However, the general style to be followed throughout has been illustrated by the examples:

- The structural form of models is defined by simple, general formulas.
- Many kinds of data for use in model-fitting can be organized by data frames and related classes of objects.
- Different kinds of models can be fitted by similar calls, typically specifying the formula and data.
- The objects containing the fits can then be used by generic functions for printing, summaries, plotting, and other computations, including fitting updated models.
- The computations are designed to be very flexible, and users are encouraged to adapt our software to their own needs and interests.

In presenting our appetizer, we did not emphasize the last point heavily, but it is central to the philosophy behind this book. Even though a large number of functions and methods are presented, we intend these to be a starting point for the computations *you* want, rather than some rigid prescription of how to use statistical models.

Statistical Models in S
Ed Chambers and Hastie

Chapter 2

Statistical Models

John M. Chambers
Trevor J. Hastie

This is a book about statistical models — how to think about them, specify them, fit them, and analyze them. Statistical models are simplified descriptions of data, usually constructed from some mathematically or numerically defined relationships. Modern data analysis provides an extremely rich choice of modeling techniques; later chapters will introduce many of these, along with S functions and classes of S objects to implement them. All these techniques benefit from some general ideas about data and models that allow us to express what data should be used in the model and what relationships the model postulates among the data. You should read this chapter (at least the first two sections) for a general notion of how models are represented. You can do this either before you start to work with specific kinds of models or after you have experimented a little. Getting some hands-on experience first is probably a good idea—for example, by looking at the first two sections of Chapter 4 on linear models, or by experimenting with whatever kind of model interests you most.

The first two sections of this chapter introduce our way of representing models, and are likely to be all you need for direct use of the software in later chapters. When and if you come to modify our software to suit your own ideas, as we hope many users will do, then you should eventually read further into Sections 2.3 and 2.4.

Throughout the book, we will be expressing statistical models in three parts:

- a *formula* that defines the structural part of the model—that is, what data are being modeled and by what other data, in what form;

- *data* to which the modeling should be applied;
- the stochastic part of the model—that is, how to regard the discrepancy or residuals between the data and the fit.

This chapter and the next concentrate on the first two of these. They discuss how formulas are represented, what objects hold the data, and how the two are brought together. The rest of the book then brings together the three parts in the context of different kinds of models.

Formulas are S expressions that state the structural form of a model in terms of the variables involved. For example, the formula

```
Fuel ~ Power + Weight
```

reads "`Fuel` is modeled as `Power` plus `Weight`." More precisely, it tells us that the response, `Fuel`, is to be represented by an additive model in the two predictors, `Power` and `Weight`. There is no information about what method should be used to fit the model. Formulas of this general style are capable of representing a very wide range of structural model information; for example,

```
100/Mileage ~ poly(Weight, 3) + sqrt(Power)
```

says to fit the derived variable `100/Mileage` to a third-order polynomial in `Weight` plus the square-root of the `Power` variable. Transformations are used directly in the formula, and the basis for the polynomial regression in `Weight` is generated automatically from the formula. Here is a formula to fit separate B-spline regression curves within the two levels of `Power` obtained by cutting `Power` at its midrange:

```
Fuel ~ cut(Power, 2) / bs(Weight,df=5)
```

In the next example, nonparametric smooth curves will be used to model the transformed `Fuel` additively in `Weight` and `Power`, using 5 degrees of freedom for each term:

```
sqrt(Fuel-min(Fuel)) ~ s(Weight, df=5) + s(Power, df=5)
```

The details of these formulas will be explained later in the chapter.

The models above imply the presence of some data on `Fuel`, `Power` and `Weight`; in fact, reasonable models are inspired by data, since models without data are hard to think about. These data actually do exist, and form part of a large collection of data on automobiles described in Chapter 3 and used throughout the book; the present model relates fuel consumption to two vehicle characteristics. Part of the model-building process is collecting and organizing the relevant dataset, and looking at it in many different ways. Some of the useful views are simple, such as summaries and plots. The next chapter is about tools for organizing data into objects that are convenient both for studying the data directly and as input for more sophisticated procedures. For the moment we assume that such data organization has already taken place, and that all the variables referred to in formulas are available.

2.1 Thinking about Models

Models are objects that imitate the properties of other, "real" objects, but in a simpler or more convenient form. We make inferences from the models and apply them to the real objects, for which the same inferences would be impossible or inconvenient. The differences between model and reality, the *residuals*, often are the key to reaching for a deeper understanding and perhaps a better model.

2.1.1 Models and Data

A road map models part of the earth's surface, attempting to imitate the relative position of towns, roads, and other features. We use the map to make inferences about where real features are and how to get to them. Architects use both paper drawings and small-scale physical models to imitate the properties of a building. The appearance and some of the practical characteristics of the actual building can be inferred from the models. Chemists use "wire frame" models of molecules (by either constructing them or displaying them through computer graphics) to imitate theoretical properties of the molecules that, in turn, can be used to predict the behavior of the real objects.

A good model reproduces as accurately as possible the relevant properties of the real object, while being convenient to use. Good road maps draw roads in the correct geographical position, in a representation that suggests to the driver the important curves and intersections. Good road maps must also be easy to read. Any good model must facilitate both *accurate* and *convenient* inferences. A large diorama or physical model of a town could provide more information than a road map, and more accurate information, but since it can be used only by traveling to the site of the model, its practical value is limited. The *cost* of creating or using the model also limits us in some cases, as this example illustrates: building dioramas corresponding to every desirable road map is unlikely to be practical. Finally, a model may be attractive because of *aesthetic* features — because it is in some sense beautiful to its users. Aesthetic appeal may make a model attractive beyond its accuracy and convenience (although these often go along with aesthetic appeal).

Statistical models allow inferences to be made about an object, or activity, or process, by modeling some associated observable data. A model that represents gasoline mileage as a linear function of the weight and engine displacement of various automobiles,

```
Mileage ~ Weight + Disp.
```

is directly modeling some observed data on these three variables. Indirectly, though, it represents our attempt to understand better the physical process of fuel consumption. The accuracy of the model will be measured in terms of its ability to imitate the data, but the relevant accuracy is actually that of inferences made about the

real object or process. In most applications the goal is also to use the model to understand or predict beyond the context of the current data. (For these reasons, useful statistical modeling cannot be separated from questions of the design of the experiment, survey, or other data-collection activity that produces the data.) The test data we have on fuel consumption do not cover all the automobiles of interest; perhaps we can use the model to predict mileage for other automobiles.

The convenience of statistical models depends, of course, on the application and on the kinds of inference the users need to make. Generally applied criteria include simplicity; for example, a model is simpler if it requires fewer parameters or explanatory variables. A model that used many variables in addition to weight and displacement would have to pay us back with substantially more accurate predictions, especially if the additional variables were harder to measure.

Less quantifiable but extremely important is that the model should correspond as well as possible to concepts or theories that the user has about the real object, such as physical theories that the user may expect to be applicable to some observed process. Instead of modeling mileage, we could model its inverse, say the fuel consumption in gallons per 100 miles driven:

```
100/Mileage ~ Weight + Disp.
```

This may or may not be a better fit to the data, but most people who have studied physics are likely to feel that fuel consumption is more natural than mileage as a variable to relate linearly to weight.

2.1.2 Creating Statistical Models

Statistical modeling is a multistage process that involves (often repeated use of) the following steps:

- obtaining data suitable for representing the process to be modeled;
- choosing a candidate model that, for the moment, will be used to describe some relation in the data;
- fitting the model, usually by estimating some parameters;
- summarizing the model to see what it says about the data;
- using diagnostics to see in what relevant ways the model *fails* to fit as well as it should.

The summaries and diagnostics can involve tables, verbal descriptions, and graphical displays. These may suggest that the model fails to predict all the relevant properties of the data, or we may want to consider a simpler model that may be

nearly as good a fit. In either case, the model will be modified, and the fitting and analysis carried out again.

If we started out with a model for mileage as a linear function of weight and displacement, we would then want to look at some diagnostics to examine how well the model worked. The left panel of Figure 2.1 shows `Mileage` plotted against the values predicted by the model. The model is not doing very well for cars with high

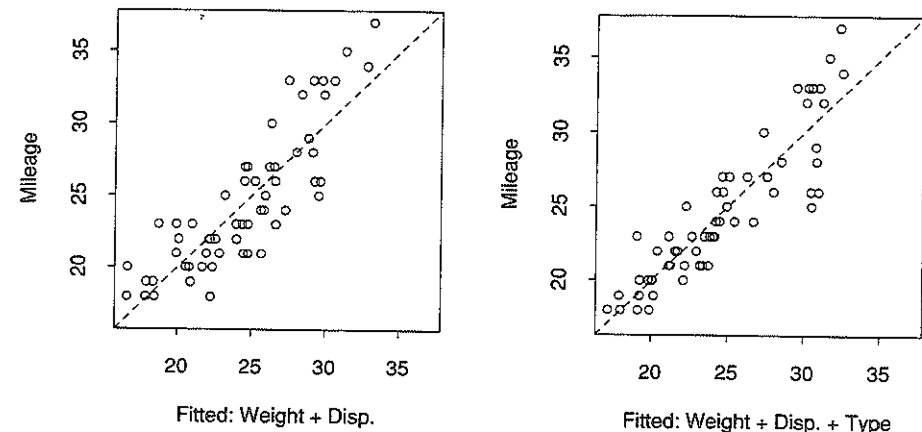

Figure 2.1: *Mileage for 60 automobiles plotted against the values predicted by a linear model in weight and displacement (the left panel) or weight, displacement and type of automobile (the right panel).*

mileage: they all fall above the line. The change to 100/`Mileage` helps some (there is a plot on page 104). If we add in a coefficient for each type of car (compact, large, sporty, van, etc.) the fit improves further. In practice, we would continue to study diagnostics and try alternative models, seeking a better understanding of the underlying process. This model is our most commonly used simple example, and will recur many times, to introduce various techniques.

Research in statistics has led to a wide range of possible models. Later chapters in this book deal with specific classes of models: traditional models such as linear regression; recent innovations, such as models involving nonparametric smooth curves or tree structures; important specializations such as models for designed experiments, and general computational techniques such as minimization, which can be used to fit models not belonging to any of the standard classes. This rich choice of possible models is of real benefit in analyzing data. Whenever we can specify

a model that is close to our intuitive understanding or is able to respond to some observed failure of a standard model, chances are we will more easily discover what is really going on. A limited computational or statistical framework that requires us to distort or approximate the model we would like to fit makes such discovery more difficult. It can also hide from us some important information about the data. The methods presented in this book and the functions that implement the methods are designed to give the widest possible scope in creating and examining statistical models.

Of course, all this rich variety will only be helpful if we can use it easily enough. We must be able to carry out the steps in specifying the models without too much effort on our part. The fitting must be accurate and efficient enough to be used in practical problems. There must be appropriate summaries and diagnostics so that we can assess the adequacy of the models. In later chapters, each of these questions will be considered for the various classes of models.

Fortunately, many different classes of models share a substantial common structure. The steps we listed above apply to many models, and important summaries and diagnostics can be shared directly or, at worst, adapted straightforwardly from one class of models to another. The organization of the S computations for the various classes of models is designed to take advantage of this common structure.

This chapter describes a way to express the structural formula for the model. What about the data? For the moment we can assume the data are around in our global environment, and simply refer to variables by name. In Chapter 3 we describe *data frames*, a more systematic way of organizing and providing the data for a model. Depending on the class of models, formulas and data frames may be all we need to specify; for example, if we are using linear least-squares fitting, there is not much more to say in step 3. Other kinds of models may require some further specifications; generalized linear models, for example, require choosing link and variance functions. The choice of the kind of model and the provision of these additional specifications fix the stochastic part of the model to be fit.

2.2 Model Formulas in S

The modeling formula defines the structural form of the model, and is used by the model-fitting functions to carry out the actual fitting. Most readers will already be familiar with conventional modeling formulas, such as those used in textbooks or research papers to describe statistical models, as in (2.1) below. The formulas used in this book have evolved from mathematical formulas as a simpler and in some ways more flexible approach to be used when computing with models.

A formula in S is a symbolic expression. For example,

```
Fuel ~ Weight + Disp.
```


just stands for the structural part of a model. If you evaluate the formula, you will just get the formula. In particular, use of a formula such as the one above does not depend on the values of the named variables; indeed, the variables need not even exist! The expression to the left of the "~" is the *response*, sometimes called the *dependent* variable. In this case the response is simply the name `Fuel`. The right side is the expression used to fit the response, made up in an additive model of *terms* separated by "+". The variables appearing in the terms are called the `predictors`. Experienced S users are by now probably very curious, so this comment is for their benefit: `"~"()` is an S function that does nothing but save the formula as an unevaluated S expression, a *formula* object.

The formula above expresses most of the ingredients of a statistical model of the form

$$Fuel = \alpha + Weight\beta_1 + Disp.\beta_2 + \varepsilon \tag{2.1}$$

For most of the models in this book, the formula does not specifically refer to the parameters β_j in the linear model. These can be inferred and so we save typing them. In a sense, we also avoid mental clutter, in that the names of the parameters are not relevant to the model itself. When we come to general nonlinear models in Chapter 10, however, the formula will have to be completely explicit, since it is no longer additive.

The formula makes no reference to the errors ε either. These, of course, are the stochastic part of the model specification. When formulas are used in a call, say to the linear regression model-fitting function

```
lm(Fuel ~ Weight + Disp.)
```

we complete the rest of the modeling specification; `lm()` assumes the mean of `Fuel` is being modeled by the linear predictor, and uses least squares to compute the fit. Expressions such as the one above were encountered in Chapter 1; in fact all the model-fitting functions take a formula as their first argument, and in most cases the same formula can be used interchangeably among them (hopefully with different consequences!).

The formula above is equivalent to

```
Fuel ~ 1 + Weight + Disp.
```

where the `1` indicates that an intercept α is present in the model. Since we usually want an intercept, it is included by default; on the other hand, we can explicitly exclude an intercept by using `-1` in the formula

```
Fuel ~ -1 + Weight + Disp.
```

In using formulas it is important to keep in mind that we are writing a shorthand for the complete model expression. In particular, there is no operation going on that adds `Weight` and `Disp.`; the operator "+" is being used in a special sense, to

separate items in a list of terms to be included in the model. The formula expression is, in fact, used to generate such a list, from which the terms and the order in which they appear in the model will be inferred. This inference poses no problem for most models, but with complicated formulas, some care may be needed to understand the model implied. The remainder of this section gives enough information for most uses of model formulas; Section 2.3.1 provides a complete description.

2.2.1 Data of Different Types in Formulas

The terms in a formula are not restricted to names: they can be any S expression that, when evaluated, can be interpreted as a variable. For example, if we wanted to model the logarithm of `Fuel` rather than `Fuel` itself, we could simply use that transformation in the formula

```
log(Fuel) ~  Weight + Disp.
```

A variable may be a *factor*, rather than numeric. A factor is an object that represents values from some specified set of possible *levels*. For example, a factor `Sex` might represent one of two values, `"Male"` or `"Female"`. Readers familiar with S might wonder what happened to the *category*, which is also an object with levels. Factors have all the features of categories, with some added class distinctions; in particular there is a distinction between *factors* and *ordered factors*. Factors can be created in a number of ways, as will be discussed in Section 3.2. For the moment the distinction between factors and categories is not important, and we will simply refer to them as factors.

Factors enter the formula in the same way as numeric variables, but the interpretation of the corresponding term in the model is different. In a linear model, one fits a *set* of coefficients corresponding to a factor. Consider the model

```
Salary ~  Age + Sex
```

where `Salary` and `Age` are numeric vectors and `Sex` is a two-level factor. This is now shorthand for a model of the form

$$Salary_i = \mu + Age_i\beta + \begin{cases} \alpha_M & \text{if } Sex_i \text{ is } Male \\ \alpha_F & \text{if } Sex_i \text{ is } Female \end{cases} + \varepsilon_i \tag{2.2}$$

where α_F and α_M are two parameters representing the two levels of `Sex`. The coding of factors proceeds from observing that this model is equivalent to one in which the factor is replaced by one "dummy" variable for each level—namely, a numeric variable taking value 1 wherever the factor takes on that level, and 0 for all other observations. In this case, for example, suppose `XMale` is a dummy variable set to 1 for all `Male` observations and `XFemale` is set to 1 for all `Female` observations. The original model is then equivalent to


```
Salary ~ Age + XMale +XFemale
```

Often in models such as this not all of the coefficients can be determined numerically; for example, in (2.2) we could replace μ by $\mu+\delta$, and then compensate by replacing α_F and α_M by $\alpha_F - \delta$ and $\alpha_M - \delta$. Numerically such indeterminacies can be detected by collinearities in the variables used to represent the terms (`Xmale` and `Xfemale` add to a vector of ones, which is also used to represent the constant μ), and will be handled automatically during the model-fitting. Occasionally, you may want to control the parametrization of a term explicitly; Section 2.3.2 will show how.

Other non-numeric variables enter into the models by being interpreted as factors. A logical variable is a factor with levels `"TRUE"` and `"FALSE"`. A character vector is interpreted as a factor with levels equal to the set of distinct character strings. A category object in S will be treated as a factor in the modeling software. Section 3.2.1 deals with these issues in more detail.

A term in a formula can also refer to a matrix. Each of the variables represented by the columns of the matrix will appear linearly in the model with its own coefficient. However, the entire matrix is interpreted as a single term.

To sum up so far, the following S data types can appear as a term in a formula:

1. a numeric vector, implying a single coefficient;
2. a factor or ordered factor, implying one coefficient for each level;
3. a matrix, implying a coefficient for each column.

Transformations increase the flexibility greatly, since the final element in this list is

4. any S expression that evaluates to a variable corresponding to one of the three types above.

To appreciate this last item, consider these examples of valid expressions that can appear as terms within a formula:

- `(Age > 40)`, which evaluates to a logical variable;
- `cut(Age,3)`, which evaluates to a three-level category;
- `poly(Age,3)`, which evaluates to a three-column matrix of orthogonal polynomials in `Age`.

The classical computational model for regression is an X matrix and a coefficient vector β. The rich syntax of our modeling language allows us instead to think of each of the terms as an entity, even though they eventually will be expanded into one or more columns of a model matrix X in most of the models discussed. But the formulas and the modeling language put no restrictions on the form of a term

or on the interpretation given to the term by a particular model-fitting function. The contribution of a term to the fit can often be thought of as a function of the underlying predictor; factors produce step functions, and terms based on functions like `poly()` produce smooth functions. See Section 2.3.1 and Chapter 6. For other models, like local regression and tree-based models, the contribution of the terms is interpreted differently. In particular, the contribution of a term to a tree-based model is invariant under monotone transformations of the variable.

2.2.2 Interactions

Terms representing the *interaction* of two or more variables lead to further shorthand in formulas. We may suspect that the effect of a variable in a model will be different depending on the level of some factor variable. In this case we need to fit an additional term in the model.

As an example, we consider some factors that describe the solder experiment in Chapter 1(these data are described in more detail in Chapter 3 and used throughout the book). `Opening` and `Mask` are two factors in the experiment, having three and five levels respectively. To allow for interactions, we will fit a term for each of the individual factors and in addition a coefficient for each level of the interaction—that is, for each combination of levels for the two factors. This is expressed in the formula language as

```
Opening + Mask + Opening:Mask
```

which implies fitting coefficients for the 3 levels of `Opening`, the 5 levels of `Mask`, and the 15 levels of their combination. The idea behind this separation into *main effects* and *interaction effects* is that for simplicity, we would prefer the interactions to be absent; by fitting them separately, we can examine the additional contribution of the interaction terms. (Once again, not all these coefficients can be determined independently.)

Rather than writing out the three terms, we allow a special use of the "*" operator in formulas to imply the inclusion of the two terms that are operands of the "*" *and* of their interaction. Thus

```
Opening * Mask
```

is equivalent to the previous expression.

When one of the variables is numeric, the interaction notation is still recognized, but it reduces to fitting coefficients for the factor variable and separate coefficients for the numeric variable within each level of the factor (see Section 2.3.1 for details).

Interactions may be defined between more than two variables; for example,

```
Opening * Mask * Samt
```


is interpreted to produce terms for each of the individual variables, for each of the two-way combinations, and for the three-way combination (that is, a coefficient for each of the $3 \times 5 \times 2$ levels of the factor defined jointly by all three variables). Another form of interaction is known as *nesting*, which we discuss in Section 2.3.1.

The full repertoire of special operators in formulas is discussed in Section 2.3.1. The same section discusses how formulas are interpreted, which may be relevant if your application is very specialized. Try, however, to build up formulas in as simple and unambiguous a way as possible.

2.2.3 Combining Data and Formula

The data and the formula for a model come together when we actually fit a particular model—e.g., when we estimate coefficients. The model-fitting functions will generate an appropriate internal form for the data in preparation for the fitting. For linear models and most of their extensions, this form is the model matrix or X matrix, in which one or more columns correspond to each of the terms in the model. Experienced modelers may have imagined the construction of this model matrix while reading the previous section, a tedious task traditionally regarded as part of the "art" of regression. The function `model.matrix()` does just this; in its simplest form it takes a single `formula` argument (with or without a response) and produces a matrix. Try it on a simple formula and see what happens! While it might be comforting for you to read Section 2.3–4 to see how we construct the ingredients of this matrix, such detailed knowledge is not necessary for standard use of the techniques we present in later chapters.

Nonstandard situations that may make model matrices of more interest include the handling of very large problems, where the size of the model matrix may force the use of special techniques, and various kinds of updating, subsampling, and iterative computations using some of the observations in the data. In these computations, practical considerations may require working directly with the model matrix.

The columns of model matrices contain coded versions of the factors and interactions in the model. The particular choice of coding will be of concern only if you want to interpret particular coefficients; Section 2.3.2 discusses how to control the coding. Section 2.4.3 contains further discussion of model matrix objects. That section is intended for those who need or want to know how the computations actually take place. In particular, to develop a *new* approach to fitting models, not covered by any of the chapters of the book, you would need to understand something about the steps that go into creating a model matrix.

2.3 More on Models

The third section of each chapter in this book expands on the S functions and objects provided in the chapter. Here we discuss the options and extended versions available that add new capabilities to the basic ideas. The material in section 3 should usually be looked at *after* you have tried out the essentials presented in section 2 on a few examples. Experience shows that, after trying out the ideas for a while, you will have a better feeling for how to make use of the functions, and will begin to think "This would be a bit better if only ...." Section 3 is intended to handle most of the "if-only." When the extra feature needed is not here, and there is no obvious way to create it by writing some function yourself, the next step is to look at section 4, which reveals how it all works. There you can learn what would be involved in modifying the basics. However, you should not take that step before thoroughly understanding what can be done more directly.

2.3.1 Formulas in Detail

In Section 2.2 we introduced model formulas and gave some examples of typical S expressions that can be used to give a compact description of the structural form of a model. Simple model-fitting situations can often be handled by the simple formulas shown there, but the full scope of model formulas allows much more detailed control. In this section, we give the full syntax available and explain how it is interpreted to generate the terms in the resulting model. Unless otherwise stated, we will always be talking about linear or additive models, in which the coefficients to be fitted do not have to appear explicitly in the formula. Formulas as discussed here follow generally the style introduced by Wilkinson and Rogers (1973), and subsequently adopted in many statistical programs such as GLIM and GENSTAT. While originally introduced in the context of the analysis of variance, the notation is in fact just a shorthand for expressing the set of terms defined separately and jointly by a number of variables, typically factors. Its application is therefore much more general; for example, it works for tree-based models (Chapter 9), where there is no direct link to linear models. Two additional extensions appear in our use of formulas:

- a "variable" can in fact be an arbitrary S expression, and
- the response in the model is included in the formula.

Of course the "any expression" in the first item had better evaluate to one of the permissible data types: numeric vector, factor (including categories and logicals), or matrices. The discussion here focuses on special operators for the predictors, and so, in the examples below, we will omit the response expressions.

A model formula defines a list of terms to appear in a model. Each term identifies some S expression involving the data. This expression, in a linear model, generates

one or more columns in a model matrix. These columns, each multiplied by the appropriate coefficient, are the contribution of this term to the fit. For other types of models, the contribution of the term may be computed in a slightly different way, but in any case the *expanded* definition of the model corresponds to this list of terms. A corresponding expanded formula has one expression for each term, separated by + operators. You will hardly ever write fully expanded formulas, but any formula you do write will first be expanded (and simplified) before being evaluated. In this section we proceed by first discussing the meaning of expanded formulas. Then we give the complete rules by which arbitrary formulas are expanded.

Interaction and Nesting

Expanding a formula reverses the process shown in Section 2.2 of choosing a shorthand for a formula. For example, the formula

```
Opening * Mask
```

says that we want a model which fits coefficients for `Opening`, for `Mask`, and for the interaction of the two. When `Opening` and `Mask` are factors, this means a coefficient for each level of the factor; if either is a numeric variable or a numeric matrix, there will instead be one coefficient for the variable or for each column of the matrix. (We will discuss the meaning of interaction in this case later in this section.) In the more customary textbook notation,

- a `factor:factor` interaction represents a term of the form γ_{ij}, which is a set of IJ constants for each cell in the two-way table obtained by crossing the two factors (assuming the factors have I and J levels, respectively);
- a `factor:numeric` interaction represents a term of the form $\beta_j x$, or a varying slope model, in which the coefficient of the numeric variable x is different for each of the J levels of the factor;
- a `numeric:numeric` interaction represents a term βxz, where xz is simply the pointwise product of the variable x with the variable z. This is probably the least meaningful form of interaction, but of course the syntax allows far more meaningful terms to be created in cases such as this. For example, `poly(x,z,2)` will specify a bivariate quadratic surface in the two numeric variables `x` and `z`.

The formula `Opening * Mask` in expanded form is then

```
1 + Opening + Mask + Opening:Mask
```

The formula above brings in factors in a *crossed* model; that is, the model says that the individual factors should be included and, in addition, that the contribution of one factor to the fit may change depending on the level of the other factors.

Nested terms in a model, on the other hand, arise when the levels of one factor are only meaningful within a particular level for some other factor or combination of factors. For example, suppose we have some geographic data in which the variable `state` defines the state for each location and the factor `county` indexes counties within each state. Clearly, `county` was generated by coding whatever county names appeared for each state, so that level `1` of `county` means something different in different states. In this context, a main effect for `county` is meaningless, and a typical model will fit a main effect for `state` and then look at the coefficients for `county` within each level of `state`. In expanded form this could be written

```
1 + state + county:state
```

However, to emphasize that the last term is thought of as a nested term, not an interaction, we allow (and encourage) writing the model as

```
1 + state + county %in% state
```

The formula written above in expanded form has the shorthand notation

```
state / county
```

meaning "`state` and then `county` within `state`." Notice that while factors joined by `*` can be permuted without changing the meaning (except for the order of the expanded terms), factors joined by `/` can never be meaningfully permuted: if `county` is nested in `state`, then `state` cannot be nested in `county`.

While the model implies a coefficient for each level of each term, in practice the coefficients have built-in dependencies. When a model matrix is created that represents a particular model, columns coding each term are included for all the coefficients that can be estimated; this is the condition for a *valid* coding of the model. One would like the individual coefficients to be meaningful in terms of the overall model and to avoid too many redundant coefficients that will have to be removed in the fitting. In Section 2.4.1, the general rules for coding will be outlined, but for practical purposes you need not worry about the coding unless you want to understand or control the specific choice of coefficients.

The coding of factors depends on the overall model. In the two-factor crossed model, main effects are included for both of the factors in the interaction term. All the possible coefficients for the interaction term can be estimated by representing each factor by contrasts among the levels of the individual factors. The contrasts will be chosen by default in a standard way, but the coding can be controlled, as shown in the next section. Unordered factors are coded as successive differences using the Helmert contrasts (Section 2.3.2), and ordered factors are coded to give a polynomial fit to a hypothetical underlying numeric variable. In the nested model, there is no main effect for `county`, so that the coding of `county %in% state` proceeds differently: `state` is coded by dummy variables and `county` by contrasts. This produces the computational equivalent of "`county` within `state`" and also guarantees that the model will fit as many coefficients as can be meaningfully defined. The details of coding affect only the meaning of the individual coefficients estimated. Any valid representation will give the same contribution to the overall fit in the model for each of the terms. If you don't care about the individual coefficients, leave the default coding in place; if you do care, look at page 32 to see how to change it.

When one of the variables in an interaction is numeric, the term will be computed formally the same way, but some extra remarks are needed on how crossed and nested interactions are interpreted. A numeric factor is always "coded" as itself. In interactions, the numeric factor will be multiplied by either the dummy variables or the contrasts for a factor. Consider a simple example using the automobile data: suppose `Weight` is numeric and `Foreign` is a logical variable, which will be turned into a factor with two levels corresponding to `FALSE` and `TRUE`. Both the crossed formula

```
Foreign * Weight
```

and the nested one

```
Foreign / Weight
```

make sense, but they mean something different. Consider the nested version. As before, this expands into the main effects for `Foreign` followed by `Weight` within each level of `Foreign`. In terms of the actual coefficients, one coefficient will be fitted for `Weight` using only data from level 1 of `Foreign` and one using data for level 2. There will only be one coefficient for `Foreign`, estimating the contrast between the two levels. This is equivalent to fitting a model to observations for which `Foreign` is `TRUE` as

$$\mu + \alpha_F + \beta_1 \times \texttt{Weight}$$

and another model to observations for which `Foreign` is `FALSE` as

$$\mu - \alpha_F + \beta_2 \times \texttt{Weight}$$

There are four coefficients: the intercept μ, the contrast α_F for `Foreign`, and coefficients β_i for `Weight` within each level of `Foreign`. This formula therefore corresponds to the concept of fitting "separate slopes" to the different levels of the factor.

The crossed formula fits main effects for both `Foreign` and `Weight`, and then fits the product of `Weight` with the coded contrasts of levels for the factor. In terms of specific coefficients this is

$$\mu + \alpha_F + \beta \times \texttt{Weight} + \gamma \times \texttt{Weight}$$

when `Foreign` is `TRUE` and

$$\mu - \alpha_F + \beta \times \texttt{Weight} - \gamma \times \texttt{Weight}$$

when `Foreign` is `FALSE`. Again there are four coefficients, but this time there is an overall slope β for `Weight` and a contrast γ estimating the interaction of `Foreign` and `Weight`. This is an appropriate way to code the model if we want to look at an overall fit to `Weight` and *then* to examine whether something substantial would be added to the model by allowing the regression to depend on the level of the factor. The distinction between the crossed and nested versions is not so strong here as when both predictors are factors, because a numeric factor is always just itself, but the treatment is entirely analogous. (Section 2.4.1, where we discuss how the coding works, will show *why* the computations can be the same.)

When *both* factors in an interaction are numeric, the formula expands as usual; now the pure interaction `x:z` amounts to fitting an ordinary product. In this case, however, you may really have wanted to use `*` or `/` in its ordinary S sense, in which case you ought to have protected the expression with the identity function `I()`, as we will show when we go into details about general formulas next.

Syntax of Formulas

We now give the full rules for writing model formulas. A model formula is created by separating a response term from the predictor terms by the operator ~; the response can be absent. Expressions appearing in a model formula are interpreted as ordinary S expressions, except for the following operators:

```
+   -   *   /   :   %in%   ^
```

The operator `-` is used to delete terms; for example,

```
Padtype * Opening * Mask - Padtype:Opening:Mask
```

deletes the third-order interaction term that was implied by the expansion of the `*` operator, so that the formula expands to

```
Padtype + Opening + Mask + Padtype:Opening + Padtype:Mask + Opening:Mask
```

As in this example, the `-` operator is useful for compactly dropping a few interactions, when we are prepared to assume these particular terms are negligible. A simple use of `-` is to exclude the intercept from a model:

```
Yield ~ Mass - 1
```

In Chapter 1 we describe the `update()` function for changing fitted models, typically by altering the formula. The "-" operator plays a special role there as well (illustrated again in the first example in the list below).

The use of ":" to denote interaction is a break from the traditional Wilkinson and Rogers syntax, where "." is used instead. A "." is a valid part of a name in S, as in `wind.speed`, so it could not serve as an interaction operator. A single "." does have a special meaning though; it serves as the *default* left or right side of a formula wherever that makes sense. We made use of "." in some of the examples in Chapter 1. Other examples are

- `update(lmob, . ~ . - Age)` is used to update the fitted linear-model object `lmob` by modifying its formula and then refitting it. The "." on the left of ~ implies that the response is the same as in `lmob`, while the "." on the right of ~ gets replaced by whatever was on the right in the formula used to fit `lmob`.
- `lm(Mileage ~ ., data = car.test.frame)`; here the "." is interpreted relative to the data frame `car.test.frame`, which is a dataset to be used in fitting the linear model. Data frames are described in Chapter 3. The "." here means that all the variables in `car.test.frame`, except `Mileage`, are to be used additively, which is equivalent to the explicit formula

  ```
  Mileage ~ Price + Country + Reliability + Type + Weight + Disp. + HP
  ```
- `lm(skips ~ .^2, data = solder.balance)`; similar to the previous item, except all the main effects and second-order interactions of the variables in `solder.balance` are to be used.

The following table summarizes the special meanings of the operators in formulas:

Expression	Meaning
$T \sim F$	T is modeled as F
F_a + F_b	Include both F_a and F_b
F_a - F_b	Include all F_a except what is in F_b
F_a * F_b	F_a + F_b + F_a : F_b
F_a / F_b	F_a + F_b %in% (F_a)
F_a : F_b or F_b %in% F_a	The factor jointly indexed by F_a and F_b
F^m	All the terms in F crossed to order m

The expression T is a term (with no special operators included), but F, F_a, and F_b can be arbitrary formulas, not just single terms. The operators in the table are special in their semantics (that is, in the way that S interprets them) but they otherwise act as they would in ordinary expressions, with the same precedence and association they would normally have (see [S], Section 3.2.6). Parentheses can be used to change the grouping implied by precedence rules—for example, to force a combination of terms to act like one term. The formula

```
Panel / (Opening * Mask)
```

says to fit all the terms in `Opening * Mask`, within each level of `Panel`.

Slightly more subtle is

```
(Panel + Mask)/ Opening
```

which expands to

```
1 + Panel + Mask + Opening  %in%   (Panel + Mask)
```

The parentheses around `Panel + Mask` here do not get expanded further, and the last term is equivalent to

```
Opening:Panel:Mask
```

The last item in the table is a shorthand for creating interactions. For example

```
(Opening + Mask + Panel)^2
```

expands to the same formula constructed from using "-":

```
Padtype + Opening + Mask + Padtype:Opening + Padtype:Mask + Opening:Mask
```

Composite Terms in Formulas

The special meaning of the operators applies only at the top level in the formula expressions, and only on the right of the "~". If the operators appear as the arguments to other functions, they behave as they always do in S. As the user certainly intended, the term

```
atan( Length / Width )
```

fits a single coefficient to the ordinary value of the S expression, and does not treat / as a nesting operator. Similarly,

```
sqrt (x - min(x))
```

does not treat - specially. It is also possible to *force* the operators to be treated in an ordinary way, by using the identity function, `I()`. This function returns its argument and exists only to protect special operators. For example, to fit as a single term the product of `Length` and `Width`, use

```
I( Length * Width )
```

to prevent the operator * from getting its special interpretation.

As emphasized before, any variables in the formula (either the response or the factors in the terms) can be arbitrary S expressions, so long as they evaluate to objects having a valid data type, namely: numeric variables or matrices, factors including ordered factors, and non-numeric variables which will be converted into factors.

Matrices that appear in the formula are treated as a single factor. This is how special curves can most easily be generated, and functions are provided that generate suitable matrices for common kinds of curves. The following are some examples:

- The expression `poly(x, degree)` returns a matrix whose columns are an orthogonal basis for fitting a polynomial of degree `degree` in the numeric variable `x`. Similarly `poly(x,y, degree)` returns the matrix of bivariate polynomial terms of degree no more than `degree`, and so on.

- The expression `bs(x, df)` returns a matrix which is a B-spline basis for piecewise-cubic regression on `x`. The parameter `df` is the degrees of freedom, which determines the number of interior knots. These knots are automatically placed by the function; otherwise, the `knots` argument can be used to place them explicitly.

See Chapters 6 and 7 for a general discussion of composite terms such as these.

Functions can be used that produce factors and categories as well; for example, `ordered(x,breaks)` will return an ordered factor cutting the numeric variable `x` at the breakpoints `breaks`. This is similar to the S function `cut()`, which produces a category. Expressions that produce factors or categories can be used in conjunction with the special operators, so that

```
cut(Weight,5) * Country
```

creates a five-level category from the numeric variable `Weight` and then uses it in a crossed model with the factor `Country`. Similarly

```
( Age < 45 ) * Cholesterol
```

creates different linear trends in `Cholesterol` for people under and over 45. The function `codes()` produces numbers to represent the levels in a factor or ordered factor; so if `Opening` is an ordered factor with levels `Large`, `Medium`, and `Small`, then the expression

```
codes(Opening)
```

implies a term linear in the numbers 1, 2, and 3, coding the three levels.

As always in S, you can write any functions of your own to create other suitable variables. The expressions can be more complicated than function calls as well:

```
group * (if(all(x)>0)log(x) else log(x-min(x)+.01))
```

Exotic expressions like this are perfectly legal but hard to read and not good programming style. A better approach is to define a function, say

```
plus.log <- function(x)
   if(all(x)>0)log(x) else log(x-min(x)+.01)
```

and then write the formula as `group*plus.log(x)`.

2.3.2 Coding Factors by Contrasts

On page 21 we noted that factors entering a model normally produce more coefficients than can be estimated. This is true regardless of the data being used; for example, the sum of all the dummy variables for any factor is a vector of all ones. This is identical to the variable used implicitly to fit an intercept term. This is *functional* overparametrization, as opposed to *data-dependent* overparametrization in which the number of observations is not large enough to estimate coefficients or in which some of the variables turn out to be linearly related. The functional problem is removed in most cases before any model-fitting occurs, by replacing the dummy variables by a set of functionally independent linear combination of those variables, which is arranged to be independent also of the sum of the dummy variables. For a factor with k levels, $k-1$ such linear combinations are possible. We call a particular choice of these linear combinations a set of *contrasts*, using the terminology of the analysis of variance. Computationally, the contrasts are represented as a k by $k-1$ matrix.

Any choice of contrasts for factors alters the specific individual coefficients in a model but does not change the overall contribution of the term to the fit. All the model-fitting functions choose contrasts automatically, but users can also specify the contrasts desired, either in the formula for the model or in the factor variable itself. By default, contrasts are chosen as follows:

- unordered factors are coded by what are known as the *Helmert* contrasts, which effectively contrast the second level with the first, then the third with the average of the first and second, and so on;
- ordered factors are coded so that individual coefficients represent orthogonal polynomials if the levels of the factor were actually equally spaced numeric values.

If this choice of contrasts is adequate, no user action is needed.

The simplest way to alter the choice of contrasts is to use the function `C()` , with usage `C(factor, contrast)` in the formula. The first argument is a factor, the second a choice of contrast. It returns `factor` with the appropriate contrast matrix attached as an attribute. The choice can be made in three ways:

- By giving the name of a built-in choice for contrasts: `helmert`, `poly`, `sum`, or `treatment`. For example, `C(Opening, sum)` uses the function `contr.sum()` to generate the appropriate sized contrast matrix. We will explain the meaning of these choices below.
- By giving a function, which when called with either a factor or the number of levels of the factor as its argument, returns the k by $k-1$ matrix of constraints: `C(Opening, myfun)` calls `myfun(Opening)` to generate the contrast matrix (if `myfun` exists as a function).
- By giving the contrast matrix *or* some columns for the contrast matrix. `C(Opening, mymat)` uses the matrix `mymat` as the contrast matrix.

The function `C()` tests for each of these cases to determine how it has been called.

The four standard choices correspond to four functions to generate particular flavors of contrasts. The polynomial contrasts are the result of the function `contr.poly()`:

```
> contr.poly(4)
              L    Q          C
[1,] -0.6708204  0.5 -0.2236068
[2,] -0.2236068 -0.5  0.6708204
[3,]  0.2236068 -0.5 -0.6708204
[4,]  0.6708204  0.5  0.2236068
```

The coefficients produced by this transformation of the dummy variables correspond to linear, quadratic, and cubic terms in a hypothetical underlying numeric variable that takes on equally spaced values for the four levels of the factor. In general, `contr.poly` produces $k-1$ orthogonal contrasts representing polynomials of degree 1 to $k-1$.

Similarly, the function `contr.helmert()` returns the Helmert parametrization. The first linear combination is the difference between the second and first levels, the second is the difference between the third level and the average of the first and second, and the jth linear combination is the difference between the level $j+1$ and the average of the first j—for example,

```
> contr.helmert(4)
  [,1] [,2] [,3]
1   -1   -1   -1
2    1   -1   -1
3    0    2   -1
4    0    0    3
```

These two are the default choices.

The `sum` choice and the corresponding function `contr.sum()` produce contrasts between each of the first $k-1$ levels and level k.

```
> contr.sum(4)
  [,1] [,2] [,3]
1    1    0    0
2    0    1    0
3    0    0    1
4   -1   -1   -1
```

This corresponds to a parametrization got by applying the constraint that the sum of the coefficients be zero.

The `treatment` form of coding is commonly used in models for which the first level of *all* the factors is considered to be the standard or control case, and in which one is interested in differences between any of the nonstandard or treatment situations. As constraints on the coefficients, this is usually expressed as saying that any coefficient in which any of the factors appears at its first level is set to 0. The equivalent coding uses the dummy variables for levels 2 through k. The function `contr.treatment()` gives this coding:

```
> contr.treatment(4)
  2 3 4
1 0 0 0
2 1 0 0
3 0 1 0
4 0 0 1
```

This is a legitimate coding, in that it captures all the coefficients. However, it is not a set of contrasts, in that the columns do not sum to zero and so are not orthogonal to the vector of ones. For applications to linear models in designed experiments, the coefficients will not be statistically independent for balanced experiments. This complicates the interpretation of techniques such as the analysis of variance, so that the control-treatment coding should generally not be used in this context. For some other models, such as the GLM models in Chapter 6, the lack of orthogonality is less obviously a defect, since the assumptions of the models do not produce statistical independence of the estimated coefficients anyway. Probably for this reason, the control-treatment coding is popular among GLM modelers, since what it lacks in orthogonality it gains in simplicity.

Any of these can be selected in a formula to override the default. You can also implement any function you like, perhaps by modifying one of the four standard functions, to produce a different set of contrasts. The matrix must be of the right dimension and the columns must be linearly independent of each other and of the vector of all ones. If this fails, model-fitting with complete data will produce singular models. An easy way to test this condition is to bind a column of 1 to the matrix and pass the result to the `qr()` function. The value of this function has a component `rank` that is the computed numerical rank of the matrix. For a set of contrasts on k levels, the rank should be k—for example

```
> qr(cbind(1,contr.treatment(4)))$rank
[1] 4
```

A function to generate contrasts must also, by convention, take either the levels of the factor or the number of levels as its argument. See any of the four standard functions for code to copy.

The third way to specify contrasts is directly by numeric data. You can start from the value of one of the functions, but a more typical situation in practice is that

you want to estimate one or more specific contrasts, but will take anything suitable for the remainder of the $k - 1$ columns. Suppose `quality` is a factor (unordered) with four levels:

```
> levels(quality)
[1] "tested-low"  "low"    "high"   "tested-high"
```

Suppose that we want the first contrast of `quality` to measure the difference between tested and nontested—that is, levels 1 and 4 versus levels 2 and 3—and we don't care about the other contrasts. Then we can give the factor in the formula as

```
C(quality, c(1, -1, -1, 1))
```

Two additional contrasts will be chosen to be orthogonal to the specified contrast. If we had wanted the second contrast to be between the two `low` and the two `high` levels, we would have supplied `C()` with the matrix

```
     [,1] [,2]
[1,]    1    1
[2,]   -1    1
[3,]   -1   -1
[4,]    1   -1
```

and one further column would be supplied.

One additional detail is sometimes needed. Sometimes the user is willing to assert that *only* some specified contrasts in the levels of a factor can be important; the others should be regarded as *known* to be zero and omitted from the model. This is risky, of course, but is done in some experiments where the number of runs is limited and the user has considerable prior knowledge about the response. The specification can be done by giving `C()` a third argument, the number of contrasts to fit. For example, suppose we are fitting polynomial contrasts to an ordered factor, `Reliability`, and assert that no more than quadratic effects are important. The corresponding expression in the model would be

```
C(Reliability, poly, 2)
```

Since the ith contrast generated by `contr.poly()` corresponds to an orthogonal polynomial of degree i, this term retains only linear and quadratic effects.

The function `C()` combines a factor and a specification for the contrasts wanted, and returns a factor with those contrasts explicitly assigned as an attribute. The companion function `contrasts()` extracts the contrasts from a factor, and returns them as a matrix. The contrasts may have been explicitly assigned as an attribute or may be the appropriate default, according to whether the factor is ordered or not. If you want to set the contrasts for a particular factor *whenever* it appears, the function `contrasts()` on the left of an assignment does this. In the example of one specific contrast,

```
> contrasts(quality) <-  c(1, -1, -1, 1)
> contrasts(quality)
             [,1] [,2] [,3]
  tested-low    1 -0.1 -0.7
         low   -1 -0.7  0.1
        high   -1  0.7 -0.1
 tested-high    1  0.1  0.7
```

two additional linear combinations have been added to give a full contrast specification. Now, `quality` will have this parametrization by default in any formula, with the opportunity still available to use the `C()` function to override. As with `C()`, the function `contrasts()` on the left of an assignment takes an optional additional argument, `how.many`, that says to assign fewer than the maximum number of contrasts to the factor.

You can also change the default choice of contrasts for *all* factors using the `options()` command in S, once you know a little more about how coding is done.

```
> options()$contrasts
          factor       ordered
 "contr.helmert" "contr.poly"
```

shows us what the defaults are. These options are the names of functions that provide contrasts for unordered and ordered contrasts, respectively. To reset the defaults, use:

```
options(contrasts=c("contr.treatment","contr.poly"))
```

Redefining one or both of the elements changes the default choice of contrasts. The effect of using `options()` to change the default contrast functions lasts as long as the S session; each time S is started up, the permanent default is assumed. If you really want to have your own private default coding every time you run S, you can invoke `options()` automatically via the `.First()` function (S, Section 3.4.9). Notice that explicit choices for individual factors can still be used to override the new default coding by assigning the contrasts as before.

Strictly speaking, the term *contrast* implies that all the linear combinations are contrasts of levels. In this case, the sum of the numbers in any column of the matrix should be zero. *Orthogonal* contrasts have the additional property that the inner products of any two columns of the contrast matrix is also zero. The Helmert and polynomial contrasts have both these properties. The contrast and orthogonal contrast properties are particularly important for linear models in designed experiments. Otherwise, the choice of contrasts can introduce artificial correlations between coefficient estimates, even if the design is balanced. Additional details on the implications of contrasts for fitted models appear in Section 5.3.1 in the context of analysis of variance, and in Section 6.3.2 in the context of GLMs.

2.4 Internal Organization of Models

In this section, as in the fourth section of later chapters, we will reveal how things work: the internal structure of the objects that have appeared earlier in the chapter and the computational techniques used in the functions. This section is *not* required reading if you only want to use the functions described so far. It will be useful, however, for those who want to extend the capabilities and/or to specialize them in a serious way for particular applications. Extensions and modification of the software we provide is not only allowed but is one of the goals of our approach to statistical software in this book. Rather than trying to provide a complete approach to the topics we cover (probably an impossible task anyway), we present functions and objects that form a kernel containing the essential computations. The functions include what we see as the natural approach to common, general use of the statistical methods. The classes of objects organize the key information involved, with the goal of making subsequent use of the information as easy and general as possible.

Users with special needs, and researchers who want to extend the statistical techniques themselves, will want to go beyond what we provide. Understanding the material in this section will likely help.

2.4.1 Rules for Coding Expanded Formulas

This section gives the rules underlying the coding of factors in the expanded formulas of Section 2.3.1. To produce a model matrix for use in linear models, factors and their interactions are represented by columns of numeric data, either dummy variables or contrasts. To be valid, the representation must estimate the full linear model. Since such models are generally overparametrized, there will be many different valid representations in this sense. The goal of a particular representation is to be meaningful and reasonably parsimonious. The actual coefficients estimated should mean something in the application of the model. For example, a coefficient value significantly different from zero should say something useful about the data. A parsimonious parametrization is desirable numerically, since the size of the model matrix can in some cases be much larger than necessary. The mathematical discussion that follows provides the basis for understanding how the representation can be chosen for various models.

Each term in an expanded formula can be written using only one special operator, ":". Suppose we have an expanded formula with p factors:

$$F_1, F_2, \cdots, F_p$$

and m terms:

$$T_1 + T_2 + \cdots + T_m$$

The F_j need not be simple variables, but can be essentially arbitrary S expressions. The expanded term T_i can always be written as an interaction of 0, 1, or more of

the F_j; say,

$$F_{i_1} : F_{i_2} : \cdots : F_{i_{o_i}}$$

where $1 \leq i_j \leq p$. The value of o_i is the *order* of T_i—that is, the number of factors. We will assume that the expanded formula is sorted by the order of the terms, so that all terms of order 1 appear first, then all terms of order 2, and so on.[1] The intercept term is the only term of order 0, and is written as `1`. If it is present, it comes first.

As discussed in Section 2.2, a factor corresponding to F_j can be represented in the model by a matrix whose columns are the dummy variables corresponding to each level of F_j. The interaction of F_{j_1} and F_{j_2} is represented by the matrix containing all possible products of pairs of columns from the matrices representing the main effects. A three-way interaction is represented by all products of columns of this matrix and columns of the matrix representing F_{j_3}, and so on. For details, see the function `column.prods()`, which carries out just this computation.

If all factors were represented by dummy variables, there would be nothing more to the interpretation of expanded formulas. However, both numerical and statistical arguments require more careful coding of factors. Usually, coefficients cannot be estimated for all the levels of the factor. For example, the sum of all the dummy variables for any main effect is the constant 1, and so is functionally equivalent to the intercept. Coding all levels by dummy variables would produce a model matrix with more columns than necessary (in some cases *many* more), and the model matrix would nearly always be singular, so that numerical solutions would not produce estimates for all the requested coefficients. These are the numerical reasons for choosing a good coding for the terms, but the statistical reason is more important—namely, to allow a *meaningful* choice of coefficients for the particular model. The functional dependencies among the dummy variables in the terms imply that only certain linear combinations of the coefficients for the dummy variables are estimable. The goal is to represent those linear combinations so that the individual computed coefficients are useful for the particular model. Section 2.3.2 showed how this coding could be controlled.

The following rule specifies which factors should be coded by dummy variables and which should be coded by contrasts in producing the columns of the model matrix:

> Suppose F_j is any factor included in term T_i. Let $T_{i(j)}$ denote the *margin* of T_i for factor F_j—that is, the term obtained by dropping F_j from T_i. We say that $T_{i(j)}$ has appeared in the formula if there is some term $T_{i'}$ for $i' < i$ such that $T_{i'}$ contains all the factors appearing in $T_{i(j)}$. The usual case is that $T_{i(j)}$ itself is one of the preceding terms. Then F_j is coded by contrasts if $T_{i(j)}$ has appeared in the formula and by dummy variables if it has not.

[1]The only ordering that we actually need is that any term T_i appear in the formula *after* its margins. It does not make sense for a factor to appear in the formula after some interaction including that same factor.

In interpreting this rule, the empty term is taken to be the intercept.

The application of this rule corresponds to generating a matrix—the model matrix—with n rows and some number of columns, to represent the whole model. This matrix comes from binding together the columns of the matrices produced by the rule for each term. We can compare this matrix with the overspecified but valid coding we would get if we used dummy variables for all the factors. Our rule is valid if the dummy variables, say X^*, introduced for term T_i in this overspecified coding, can be represented as a linear combination of columns from the matrices produced by our rule for terms up to and including T_i, for all i.

Here is an informal proof that the rule is valid. Start with an inductive assumption: suppose that the rule is valid for terms of order less than the order of T_i; specifically, for any such term, assume that its dummy matrix can be written as a linear combination of the matrices given by our rule for that term and those of its margins that are in the formula. Suppose F_j is one of the factors for which the rule says we can use contrasts. Let X_j be the n by $k_j - 1$ matrix of contrast variables for F_j, and X_j^* the corresponding n by k_j matrix of dummy variables. We will need to refer to the lth columns of these matrices; let's call them $x_{j:l}$ and $x_{j:l}^*$. Any column of X^* can be written as the product of one column from each of the dummy matrices, X_J^*, for factors J in T_i, so in particular it can be written as:

$$\Big(\prod_{J \in T_{i(j)}} x_{J:l_J}^* \Big) x_{j:l}^*$$

Note that this is ordinary multiplication of the n-vectors, not matrix product. Now look at the two parts of the above expression separately, the left part in parentheses and the single vector on the right.

1. By the inductive assumption, the left part is a linear combination of our matrices for $T_{i(j)}$ and *its* margins.

2. From the definition of a valid coding of the individual factors, the right part is a linear combination of 1 and the $x_{j:r}$

If we were to expand these two linear combinations, the result would be a linear combination of column products from our coding for T_i and for its various margins. Therefore, the inductive assumption holds for T_i as well. By looking directly at the cases of the empty term and terms of order 1, the inductive assumption holds for these cases, and so is believable in general. This is not quite precise; in particular, extra arguing needs to be added for the (somewhat strange) case that $T_{i(j)}$ is not in the model, but is contained in some other preceding term of order equal that of T_i.

The argument above should nevertheless be sufficiently convincing for our purposes here.

The rule does not always produce a minimal coding; that is, in some cases there may be functional dependencies between columns of the matrix representing T_i and those representing earlier $T_{i'}$. In particular, this will be the case again when there is no $T_{i'}$ that is exactly equal to $T_{i(j)}$. However, models of that form are usually questionable; for most sensible model formulas, the rule above produces a minimal and meaningful coding of the terms.

Numeric variables appear in the computations as themselves, uncoded. Therefore, the rule does not do anything special for them, and it remains valid, in a trivial sense, whenever any of the F_j is numeric rather than categorical.

2.4.2 Formulas and Terms

Formula objects pass through an intermediate stage before being combined with the data. This stage produces objects of class "terms", which contain the formula after it is processed to have, in a convenient form, all the information needed to create the model. Users of model-fitting functions will not see this intermediate stage, but those of you who want to modify model-fitting techniques or to create a new class of models may find it helpful to know what information the terms objects contain. A terms object is an object of mode "expression" with extra attributes. The elements of the expression are the individual terms in the expanded right side of the formula:

```
> form1 <- skips ~ Panel * Opening
> terms1 <- terms(form1)
> as.vector(terms1)
expression(skips, Panel, Opening, Panel:Opening)
```

Now let's consider the attributes:

```
> names(attributes(terms1))
[1] "formula"     "factors"     "order"     "variables"
[5] "term.labels" "intercept"   "response"  "class"
```

The meaning of the attributes is as follows:

"formula": the actual formula used to construct terms, in this case the contents of form1:

```
> attr(terms1,"formula")
skips ~ Panel * Opening
```

There is a generic function formula() for extracting formulas from a variety of objects; in this case, formula(terms1) would extract the formula from the terms object.

"factors": a matrix with factors along the rows and terms along the columns. The jth column says what factors appear in the jth term and also whether they are coded as contrasts or dummy variables. Values in the column are 1 for contrasts, 2 for dummy variables, and 0 if the factor does not appear in the term:

```
> attr(terms1,"factors")
        Panel Opening Panel:Opening
  skips     0       0             0
  Panel     1       0             1
Opening     0       1             1
```

The coding is specified according to the general rule in the previous section.

"order": a vector giving the order of each term: 1 for main effects, 2 for second-order interactions, and so on:

```
> attr(terms1,"order")
[1] 1 1 2
```

"variables": an expression whose elements are the expressions for each of the variables, including the response (remember that these need not simply be names):

```
> attr(terms1,"variables")
expression(skips, Panel, Opening)
```

"term.labels": the character form of the terms, only included to save repeated deparsing later:

```
> attr(terms1,"term.labels")
[1] "Panel"        "Opening"       "Panel:Opening"
```

which can also be extracted using the labels() generic function.

"intercept": a logical variable that will be TRUE unless the term -1 appears in the formula. Notice that the intercept term does not appear in the expression vector itself nor in the term labels. The label "(Intercept)" is used to label coefficients, etc. corresponding to the intercept.

"response": which variable in the "variables" attribute is the response (0 if there is no response specified).

Since all the model-fitting functions include the terms object for a particular model in the object that represents the fitted model, you can use the information above to conveniently get at information about pieces of the model when designing new summary functions or modifying the model-fitting. Section 7.4 describes some additional arguments in the call to terms that add to its flexibility.

2.4.3 Terms and the Model Matrix

The process of putting together data and formula to construct a model matrix involves three basic steps:

1. Convert the formula into a `terms` object, in which all the interactions and nested terms have been expanded, and any simplifications resulting from subtractions, parentheses, dots, and powers have been applied.

2. Compute a *model frame* from the terms and the data, containing variables corresponding to the expressions needed to compute the terms defined in step 2.

3. Generate the model matrix itself from the model frame.

A model frame is a special type of data frame, described in the next chapter. For the moment, simply think of it as a list of the response variable and variables corresponding to all the terms in the formula. The function `model.frame()` uses the terms object, specifically its attribute `variables`, to determine which expressions will be used in generating the model matrix. It returns a special data frame containing those variables. Notice that there is no restriction on the expressions appearing in the formula for the terms. The names of the variables will not necessarily be syntactic names in S; if one of the terms is `log(Fuel)` then the corresponding variable will have name `"log(Fuel)"`. Two other, optional computations take place during the evaluation of the model frame. If a `subset` argument is supplied, the corresponding subset will be extracted from each computed variable before it is inserted into the model frame. Similarly, if a `na.action` function is supplied either as an argument or as an attribute to the data frame, this function will be applied to the model frame. See Section 3.3.3 for further details.

Once the model frame has been computed, it is used to generate a model matrix, with columns corresponding to each of the terms in the model formula. A model matrix is a numeric matrix of suitably coded dummy variables, contrasts, or numeric variables, plus some attributes related to the model.

We don't have to worry about the steps described above; the fitting functions such as `lm()` do the work for us, and will return both the model frame and model matrix if requested. On the other hand, we can create a model matrix from some data directly from the `model.matrix()` function. To illustrate the structure in model matrices, we will compute a model matrix from a market study data frame described in the next chapter. The model chosen will use the numeric variable `usage` along with a complete model (main effects and interaction) for two factors, `nonpub` and `education`:


```
> model1 <- model.matrix(~ usage + nonpub * education)
> print(model1[1:5,], abbreviate = T)
   (Intrc) usage nonpub education1 education2 education3 education4
 1       1     9      1          1         -1         -1         -1
 2       1     2      1          1         -1         -1         -1
 7       1     3     -1         -1         -1         -1         -1
 8       1     1     -1          1         -1         -1         -1
10       1     2     -1         -1         -1         -1         -1

   education5 nnp:edc1 nnp:edc2 nnp:edc3 nnp:edc4 nnp:edc5
 1         -1        1       -1       -1       -1       -1
 2         -1        1       -1       -1       -1       -1
 7         -1        1        1        1        1        1
 8         -1       -1        1        1        1        1
10         -1        1        1        1        1        1
> dim(model1)
[1] 1000   13
```

Notice we used no response in the formula; had one been there it would have been ignored. The model matrix has as many columns as are required by the coding of the expanded formula. The `abbreviate=` argument to the printing method for `matrix` objects abbreviates the column labels. The unabbreviated column labels are

```
> dimnames(model1)[[2]]
 [1] "(Intercept)"          "usage"                "nonpub"
 [4] "education1"           "education2"           "education3"
 [7] "education4"           "education5"           "nonpub:education1"
[10] "nonpub:education2" "nonpub:education3" "nonpub:education4"
[13] "nonpub:education5"
```

Model matrices have additional attributes:

```
> names(attributes(model1))
[1] "dim"        "formula"    "class"     "order"     "term.labels"
[6] "assign"     "dimnames"
```

The `class` attribute has value `c("model.matrix", "matrix")`, which means that model matrices inherit from the more general class `"matrix"`. The `formula`, `order` and `term.labels` attributes are retained from the `terms` object. The `assign` attribute is a list, with length equal to the number of terms. The elements of `assign` define which columns of the matrix belong to the corresponding terms:

```
> attr(model1,"assign")
$"(Intercept)":
[1] 1
```

```
$usage:
[1] 2

$nonpub:
[1] 3

$education:
 1 2 3 4 5
 4 5 6 7 8

$"nonpub:education":
 1  2  3  4  5
 9 10 11 12 13
```

The `model.matrix` function produces the matrix of predictors for linear and generalized linear regression and anova model-fitting routines; other model-fitting functions, such as those that build trees, require other constructions. These are discussed further in the relevant chapters, as well as in Section 3.3.3. There we also expand on the sequence of steps needed to create model frames and model matrices, to allow facilities for weights, missing data, and subsets.

Bibliographic Notes

The formula language described in this chapter was inspired by the the Wilkinson and Rogers (1973) formula language used in the package GLIM. Several of the enhancements introduced here, such as `poly()`, for example, were mentioned in the Wilkinson and Rogers paper, but not fully implemented in GLIM.

Chapter 3

Data for Models

John M. Chambers

This chapter describes the general structure for data that will be used throughout the book. In particular, it introduces the *data frame*, a class of objects to represent the data typically encountered in fitting models.

Section 3.1 presents some datasets that recur as examples throughout the book. S functions to create, manipulate, modify, and study data frames are described in Section 3.2. Section 3.3 discusses the computations on data frames and related classes of objects at a detailed level, suitable if you want to modify functions dealing with these objects.

As with Chapter 2, the ideas in this chapter underlie all the computations for various models in the following chapters. To get a general view of our approach to data, you should read some of this chapter before going on to specific models. Sections 3.1 and 3.2 should be plenty. Your data analysis will benefit from studying graphical and other summaries of the data *before* any commitment to a particular model. This chapter describes a number of such summaries and also shows how to apply S functions generally to the data in data frames. Therefore, we recommend reading through the first two sections of the chapter before fitting particular models.

3.1 Examples of Data Frames

The statistical models discussed in this book nearly always think of the underlying observational data as being organized by *variables*—statistical abstractions for different things that can be observed. Values on these variables can be recorded for a