Chapter 12

Interfaces II: Between R and Other Systems

This chapter discusses general inter-system interfaces between computations in R and those done in other languages and systems. "Other" generally has two senses here: The implementation usually involves communicating with another application; and more fundamentally, the computational model for the other system may be different from that in R.

The chapter discusses several approaches, the best choice depending on the other system and on the nature of the particular task: file or text-based (Section 12.2, page 432), functional (12.3, 433), or object-based (12.4, 435). Moving down the list generally provides greater flexibility and efficiency, paid for by more effort in installing the interface and programming the application. The programming model of some systems needs to be considered explicitly, as with OOP systems (Section 12.5, page 437), and C++ in particular (12.6, 440), and with database or spreadsheet systems (12.7, 446).

With our emphasis on programming with R, most of the examples invoke computations in another system *from* R, but a number of the packages implementing the interface support communication *to* R as well. In addition, some applications benefit from interfaces that avoid R altogether (Section 12.8, page 450).

12.1 Choosing an Interface

Chapter 11 discussed computations in R that called routines written in C or related languages. These routines could be called directly in the R session, so long as the routine was loaded with a package or from a library, thanks to the fact that R itself is implemented in C.

Once we think of software in other languages and systems, the picture changes. Now we are communicating in a more equal sense: The other system will typically have its own ideas of how programming is expressed (its "programming model") and quite likely its own ideas of objects and data. We have a number of choices to make in communicating between the systems.

Many such systems are of potential value for data analysis. Interfaces to some important systems are summarized in Table 12.1. Unless noted, the

System	Applications	Package	Source
Perl	Text, WWW, coding, interfaces, ...	`RSPerl`	omegahat
Python	(similar to Perl)	`RSPython` `rpy` (to R)	omegahat Sourceforge
Java	User interfaces, events, graphics, ...	`rJava` (from R) `JRI` (to R) `RSJava`	CRAN RForge omegahat
C++	Algorithms, processes	`.C()`, `.Call()`	(built in)
Oracle, MySQL, ...	Relational databases	`ROracle` `RmySQL`	CRAN CRAN
Tcl/Tk	User interface	`tcltk`	(built in)

Table 12.1: Some inter-system interfaces. (Web pages for the sources: `cran.r-project.org`, `omegahat.org`, `sourceforge.net`, `rforge.net`)

interfaces provide for communication both from R and to R. Communication from R generally means that the user calls an R function that then invokes the other system as an application, communicates with a running evaluator for that system, or invokes some built-in or compiled code. Communication in the other direction usually involves embedding or dynamically linking R to a process or application running the other system. Installing an interface in this form does require some extra steps beyond a minimal installation of R, so that the embedding or linking of the R software is possible. Interfaces

that are built-in run as code in the R process itself; for these interfaces, communication to R is usually through the mechanisms for evaluating function calls in R from C.

This chapter concentrates on obtaining computations in R from other systems, given our focus on building software for data analysis, and on programming with R.

Forms of interface

There are many variations in how the computations are done and communicated. We can usefully group these into three types.

1. *Text based*: In this form, text is communicated including a command that the other system executes. Text output from the command is communicated back. In effect, this is the model provided by the function `system()` in R, also sometimes referred to as the "Unix pipe" model for computation.

2. *Function based*: One or more functions in R communicate requests to the other system and return a resulting value. Usually, the arguments to the R functions identify a function or something similar in the other system and then provide arguments to that function. The interface functions to C, such as `.C()`, are the paradigm for this approach.

3. *Object based*: At least some of the computations may create and refer to objects in the other system. In particular, there may be what we call *proxy* objects in R that stand for objects in the other system.

The three models for communication are listed in order of increasing generality, in terms of what can be done, and also of an increasing level of organization required. Setup requirements also tend to increase as we go down the list, although these also depend on which system is involved.

Text- or file-based interfaces can often avoid installing an explicit interface package, in two ways. If it's sufficient to occasionally export data from one system and import it into the other, one can use the techniques of Section 8.2, page 294, on importing text or those of Section 6.5, page 173, on exporting and importing in spreadsheets and database systems. If the other system can be invoked as a "shell" command, one can use the `system()` function as described in section 12.2. Otherwise, and generally for both function-based and object-based interfaces, there must be an inter-system interface package, such as those in Table 12.1. The interface package needs to be installed on your computer, if it isn't already. Installation may

not be quite as simple as with other packages, since both R and the other system must be available in a compatible form. If you encounter problems, check the hints and instructions on the package's Web site and also search for relevant discussions on the R mailing lists. Unfortunately, there are no definitive techniques for all situations.

The distinctions among the levels are not rigid; each level is capable of implementing the next level down, at least partially. We could also have added a fourth level, *Component based*. In this model, the interfaces are made up of components that advertise services or methods. Interfaces of this form have much promise for future work, but at the time of writing the activity is either restricted to the Windows operating system (DCOM) or to specialized communication systems that are not much used with statistical computing.

12.2 Text- and File-Based Interfaces

Interfaces can be established from R to any system that can be invoked as a shell-style command. The function `system()` is the general tool to invoke a command. How well an interface of this form works in practice, however, depends on the other system and somewhat on the platform. Shell commands are at the heart of the UNIX operating system's programming model, and are fully compatible with Linux or Mac OS X; on Windows some extra software may need to intervene to provide UNIX-style commands, but the `system()` function itself has been designed to be platform-independent. The function provides for specifying standard input as a character vector in R, or any class of objects that can be interpreted as a character vector. The catch, particularly on Windows, is that the commands invoked via `system()` must be available. If you plan to do any significant programming with non-R software on Windows, see the Appendix on the Windows toolset in the *Installing and Administering R* manual at the R Web site.

Interfaces to scripting languages such as Perl or Python often fit easily into a command-style interface. In Section 8.5, page 310, a simple Perl program was shown that removed HTML tags. In the form shown, the program used the UNIX style of reading data from its input and writing to its output, which fits naturally with the `system()` function, using its `input=` argument to specify the standard input.

A command-style interface for Perl and similar languages must start the interpreter for the language each time and open files or other connections for input and output. If the application involves many evaluations of small text

processing jobs, a more efficient mechanism is to start the interpreter and give it successive tasks directly—the approach of the functional interface. A functional interface may be more natural as well, if there are several related tasks to be requested, since those may naturally map into corresponding functions or methods in the other system.

The balance shifts in favor of text-based interfaces when the text comes from an external source and is either extensive or structured in a non-trivial way by another language. Both conditions often apply. If the text arises as a document or data stream in some markup or display language (XML or HTML, for example), extracting the relevant text for R may need some flexible programming in Perl.

12.3 Functional Interfaces

Functional interfaces, such as `.Perl()` in the `RSPerl` package, allow the R programmer to execute a function or similar programming construction in another system, and retrieve the result in R. At a basic level, the arguments and result may be treated as R objects, provided there are unambiguous analogs in the two systems. This level is functional, in the sense we use the term frequently in the book: the effect of the computation can be entirely described functionally, in terms of the arguments and value with no discussion of side effects.

In a simple functional interface, all arguments will be converted to the foreign system and all results converted back. This is the model for the `.C()` interface to C. If we examine the way that interface works a little more closely, it will illustrate the essential points to understand in other interfaces with a similar model.

Each of the arguments to be passed through `.C()` is required to be an array of one of the basic vector datatypes (those listed in Table 11.1 on page 415). Furthermore, all results are returned by the C routine by modifying the contents of these arrays. So only a very special set of routines will qualify. The strategy may not be perfect, but it is one approach to managing the different programming models of C and R.

Interfaces to other systems will also impose some restrictions in order to make a simple functional interface possible. For a particular interface, see what the documentation says about converting arguments and results. Some experimentation may be needed, and perhaps some extra techniques to convert objects you need to work with. For example, tables of objects indexed by strings are important in many systems (*hashes* in Perl and *dic-*

tionaries in Python, for example). If you are supplying such data as an argument or getting it back as the value of a call, you need to know whether there is an automatic conversion and, if so, what the corresponding object in R will be. If there are no practical conversions for objects appearing as arguments or as the value returned, then you may need to use the notion of proxy objects in the other system as discussed in the next section. Even if a conversion is possible, there may be computational advantages to the object-based approach.

For a specific example of a functional interface, consider `.Perl()`. This function in package `RSPerl` constructs and evaluates a call to a Perl function. For a simple functional interface to work in this case, it's important to understand how arguments are passed to a function in Perl. Basically, the argument list is a single array of scalars, either basic types such as numbers or strings, or else references to other Perl objects. The `.Perl()` interface applies some heuristics to interpret arbitrary R objects, but these inevitably are imperfect; for trustworthy software, don't rely on them. Supply the argument as a list, each element of which is a single basic value or a reference to a proxy object, and ensure that each value has been coerced in R to a type that corresponds to what the Perl function expects as that element of the argument array.

Consider the Perl function `&quotewords` in the `Text::ParseWords` module. Given an array of strings representing lines of text, it returns the separate words in all the text, based on specified delimiters and taking account of quotes to group words together. This function appears to take three arguments, with Perl types as follows:

```
&quotewords($delim, $keep, @lines);
```

where `$delim` is a string with the delimiter as a regular expression, `$keep` is a flag saying whether to keep quotes, and `@lines` is an array of the text to process. Watch out, however: The third argument is an array, and not a reference to an array, and Perl flattens all the arrays in argument lists. So if the third argument is a vector of n strings, Perl in effect expects $n + 2$ scalar values as arguments. Don't give `.Perl()` three arguments, the third being a character vector of arbitrary length; that might generate an array reference. Instead, we need to pass a list with $n + 2$ "scalar" elements, as is done by the simple interface function in R:

```
quotewords <- function(x, delim = "\\s+")  {
    .PerlPackage("Text::ParseWords")
    .Perl("quotewords", .args = c(list(delim, 1), x))
}
```

The `.args` argument to `.Perl()` is interpreted as a list, each element of which will be one element in the argument to the Perl function. The call to `c()` concatenates a list of 2 elements with the character vector, which has the desired effect of converting the latter to a list of length-1 vectors.

12.4 Object-Based Interfaces

Objects are certainly involved in the functional interface described above, but there is no need for the user to consider objects in the other system that are not convertible to local classes of data. In the interface defined for C by `.C()`, there is no attempt to consider C types that have no analog as R objects. The interface to Perl provided by `.Perl()` is more general, in that objects can be returned that do not correspond to R objects and Perl functions can expect such objects or references to them as arguments.

The ability to refer in R to an object in another system, even when it does not correspond exactly to any R object, opens up many valuable techniques; after all, it's often precisely the ability to do computations outside the current system's tools that makes an inter-system interface valuable. We call the R reference a *proxy* object, standing in for an object reference in the other system. And the proxy object is indeed a reference; that is, it refers to some object in the other system that can be modified, in most cases, and such that the reference will then be to the modified object. In contrast, the functional model applying to most R computations deals with objects, not references, so that modifications are local to the function involved.

The form of the proxy object reference—how its contents are extracted or modified—depends on the other system. Section 12.6, page 440, discusses proxy objects for "object-oriented" systems and Section 12.7, page 446, for relational database management systems, two important special cases. In the rest of this section, we will look at object references for Perl, which is a simpler case in some respects and so can usefully introduce general points that will apply to the other systems as well.

The most basic issue with proxy objects is to arrange for them to be created and to persist for as long as they are needed. (And then, in some cases, to arrange that they will not persist after they are needed.) When you are computing in R, such questions don't usually require your attention in any detail. If you need some object, you assign it a name in the current context, inside a function that you are writing or interactively at the top

level. Assignments inside a function persist until the call to that function returns, then R will clean up at some point without your intervention.

Proxy object references, in contrast, are generated first in the interface. An object will be created in the other system and a reference to that object will be passed back as the proxy in R. Something will likely have happened in the other system so that the reference persists there, otherwise your proxy object would not be of much use. Then what? On the R side, you would assign the proxy like any object. However, can you count on the foreign object it refers to persisting and disappearing along with the proxy? In computing terminology, what is the *scope* of the object referred to? Many of the inter-system interface packages will maintain a table of such references (`RSPerl` and `rJava`, for example). In addition, packages may to varying degrees arrange to delete the referenced object and/or to modify the proxy when the reference is no longer valid. The `RSPerl` package arranges to zero-out a reference when the proxy object is saved, so that using the reference in a new session will just warn you about a zero reference, rather than giving an addressing error because the pointer referred to the old process.

In general, you should read the documentation for the particular interface package carefully and/or do some experiments to see when objects in the foreign system are saved and whether the user is expected to explicitly delete them. Fortunately, the really serious danger is that the object will be prematurely deleted, and this is less likely. Most interfaces use a mechanism such as a hash table or global environment to assign the object when passing back a reference. The reference should then persist through the life of the current session. Users of the interface may need to arrange for explicitly deleting objects no longer needed. The issue here is one of wasting memory, potentially serious for efficiency but at least not likely to destroy valuable information.

As an example, let's write an interface to some Perl routines for text data.

Example: Perlfunctions for counting chunks of text

In Section 8.5, page 316, we showed two Perl functions that took a reference to a Perl hash object and an array of strings. The hash contains counts of strings. The functions either added or subtracted to the appropriate count for each of the new strings. The intent is to maintain counts of patterns in text.

R functions `chunksAdd()` and `chunksDrop()` in package `SoDA` are interfaces to the Perl functions. A slightly simplified version of `chunksAdd()` is:

```
chunksAdd <- function(
     table = .PerlExpr("\\%{0};", .convert = FALSE),
     data = character(),
     convert = length(data) == 0) {
    if(!is(table, "PerlHashReference"))
      stop(
       "Argument table must be reference to a Perl hash object;",
        " got an object of class ", class(table))
    args <- c(list(table), as.list(as.character(data)))
   .Perl("chunks_add",  .args = args, convert = convert)
}
```

The function takes three arguments: `table` is the proxy for the hash; `data` is the new data; `convert` is a flag saying whether to return the reference or convert the hash (which `RSPerl` does by making a named vector of, in this case, the numeric counts). If `table` is omitted the function initializes it to an empty hash. Omitting the `data` argument, on the other hand, is the easy way to get the converted counts back, without modifying the hash—that's why the default for `convert` is `TRUE` when no data is being added.

The function assembles the proxy reference and the new data as a list, which when transmitted to Perl will give the suitable argument array for routine `chunks_add`. Initialization (of the table) and most error checking are done in R. That reflects our general preference for programming with R, including its facilities for interactive debugging. The Perl code can do error checking as well, and does, validating the individual data items. But as a general rule, giving the other system as clean and well-checked a set of arguments as possible is likely to save you time learning about debugging other systems.

See the code in package `SoDA` for more details.

12.5 Interfaces to OOP Languages

Throughout this book, the term "object-oriented programming" and its acronym OOP are reserved for the languages or systems with a programming model having the following features.

1. Objects are generated from class definitions. The data content of the objects is usually defined in terms of named slots which have a specified class or type. The terms *property* or *attribute* may be used instead of *slot* depending on the system.

2. Programming with these objects is exclusively, or at least largely, done by invoking methods on the objects.

3. The definitions of the methods come from the definition of the object's class, directly or through inheritance.

4. Although it's not a requirement, nearly all OOP objects are passed by reference, so that methods can alter the object.

The OOP programming model differs from the S language in all but the first point, even though S and some other functional languages support classes and methods. Method definitions in an OOP system are local to the class; there is no requirement that the same name for a method means the same thing for an unrelated class. In contrast, method definitions in R do not reside in a class definition; conceptually, they are associated with the generic function. Class definitions enter in determining method selection, directly or through inheritance. Programmers used to the OOP model are sometimes frustrated or confused that their programming does not transfer to R directly, but it cannot. The functional use of methods is more complicated but also more attuned to having meaningful functions, and can't be reduced to the OOP version.

Languages such as Java use the OOP model as essentially their only programming style. Other languages such as Perl, Python and C++ have added OOP programming to a functional or procedural language. Interfaces from R to these languages open up many new computations.

Method invocation usually has a different appearance from a function call, emphasizing that the method definition comes from the class definition. Usually, the expression for the object comes first, then some operator symbol, then the name of the method followed by a parenthesized list of additional arguments. The dot, `"."`, is a common choice for the operator symbol, used by Java and Python, among others. In these languages, a method named `print` defined for an object `x` might be called in the form:

```
x.print()
```

C++ and Perl (version 5) use the operator `"->"` instead of `"."` (but the proposed Perl6 uses `"."`).

Invoking methods in the OOP system

Interfaces from R to systems that support the OOP programming model must provide a mechanism to invoke methods. The requirements vary from one

system to another but are basically that one starts with a proxy reference in R to an object in the other system, along with the name of the method, and any additional arguments the method requires. For systems supporting both functions and OOP methods, some indication may be needed as to which is wanted. It is quite feasible to mimic the syntax of method invocation in R, but as this book is written most interfaces don't do so, but instead use their functional interface.

For example, in the `rJava` interface package, the function `.jcall()` invokes Java methods. The equivalent to the example above would be:

```
.jcall(x, method = "print")
```

Similarly, in the `RSPerl` package, the function `.Perl()` handles methods if given a proxy object via the `ref=` argument:

```
.Perl("print", ref = x)
```

In either case, the interface code will arrange to dispatch the appropriate method in the OOP system applied to the OOP object for which `x` is the proxy.

Finding the "appropriate" method is the job of the foreign system. In some systems (Java, for example) there will be metadata which determines the method corresponding to the method name and the class of the object. In other systems (current Perl, for example) an interpreter for the language will evaluate the equivalent method invocation. In any case, the R software does not select the method, as it would for an R generic function.

Constructors and class methods

The notation for invoking a method in an OOP language suggests that the method belongs to the object. In fact, nearly all OOP systems associate the methods with the *class* of the object, not with the instance (the individual object itself). Method dispatch uses the known class of `x` to select a method with a given name; the selection will be identical for all objects with the same class. The object for which the method is invoked plays two roles: it defines the method, but only through its class; and it is passed as an argument to that method. (For example, the discussion of "method invocation" in the *Programming Perl* book [25] describes the mechanism.)

If programming is only by invoking methods on objects, how are new objects from a class generated in the first place? Methods whose purpose is to generate new objects are usually called *constructors* in OOP languages. Some languages have a separate syntax for constructors (Java, for example),

but a more revealing version has constructors invoked as a method, but on the class itself, with no instance involved. Such methods are called *class methods* to emphasize that they are invoked by providing the name of the class in place of an object. So if `Chunks` is a class in Perl and `new()` is a class method that generates a new object from the class, that method is invoked on the class name, whereas an ordinary method, say `add()` is invoked on an object from the class (to be precise, in Perl, a reference to such an object). The following piece of Perl code creates an object and invokes one of its methods.

```
my $counter = Chunks->new();
$counter->add(data);
```

The class method is invoked on the literal `"Chunks"` whereas the instance method is invoked on the variable `"$counter"`. Class methods in Java are distinguished in a similar way, but by the use of declarations rather than through syntax. Constructors are the obvious example of a class method, and no ordinary class can get along without them. Other class methods can exist in most systems as well, and would be invoked in a similar way.

Functional interfaces from R, such as `.jcall()` or `.Perl()`, will expect a class method if the argument referring to an object is a character string (the name of a class), rather than a proxy object. Constructors are often supplied as a special case, with their own interface function (`.jnew()` and `.PerlNew()`, for example).

12.6 Interfaces to C++

The C++ language started as a preprocessor to C, and is still compiled into object code compatible with C. The close relation between the languages and the fact that R is itself based on an implementation in C simplify some aspects of interfacing to C++. Instead of calling a general interpreter for the other system or communicating with another process, the computations will use one of the C interface functions, `.C()` or `.Call()`. The interface code in C++ can be included in the standard `src` directory where C code would be kept, but in a file with a suffix such as `"cpp"` that identifies it as C++. We began the discussion of interfacing to C++, therefore, in the previous chapter, in Section 11.4, page 425. We continue it here because the computational model for C++ is similar to other OOP languages and because some extra steps are needed to write the interfacing C code.

As shown in section 11.4, the usual approach is to write some special code in your package that contains one C-callable routine for each computation

needed from C++. The mechanism is simple: Write the new C code in a C++ source file and enclose the definitions of the C routines in a declaration that says the external names should be interpreted as C, not C++:

```
extern "C" {

}
```

While using this mechanism, there is a range of possible strategies as to how much of the C++ structure to make available to the R user, from "None" to a mirror image of the C++ methods, and corresponding questions about the R objects that should be returned to the user.

As an example, and to make the general approach clearer, let's look at the CRAN package gbm written by Greg Ridgeway. This package provides an interface to some C++ code, mostly by Jerome Friedman, for "gradient boosting", a technique for fitting statistical models. For the statistical techniques, see Chapter 10 of *Elements of Statistical Learning* [15] and the overview documentation for the gbm package. All we need to keep in mind is that the techniques iteratively refine a statistical model using the general structure we've discussed in Section 6.9, page 218, including a formula and optionally an associated data frame. The user can fit a model with an expression of the form

```
gbm1 <- gbm(formula, data, ...)
```

The formula and data arguments are similar to linear regression models and the like; in addition, there are a number of arguments special to the boosting techniques. After fitting, the user has access to plotting and other general summaries, as well as specialized performance analysis for boosting. The function gbm.more() continues the iterative fitting of an existing model. The gbm package is valuable as a bridge between the specialized computations of the C++ software and the familiar ideas provided in R for dealing with statistical models.

Functions gbm() and gbm.more() both call a C routine that creates and manipulates a CGBM object from a C++ class, CGBM, representing the models. C++ methods exist to construct and initialize the objects, to iterate fitting and to provide utilities such as prediction. The interface to the C++ computations in the R function gbm.fit() uses the .Call() interface to call the C subroutine gbm:

```
gbm.obj <- .Call("gbm",
      Y = as.double(y),
       # ... and many more arguments ...
      PACKAGE = "gbm")
```

Here is a sketch of some important steps in the C routine gbm. Arguments, allocation of R objects and error checking have all been omitted, but what's left gives an idea of the essential steps, and helps illustrate alternative strategies.

```
extern "C" {

SEXP gbm (
   // The corresponding arguments
  ) {
    CGBM *pGBM,

   // initialize R's random number generator
    GetRNGstate();

    // initialize some things
    gbm_setup( .... );

    pGBM = new CGBM();
    pGBM->Initialize(  .... );
    pGBM->iterate( .... );
    gbm_transfer_to_R( .... );

    // dump random number generator seed
    PutRNGstate();

    delete pGBM;
    return rAns;
}

} // end extern "C"
```

The call to GetRNGState is a core R routine that initializes the random number generator in C to its current state (see Section 6.10, page 234); the gbm_setup call does other initialization. As with any estimation procedure using simulated random values, some extra steps would be needed to make the results reproducible; see Section 6.10, page 229.

The next lines of C++ code create and work with the object representing the model: the new expression creates the object, and the methods Initialize() and Iterate() do what their names imply. As usual in OOP computations, the object referred to by pGBM is modified to reflect the iterative fitting that has been applied. The routine gbm_transfer_to_R copies information from that C++ object into various components of the R ver-

sion of the model. The C++ directive `delete` removes the object now that information has been copied.

One point to note is that the single C routine `gbm` takes the key C++ object, `pGBM`, through its entire lifetime: initializing, iterating, extracting information to return to R, and finally deleting it. The R object representing the model does not contain any proxy to a C++ object.

Hiding the C++ structure from the user has the effect, in this package, of emphasizing the similarity to other software for models in R. Users new to the package will find much of the functionality familiar, with no need to adjust to a different programming model. Their ability to explore data with these techniques will be enhanced, and that is indeed the *Mission*.

For applications in which the user needs to control computations at the level of individual C++ methods, a different organization is needed.

C++ objects in R

Once we decide not to insulate R users from the C++ objects and methods, we need a way to represent such objects. The C++ object is handled by a pointer (a reference, to sound more elegant), which will not be manipulated at all in R. In the example sketched above, `pGBM` was a pointer to an object of C++ class `CGBM`. To handle such objects in R, data of the `"externalptr"` type is the natural choice. Objects of this basic type have a single pointer as their value, only set and used in C. A value is inserted in such an object by C code and left untouched in R functions. To create explicit access to the objects and methods requires only two basic programming techniques.

- An initializing routine returns the pointer to the object, in the value field of an `"externalptr"` object;

- To each C++ method to be called from R there corresponds a C-callable routine of a known name, designed to be invoked via a `.Call()` interface and taking as its arguments the `"externalptr"` object plus whatever other arguments the method requires.

There are other ways to do it, but these choices are simple and natural.

The R package will usually have one function for each of the routines implied by these steps, each function using the `.Call()` interface to call the corresponding routine. A minimal rearrangement of the `gbm` example above to expose the C++ structure in R would have four new routines, each wrapping a corresponding C++ method:

1. `gbm_new`, a constructor to create and initialize an object of the C++ `"CGBM"` class;

2. `gbm_iterate`, a routine to invoke the `Iterate` method on the object;

3. `gbm_results`, a routine to return in R form the information in the current object;

4. `gbm_delete`, a destructor to delete the object.

The invocation of methods in the `gbm` routine is now broken up into separate user-callable pieces. Each of the four routines has its value and all arguments declared as pointers to R objects, conforming to the requirements for any C software to be called from the `.Call()` interface (see Section 11.3, page 422, for an example).

```
extern "C" {

SEXP gbm_new ( SEXP ext, ....
   ) {
    GetRNGstate();

    // initialize some things
    gbm_setup( .... );

    CGBM *pGBM = new CGBM();
    pGBM->Initialize(  .... );

    R_SetExternalPtrAddr(ext, (void *)pGBM);
    return ext;

}

SEXP gbm_iterate ( SEXP ext)
{
    CGBM *pGBM = (CGBM *) R_ExternalPtrAddr(ext);
    pGBM->iterate( .... );
    return ext;
}

SEXP gbm_results(SEXP ext)
{
    CGBM *pGBM = (CGBM *) R_ExternalPtrAddr(ext);
    gbm_transfer_to_R( .... );
   // construct list as in routine gbm
```

```
    return(rAns);
}

SEXP gbm_delete(SEXP ext)
{
    CGBM *pGBM = (CGBM *) R_ExternalPtrAddr(ext);
    delete pGBM;
    return ext;

}

} // end extern "C"
```

Each of the four C routines takes as an argument a pointer to an R object of class `"externalptr"` and returns the same object. The constructor, `gbm_new` fills in the pointer value with the newly allocated and initialized object; all the other routines extract the corresponding pointer and operate on the C++ object. (The two routines `R_ExternalPtrAddr` and `R_SetExternalPtrAddr` are R utilities that extract and set the pointer contained in an `"externalptr"` object.) Everything else in the example, including the code we haven't shown in this sketch, essentially rearranges the same computations done before, but now the programming model is that computations in R will control the sequence of creating, iterating, extracting and deleting.

The R software to complete the interface can be as simple as one function for each C routine, doing little more than using `.Call()` for the corresponding routine. If there are additional arguments to the C++ method, however, these need to be coerced to the correct datatype, either in the R function (usually the best place) or in the C routine. The construction and initialization of the `CGBM` object, for example, takes a number of inputs that would be arguments to `gbm_new` and to the corresponding R function. Here's a sketch of a fairly minimal version.

```
gbmNew <- function( x, y,
    .... (Many more arguments)) {
    .Call("gbm_new",
        new("externalptr"),
        as.double(x), as.double(y),
        .... )
}
```

The arguments all need to be carefully coerced to a specific basic datatype since the C routine `gbm_new` just passes the arguments on without checking them. The first argument is the `"externalptr"` object into which the C++

pointer will be inserted, and the object then returned, to be supplied in future calls to the other routines in the interface to the C++ class.

We now have a working interface to the C++ software, but usually one more layer is desirable: an R class for the objects. C++ proxy objects don't go through a single interface function, in contrast to the case for Java or other external systems; because the interface can use the standard C interface functions, no metadata is provided automatically to identify the C++ class corresponding to the object. The responsibility falls to the programmer, and for many reasons the extra effort is worth taking. In this example, a class corresponding to the C++ class would have a slot for the `"externalptr"` proxy, plus whatever additional slots are needed to complete the definition of the model (including states for the random number generator, in order to make the computations reproducible, as discussed in Section 6.10, page 229). Note that `"externalptr"` objects do not follow the standard R model to be duplicated when needed, so that the new class can not extend `"externalptr"`.

For extensive C++ software it would be better to create the mappings to C and to R automatically. Why take a chance on human error in reading the C++ definitions? As this book is written, we aren't quite able to hand over the job, but some promising work has been done, based on data available from the gcc compiler; see, for example, the `RGCCTranslationUnit` package by Duncan Temple Lang at the omegahat Web site. Check out the current status if such automation would be helpful in your application.

12.7 Interfaces to Relational Database Systems and to Spreadsheets

Database and spreadsheet programs share typical roles and data models in their relation to data analysis, even though they differ from each other in form. The typical role is as a data repository: These are the systems where the data often resides, where the underlying process keeps information. We need to interface to these repositories to have direct access to the data.

The data model suitable for both kinds of programs is the general *data frame* model discussed many times in the book; that is, the notion of some defined observable variables, for each of which values will be recorded for a range of observations. Spreadsheets and relational databases essentially visualize data frames as tables, with columns for variables and rows for observations (not that either the creators or the users of these systems would necessarily think of their data in terms of variables and observations). It's natural then that interfaces to these systems should relate tables to data

frames, both in the general sense of this book and in the narrower sense of the "data.frame" class of objects.

The simplest interface from database and spreadsheet programs to R is to create files from the other system that can be read as data frames in R, in other words a text-based interface. There are a number of possible file formats, but the most widely available and convenient to use are the *comma-separated-values files* and the *tab-delimited files*. These are both standard file formats, which can be read into and exported from nearly any spreadsheet program and many database systems. Section 6.5, page 169, showed how to read such files into R, how to import and export the files in spreadsheet programs (page 173), and how to create tables from database systems (page 178).

For spreadsheet programs, this form of interface is the way to start, so long as a text-based interface is suitable for your application (mainly, that you can live with getting a copy of the non-R data and that rapid, dynamic change in the data is unlikely). Follow the discussion in Chapter 6. For database programs, such files may still be a reasonable option. Importing data from a ".csv" file is usually straightforward, but exporting a table to one may not be as simple, depending on the particular program. If your database setup does support easy export, you can follow the same route. (For example, MySQL supports ".csv" files as one of its engines; if that option is suitable to your application, it could provide an excellent interface.)

For spreadsheet programs, and in particular for Excel, some more specialized options may be available. On the Windows platform, the R-DCOM interface provides a very flexible and potentially very sophisticated communication mechanism based on the notion of components and services. On a non-Windows platform, the practical interface is to use the data export/import features in Excel.

Interfaces to database systems

For most database systems, interface packages allow flexible access with less human intervention than required to export tables explicitly. These interfaces support a functional or object-based view. Whole tables can be accessed straightforwardly. Portions of tables can be accessed using the standard query language, SQL, as supported by all major database systems. SQL was introduced briefly in Section 6.5, page 178. If you are willing to program in SQL, a functional interface is available for very general queries. Access can be functional (returning the result as a data frame) or object-based, using proxy objects as the basis for further queries or incremental

access. An additional advantage is that the related R packages provide a uniform programming interface from R to the major database systems.

The packages use a standardization provided by the DBI package of David James. This package defines a uniform interface for the R programmer, via generic functions and virtual classes. Functions in the DBI package define access to databases, tables, SQL queries, and other computations, expressed in an essentially identical form regardless of the actual database system. The specific database interface package implements methods and defines subclasses to realize the programming interface for a particular database system. If the specific interface is "DBI-compliant", software can be written once and used on any of the database systems. Compliant interfaces exist for Oracle (ROracle), MySQL (RMySQL), and SQLite (RSQLite).

The key concept is the mapping between a *table* in the database system and a data frame in R. Related tables are organized into a *database*, as files are organized in directories (in some database systems this is actually the implementation). The DBI package reflects this organization, top-down from choosing a database system, to specifying a database, to techniques that manipulate individual tables.

The database system corresponds to a *driver*, created and kept through a session. If we're using the SQLite system:

```
drv <- dbDriver("SQLite")
```

The driver is now used to open a *connection* to a particular database in this system. Depending on the system, you may need to supply user and password information, as well as the name of the database. SQLite just needs the name of the database:

```
conn <- dbConnect(drv, "myDatabase")
```

It's relevant that we call these database connections; they do act much like the R connections in Section 5.5, page 131. The difference is that data transfer uses the facilities of the database system here rather than low-level input and output. The units of data are tables. So, if "MarsData" has been established as a table in "myDatabase" to hold our example of the Mars declination data, then reading the whole table is just a call to dbReadTable():

```
> mars <- dbReadTable(conn, "MarsData")
> dim(mars)
[1] 923  21
```

Similarly, functions dbWriteTable(), dbExistsTable(), and dbRemoveTable() perform the operations their names suggest. The concepts here are again

closely tied to the ideas of the S language, and in particular to the basic computations on environments as databases in Section 5.3, page 124.

Database tables may be very large, so that suitable access must be to a selected portion of the table. Also, a database may contain related tables and a selection may combine information from more than one of them. This is the stuff of classical relational database computation and the SQL language. Queries more complex than transferring whole tables will need to be expressed in SQL, but can be transmitted via the `dbSendQuery()` function. This takes as arguments a connection and a string containing the SQL query. If `myQuery` is an object containing such a query, the result of the query is obtained as:

```
res <- dbSendQuery(conn, myQuery)
```

We introduced SQL in Section 6.5, page 178, but for learning how to write queries, I'm afraid you will have to look up some books or other references on SQL itself. Whatever the actual query, the result is conceptually another table-like data object made up from information in one or more tables in the database. Query results are not data frames, however. In the terminology of this chapter they are proxy objects, standing in for an object in the database.

The only thing you can do in general with the result is to call function `fetch()` to fetch a specified number of "records" from the result. A record is a row of the implied table and the result of the call is a data frame with that many rows and with whatever columns were defined by the query.

The fetched results can be processed anyway you like. The paradigm is to check for the end of data by calling `dbHasCompleted()`, and then to fetch as much data as you want to handle at one time. If you wanted to just create the entire data frame:

```
> out <- NULL
> while(!dbHasCompleted(res))
+   out <- rbind(out, fetch(res, n))
```

However, if you really wanted to do this, just call `dbGetQuery()` instead of `dbSendQuery()`. The direct use of `fetch()` is usually to do some computations that don't require the entire data frame.

The concepts and terminology derive from the old days when records really were records, perhaps on a magnetic tape. As a consequence, bare SQL in this form does not support general manipulations of results, although database systems often do support various extensions. The simple version is adequate for many applications, and does scale well to very large datasets.

Once you have mastered enough SQL for your applications, the DBI-based interface should be straightforward to use.

As with regular connections, but sometimes even more, it may be essential to close down your connection to the database when computation is finished:

```
> dbClearResult(res)
> dbDisconnect(conn)
> dbUnloadDriver(drv)
```

12.8 Interfaces without R

Other users, not just those of us doing data analysis, can benefit from intersystem interfaces; as a result, many systems offer convenient access to other software. Keep these in mind for applications where there is no need to bring data into R from another system just to pass it on to a third. Using an interface between the other two systems can simplify programming and save computing time.

For example, many systems will have interfaces to both spreadsheet software and relational databases, for the same reasons such interfaces are useful in R: that's often where data resides. This gives us some more choices whenever data in a database or spreadsheet is to be used eventually in some other non-R computations: Either access the data indirectly through R or write some code in Perl to access the data directly, in addition to the text manipulation software.

As an example consider applying some text manipulation in Perl to data residing in a relational database. What are some tradeoffs to guide the choices?

- If the original data from the database is not needed in R for other purposes, there will be some computational efficiency to direct access, particularly if the programming style of access is made more natural for the other system (see the example below). My usual caution about "efficiency" applies: Does it matter in this case?

- The additional programming effort required for direct access will vary. If R creates a data frame from the database and then presents part of it in a different form, such as a single variable, the existing Perl code will not likely be directly usable given Perl's approach to a database interface.

The best solution in different applications can vary from ignoring R for database access (when only the results of the Perl-processed data are needed) to using only R for database access (when the data needed for Perl is more naturally supplied as columns of the data frame formed in R).

To see the different styles of database access, we can compare typical use of the DBI package in R and the DBI module in Perl. The two share a name and a basic design: to act as an interface to relational database software, including the SQL query language, with programming that is independent of the specific database system. They also take a similar approach to the initial programming required. The user establishes a connection to a particular database. In the OOP form of Perl, one invokes the connect() method of the module:

```
my $dbcon = DBI->connect('DBI:MySQL:myData');
```

As with the function dbConnect() in the DBI package, this returns a connection that can then be used for all queries on this database in this session.

The next step in both systems is to obtain a result set, the database's version of the result of executing a query. Using the DBI package in R, we call dbSendQuery(). The actual SQL is essentially the same for access from Perl, but the standard approach includes an intermediate step to *prepare* the query. The prepared but unexecuted query is returned as an object.

```
my $query = $dbcon->prepare($someQueryString);
```

Preparing is essentially compiling the SQL query; one can leave parameters (typically to be filled in by names) unspecified. The execute() method of the query then creates the result set; if there are parameters in the prepared query, these are supplied as arguments to execute(), as in:

```
$query->execute($thisName);
```

We are now at the same state that the interface from R would be after evaluating a call to dbSendQuery(), with two minor differences. The standard DBI interface in R does not include preparing queries, although specific database interface packages may do so; also, the Perl execution of the query does not return a result set as an object, but instead modifies the $query object to be ready for fetching data.

In both interfaces, the actual data transfer takes place by fetching rows from the result set. As usual, the Perl method is oriented to iteration: The method fetchrow_array() always fetches a single row, which is returned as a Perl array. The elements of the array are the (scalar) values for each variable

in the result set, for the next available row. The R function `fetch()` fetches an arbitrary number of rows, as a data frame.

It's the one-row-at-a-time nature of the Perl fetch that suggests organizing the Perl computation differently when accessing data from a database rather than from R. In the first case, typical Perl style would do all the immediately relevant computations on all the variables, incrementally one row at a time, rather than collecting one or more variables as separate arrays. The second approach is feasible, but then it may be simpler just to collect the data in R.