-object computational picture in R makes them quite distinct. Considerations of numerical accuracy and in particular of the effect of numerical error in floating-point representation (so-called “rounding error”) have some relevance to both. But it’s testing that provides the real challenge, and rounding error is often secondary to other limitations on accuracy.

Having said that, we still need a basic understanding of how numerical comparisons behave in order to produce trustworthy software using them. The six comparison operators will produce logical vectors. The rules for dealing with arguments of different length are those for operators generally (see page 189). As with other operators, the wise design sticks to arguments that are either the same length and structure, or else with one argument a single numeric value.

The elements of the object returned from the comparison will be `TRUE`, `FALSE`, or `NA`, with `NA` the result if either of the corresponding elements in the arguments is either `NA` or `NaN`.

The two equality operators, ``==`` and ``!=``, are dangerous in general situations, because subtle differences in how the two arguments were computed can produce different floating-point values, resulting in `FALSE` comparisons in cases where the results were effectively equal. Some equality comparisons are safe, if we really understand what’s happening; otherwise, we usually need to supply some information about what “approximately equal” means in this particular situation.

Floating-point representation of integer numbers is exact as long as the integers are within the range of the fractional part as an integer (for 64-bit double precision, around 10^{17}). Therefore, provided we know that all the numeric computations for both arguments used only such integer values, equality comparisons are fine.

```
x[ x == 9999 ] <- NA
```

If the data were scanned from decimal text, for example, this test should be valid. A safer approach is to apply the tests to the data as character vectors, before converting to numeric, but this might not be simple if a function such as `read.table()` was reading a number of variables at once.

As should be obvious, just the appearance of integer values when an object is printed is no guarantee that equality comparisons are safe. The following vector looks like integers, but examining the remainder modulo 1 shows the contrary:

```
> x
[1] 10 9 8 7 6 5 4 3 2 1 0
> x%%1
[1] 0.000000e+00 1.776357e-15 0.000000e+00 8.881784e-16
[5] 0.000000e+00 0.000000e+00 8.881784e-16 0.000000e+00
[9] 4.440892e-16 8.881784e-16 0.000000e+00
```

No surprise, after we learn that `x<-seq(1.5,0,-.15)/.15`, given the example on page 192.

Using greater-than or less-than comparisons rather than equality comparisons does not in itself get around problems of inexact computation, but just shifts the problem to considering what happens to values just on one side of the boundary. Consider:

```
x[ x<0 ] <- NaN
```

This converts all negative numbers to numerically undefined; the problem is then whether we are willing to lose elements that came out slightly negative and to retain elements that came out slightly positive. The answer has to depend on the context. Typically, the filtering comparison here is done so we can go on to another step of computation that would be inappropriate or would fail for negative values. If there is reason to retain values that might be incorrectly negative as a result of numerical computations, the only safe way out is to know enough about the computations to adjust small values. It's not just comparison operations that raise this problem, but any computation that does something discontinuous at a boundary. For example,

```
xx <- log(x)
```

has the same effect of inserting NaN in place of negative values. Once again, if we need to retain elements computed to be slightly negative through inexact

computation, some adjustment needs to be made and that in turn requires considerable understanding of the context.

When we turn from filtering during the computation to testing for control purposes, we have the additional requirement of choosing a single summary value from the relevant object. R deals with objects, but tests and conditional computations can only use single `TRUE` or `FALSE` values. Typical tests discussed in the numerical analysis literature involve comparisons allowing for a moderate difference in the low-order bits of the floating-point representation, plus some allowance for the special case of the value 0. We saw on page 192 that seemingly identical computations can produce small differences in the floating-point representation of the result. For a non-zero correct result, then, the recommendation is to allow for relative error corresponding to a few *ulps* (*units in the last place*). A correct value 0 is a special case: If the correct test value is 0 then the computed value has to be tested for sufficiently small absolute value, because relative error is meaningless. There are some good discussions of how to do such comparisons very precisely (reference [18]; also search the Web, for example for “IEEE 754”).

The first problem in applying tests in this spirit in practice is to deal with objects, not single values. The function `all.equal.numeric()` in basic R implements a philosophy designed to treat objects as equal if they differ only in ways that could plausibly reflect inexact floating-point calculations. Given arguments `current` and `target` and a suitable small number, `tolerance`, it tests roughly:

```
mean(abs(current - target))/mean(abs(target)) < tolerance
```

as long as `mean(abs(target))` is non-zero, and

```
mean(abs(current - target)) < tolerance
```

otherwise. Tests of this sort are fine as far as they go, but unfortunately only apply to a small minority of practical situations, where there is a clear target value for comparison and usually where deviations from the target can be expected to be small.

We created the `all.equal` methods originally to test software for statistical models, when it was installed in a new computing environment or after changes to the software itself. The tests compared current results to those obtained earlier, not “exact” but asserted to be correct implementations, run in an environment where the needed numerical libraries also worked correctly for these computations. (This is “regression” testing in the computational, not statistical, sense.) Numerical deviations were only

one aspect; `all.equal()` methods also check various structural aspects of the `current` and `target` objects.

The general problem, unfortunately, is much harder and no automatic solution will apply to all cases. Testing numerical results is an important and difficult part of using statistical software. Providing numerical tests is an equally important and difficult part of creating software. Most of the difficulty is intrinsic: It is often harder to test whether a substantial numerical computation has succeeded (or even to define clearly what “succeeded” means) than it is to carry out the computation itself. Computing with R does have the advantage that we can work with whole objects representing the results of the computation, providing more flexibility than computations with single numbers or simple arrays. Also, the interactive context of most R computation provides rich possibilities for choosing from a variety of tests, visualizations, and summaries. It’s never wise to base an important conclusion on a single test.

With all these disclaimers firmly in mind, we can still consider a simple style of test, analogous to the `all.equal.numeric()` logic above, to be adapted to specific testing situations. Two relevant considerations correspond to *convergence* and *uncertainty*. In convergence tests, one would like to test how near the iterative computation is to the target, but naturally the target is generally unknown. With luck, some auxiliary criterion should apply at the target. In linear least-squares regression, for example, at the target model the residuals are theoretically orthogonal to the predictor; therefore, comparing the inner products of residuals with columns of `X` to the value 0 would be a way of testing an iterative computation. Care is still needed in choosing tolerance values for the comparison.

New methods for complicated problems often have no such outside criterion. The natural inclination is then to test the iteration itself. The `target` and `current` objects are taken to be the parameters of the model or some other relevant quantity, from the current and previous iteration. Such a test may work, and in any case may be all you can think of, but it is rarely guaranteed and can be dangerously over-optimistic. Use it by all means, if you must, but try to experiment thoroughly and if possible replace it or calibrate by whatever theory you can manage to produce. Some examples of related test criteria are discussed when we consider software for statistical models in Section 6.9, page 218.

Issues of uncertainty, on the other hand, correspond to questions about the data being used. Limitations in our ability to measure the data used in a model or other statistical computation, or even limitations in our ability to define what is being measured, must naturally translate into uncertainty

in any parameters or other summaries computed. Some tests or measures based on these uncertainties are essential as part of reporting a model if we are not to run the risk of implying more accurate knowledge than we really possess. If nothing else, it's a good idea to do some simulations using as plausible assumptions as can be made about the uncertainties in the data, to see how reproducible are the results of the computation.

6.8 Matrices and Matrix Computations

Matrix computations are the implementation for much fundamental statistical analysis, including model-fitting and many other tasks. They also have a central role in a wide variety of scientific and engineering computations and are the basis for several software systems, notably MATLAB[®]. Matrices play an important role in R as well, but less as the basic objects than as an example of some general approaches to data and computation.

In spite of the conceptual difference, many matrix computations will look similar in R to those in other systems. A matrix object can be indexed by rows and columns. R includes most of the widely used numerical methods for matrices, either in the base package or in add-on packages, notably `Matrix`. The usual search lists and tools will likely find some functions in R to do most common matrix computations.

This single section has no hope of covering matrix computations in detail, but it examines the concept of a matrix in R, its relation to some other concepts, such as general vector structures, and a variety of techniques often found useful in programming with matrix objects. We discuss different techniques for indexing matrices and use these in an extended example of constructing matrices with a particular pattern of elements (page 206). We discuss the `apply()` family of functions (page 212), consider some basic numerical techniques (page 214), and finally look briefly at numerical linear algebra (page 216).

Moving from the inside out, first, the "matrix" class extends the "array" class: a matrix is a multi-way array in which "multi" happens to be "two". The defining properties of an R matrix, its dimension and optional `dimnames` are simply specializations of the same properties for a general multi-way array. Indexing of elements is also the same operator, specialized to two index expressions.

An array, in turn, is a special case of the classic S language concept of a *vector structure* discussed in Section 6.3, page 154; that is, a vector that has additional properties to augment what one can do, without losing the

built-in vector computations, such as ordinary indexing by a single variable and the many basic functions defined for vectors.

As a result of what a matrix is in R, there are some important things it is not; most importantly, it is not itself a basic concept in the system, contrasting with MATLAB, for example. Neither is the multi-way array a basic concept, contrasting with the APL language, from which many of the array ideas in the S language were derived. Arrays never stop being vectors as well. Many useful computational techniques come from combining vector and matrix ways of thinking within a single computational task.

To make a vector, \mathbf{x} , into a matrix, information is added that defines $\dim(\mathbf{x})$, its dimensions. For a matrix, $\dim(\mathbf{x})$ is the vector of the number of rows and columns. The elements of \mathbf{x} are then interpreted as a matrix stored by columns. For general k -way arrays, the same information is added, but $\dim(\mathbf{x})$ has length k . As a result, any vector can be made a matrix, not just a numeric vector. The data part of a matrix can be a vector of any of the basic types such as "logical", "character", or even "list".

The programming model for matrices, and for arrays in general, includes the mapping between matrix indexing and vector indexing. That mapping is defined by saying that the first index of the matrix or array varies most rapidly. Matrices are stored as columns. Three-way arrays are stored as matrices with the third index constant, and so on. Fortran prescribed the same storage mechanism for multi-way arrays (not a coincidence, given the history of S).

Because matrices in R are an extension of basic vectors rather than a built-in structure at the lowest level, we might expect more specialized matrix languages, such as MATLAB, to perform more efficiently on large matrix objects. This is fairly often the case, particularly for computations that in R are not directly defined in C.

Extracting and Replacing Subsets of a Matrix

To see how matrix and vector ideas work together, let's consider expressions to manipulate pieces of a matrix. Subsets of a matrix can be extracted or replaced by calls to the `[` operator with four different forms for the index arguments:

1. two index arguments, indexing respectively the rows and columns;
2. a single index argument that is itself a two-column matrix, each row specifying the row and column of a single element;

3. a single logical expression involving the matrix and/or some other matrix with the same number of rows and columns;
4. a single numeric index, using the fact that matrices are stored by column to compute vector index positions in the matrix.

All four techniques can be useful, and we will look at examples of each. The first case returns a matrix (but see the discussion of `drop=` on page 203), the other three return a vector. All four can be used on the left side of an assignment to replace the corresponding elements of the matrix.

The third and fourth methods for indexing are not specially restricted to matrices. In fact, we're using some basic properties of any vector structure: logical or arithmetic operations produce a parallel object with the same indexing of elements but with different data. And any structure can be subset by a logical vector of the same size as the object. Because a matrix or other structure is also a vector by inheritance, comparisons and other logical expressions involving the object qualify as indexes. This is the fundamental vector structure concept in R at work.

Similarly, the indexing in the fourth form is matrix-dependent only in that we have to know how the elements of the matrix are laid out. Similarly for time-series or any other structure class, once we use knowledge of the layout, any vector indexing mechanism applies.

Indexing rows and columns

The obvious way to index a matrix, `x[i, j]`, selects the rows defined by index `i` and the columns defined by index `j`. Any kind of indexing can be used, just as if one were indexing vectors of length `nrow(x)` and `ncol(x)`, respectively. Either index can be empty, implying the whole range of rows or columns. Either index can be positive or negative integers, or a logical expression. Consider, for example:

```
x[ 1:r, -1 ]
```

The first index extracts the first `r` rows of the matrix, and in those rows the second index selects all the columns except the first. The result will be an `r` by `ncol(x)-1` matrix. As with vectors, the same indexing expressions can be used on the left of an assignment to replace the selected elements.

The result of selecting on rows and columns is itself a matrix, whose dimensions are the number of rows selected and the number of columns selected. However, watch out for selecting a single row or column. In this case there is some question about whether the user wanted a matrix result or

a vector containing the single row or column. Both options are provided, and the choice is controlled by a special argument to the ``[`` operator, `drop=`. If this is `TRUE`, single rows and columns have their dimension “dropped”, returning a vector; otherwise, a matrix is always returned. The default is, and always has been, `drop=TRUE`; probably an unwise decision on our part long ago, but now one of those back-compatibility burdens that are unlikely to be changed. If you have an application where maintaining matrix subsets is important and single rows or columns are possible, remember to include the `drop=FALSE` argument:

```
model <- myFit(x[whichRows,,drop=FALSE], y[whichRows])
```

Indexing with a row-column matrix

Row and column indices can be supplied to ``[`` as a single argument, in the form of a two-column matrix. In this form, the indices are not applied separately; instead, each row i of the index matrix defines a single element to be selected, with $[i, 1]$ and $[i, 2]$ being the row and column of the element to select. For an example, suppose we wanted to examine in a matrix, `x`, the elements that corresponded to the column-wise maxima in another matrix, `x0` (maybe `x0` represents some initial data, and `x` the same process at later stages). Here’s a function, `columnMax()`, that returns a matrix to do this indexing.

```
columnMax <- function(x0) {
  p <- ncol(x0)
  value <- matrix(nrow = p, ncol = 2)
  for(i in seq(length = p))
    value[i,1] = which.max(x0[,i])
  value[,2] <- seq(length = p)
  value
}
```

The function `which.max()` returns the first index of the maximum value in its argument. The matrix returned by `columnMax()` has these (row) indices in its first column and the sequence `1:p` in its second column. Then `x[columnMax(x0)]` can be used to extract or replace the corresponding elements of `x`.

```
> x0
      [,1] [,2] [,3]
[1,] 11.4 11.0  9.2
[2,] 10.0 10.1 10.4
```

```

[3,]  9.2  8.9  8.7
[4,] 10.7 11.5 11.2
> columnMax(x0)
      [,1] [,2]
[1,]    1    1
[2,]    4    2
[3,]    4    3
> x
      [,1] [,2] [,3]
[1,] 11.1 11.0  9.0
[2,]  9.6 10.1 10.5
[3,]  9.2  8.7  9.0
[4,] 10.7 11.6 11.0
> x[columnMax(x0)]
[1] 11.1 11.6 11.0

```

It takes a little thought to keep straight the distinction between indexing rows and columns separately, versus indexing individual elements via a matrix of row and column pairs. In the example above, suppose we take the row indices, the first column, from `columnMax(x0)`, and index with that:

```

> rowMax <- unique(columnMax(x0)[,1]); x[rowMax,]
      [,1] [,2] [,3]
[1,] 11.1 11.0   9
[2,] 10.7 11.6  11

```

This does something different: it creates a new matrix by selecting those rows that maximize some column of `x0`, but keeps all the corresponding columns of `x`.

Notice the use of `unique()`, so that we don't get multiple copies of the same row. In indexing any object, R allows a positive integer index to appear any number of times, and then repeats the same selection each time. Your application may or may not want to replicate the selection, so remember to eliminate any duplicates if it does not.

Indexing matrices with logical expressions

Logical index expressions typically involve the matrix whose values we want to select or replace, or perhaps some companion matrix of the same dimensions. For example, the value of a comparison operation on a matrix can be used as a single index to subset that matrix. To set all the negative values in `x` to `NA`:

```
x[ x < 0 ] <- NA
```

(When dealing with missing values, watch out for the opposite computation, however. To refer to all the missing values in `x`, use `x[is.na(x)]`, and never use `NA` in comparisons; see Section 6.7, page 192.)

Indexing in this form applies to any vector or vector structure, and uses nothing special about matrices. However, two auxiliary functions for matrices can be useful in logical indexing, `row()` and `col()`. The value of `row(x)` is a matrix of the same shape as `x` whose elements are the index of the rows; similarly, `col(x)` is a matrix containing the index of the columns of `x`. The function `triDiag()` on page 207 shows a typical use of these functions.

Indexing matrices as vectors

The fourth indexing technique for matrices is to use knowledge of how a matrix is laid out as a vector; namely, by columns. Logical indices are in a sense doing this as well, because the logical expression ends up being treated as a vector index. However, when the expression involves the matrix itself or other matrices of related shape, the code you write should not require knowledge of the layout.

In contrast, we now consider numeric index expressions explicitly involving the layout. Using these is somewhat deprecated, because the reliance on the physical storage of matrix elements in R tends to produce more obscure and error-prone software. On the other hand, knowledge of the layout is required if you write C software to deal with matrices (Section 11.3, page 424). And computations for some general indexing are more efficient if done directly. Don't worry that the column-wise layout of matrix elements might change. It goes back to the original desire to make objects in the S language compatible with matrix data in Fortran.

If the matrix `x` has n rows and p columns, elements 1 through n of `x` are the first column, elements $n + 1$ through $2n$ the second, and so on. When you are programming in R itself, the arrangement works identically regardless of the type of the matrix: "numeric", "logical", "character", or even "list". Be careful in using non-numeric matrix arguments in the C interface, because the declaration for the argument corresponding to the R matrix must match the correct C type for the particular matrix (Section 11.3, page 424).

From examination of data manipulation functions in R, particularly the `seq()` function and arithmetic operators, you can construct many special sections of a matrix easily. For example, suppose we wanted to select or replace just the elements of `x` immediately to the right of the diagonal; that is, elements in row-column positions `[1,2]`, `[2,3]`, and so on. (Sections such

as this arise often in numerical linear algebra.) As vector indices, these are positions $n + 1, 2n + 2, \dots$. A function that returns them is:

```
upper1 <- function(x) {
  n <- nrow(x); p <- ncol(x)
  seq(from = n+1, by = n+1, length = min(p-1, n))
}
```

There is one small subtlety in defining functions of this form: computing the length of the sequence. Often the length depends on the shape of the matrix, specifically whether there are more columns or rows. Try out your computation on matrices that are both short-and-wide and long-and-skinny to be sure.

```
> xLong <- matrix(1:12, nrow = 4)
> xLong
      [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> xLong[upper1(xLong)]
[1]  5 10
> xWide <- matrix(1:12, nrow = 2)
> xWide
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   11
[2,]    2    4    6    8   10   12
> xWide[upper1(xWide)]
[1] 3 6
```

Example: Matrix indexing and tridiagonal matrices

To illustrate some additional techniques and to clarify the different mechanisms, we will develop a function that can be implemented in different ways by three of the four matrix indexing techniques. Let's consider what are called *banded* matrices, and in particular *tri-diagonal matrices*. A number of numerical techniques with matrices involve special forms in which all the elements are zero except for those on the diagonal or next to the diagonal. In general, this means there are at most 3 nonzero elements in each row or column, leading to the term tri-diagonal matrix. Multiplication by a banded matrix applies linear combinations to nearby elements of each column or row of another matrix. This technique aids in *vectorizing* a computation that might otherwise involve looping over the rows of the matrix. We used this

technique in the example on computing binary representations (Section 6.4, page 163).

Suppose we wanted a function to construct a general tri-diagonal matrix. The natural way to define the matrix is usually by specifying three vectors of numbers, the diagonal, the values above the diagonal, and the values below the diagonal. For example a 5 by 5 matrix of this form is:

```

      [,1] [,2] [,3] [,4] [,5]
[1,]    2    1    0    0    0
[2,]   -1    2    1    0    0
[3,]    0   -1    2    1    0
[4,]    0    0   -1    2    1
[5,]    0    0    0   -1    2

```

In this case the diagonal is `rep(2, 5)`, the upper off-diagonal elements are `rep(1, 4)`, and the lower off-diagonal elements `rep(-1, 4)`. A nice utility would be a function:

```
triDiag(diagonal, upper, lower, nrow, ncol)
```

If it adopted the usual R convention of replicating single numbers to the length needed, and set `ncol = nrow` by default, we could create the matrix shown by the call:

```
triDiag(2, 1, -1, 5)
```

Three different matrix indexing techniques can be used to implement function `triDiag()`. (All three versions are supplied with the `SoDA` package, so you can experiment with them.)

An implementation using logical expressions is the most straightforward. The expressions `row(x)` and `col(x)` return matrices of the same shape as `x` containing the corresponding row and column indices.

```

> row(x)
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    2    2    2    2    2
[3,]    3    3    3    3    3
[4,]    4    4    4    4    4
[5,]    5    5    5    5    5

```

This immediately tells us how to implement the `triDiag()` function: the upper diagonal elements always have a column index one greater than the row index, and conversely the lower diagonal elements have row index one greater than the column index. The diagonal has equal row and column

indices, but another useful auxiliary matrix function, `diag()`, lets us construct the matrix with its diagonal elements already in place, and all other elements set to 0. Here, then, is a definition of `triDiag()`:

```
triDiag <- function(diagonal, upper, lower,
                   nrow = length(diagonal), ncol = nrow) {
  value <- diag(diagonal, nrow, ncol)
  R <- row(value)
  C <- col(value)
  value[C == R + 1] <- upper
  value[C == R - 1] <- lower
  value
}
```

The value is created initially with the specified diagonal elements. Then the upper and lower off-diagonal elements are inserted using logical expressions, on the left of an assignment, to replace the correct elements. The function `diag()` and the two replacements use the standard R rule of replicating single values to the necessary length.

A second version of `tridiag()` can be implemented using a single matrix. The implementation is not as simple, but has some efficiency advantages for large problems that are typical of using explicit indices. Once again we use the fact that the upper diagonal has column indices one greater than row indices, and the lower diagonal has column indices one less than row indices. But in this case we will construct explicitly the two-column matrix with the row and column indices for each of these. For the moment, assume the desired matrix is square, say r by r . Then the upper diagonal is the elements $[1, 2], [2, 3], \dots, [r-1, r]$. The matrix index corresponding to the upper diagonal has $1:(r-1)$ in its first column and $2:r$ in its second column. Given these two expressions as arguments, the function `cbind()` computes just the index required. The whole computation could then be done by:

```
value <- diag(diagonal, nrow = nrow, ncol = ncol)
rseq <- 2:r
value[cbind(rseq-1, rseq)] <- upper
value[cbind(rseq, rseq-1)] <- lower
```

What makes this version more efficient than the logical expressions above for large problems? Only that it does not create extra matrices of the same size as x , as the previous implementation did. Instead it only needs to create two matrices of size $2*r$. Don't take such considerations too seriously for

most applications, but it's the sort of distinction between “quadratic” and “linear” requirements that can be important in extreme situations.

What makes this version more complicated is that the precise set of elements involved depends on whether there are more rows or columns. The expressions shown above for the case of a square matrix will not work for the non-square case. There will be `nrow` upper-diagonal elements, for example, if `ncol > nrow`, but only `nrow-1` otherwise. Conversely, there are `min(ncol, nrow-1)` lower diagonal elements.

```
> triDiag(2, 1, -1, 4, 6)
      [,1] [,2] [,3] [,4] [,5] [,6]
[1,]  2    1    0    0    0    0
[2,] -1    2    1    0    0    0
[3,]  0   -1    2    1    0    0
[4,]  0    0   -1    2    1    0
> triDiag(2, 1, -1, 6, 4)
      [,1] [,2] [,3] [,4]
[1,]  2    1    0    0
[2,] -1    2    1    0
[3,]  0   -1    2    1
[4,]  0    0   -1    2
[5,]  0    0    0   -1
[6,]  0    0    0    0
```

The general implementation of `triDiag()` using matrix index arguments then has the following form.

```
triDiag2 <- function(diagonal, upper, lower,
                     nrow = length(diagonal), ncol = nrow) {
  value <- diag(diagonal, nrow = nrow, ncol = ncol)
  n <- min(nrow, ncol-1)
  if(n>0) {
    rseq <- 1:n
    value[cbind(rseq, rseq+1)] <- upper
  }
  n <- min(nrow-1, ncol)
  if(n > 0) {
    rseq <- 1:n
    value[cbind(rseq+1, rseq)] <- lower
  }
  value
}
```

We also needed to look out for “degenerate” cases, where the resulting lower- or upper- diagonal was missing altogether (of length 0). Convince yourself

that the logical expressions involving `row(x)` and `col(x)` take care of all these variations.

As a third implementation, let's consider using the explicit layout of a matrix to analyze the index values needed to fill in the data elements, and derive a simple computation to generate them. The implementation will repay presenting in detail (perhaps somewhat more than the actual function deserves) because the process of analysis illustrates a useful approach to many problems. We will derive a pattern for the data needed, and find some R utilities that generate this pattern as an object.

Let's look again at the example of a tridiagonal matrix, but this time thinking about the storage layout.

| | [,1] | [,2] | [,3] | [,4] | [,5] |
|------|------|------|------|------|------|
| [1,] | 2 | 1 | 0 | 0 | 0 |
| [2,] | -1 | 2 | 1 | 0 | 0 |
| [3,] | 0 | -1 | 2 | 1 | 0 |
| [4,] | 0 | 0 | -1 | 2 | 1 |
| [5,] | 0 | 0 | 0 | -1 | 2 |

Starting with the first column, what are the index values for the non-zero elements? The first and second row of the first column are the first two elements; then the first three elements in the second column; then the second through fourth in the third column; and so on, shifting down one index for the three non-zero elements in each successive column. With a matrix having n rows, the non-zero elements appear in positions with the following pattern: 1, 2, $n + 1$, $n + 2$, $n + 3$, $2n + 2$, $2n + 3$, $2n + 4$, $3n + 3$, $3n + 4$, $3n + 5$, ... These come in triples, except for the first two. Let's make each triple correspond to the row of a matrix. Notice that the first element of each row is a multiple of $n + 1$, the second adds 1 to the first and the third adds 2 to the first.

$$\begin{bmatrix} & & & 1 & & 2 \\ (n+1) & & & (n+1)+1 & & (n+1)+2 \\ 2(n+1) & & & 2(n+1)+1 & & 2(n+1)+2 \\ & & & \dots & & \end{bmatrix}$$

If we fill the empty upper-left element with 0, it becomes obvious that the matrix can be computed by adding $(0, 1, 2)$ for the columns and $(0, (n + 1), 2(n + 1), \dots)$ for the rows.

The pattern of applying a function to a set of row values and a set of column values occurs in many matrix computations. It is handled by the R function `outer()`, which takes row values, column values, and the applied function as its arguments. The name `outer` refers to the outer product of

two vectors, but instead of multiplying elements here, we add them; with $n = 5$,

```
> outer((0:4)*6, 0:2, `+`)
      [,1] [,2] [,3]
[1,]    0    1    2
[2,]    6    7    8
[3,]   12   13   14
[4,]   18   19   20
[5,]   24   25   26
```

Convince yourself that the 3 columns are in fact the positions of the upper-diagonal, diagonal, and lower-diagonal non-zero elements of a 5 by 5 matrix, with the exception that the [1,1] element and the [5,3] element of the index matrix are outside the range, and have to be dropped out by our function. The third argument to `outer()` uses “backtick” quotes to pass in a name for the operator, ``+``.

By extracting the suitable elements of the three columns from the index matrix, we can insert the correct upper, diagonal, and lower values. Here then is a third definition of `triDiag()`:

```
triDiag3 <- function(diagonal, upper, lower,
                    nrow = length(diagonal), ncol = nrow) {
  value <- matrix(0, nrow = nrow, ncol = ncol)
  r <- max(nrow, ncol)
  if(r > 1) {
    nu <- min(nrow, ncol-1)
    nl <- min(nrow-1, ncol)
    index <- outer((0:nu)*(nrow+1), 0:2, `+`)
    value[index[1:min(nrow, ncol), 2]] <- diagonal
    if(nu > 0)
      value[index[-1, 1]] <- upper
    if(nl > 0)
      value[index[1:nl, 3]] <- lower
  }
  value
}
```

As with the second version of `triDiag()`, the number of lower- and upper-diagonal elements depends on whether there are more rows or columns in the matrix. By experimenting with the function (supplied in package `SoDA`), you can test whether the range of values inserted is indeed correct for square, wide, and skinny matrices.

In this version, we need to be even more careful about special cases. The compensation is that the call to `outer()` does all the actual index calculations at once. This version also generalizes to an arbitrary “bandwidth”, that is to upper- or lower-diagonal elements removed by more than just one place from the diagonal.

The `apply()` functions

One frequently wants to assemble the results of calling a function repeatedly for all of the rows or all of the columns of a matrix. In the `columnMax()` example on page 203, we assembled a vector of all the row indices maximizing the corresponding columns, by iterating calls to the function `which.max()` for each column. The function `apply()` will perform this computation, given three arguments: a matrix, a choice of which dimension to “apply” the function to (1 for rows, 2 for columns), and a function to call. A single call to `apply()` will then produce the concatenated result of all the calls.

In the case of `columnMax()`, using `apply()` allows the function to be rewritten:

```
columnMax <- function(x0) {
  p <- ncol(x0)
  cbind(apply(x0, 2, which.max),
        seq(length = p))
}
```

We showed `apply()` used with a matrix, and this indeed is the most common case. The function is defined, however, to take a general multi-way array as its first argument. It also caters to a wide range of possibilities for details such as the shape of the results from individual function calls and the presence or not of `dimnames` labels for the array. See `?apply` for details.

The `apply()` idea is more general than arrays, and corresponds to the common notion of an *iteration operator* found in many functional languages. The array version came first, and stole the general name “`apply`”, but a number of other functions apply a function in iterated calls over elements from one or more lists: `lapply()`, `mapply()`, `rapply()`, and `sapply()`. The first three differ in the way they iterate over the list object(s), while the last attempts to simplify the result of a call to `lapply()`. For an example of `mapply()`, see Section 8.6, page 319.

There are at least two reasons to prefer using `apply()` and friends to an explicit iteration.

1. The computation becomes more compact and clearer.

2. The computation should run faster.

The first of these is often true, and certainly applies to the `columnMax()` example. The second reason, however, is more problematic, and quite unlikely for `apply()` itself, which is coded in R, though carefully. The other functions do have a chance to improve efficiency, because part of their computation has been implemented in C. However, none of the apply mechanisms changes the number of times the supplied function is called, so serious improvements will be limited to iterating simple calculations many times. Otherwise, the n evaluations of the function can be expected to be the dominant fraction of the computation.

So, by all means use the `apply()` functions to clarify the logic of computations. But a major reprogramming simply to improve the computation speed may not be worth the effort.

One detail of `apply()` that sometimes causes confusion is its behavior when we expect to construct a matrix or array result. The function works by concatenating the results of successive calls, remembering each time the length of the result. If the length is identical each time, the result will be a vector (for results of length 1) or a matrix (for vector results of length greater than 1). But the matrix is defined, naturally enough, by taking the length as the first dimension, because that's the way the values will have been concatenated.

Users may be surprised, then, if they apply a function to the rows that always returns a result that looks like the row (i.e., of length `ncol(x)`). They might expect a matrix of the same shape as `x`, but instead the result will be the transpose of this shape. For example:

```
> xS
      [,1] [,2] [,3]
[1,]    6    9   12
[2,]    2    3    5
[3,]    8   11   10
[4,]    1    4    7
> apply(xS,1,order)
      [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    2    2    3    2
[3,]    3    3    2    3
> apply(xS,2,order)
      [,1] [,2] [,3]
[1,]    4    2    2
[2,]    2    4    4
```

| | | | |
|------|---|---|---|
| [3,] | 1 | 1 | 3 |
| [4,] | 3 | 3 | 1 |

Just remember to transpose the result when applying over rows.

Numerical computations with matrices

Numerical matrix computations, including those based on mathematical ideas of linear algebra, are fundamental to many statistical modeling and analysis techniques. Matrix computations are also useful ways to formulate computations that might otherwise be programmed as iterations of elementary arithmetic, with the matrix version being significantly more efficient for large problems.

In fact, some numerical computations with matrices may be measurably more efficient in terms of CPU time than other computations that do the same number of arithmetic operations. Examples include matrix multiplication and other similar operations, usually based on *inner products*, $\sum_{i=1}^n x_i y_i$, or *scalar products*, $\{y_i + ax_i, i = 1, \dots, n\}$. Subprograms for these, usually in Fortran, are known as Basic Linear Algebra Subroutines, or BLAS. The computations done by these subprograms are themselves quite simple and capable of being programmed efficiently. But at the same time a number of higher-level operations can be programmed to do much of their computation through calls to the BLAS routines, leading to efficient implementations, even for quite large problems. Matrix multiplication, for example, is just an iteration of inner products. Many decompositions of numerical linear algebra use BLAS for most of their numerical computation. Linear least-squares fitting, in turn, can be written in terms of numerical decompositions and other efficient matrix operations.

R users don't need to be aware of the underlying operations directly. Functions for numerical linear algebra, statistical models and other applications will make use of the operations through interfaces to compiled code.. If you are installing R from source on a special computing platform, some extra steps may be needed to ensure you get efficient versions of the subprograms. See the installation instructions and related Web pages and FAQ lists for your particular platform.

But when does the speed really matter? Throughout this book, our *Mission* is exploring data—asking questions and trying out new ideas—and the other essential criterion is the *Prime Directive*—providing trustworthy software. Blazingly fast numerical computation does not directly relate to the *Prime Directive* and only serves the *Mission* when it widens the range of potential computations. The CPU unit on your computer is likely to be

idle most of the time, and so it should be if you are spending that time constructively thinking about what you want to do or contemplating some interesting data analysis. For many applications the difference between a very fast numerical method and a similar, less optimized computation would be barely noticeable. It would be a mistake to choose a computation because of numerical speed, if an alternative choice would give more informative results. Having said all that, there remain applications that test the speed of even current computers. Knowing about fast methods can occasionally let us ask questions for such applications that would otherwise not be practical. If you think you have computing needs at this level, by all means try to apply the matrix techniques to your application (see, for example, Section 6.4, page 161, where some of these techniques are applied to “vectorize” a computation).

Matrix operations and matrix arithmetic

Matrices in R that contain numerical data can be used with all the standard arithmetic and comparison operators. The computations work element-by-element and generally return a matrix of the same dimension as the arguments. Consider computations such as:

```
x + y; x / y; x ^ y; x %% y; x >= y;
```

When either x or y is a matrix, the correct answer is obvious if the other argument is either another matrix of the same dimensions or a numeric vector of length 1 (a “scalar”). So, if x is an n by p matrix, the result of each of the following expressions is also a matrix of this shape:

```
x ^ 2; x + 1/x; abs(x) >= .01
```

As the third example shows, the various mathematical and other functions that operate elementwise return a matrix with unchanged dimensions when called with a matrix argument.

Operators called with matrix arguments other than the two cases above are always less clearly defined. R allows a few without comment, warns on some, and generates an error on others.

1. An error results if the arguments are two matrices of different dimensions.
2. An error also results if one argument is a vector larger than the matrix (i.e., of length greater than np).

3. If one argument is a vector exactly of length np , the computation completes silently.
4. If one argument is a vector of length less than np , the vector is implicitly repeated to be of length np . If the original vector was the length of a column, n , the computation silent; otherwise, a warning is generated.

Getting the answer you expect from the last two cases depends on knowing the representation of a matrix; namely, as a structure with data stored by columns.

Numerical linear algebra

This section outlines some of the central tools for numerical computations on matrices, the functions implementing key concepts in numerical linear algebra, particularly computations related to statistical computing. Most of the functions interface to implementations in Fortran or C of corresponding algorithms. Many of the algorithms are taken directly or with some modification from LAPACK, a collection of Fortran subroutines well tuned for both accuracy and speed.

Applications of linear algebra in statistical computing largely come from considering linear combinations of quantitative variables. The standard functions for linear models and analysis of variance, as supplied in the `stats` package, and extended in a number of other R packages, provide users with an interface to the models that largely hides the underlying linear algebra. You should use functions at that higher level unless you really need to work with the fundamental linear algebra relations directly. If you're uncertain, read the documentation for related statistical model software and determine whether it could meet your needs. If so, use it, because many details will be handled that could otherwise compromise the quality of the computed results.

If you really do want to deal with the linear algebra directly, press on, but you may still want to use utilities from the statistical models software to convert from variables such as factors into the matrix arguments used for linear algebra; see, for example, `?data.matrix` and `?model.frame`.

The fundamental role of linear algebra in statistical computing dates back a couple of centuries, and is based on the ability to solve two related computational problems. The first is to find a vector or matrix β that satisfies a set of linear equations. Using the S language operator, `~*~`, for matrix multiplication:

$$\mathbf{a} \%*\% \beta = \mathbf{b}$$

where \mathbf{a} is a square, p by p , matrix and \mathbf{b} is either a vector of length p or a matrix with p rows. The second problem is linear least-squares, finding a vector or matrix β that minimizes the column sums of squares

$$\mathbf{y} - \mathbf{x} \%*\% \beta$$

where \mathbf{x} is an n by p matrix and \mathbf{y} is either a vector of length n or a matrix with n rows.

R functions to solve both of these problems are available and apply to most applications. They are `solve()` for linear equations and `lm.fit()` for least squares. If your application seems to be expressed naturally as one or the other of the two numerical problems, you can probably go away now and use the appropriate function. If you think you need to dig deeper, read on.

The main computational tools for these problems use some fundamental *matrix decompositions*, that is, the computation of special matrices and vectors which if combined, usually by matrix multiplication, would closely approximate the original matrix. The special forms of the decomposition allow them to express the solution to the two problems straightforwardly in most cases, and also make them useful tools in other problems. To see how these relate to modern numerical linear algebra, a little history is needed.

Prior to large-scale electronic computing, the linear least-squares problem would be solved by reducing it to a special linear equation. Linear equations, in turn, could be solved for at least single-digit values of p . When computers came along, software to solve linear equations was very high priority, particularly motivated by military applications and problems in physics. From about the 1960's, software based on matrix decompositions was developed for linear equations, for direct solutions to least-squares problems, and for other problems, such as solving differential equations. The program libraries implementing these results have continually improved in accuracy, speed, and reliability. In particular, the LAPACK software for linear algebra is the current reference for numerical linear algebra, and forms the base for these computations in R.

What does this history imply for modern statistical computing? First, that computations expressed in terms of the standard operations of linear algebra can be applied with confidence, even for quite large problems. If the matrices involved are not so large that manipulating them in R at all is impractical, then at least some operations of linear algebra will also likely be practical for them. Second, a fairly wide range of other computations can be usefully solved by reducing them to operations in linear algebra, either directly through some imaginative re-casting (see the discussion of vectorizing

in Section 6.4, page 158) or by an iterative computation where each iteration is carried out by linear computations (as is the case for some important statistical models). The speed and accuracy provided by LAPACK and similar software means that iterated linear computations may be competitive with other implementations, even though the amount of “arithmetic” seems to be larger.

In addition to the functions for fitting statistical models and the functions `solve()` and `lm.fit()` to solve equations and least-squares problems, the base code for R provides access to several matrix decompositions. These organize themselves naturally on two dimensions; first, on whether the matrix in question is rectangular or square; and, second, between simple decompositions and those that maximize an approximation for each submatrix. The simple decompositions are mainly the `qr()` function for the QR decomposition, for rectangular matrices; and the `chol()` function for the Choleski decomposition, essentially for cross-products and other matrices with similar form. The maximizing decompositions are function `svd()` for the singular-value decomposition of rectangular matrices and `eigen()` for the eigenvalue decomposition of symmetric square matrices. See the online documentation and the references there for some of the details. To really understand the decomposition will require digging into the numerical analysis background. Try the documentation of LAPACK and good books on the subject, such as *Matrix Algorithms* by G. W. Stewart [22].

Beyond the base code for R, there are now a number of packages that extend the range of numerical linear algebra software available. If you have special needs, such as computations for sparse matrices or other decompositions, browse in the documentation for the `Matrix` package by Douglas Bates and Martin Maechler.

6.9 Fitting Statistical models

R and S-Plus both contain software for fitting and studying the types of statistical model considered in the book *Statistical Models in S* [6]. Many of the techniques described in the book are supplied in the `stats` package; a number of other packages fill in the gaps and add other similar types of models or additional techniques. From a programming viewpoint the essential property of the software is that it takes a functional, object-based view of models. For software details, see documentation of the `stats` package. In addition to the original reference, nearly all the general introductions to statistics using R cover the basics; *Modern Applied Statistics with S* by

Venables and Ripley [23] gives a broad, fairly advanced treatment.

There are many other sources of software for statistical models as well (notably for Bayesian model inference and graph-based model formulations). We will not cover any of these here; in many cases, there are either R implementations or R interfaces to software implemented in other languages. Good starting points for a search are the "Related Projects" pointer on the R home page and the usual Web search resources, such as rseek.org.

In the *Statistical Models in S* approach, the functions to fit various types of model all take two primary arguments, a formula expressing the structure of the model and a source of data; they then return an object representing a fitted model estimated according to the arguments. The type of model (linear least-squares, etc.) depends on the choice of function and possibly also on other arguments.

The various top-level fitting functions can be viewed as generators for corresponding classes of objects. For example, the function `lm()` fits a linear model using least-squares to estimate coefficients corresponding to the formula and data supplied, and returns an object of (S3) class "lm", whose elements define the fit.

Other functions then take the fitted model object as an argument and produce auxiliary information or display the model in graphical or printed form. Functions specifically related to the model software include `residuals()` and `fitted()`, with the interpretation depending on the type of model. The function `update()` will allow modification of the fitted model for changes in the data or model. These are S3 generic functions, with methods corresponding to the class of the fitted model. General functions such as `plot()`, `print()`, and `summary()` also have S3 methods for most classes of models.

The various model-fitting functions share the same main arguments and, for the most part, similar computational structure. Here are the main steps, using `lm()` as an example. The functions generally have arguments `formula` and `data`, and it is from the combination of these that the model-fitting proceeds:

```
lm(formula, data, ...)
```

Caution: `formula` is always the first argument, but `data` is not always the second: check the arguments for the particular function.

The `formula` argument must be an object of the corresponding "formula" class, which is generated by a call to the `~` operator. That operator returns its call as its value, promoted to the "formula" class, meaning that the formula is essentially a symbolic constant describing the structure of

the model, with one sometimes crucial addition: It has an attribute with a reference to the environment in which the call took place.

The convention is to read the \sim as “is modeled by”, so the left-side argument to the operator is the variable or derived quantity being modeled and the right-side argument is the expression for the predictor. Just how the arguments are interpreted depends on the kind of model. Linear models and their extensions use conventions about the meaning of other operators ($+$, $*$, and $:$) to indicate combinations of terms, along with other functions that are interpreted as they usually would be in R. Other models will use other conventions.

The `data` argument is optional and if supplied is usually an object of S3 class `"data.frame"`, containing named variables, each of which corresponds to values on the same n observations. Some of the names will typically appear in the `formula` object, and if so, those variables will be used in fitting the model.

Section 6.5, page 168 discusses `"data.frame"` objects generally. For model-fitting, some classes of variables that are valid in a data frame may not be valid in a particular type of model. Linear models and their extensions in `glm()` and `gam()` essentially only allow numeric predictors, which can be supplied as `"numeric"`, `"matrix"` or `"factor"` objects. The matrix must be numeric with n rows. A factor is included by being coded numerically using *contrasts* to differentiate observations according to the levels of the factor.

The `formula` and `data` arguments are used to prepare the more explicit data required to fit the model. The form depends on the type of model but again linear models provide an example typical of many types of model.

The preparation of a linear model for actual fitting by `lm.fit()` proceeds in two steps. First, a data frame containing the specific variables implied by the `formula` and `data` arguments is computed by the `model.frame()` function. Then the matrix for the linear model fitting itself is computed from the model frame by the `model.matrix()` function.

The computation of the model frame brings the `formula` and the supplied data frame together, evaluating expressions derived from the `formula` by an explicit call to `eval()`. The purpose of the call is to form a data frame containing all the variables appearing in the `formula`; this is the “model frame” object. The model frame also has a `"terms"` object as an attribute; essentially, this is the `formula` with some extra attributes. When the `data` argument is supplied and all the variables in the model are found in that object, the result is to select the suitable variables, and all is well. That’s the trustworthy approach: Assemble all the relevant variables explicitly in a

data frame, and supply that data frame to the model-fitting function (`lm()`, `gam()`, etc.).

Otherwise, the computations for the model frame must look elsewhere for some or all of the variables. The critical fact is that the computations look in the environment of the formula object, stored in the object as the environment where the formula was created.

If the formula was typed by the user interactively, then the call came from the global environment, meaning that variables not found in the data frame, or all variables if the `data` argument was missing, will be looked up in the same way they would in ordinary evaluation. But if the formula object was precomputed somewhere else, then its environment is the environment of the function call that created it. That means that arguments to that call and local assignments in that call will define variables for use in the model fitting. Furthermore, variables not found there will be looked up in the parent (that is, enclosing) environment of the call, which may be a package namespace. These rules are standard for R, at least once one knows that an environment attribute has been assigned to the formula. They are similar to the use of closures described in Section 5.4, page 125.

Where clear and trustworthy software is a priority, I would personally avoid such tricks. Ideally, all the variables in the model frame should come from an explicit, verifiable data source, typically a data frame object that is archived for future inspection (or equivalently, some other equally well-defined source of data, either inside or outside R, that is used explicitly to construct the data for the model).

Once the model formula and (in the case of linear-style models) the model matrix have been constructed, the specific fitting mechanism for this class of models takes over, and returns an object from the corresponding S3 class, such as `"lm"`, `"gam"`, `"nls"` and many more. The mechanisms and the interpretation of the fitted model objects that result vary greatly. Generally, however, you can get a good picture of the programming facilities provided by looking for S3 methods associated with the generic functions for models (`residuals()`, `update()`, etc.) and for printed or graphical summaries (`print()`, `summary()`, `plot()`, etc.).

6.10 Programming Random Simulations

This section considers some programming questions related to the use of pseudo-random generators, or less directly, computations involving the *Monte-Carlo method*.

We begin by summarizing the overall organization of simulation functions in R, with an assessment of the level of trust one can have in their “correctness”. A second issue for trustworthy simulation results is that others can reproduce them; on page 226 we discuss techniques for this goal. We then show a related example that examines how robust a simulation is to small failures in reproducibility (page 230). Finally, on page 234, we consider the use of generators in low-level code, such as C or C++, which we may want to use for efficiency.

The starting point for simulating in R is a set of “random generator” functions that return a specified number of values intended to behave like a sample from a particular statistical distribution. Because no common generators in practice use an external source thought to be truly random, we are actually talking about *pseudo-random generators*; that is, an ordinary computation that is meant to simulate randomness. We follow common custom in this section by dropping the “pseudo-” when we talk about random generators; you can mentally put it back in, and prepend it to statements about how “likely” some event is, or to other properties of sequences from pseudo-random generators.

Basic concepts and organization

R provides random generators for a variety of statistical distributions as well as some related generators for events, such as sampling from a finite set. Conceptually, all the generators in the `stats` package work through the package’s uniform generator, either at the R level or the C level. This leads to the key techniques for achieving trustworthy software for simulation, as we explore on page 226, but the concept is worth noting now.

Random generators don’t follow the functional programming model, as implemented in R or in most other systems, because they depend on a current global state for the generator. How would we formulate a functional approach to simulation? Given that all generators work through the uniform generator, we could imagine an object that is our personal stream of uniform random numbers. If this stream was an argument to all the actual generators and to any other function for simulation, then all the remaining computations can be defined in terms of the stream. In practice, this would require quite a bit of reorganization, but the essential point is that no other external dependence is required.

In practice, such a stream is represented by a seed for the generator. A combination of techniques in the `stats` package and some extensions in the `SoDA` package can get us trustworthy software, essentially by incorporating

sufficient state information with the computed results.

Functions for probability distributions in R are organized by a naming tradition in the S language in which the letters "r", "p", "q", and "d" are prepended to a fixed name for a distribution to specify functions for random numbers, cumulative probability, quantiles, and density function values for the distribution. So, "unif" is the code for the uniform distribution, resulting in functions `runif()`, `punif()`, `qunif()`, and `dunif()` (admittedly, none but the first of these is hard to program). Similar functions are defined for the normal ("norm"), Poisson ("pois"), and a number of others, all on the core package `stats`; to look for the distribution you need, start with:

```
help.search("distribution", package = "stats")
```

Some distributions may not have all four functions. If no random generator is provided but a quantile function does exist, you can get a random sample by what's known as the *probability integral transformation*, which just says to compute the quantiles corresponding to a sample from the standard uniform distribution. The Studentized range ("`tukey`") distribution has a quantile version but no random generator, so you could define a rough version for it as:

```
rtukey <- function(n, ...)
  qtkey(runif(n), ...)
```

Where some functions are missing, there may be numerical issues to consider, so it's a good idea to check the documentation before putting too much faith in the results.

Additional random generators are supplied in other packages on CRAN and elsewhere; for these non-`stats` generators, especially, you should check whether they are integrated with the basic uniform generator if trustworthy results are important.

Are pseudo-random generators trustworthy?

From early days, statistical users of generators have asked: "Can the numbers be treated as if they were really random?". The question is difficult and deep, but also not quite the relevant one in practice. We don't need real randomness; instead, we need to use a computed simulation as the approximate value of an integral (in simple cases) or an object defined as a probabilistic limit (for example, a vector of values from the limiting distribution of a Markov chain). All we ask is that the result returned be as good

an approximation to the limit value as probability theory suggests. (We'd even settle for "nearly as good", if "nearly" could be quantified.)

Good current generators justify cautious confidence, with no absolute guarantees, for all reasonably straightforward computations. Popular generators, such as the default choice in R, have some theoretical support (although limited), have performed adequately on standard tests, and are heavily used in an increasingly wide range of applications without known calamitous failures. R's default generator is also blazingly fast, which is an important asset as well, because it allows us to contemplate very extensive simulation techniques.

If this sounds like a lukewarm endorsement, no radically better support is likely in the near future, as we can see if we examine the evidence in a little more detail.

As in the `stats` package, we assume that all simulations derive from a *uniform pseudo-random generator*, that is, a function that simulates the uniform distribution on the interval $0 < x < 1$. Function `runif()` and a corresponding routine, `unif_rand`, at the C level are the source of such numbers in R.

If one completely trusted the uniform generator, then that trust would extend to general simulations, as far as approximating the probabilistic limit was concerned. There would still be questions about the logical correctness of the transformation from uniforms to the target simulation, and possibly issues of numerical accuracy as well. But given reassurances about such questions, the statistical properties of the result could be counted on.

The evidence for or against particular generators is usually a combination of mathematical statements about the complete sequence, over the whole period of the generator, and empirical tests of various kinds. A complete sequence is a sequence from the generator with an arbitrary starting point such that, after this sequence, the output of the generator will then repeat.

Generators going back to the early days provide equidistribution on the complete sequence. That is, if one counts all the values in the sequence that fall into bins corresponding to different bit representations of the fractions, the counts in these bins will be equal over the complete sequence. More recent generators add to this equidistribution in higher dimensions; that is, if we take k successive numbers in the sequence to simulate a point in a k -dimensional unit cube, then the counts in these k -dimensional bins will also be equal over the complete sequence.

Such results are theoretically attractive, but a little reflection may convince you that they give at best indirect practical reassurance. The period of most modern generators is very long. The default generator for R, called

the “Mersenne-Twister” has a period a little more than 10^{6000} , effectively infinite. (To get a sense of this number, I would estimate that a computer producing one random uniform per nanosecond would generate about 10^{25} numbers, an infinitesimal fraction of the sequence, before the end of the earth, giving the earth a rather generous 10 billion years to go.)

Sequences of a more reasonable length are not directly predictable, nor would you want them to be. Exact equidistribution over shorter subsequences is likely to bias results.

A more fundamental limitation of equidistribution results, however, is that most use of generators is much less regular than repeated k -dimensional slices. Certainly, if one is simulating a more complex process than a simple sample, the sequence of generated numbers to produce one value in the final set of estimates will be equally complex. Even for samples from distributions, many algorithms involve some sort of “rejection” method, where candidate values may be rejected until some criterion is satisfied. Here too, the number of uniform values needed to generate one derived value will be irregular. Some implications are considered in the example on page 230.

Turning to empirical results, certainly a much wider range of tests is possible. At the least, any problem for which we know the limiting distribution can be compared to long runs of a simulation to test or compare the generators used. Such results are certainly useful, and have in the past shown up some undesirable properties. But it is not easy to devise a clear test that is convincingly similar to complex practical problems for which we don’t know the answer. One needs to be somewhat wary of the “standardized test syndrome” also, the tendency to teach students or design algorithms so as to score well against standardized tests rather than to learn the subject or do useful computations, respectively.

The results sound discouraging, but experience suggests that modern generators do quite well. Lots of experience plus specific tests have tended to validate the better-regarded generators. Subjective confidence is justified in part by a feeling that serious undiscovered flaws are fairly unlikely to coincide with the pattern of use in a particular problem; based on something of a paraphrase of Albert Einstein to the effect that God is mysterious but not perverse.

To obtain this degree of confidence does require that our results have some relation to the theoretical and empirical evidence. In particular, it’s desirable that the sequence of numbers generated actually does correspond to a contiguous sequence from the chosen generator. Therefore, all the generators for derived distributions should be based in a known way on uniform random numbers or on compatible R generators from the standard

packages. New generators implemented at the level of C code should also conform, by using the standard R routines, such as `unif_rand`. One wants above all to avoid a glitch that results in repeating a portion of the generated sequence, as could happen if two computations used the same method but inadvertently initialized the sequence at two points to the same starting value. Taking all the values from a single source of uniforms, initialized once as discussed below, guards against that possibility.

In using packages that produce simulated results, try to verify that they do computations compatible with R. There are packages on CRAN, for example, that do their simulation at the C level using the `rand` routine in the C libraries. You will be unable to coordinate these with other simulations in R or to control the procedures used. If you are anxious to have reliable simulations, and particularly to have simulations that can be defended and reproduced, avoid such software.

Reproducible and repeatable simulations

Ensuring that your simulation results can be verified and reproduced requires extra steps, beyond what you would need for a purely functional computation. Reproducible simulations with R require specifying which generators are used (because R supports several techniques) and documenting the initial state of the generator in a reproducible form. Another computation that uses the same generators and initial state will produce the same result, provided that all the numerical computations are done identically. The extra requirement is not just so that the last few bits of the results will agree; it is possible, although not likely, that numerical differences could change the simulation itself. The possibility comes from the fact we noted earlier: Nearly all simulations involve some conditional computation based on a numerical test. If there are small numeric differences, and if we run a simulation long enough, one of those conditional selections may differ, causing one version to accept a possible value and the other to reject the value. Now the two sequences are out of sync, a condition we discuss as *random slippage* on page 230. To avoid this possibility, the numerical representations, the arithmetic operations, and any computational approximations used in the two runs must be identical.

The possibility of numerical differences that produce random slippage emphasizes the need to verify that we have reproduced the simulation, by checking the state of the generator after the second run. To do so, we need to have saved the final state after the initial run. Verification is done by comparing this to the state after supposedly reproducing the result. Both

the two simulation results and the two states should match.

But reproducing a simulation exactly only tests that the asserted computations produced the results claimed, that is, the programming and software questions. For simulations, another relevant question is often, “How do the results vary with repeated runs of this computation?”. In other words, what is the statistical variability? Repeated runs involve two or more evaluations of the computation, conceptually using a single stream of random numbers, in the sense introduced on page 222.

A simulation result is *repeatable* if the information provided allows the same simulation to be repeated, with the same result as running the simulation twice in a single computation.

The technique to assure repeatability is to set the state of the generator at the beginning of the second run to that at the end of the first run. Once more, we see that saving the state of the generator at the end of the simulation is essential.

The SoDA package has a class, "simulationResult", and a related function of the same name that wraps all this up for you. It records the result of an arbitrary simulation expression, along with the expression itself and the first and last states of the generator. An example is shown on page 230.

The state, or *seed* as it is usually called, is a vector of numbers used by a particular uniform generator. Intuitively, the generator takes the numbers and scrambles them in its own fashion. The result is both a new state and one or more pseudo-random uniform numbers. The generators are defined so that n requests for uniform variates, starting from a given initial state will produce the same final state, regardless of whether one asks for n variates at once or in several steps, provided no other computations with the generator intervene. The information needed in the state will depend on which generator is used.

The user’s control over the generator comes from two steps. First, the generating method should be chosen. R actually has options for two generators, for the uniform and normal distributions. For completeness you need to know both, and either can be specified, via a call to the function `RNGkind()`. The call supplies the uniform and normal generator by name, matched against the set of generators provided. For those currently available, see the documentation `?RNGkind`. If you’re happy with the default choices, you should nevertheless say so explicitly:

```
RNGkind("default", "default")
```

Otherwise R will continue to use whatever choice might have been made before. Provided we know the version of R, that specifies the generators unambiguously.

Second, the numerical seed is specified. The simplest approach is to call:

```
set.seed(seed)
```

where `seed` is interpreted as a single integer value. Different values of `seed` give different subsequent values for the generators. For effective use of `set.seed()`, there should be no simple relation of the state to the numerical value of `seed`. So, for example, using `seed+1` should not make a sequence “nearer” to the previous one than using `seed+1000`. On the other hand, calling `set.seed()` with the same argument should produce the same subsequent results if we repeat exactly the same sequence of calls to the same generators.

The use of an actual seed object can extend the control of the generators over more than one session with R. As mentioned before, the generators must save the state after each call. In R, the state is an object assigned in the top-level, global environment of the current session, regardless of where the call to the generator occurred. This is the fundamentally non-functional mechanism. The call to `set.seed()` also creates the state object, `.Random.seed`, in the global environment. After any calls to the uniform generator in R, the state is re-saved, always in the global environment. Note, however, that random number generation in C does not automatically get or save the state; the programmer is responsible for this step. See the discussion and examples on page 234.

When the generator is called at the beginning of a session, it looks for `.Random.seed` in the global environment, and uses it to set the state before generating the next values. If the object is not found, the generator is initialized using the time of day. As you might imagine, the least significant part of the time of day, say in milliseconds, would plausibly not be very reproducible, and might even be considered “random”.

To continue the generator sequence consistently over sessions, it is sufficient to save `.Random.seed` at the end of the session, for example by saving the workspace when quitting. The `.Random.seed` object can also be used to rerun the generator from a particular point in the middle of a simulation, as the following example illustrates.

Example: Reproducible computation using simulations for model-fitting

One aspect of the *Prime Directive* is the ability to reproduce the result of a computation: To trust the result from some software, one would at least like to be able to run the computation again and obtain the same result. It's natural to assume that a fitted model is reproducible, given the data and knowledge of all the arguments supplied to the fitting function (and, perhaps, some information about the computing platform on which the computation ran).

For classical statistical models such as linear regression, reproducibility is usually feasible given this information. But a number of more modern techniques for model-fitting, statistical inference, and even general techniques for optimization make use of simulations in one form or another. Modern Bayesian techniques do so extensively. But other techniques often use simulated values internally and here the dependence may be less obvious. A whole range of optimization techniques, for example, use pseudo-random perturbations.

The package `gbm` by Greg Ridgeway fits models by the technique of “gradient boosting”. In Section 12.6, page 441, we look at the software as an example of an interface to C++. Examining the code shows that the method uses random numbers, but a casual reader of the literature might easily not discover this fact. If not, the non-reproducibility of the results might be a shock.

Running `example(gbm)` from the package produces `gbm1`, a model fitted to some constructed data, and then continues the iteration on that model to produce a refined fit, `gbm2`. If the results are reproducible, we should be able to redo the second stage and get an object identical to `gbm2`:

```
> gbm2 <- gbm.more(gbm1,100,
+ verbose=FALSE) # stop printing detailed progress

> gbm22 = gbm.more(gbm1,100,verbose=FALSE)
> all.equal(gbm2, gbm22)
[1] "Component 2: Mean relative difference: 0.001721101"
[2] "Component 3: Mean relative difference: 0.0007394142"
[3] "Component 4: Mean relative difference: 0.0004004237"
[4] "Component 5: Mean relative difference: 0.4327455"
```

And many more lines of output

Component 2 is the fitted values. A difference of .1% is not huge, but it's not a reproducible computation, perhaps weakening our trust in the software.

In fact, nothing is wrong except that we don't have control of the state of the generator.

Examining the implementation shows that the C++ code is importing the R random number generator, but is not providing a mechanism to set the seed. Once this is seen, a solution is straightforward. The `simulationResult()` function in the SoDA package wraps the result of an arbitrary expression to include the starting and ending states of the generator.

```
run2 <- simulationResult(
  gbm.more(gbm1, 100, verbose = FALSE))
```

By making the previous expression the argument to `simulationResult()`, we can at any time reset the generator to either the first or last state corresponding to the run.

```
> .Random.seed <- run2@firstState
> gbm22 <- gbm.more(gbm1,100,verbose=FALSE)
> all.equal(run2@result, gbm22)
[1] TRUE
```

Remember that the generator looks for the seed only in the global environment; if the computation above were done in a function, the first step would require:

```
assign(".Random.seed", run2@firstState, envir = .GlobalEnv)
```

Example: Are pseudo-random sequences robust?

Our next example investigates what happens when a sequence of generated numbers is perturbed “a little bit”. As mentioned earlier, such slippage can occur in an attempt to reproduce a simulation, if small numerical differences cause a conditional result to be accepted in one case and rejected in the other. To catch such an event is tricky, but we can emulate it and study the effect. Does such a small change completely alter the remainder of the simulation or is there a resynchronization, so that only a limited portion of the results are changed?

Consider the following experiment. We simulate $n = n_1 + n_2$ values from the normal distribution, in two versions. In the first version, nothing else is generated. In the second version, we make the smallest possible perturbation, namely that after n_1 values are generated, we generate one uniform variate, then go on to generate n_2 more normals, as before. What

should happen? And how would we examine the results of the two runs to describe the perturbation?

In the simplest concept of a generator, each normal variate is generated from a uniform variate. The obvious method is to compute the quantile corresponding to the generated uniform, in this case `qnorm(runif(1))`. With this computation, the second sample lost out on one normal value, but from then on the samples should match, but just be off by one.

As it happens, the default normal generator in R is indeed defined as this computation, known in Monte-Carlo terminology as the inversion method. We might expect the slippage as described, but in fact that does not happen. All the generated values in the second version are different from those in the first version. Why? Because the inversion code uses two uniforms in order to get a more extended argument for the quantile function. As a result, if the slippage involves an even number, $2k$, of uniforms, then the output will resynchronize after k values, but slippage by an odd number of uniforms will never resynchronize.

The default algorithm for normals shows the fragility, but an alternative algorithm gives a more typical example. Let's set the normal generator technique by:

```
RNGkind(normal.kind = "Ahrens-Dieter")
```

This technique uses some tests to choose among alternative approximations, so that the number of uniform values needed per normal variate is random (well, pseudo-random). To see how slippage affects this algorithm, let's program our experiment. The computation in a general form is done by the function `randomSlippage()` in the SoDA package, which does essentially the following computation.

We carry out the preliminary simulation, and save the state:

```
g1 <- rnorm(n1)
saveSeed <- .Random.seed
```

Now we carry out the second simulation and save the result, twice. The second time we reset the seed, but this time generate some uniforms (1 in the simplest experiment), before the repeated simulation:

```
g21<-rnorm(n2)
assign(".Random.seed", saveSeed, envir = .GlobalEnv)
u1<-runif(slip)
g22<-rnorm(n2)
```

The next step is to compare the second batch of normal variates, `g21` and `g22` from the two branches. The question of interest is whether the two sequences resynchronize and if so, where. The generator starts off producing values under the two situations. If at some point the two batches contain exactly the same number, we expect this to have been produced by the same set of uniforms in both cases, given our overall confidence in the uniform generator. From this point on, the two sequences should be exactly identical, after having slipped some amount on each sequence. The two slippage amounts measure how much we have perturbed the simulation.

How to program this? Whenever the word “exactly” comes up in comparisons, it’s a clue to use the function `match()`. We’re dealing with numerical values but are uninterested in these as numbers, only in equality of all the bits. Suppose we match the two second-part sequences:

```
m <- match(g21, g22)
```

What do we expect in `m`? Because the second sequence inserted *slip* uniforms, we expect the first few elements of `g21` won’t appear in `g22`. The corresponding elements of `m` will be `NA`. If the sequence resynchronizes, some element will match beyond some point, after which all the elements of `m` should be successive positive integers. The two numbers representing the slippage are the index of the first non-`NA` value in `m`, and the corresponding element of `m`. In the following code, we find this index, if it exists, and insert the two numbers into a row of the matrix of slippage values being accumulated.

```
seqn2 <-() seq(along = g21)
m <- match(g21, g22)
k <- seqn2[!is.na(m)]
if(length(k) > 0) {
  k <- k[[1]]
  slippage[i,] <- c(k, m[[k]])
}
```

If the normal generator uses just one uniform, then we expect the second item in the unperturbed generator to match the first in the perturbed generator if *slip* is 1. The corresponding row of the test results would be `c(2, 1)`. The Ahrens-Dieter generator uses one value most of the time, and applies various tests using more uniform values to match the generated distribution to the normal. Here is an example, doing 1000 runs, and then making a table of the results:

```
> RNGkind("default", "Ahrens")
```

```
> set.seed(211)
> xx <- randomSlippage(1000, rnorm(10), rnorm(10))
> table(xx[,1], xx[,2])
```

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|-----|----|---|----|---|---|---|---|
| 2 | 891 | 49 | 5 | 24 | 9 | 2 | 3 | 1 |
| 3 | 0 | 4 | 1 | 2 | 1 | 1 | 0 | 0 |
| 4 | 0 | 2 | 1 | 1 | 1 | 0 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

As expected, about 90% of the time the generator resynchronizes after missing one of the original values. The remainder of the pattern is more complex, depending on the algorithm's choices of alternative computation in the perturbed or unperturbed sequence.

Notice that the experiment specified both the initial seed and the types of generator to use. The initial value of `.Random.seed` will contain the internal version of both the seed and the choices of generator. This seed is included as an attribute of the value returned by `randomSlippage()`, so to redo the computations:

```
.Random.seed <- attr(xx, "seed")
newXX <- randomSlippage(1000, rnorm(10), rnorm(10))
```

The various other arguments could be inferred from the previously returned value as well. When designing simulation experiments for serious applications, try to include such information in the object returned, particularly the initial seed.

A few programming comments on `randomSlippage()`. It takes two literal expressions as arguments, the computations to be done before and after the slippage, as well as a third argument for the slippage itself, which defaults to `runif(1)`. As often in programming with R, we have turned the specific experiment into a rather general technique with only a little extra work, by computing with the language itself. The expression for the simulation after the slippage can do anything at all, so long as it returns an object for which the matching comparison makes sense. See the listing of the function in the SoDA package for details.

A note on *vectorizing*: Because the result returned uses only the first matching position, on the assumption that the two series are then synchronized, one might object to matching the whole object. However, because `match()` operates by creating a hash table, it is fast for comparing a number

of values in its first argument. Testing for the first element, then the second if necessary, and so on, would tend in fact to be slower.

The extra information from the full match also allows us to test our assumption that the two sequences are synchronized if they ever have equal elements. The `randomSlippage()` function includes a `check` argument, `FALSE` by default, that optionally tests the assumption:

```
if(check && k1 < n2
    && ( any(diff(k) != 1) || any(diff(m[k]) != 1)))
    stop("Non-synchronized . . . . .")
```

In a sufficiently large simulation, exactly identical values could in principle occur without the generator being resynchronized.

Pseudo-random generators in C

Simulating a process that is not simply a large sample of independently generated values often leads to a one-number-at-a-time computation. The next value to be generated requires tests based on the preceding values and/or involves trying various alternatives. It's natural to look for computationally efficient software in a language such as C in these cases. Software for simulation is not trivial, however, and when written in such languages needs to be both flexible to use and trustworthy, our two guiding principles. Whether you are evaluating someone else's software or planning your own, here are some suggested guidelines.

The low-level implementation of techniques should not compromise users' flexibility, their ability to use the simulation software to explore freely (the *Mission*). That's a guideline for all low-level code, and largely means that the C code should be a small set of clear, functionally designed tools. The standard approach via the `.C()` or `.Call()` interface would be to design a set of R functions. The functions should have a clear purpose, be well-defined in terms of their arguments and together give the user a flexible coverage of the new simulation techniques.

From the viewpoint of trustworthy software (the *Prime Directive*), extra care is particularly important with simulation, because programming errors can be hard to detect. Because the computations are by definition (pseudo-)random, some aspects of the code will only be tested rarely, so bugs may only show up much later. Some special requirements come from the reproducibility aspects noted above. For trustworthiness as well as convenience, the techniques should conform to standard conventions about setting seeds and choice of basic generators, in order for results of the new functions to be reproducible and therefore testable.

C-level access to the basic R generators is supplied by a simple interface with access to the uniform, normal and exponential distributions. The official interface is described in the *Writing R Extensions* manual, and consists of the routines:

```
double unif_rand();
double norm_rand();
double exp_rand();
```

For most purposes, the uniform generator is likely to be the essential interface. It is essential for consistency with other simulation computations that the C code get the state of the generator (that is, the seed) before calling any of these routines and save the state after finishing the simulation. These two operations are carried out by calling the C routines:

```
GetRNGstate();
PutRNGstate();
```

Therefore, any C routine that does some simulation and then returns control to R should be organized somewhat like the following, imaginary example. The following snippet uses the `.C()` interface, to a C routine `my_simulation` taking arguments for a vector `pars` of parameters defining the simulation and a vector `x` in which to store and return some computed values. The lower-level routine called in the loop will do something involving `unif_rand` and/or the normal or exponential routines, and return one numeric value from the simulation. The simulation in the loop is bracketed by getting the state of the generator and putting it back.

```
void my_simulation(double *x, double *pars,
                  double *nx_ref, double *npars_ref) {
    long nx = *nx_ref, npars = *npars_ref, i;

    GetRNGstate(); /* initialize random seed */
    for(i = 0; i < nx; i++) {
        x[i] = do_some_simulation(pars, npars);
    }

    PutRNGstate(); /* save random seed before returning */
}
```

A more extensive computation may prefer to use C++, as is the case with several packages on CRAN. For an example of a C++ interface to the R generators, see Section 12.6, page 442.

There are many libraries of random number generators, other than those in R. However, from our guiding principles of usefulness and trustworthy software, I think the burden of proof is on having a good reason not to use the interface to the R generators. The basic generators have been carefully studied and very extensively used in practice. They include a number of currently competitive algorithms, and have facilities for introducing user-defined generators (although that's an activity probably best left to those who specialize in the field). Most importantly, they can be combined with a wide variety of other R software, both for simulation and for general support.

If you are using some simulation software in a package, I would recommend testing whether the software is properly integrated with the basic R simulation mechanism. The test is easy: set the seed to anything, do a calculation, then reset the seed to the same value and repeat the calculation. If the two values are not identical (and you really have asked for the identical computation from the package), then there is some "slippage" as in our example. Quite possibly the slippage is total, in that the package is using a different generator, such as that supplied with the basic C library. Unless there is a good alternative mechanism, it's a definite negative for the package.