Experimental Design: Efficiency, Coding, and Choice Designs

Warren F. Kuhfeld

Abstract

This chapter discusses some of the fundamental concepts in marketing research experimental design including factorial designs, orthogonal arrays, balanced incomplete block designs, nonorthogonal designs, and choice and conjoint designs. Design terminology is introduced, design efficiency and the relationship between designs for linear and choice models are explained, and several examples of constructing designs for marketing research choice experiments are presented.*

You should familiarize yourself with the concepts in this chapter before studying the conjoint chapter (page 681) or the discrete choice chapter (page 285). After you are comfortable with the material in this chapter, consider looking at the other design chapters starting on pages 243 and 265.

Introduction

An *experimental design* is a plan for running an experiment, and it is often displayed as a matrix. The *factors* of an experimental design are the columns or variables that have two or more fixed values or *levels*. The rows of a design are the treatment combinations and are sometimes called *runs*. Experiments are performed to study the effects of the factor levels on a dependent or response variable.

Experimental designs are important tools in marketing research, conjoint analysis, and choice modeling. In a consumer product study, the rows or runs correspond to product profiles and the factors correspond to the attributes of the hypothetical products or services. In a conjoint study, the rows of the design correspond to products, and the dependent variable or response is a rating or a ranking of the products. In a discrete-choice study, the rows of the design correspond to product *alternatives*. The choice design consists of blocks of several alternatives, and each set of alternatives is called a *choice set*. The dependent variable or response is choice (product i was chosen and the other products in the set were not chosen). See page 681 for an introduction to conjoint analysis and page 289 for an introduction to choice models. The next two sections show simple examples of conjoint and choice experiments.

*Copies of this chapter (MR-2010C), the other chapters, sample code, and all of the macros are available on the Web `http://support.sas.com/resources/papers/tnote/tnote_marketresearch.html`. Specifically, sample code is here `http://support.sas.com/techsup/technote/mr2010c.sas`. For help, please contact SAS Technical Support. See page 25 for more information. Parts of this chapter are based on the tutorial that Don Anderson and Warren F. Kuhfeld presented for many years at the American Marketing Association's Advanced Research Techniques Forum.

The Basic Conjoint Experiment

A conjoint study uses experimental design to create a list of products, and subjects rate or rank the products. The conjoint analysis model is a linear model of the form $\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}}$ where $\mathbf{y}$ contains product ratings or rankings, $\mathbf{X}$ is the coded design matrix (see pages 70 and 73), and $\hat{\boldsymbol{\beta}}$ is the vector of parameter estimates or part-worth utilities. The following table displays a conjoint design and the layout of a simple conjoint experiment with two factors:

Conjoint Design

Acme	$1.99
Acme	$2.99
Ajax	$1.99
Ajax	$2.99
Comet	$1.99
Comet	$2.99

Full-Profile Conjoint Experiment

		Rate Your Purchase Interest
Acme	$1.99	
Acme	$2.99	
Ajax	$1.99	
Ajax	$2.99	
Comet	$1.99	
Comet	$2.99	

In a real experiment, the product descriptions are typically more involved and might use art or pictures, but the basic experiment involves people seeing products and rating or ranking them. The brand attribute has three levels, Acme, Ajax, and Comet, and the price attribute has two levels, $1.99 and $2.99. There are a total of six products.

The Basic Choice Experiment

A discrete choice study uses experimental design to create sets of products, and subjects choose a product from each set. Like the conjoint model, the choice model has a linear utility function, but it is embedded in a nonlinear model (see page 71). The following table displays a choice design and the layout of a simple choice experiment:

Choice Design

1	Acme	$2.99
	Ajax	$1.99
	Comet	$1.99
2	Acme	$2.99
	Ajax	$2.99
	Comet	$2.99
3	Acme	$1.99
	Ajax	$1.99
	Comet	$2.99
4	Acme	$1.99
	Ajax	$2.99
	Comet	$1.99

Discrete Choice Experiment

1	2	3	Choice
Acme $2.99	Ajax $1.99	Comet $1.99	
Acme $2.99	Ajax $2.99	Comet $2.99	
Acme $1.99	Ajax $1.99	Comet $2.99	
Acme $1.99	Ajax $2.99	Comet $1.99	

In a real experiment, the product descriptions are typically more involved and they might use art or pictures, but the basic experiment involves people seeing sets of products and making choices. This example has four choice sets, each composed of three alternative products; so subjects make four choices. Each alternative is composed of two attributes: brand has three levels, and price has two levels.

Attributes can be generic or alternative-specific. A *generic attribute* is treated the same way for each brand, like the price attribute in the design on the left. A design that consists entirely of generic attributes is called a generic design. An *alternative-specific attribute* is analyzed separately for each brand, such as the set of price factors in the design on the right. Note that the alternative-specific price effects consist of the interaction (product) of the binary brand effects and the generic price effect.

Coded Choice Design
With a Generic Price Effect

1	Acme	1	0	0	$2.99
	Ajax	0	1	0	$1.99
	Comet	0	0	1	$1.99
2	Acme	1	0	0	$2.99
	Ajax	0	1	0	$2.99
	Comet	0	0	1	$2.99
3	Acme	1	0	0	$1.99
	Ajax	0	1	0	$1.99
	Comet	0	0	1	$2.99
4	Acme	1	0	0	$1.99
	Ajax	0	1	0	$2.99
	Comet	0	0	1	$1.99

Coded Choice Design With
Alternative-Specific Price Effects

1	Acme	1	0	0	$2.99	0	0
	Ajax	0	1	0	0	$1.99	0
	Comet	0	0	1	0	0	$1.99
2	Acme	1	0	0	$2.99	0	0
	Ajax	0	1	0	0	$2.99	0
	Comet	0	0	1	0	0	$2.99
3	Acme	1	0	0	$1.99	0	0
	Ajax	0	1	0	0	$1.99	0
	Comet	0	0	1	0	0	$2.99
4	Acme	1	0	0	$1.99	0	0
	Ajax	0	1	0	0	$2.99	0
	Comet	0	0	1	0	0	$1.99

Chapter Overview

This chapter begins with an introduction to experimental design and choice design. Then it presents examples of the basic approaches to choice design. In all examples, the data are analyzed with a multinomial logit model. The examples include the following:

- The first example creates a design where all of the attributes of all of the alternatives are balanced and orthogonal. This design, which is efficient for a hypothetical linear model involving all of the attributes of all of the alternatives, is converted to the choice design format. This is a useful approach for alternative-specific models and models with cross-effects or when you are not sure what model you will ultimately use in your analysis. See page 127.

- The second example is a continuation of the first example. In this case, a design that is fully balanced and orthogonal cannot be used since there are restrictions on the design. This example uses the same tools that the first example uses, but it additionally uses a macro and options to impose restrictions on the design. See page 156.

- The third example searches a candidate set of alternatives for a design that is efficient for a specific choice model under the null hypothesis $\boldsymbol{\beta} = \mathbf{0}$. This approach is useful for generic models and whenever you are willing to specify a specific model and parameter vector before you collect your data. See page 166.

- The fourth example is a variation on the third example. This examples searches a candidate set of alternatives for a design that is efficient for a specific choice model under the null hypothesis $\boldsymbol{\beta} = \mathbf{0}$. Restrictions are imposed within each choice set across the alternatives. This approach is useful for restricted models such as restricted generic models and whenever you are willing to specify a specific model and parameter vector before you collect your data. See page 177.

- The fifth example searches a candidate set of choice sets for a design that is efficient for a specific choice model under the null hypothesis $\boldsymbol{\beta} = \mathbf{0}$. This approach can be used for generic models and whenever you are willing to specify a specific model and parameter vector before you collect your data. It is sometimes used when there are restrictions within choice sets but across alternatives. For example, this approach can be used when there are restrictions that certain levels in one alternative should not appear with certain levels in another alternative. In contrast, in the third and fourth examples, the design is constructed from a candidate set of alternatives rather than choice sets. Note, however, that the approach illustrated in the fourth example, searching a candidate set of alternatives with restrictions, is often a better approach. See page 188.

- The sixth example creates an efficient design for a purely generic experiment—an experiment that involves no brands, just bundles of attributes. The design is efficient for a choice model under the null hypothesis $\boldsymbol{\beta} = \mathbf{0}$ and for a main-effects model. The design is constructed from a candidate set of alternatives. See page 198. Also see page 102 for information about optimal generic designs. See page 198.

- The seventh example creates an optimal design for a partial-profile choice experiment (Chrzan and Elrod 1995). In each choice set, only a subset of the attributes vary and the rest remain constant. The design is optimal for a partial-profile choice model under the null hypothesis $\boldsymbol{\beta} = \mathbf{0}$ and a main-effects model. The design is constructed from a balanced incomplete block design and an orthogonal array. See page 207.

- The eighth example creates a balanced incomplete block design and uses it in a MaxDiff study (Louviere 1991, Finn and Louviere 1992). Subjects choose their most and least favorite attributes from each set, which is a subset of the full list of attributes. See page 225.

Experimental Design Terminology

The following tables display the conjoint design in four forms:

Full-Factorial Design	
x1	x2
1	1
1	2
2	1
2	2
3	1
3	2

Full-Profile Conjoint Design	
Brand	Price
Acme	1.99
Acme	2.99
Ajax	1.99
Ajax	2.99
Comet	1.99
Comet	2.99

Randomized Design	
x1	x2
2	2
1	1
1	2
3	1
3	2
2	1

Randomized Conjoint Design	
x1	x2
Ajax	2.99
Acme	1.99
Acme	2.99
Comet	1.99
Comet	2.99
Ajax	1.99

The first table contains a "raw" experimental design with two factors. The second contains the same design with factor names and levels assigned. The third contains a randomized version of the raw design. Finally, the fourth contains the randomized design with factor names and levels assigned.

Before an experimental design such as this is used, it should be *randomized*. Randomizing involves sorting the rows into a random order and randomly reassigning all of the factor levels. It is not unusual for the first row of the original design to contain all ones, the first level. Many other groupings or orderings can occur in the original design. Randomization mixes up the levels and eliminates systematic groupings and orderings. For example, randomizing a three level factor changes the original levels (1 2 3) to one of the following: (1 2 3), (1 3 2), (2 1 3), (2 3 1), (3 1 2), (3 2 1). See page 93 for more about randomization, when it is required, and when it is not.

The design in this conjoint example is a *full-factorial design*. It consists of all possible combinations of the levels of the factors. Full-factorial designs let you estimate main effects and interactions. A *main effect* is a simple effect, such as a price or brand effect (see Figure 1). For example, in a main-effects model the brand effect is the same at the different prices and the price effect is the same for the different brands. *Interactions* involve two or more factors, such as a brand by price interaction (see Figure 2). In a model with interactions brand preference is different at the different prices and the price effect is different for the different brands. In Figure 1, there is a main effect for price, and utility increases by one when price goes from \$2.99 to \$1.99 for all brands. Similarly, the change in utility from Acme to Ajax to Comet does not depend on price. In contrast, there are interactions in Figure 2, so the price effect is different depending on brand, and the brand effect is different depending on price.

In a full-factorial design, all main effects, all two-way interactions, and all higher-order interactions are estimable and uncorrelated. The problem with a full-factorial design is that it is too cost-prohibitive and tedious to have subjects consider all possible combinations, for most practical situations. For example, with five factors, two at four levels and three at five levels (denoted $4^2 5^3$), there are $4 \times 4 \times 5 \times 5 \times 5 = 2000$ combinations in the full-factorial design. For this reason, researchers often use *fractional-factorial designs*, which have fewer runs than full-factorial designs. The price of having fewer runs is that some

Figure 1

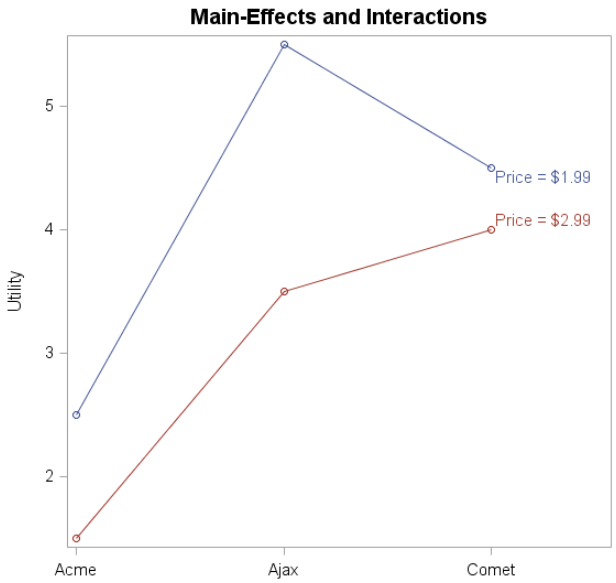


Figure 2

effects become confounded. Two effects are *confounded* or *aliased* when they are not distinguishable from each other. This means that lower-order effects such as main effects or two-way interactions might be aliased with higher-order interactions in most of our designs. We estimate lower-order effects by assuming that higher-order effects are zero or negligible. See page 495 for an example of aliasing.

Fractional-factorial designs that are both orthogonal and balanced are of particular interest. A design is *balanced* when each level occurs equally often within each factor, which means that the intercept is orthogonal to each effect. When every *pair* of levels occurs equally often across all pairs of factors, the design is *orthogonal*. More generally, a design is orthogonal when the frequencies for level pairs are proportional or equal. For example, with 2 two-level factors, an orthogonal design could have pairwise frequencies proportional to 2, 4, 4, 8. Such a design is not balanced—one level occurs twice as often as the other. Imbalance is a generalized form of nonorthogonality; hence it increases the variances of the parameter estimates and decreases the efficiency or goodness of the design.

Fractional-factorial designs are categorized by their *resolution*. The resolution identifies which effects (possibly including interactions) are estimable. For example, for resolution III designs, all main effects are estimable free of each other, but some of them are confounded with two-factor interactions. For resolution IV designs, all main effects are estimable free of each other and free of all two-factor interactions, but some two-factor interactions are confounded with other two-factor interactions. For resolution V designs, all main effects and two-factor interactions are estimable free of each other. More generally, if resolution (r) is odd, then effects of order $e = (r-1)/2$ or less are estimable free of each other. However, at least some of the effects of order e are confounded with interactions of order $e+1$. If r is even, then effects of order $e = (r-2)/2$ are estimable free of each other and are also free of interactions of order $e+1$. Higher resolutions require larger designs. Resolution III fractional-factorial designs are frequently used in marketing research. They are more commonly known as orthogonal arrays.

Orthogonal Arrays

A special type of factorial design is the *orthogonal array*. In an orthogonal array, all estimable effects are uncorrelated. Orthogonal arrays come in specific numbers of runs for specific numbers of factors with specific numbers of levels. The following list contains all (main effects only) orthogonal arrays up to 28 runs:

4	2^3	12	2^{11}	16	2^{15}	18	$2^1 3^7$	21	$3^1 7^1$	24	2^{23}	25	5^6
6	$2^1 3^1$		$2^4 3^1$		$2^{12} 4^1$		$2^1 9^1$	22	$2^1 11^1$		$2^{20} 4^1$	26	$2^1 13^1$
8	2^7		$2^2 6^1$		$2^9 4^2$		$3^6 6^1$				$2^{16} 3^1$	27	3^{13}
	$2^4 4^1$		$3^1 4^1$		$2^8 8^1$	20	2^{19}				$2^{14} 6^1$		$3^9 9^1$
9	3^4	14	$2^1 7^1$		$2^6 4^3$		$2^8 5^1$				$2^{13} 3^1 4^1$	28	2^{27}
10	$2^1 5^1$	15	$3^1 5^1$		$2^3 4^4$		$2^2 10^1$				$2^{12} 12^1$		$2^{12} 7^1$
					4^5		$4^1 5^1$				$2^{11} 4^1 6^1$		$2^2 14^1$
											$3^1 8^1$		$4^1 7^1$

The list shows the number of runs followed by the design. The design is represented as the number of levels raised a power equal to the number of factors. For example, the first design in the list (2^3) has 3 two-level factors in four runs, and the second design ($2^1 3^1$) has 1 two-level factor and 1 three-level factor in 6 runs. The first five designs in the list are as follows:

2^3

1	1	1
2	1	2
1	2	2
2	2	1

$2^1 3^1$

1	1
1	2
1	3
2	1
2	2
2	3

2^7

1	1	1	1	1	1	1
2	1	2	1	2	1	2
1	2	2	1	1	2	2
2	2	1	1	2	2	1
1	1	1	2	2	2	2
2	1	2	2	1	2	1
1	2	2	2	2	1	1
2	2	1	2	1	1	2

$2^4 4^1$

1	1	1	1	1
1	2	1	2	2
1	1	2	2	3
1	2	2	1	4
2	2	2	2	1
2	1	2	1	2
2	2	1	1	3
2	1	1	2	4

3^4

1	1	1	1
1	2	3	2
1	3	2	3
2	2	2	1
2	3	1	2
2	1	3	3
3	3	3	1
3	1	2	2
3	2	1	3

Each of these designs is balanced—each level occurs the same number of times within each factor. Each of these designs is orthogonal—every pair of levels occurs the same number of times across all of the pairs of factors in each design. In the first design, each of the four pairs appears once across all three pairs of factors. In the second design, each of the six pairs appears once. In the third design, each of the four pairs appears twice across the 21 pairs of factors. In the fourth design, each of the four pairs appears twice across the six pairs of two-level factors, and each of the eight pairs appears once across the four pairs that involve the four-level factor and each of the two-level factors. In the fifth design, each of the nine pairs appears once across the six pairs of factors. Since orthogonal arrays are both balanced and orthogonal, they are 100% efficient and optimal. Efficiency, which is explained starting on page 62, is a measure of the goodness of the experimental design.

The term "orthogonal array," is sometimes used imprecisely. It is correctly used to refer to designs that are both orthogonal and balanced, and hence optimal. However, the term is sometimes incorrectly used to refer to designs that are orthogonal but not balanced, and hence not 100% efficient and sometimes not even optimal. Such designs are made from orthogonal arrays by "coding down." Coding down consists of replacing an a-level factor by a b-level factor where $a > b$. For example, coding down might replace a three-level factor by a two-level factor (e.g., replacing all 3's with 1's), or a four-level factor by a three-level factor (e.g., replacing all 4's with 1's), or a five-level factor by a three-level factor (e.g.,

replacing all 4's with 1's and all 5's with 3's), and so on. Coding down introduces imbalance. It would be more precise to call designs such as these something like "unbalanced arrays."

Orthogonal designs are often practical for main-effects models when the number of factors is small and the number of levels of each factor is small. However, there are some situations in which orthogonal designs might not be practical, such as when not all combinations of factor levels are feasible or make sense, the desired number of runs is not available in an orthogonal design, or a model with interactions is being used. When an orthogonal and balanced design is not practical, you must make a choice. One choice is to change the factors and levels to fit some known orthogonal design. This choice is undesirable for obvious reasons. When a suitable orthogonal and balanced design does not exist, efficient nonorthogonal designs can be used instead. Often, these designs are superior to orthogonal but unbalanced designs.

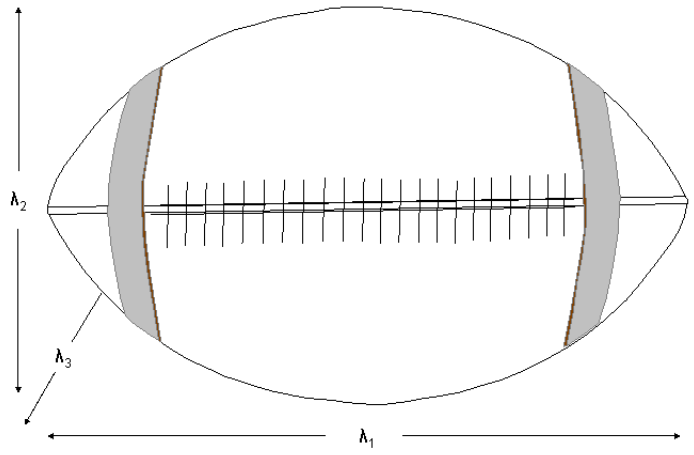
Nonorthogonal designs, where some coefficients might be slightly correlated, can be used when orthogonal designs are not available. You do not have to adapt every experiment to fit some known orthogonal array. First you choose the number of runs. You are not restricted by the sizes of orthogonal arrays, which come in specific numbers of runs for specific numbers of factors with specific numbers of levels. Then you specify the levels of each of the factors and the number of runs. Algorithms for generating efficient designs select a set of *design points* from a set of *candidate points* (such as a full-factorial design or a fractional-factorial design). Design points are selected or replaced when that operation increases an efficiency criterion.*

Throughout this book, we use the `%MktEx` macro to find good, efficient experimental designs. It has specialized algorithms for finding both orthogonal and nonorthogonal designs. The `%MktEx` macro is a part of the SAS autocall library. See page 803 for information about installing and using SAS autocall macros. There are many different construction methods for creating orthogonal arrays. In practice, since we always use the `%MktEx` macro to make them, we never have to worry about the different methods. However, if you would like to learn more, you can read the section beginning on page 95.

Eigenvalues, Means, and Footballs

The next section discusses experimental design efficiency. To fully understand that section, you need some basic understanding of *eigenvalues* and various types of means or averages. This section explains these and other concepts, but without a high degree of mathematical rigor. An American football provides a nice visual image for understanding the eigenvalues of a matrix.

*In the coordinate exchange algorithm that the `%MktEx` macro uses, the full-factorial candidate set is virtual. It is never created, but all combinations in it are available for consideration. In contrast, in the Modified Fedorov search that PROC OPTEX does (either by itself or through the `%MktEx` macro), an explicit full-factorial or fractional-factorial candidate set is created and searched.



The rows or columns of a matrix can be thought of as a swarm of points in Euclidean space. Similarly, a football consists of a set of points in a space. Sometimes, it is helpful to get an idea of the size of your group of points. For a football, you might think of three measures because a football is a three-dimensional object: the longest length from end to end, the height in the center and perpendicular to the length, and finally the width, which for a fully-inflated football is the same as the height. One can do similar things for matrices, and that is where eigenvalues come in. For many of us, eigenvalues are most familiar from factor analysis and principal component analysis. In principal component analysis, one rotates a cloud of points to a principal axes orientation, just as this football has been rotated so that its longest dimension is horizontally displayed. The principal components correspond to: the longest squared length, the second longest squared length perpendicular or orthogonal to the first, the third longest squared length orthogonal to the first two, and so on. The eigenvalues are the variances of the principal components and are proportional to squared lengths. The eigenvalues provide a set of measures of the size of a matrix, just as the lengths provide a set of measures of the size of a football.

The following matrices show a small experimental design, the coded design $\mathbf{X}$, the sum of squares and cross products matrix $\mathbf{X}'\mathbf{X}$, the matrix inverse $(\mathbf{X}'\mathbf{X})^{-1}$, and the eigenvalues of the inverse, Λ:

Design

1	1	1
1	2	2
1	2	2
2	1	2
2	2	1
2	2	1

$\mathbf{X}$

1	1	1	1
1	1	−1	−1
1	1	−1	−1
1	−1	1	−1
1	−1	−1	1
1	−1	−1	1

$\mathbf{X}'\mathbf{X}$

6	0	−2	0
0	6	0	−2
−2	0	6	0
0	−2	0	6

$(\mathbf{X}'\mathbf{X})^{-1}$

0.188	0.000	0.063	0.000
0.000	0.188	0.000	0.063
0.063	0.000	0.188	0.000
0.000	0.063	0.000	0.188

Λ

1/4	0	0	0
0	1/4	0	0
0	0	1/8	0
0	0	0	1/8

$\mathbf{X}$ is made from the raw design by coding, which in this case simply involves creating an intercept, appending the design, and replacing 2 with –1. See page 73 for more about coding. The $\mathbf{X}'\mathbf{X}$ matrix comes from a matrix multiplication of the transpose of $\mathbf{X}$ times $\mathbf{X}$. For example, the –2 in the first row comes from $\mathbf{x}_1'\mathbf{x}_3 = (1\ 1\ 1\ 1\ 1\ 1)'(1\ -1\ -1\ \ 1\ -1\ -1) = 1\times1+1\times-1+1\times-1+1\times1+1\times-1+1\times-1 = -2$. Explaining the computations involved in finding the matrix inverse and eigenvalues is beyond the scope of this chapter; however, they are explained in many linear algebra and multivariate statistics

texts.

The trace is the sum of the diagonal elements of a matrix, which for $(\mathbf{X}'\mathbf{X})^{-1}$, is both the sum of the variances and the sum of the eigenvalues: $\text{trace}\,((\mathbf{X}'\mathbf{X})^{-1}) = \text{trace}\,(\Lambda) = 0.188+0.188+0.188+0.188 = 1/4 + 1/4 + 1/8 + 1/8 = 0.75$. The determinant of $(\mathbf{X}'\mathbf{X})^{-1}$, denoted $|(\mathbf{X}'\mathbf{X})^{-1}|$, is the product of the eigenvalues and is 0.0009766: $|(\mathbf{X}'\mathbf{X})^{-1}| = |\Lambda| = 1/4 \times 1/4 \times 1/8 \times 1/8 = 0.0009766$. The determinant of a matrix is geometrically interpreted in terms of the volume of the space defined by the matrix. The formula for the determinant of a nondiagonal matrix is complicated, so when the eigenvalues are known, determinants are more conveniently expressed as a function of the eigenvalues.

Given a set of eigenvalues, or any set of numbers, we frequently want to create a single number that summarizes the values in the set. The most obvious way to do this is to compute the average or arithmetic mean. The familiar *arithmetic mean* is found by adding together p numbers and then dividing by p. A trace, divided by p, is an arithmetic mean. The arithmetic mean is an enormously popular and useful statistic, however it is not the only way to average numbers. The less familiar *geometric mean* is found by multiplying p numbers together and then taking the *pth* root of the product. The *pth* root of a determinant is a geometric mean of eigenvalues. To better understand the geometric mean, consider an example. Say your investments increased by 7%, 5%, and 12% over a three year period. The arithmetic mean of these numbers, $(7 + 5 + 12)/3 = 8\%$, is *not* the average increase that would have come if the investments had increased by the same amount every year. To find that average, we need the geometric mean of the ratios of the current to previous values: $(1.07 \times 1.05 \times 1.12)^{1/3} = 1.0796$. The average increase is 7.96%.

Experimental Design Efficiency

This section discusses precisely what is meant by an efficient design. While this section is important, it is not critical that you understand every mathematical detail. The concepts are explained again in a more intuitive and less mathematical way in the next section. Also, see page 243 for more information about efficient experimental designs.

The goodness or *efficiency* of an experimental design can be quantified. Common measures of the efficiency of an $(N_D \times p)$ design matrix $\mathbf{X}$ are based on the *information matrix* $\mathbf{X}'\mathbf{X}$. The variance-covariance matrix of the vector of parameter estimates $\hat{\boldsymbol{\beta}}$ in a least-squares analysis is proportional to $(\mathbf{X}'\mathbf{X})^{-1}$. More precisely, it equals $\sigma^2(\mathbf{X}'\mathbf{X})^{-1}$. The variance parameter, σ^2, is an unknown constant. Since σ^2 is constant, it can be ignored (or assumed to equal one) in the discussion that follows. The diagonal elements of $(\mathbf{X}'\mathbf{X})^{-1}$ are the parameter estimate variances, and the standard errors are the square roots of the variances. Since they depend only on $\mathbf{X}$ (and σ^2), they can be reported by design software before any data are collected. An efficient design has a "small" variance matrix, and the eigenvalues of $(\mathbf{X}'\mathbf{X})^{-1}$ provide measures of its "size." The process of minimizing the eigenvalues or variances only depends on the selection of the entries in $\mathbf{X}$ not on the unknown σ^2 parameter.

The two most prominent efficiency measures are based on quantifying the idea of matrix size by averaging (in some sense) the eigenvalues or variances. *A-efficiency* is a function of the arithmetic mean of the eigenvalues, which is also the arithmetic mean of the variances, and is given by $\text{trace}\,((\mathbf{X}'\mathbf{X})^{-1})/p$. A-efficiency is perhaps the most obvious measure of efficiency. As the variances get smaller and the arithmetic mean of the variances of the parameter estimates goes down, A-efficiency goes up. However, as we learned in the previous section, there are other averages to consider. *D-efficiency* is a function of the geometric mean of the eigenvalues, which is given by $|(\mathbf{X}'\mathbf{X})^{-1}|^{1/p}$. Both D-efficiency and A-efficiency are based on the idea of average variance, but in different senses of the word "average." We

usually use *D*-efficiency for two reasons. It is the easier and faster of the two for a computer program to optimize. Furthermore, relative *D*-efficiency, the ratio of two *D*-efficiencies for two competing designs, is invariant under different coding schemes. This is not true with *A*-efficiency. A third common efficiency measure, *G-efficiency*, is based on σ_M, the maximum standard error for prediction over the candidate set. All three of these criteria are convex functions of the eigenvalues of $(\mathbf{X}'\mathbf{X})^{-1}$ and hence are usually highly correlated.

For all three criteria, if a balanced and orthogonal design exists, then it has optimum efficiency; conversely, the more efficient a design is, the more it tends toward balance and orthogonality. A design is balanced and orthogonal when $(\mathbf{X}'\mathbf{X})^{-1}$ is diagonal and equals $\frac{1}{N_D}\mathbf{I}$ for a suitably coded $\mathbf{X}$. A design is orthogonal when the submatrix of $(\mathbf{X}'\mathbf{X})^{-1}$, excluding the row and column for the intercept, is diagonal; there might be off-diagonal nonzeros for the intercept. A design is balanced when all off-diagonal elements in the intercept row and column are zero. How we choose $\mathbf{X}$ determines the efficiency of our design. Ideally, we want to choose $\mathbf{X}$ so that the design is balanced and orthogonal or at least very nearly so. More precisely, we want to choose $\mathbf{X}$ so that we maximize efficiency.

These measures of efficiency can be scaled to range from 0 to 100 (see pages 73–73 for the orthogonal coding of $\mathbf{X}$ that must be used with these formulas) as follows:

$$\begin{aligned} A\text{-efficiency} &= 100 \times \frac{1}{N_D \text{ trace}\left((\mathbf{X}'\mathbf{X})^{-1}\right)/p} \\ D\text{-efficiency} &= 100 \times \frac{1}{N_D \left|(\mathbf{X}'\mathbf{X})^{-1}\right|^{1/p}} \\ G\text{-efficiency} &= 100 \times \frac{\sqrt{p/N_D}}{\sigma_M} \end{aligned}$$

These efficiencies measure the goodness of a design relative to hypothetical orthogonal designs that might not exist, so they are not useful as absolute measures of design efficiency. Instead, they should be used relatively, to compare one design to another for the same situation. Efficiencies that are not near 100 might be perfectly satisfactory. When *D*-efficiency is 0, one or more parameters cannot be estimated. When *D*-efficiency is 100, then the design is balanced and orthogonal. Values in between mean that all of the parameters can be estimated, but with less than optimal precision. Precisely what this means can vary from design to design. It might be that all of the variances are larger than the optimal value, or it might be that only some are larger than the optimal value. When the standardized orthogonal contrast coding on pages 73–73 is used, *D*-efficiency computed this way can never vary outside the 0 to 100 range. The range for *D*-efficiency can be quite different (either larger or smaller) with other coding schemes.

Experimental Design: Rafts, Rulers, Alligators, and Stones

A good physical metaphor for understanding experimental design and design efficiency is a raft. A raft is a flat boat, often supported by flotation devices attached to the corners. The raft in Figure 3 has four Styrofoam blocks under each corner, which provide nice stability and equal support. This raft corresponds to 2 two-level factors from a 16-run design (see Table 1). The four corners correspond to each of the four possible combinations of 2 two-level factors, and the four blocks under the raft form an up-side-down bar chart showing the frequencies for each of the four combinations. Looking at the

Table 1
Two-Level Factors in 16 Runs

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	2	2	2	2	1	1	2	2	1	2	2
1	1	2	2	1	1	2	2	1	2	1	2	2	1	2
1	1	2	2	2	2	1	1	1	2	2	1	2	2	1
1	2	1	2	1	2	1	2	2	2	2	2	1	1	1
1	2	1	2	2	1	2	1	2	2	1	1	1	2	2
1	2	2	1	1	2	2	1	2	1	2	1	2	1	2
1	2	2	1	2	1	1	2	2	1	1	2	2	2	1
2	1	1	2	1	2	2	1	2	1	1	2	2	2	1
2	1	1	2	2	1	1	2	2	1	2	1	2	1	2
2	1	2	1	1	2	1	2	2	2	1	1	1	2	2
2	1	2	1	2	1	2	1	2	2	2	2	1	1	1
2	2	1	1	1	1	2	2	1	2	2	1	2	2	1
2	2	1	1	2	2	1	1	1	2	1	2	2	1	2
2	2	2	2	1	1	1	1	1	1	2	2	1	2	2
2	2	2	2	2	2	2	2	1	1	1	1	1	1	1

Table 2
Unbalanced
$2^2 3^3$ in 18 Runs

1	1	1	1	1
1	1	2	3	3
1	1	1	3	2
1	1	3	2	3
1	2	2	2	1
1	2	3	1	2
1	1	1	2	3
1	1	3	3	2
1	1	2	2	2
1	1	3	1	1
1	2	1	3	1
1	2	2	1	3
2	1	2	1	2
2	1	3	2	1
2	1	1	1	3
2	1	2	3	1
2	2	1	2	2
2	2	3	3	3

Table 3
Making Twos from Threes

1		1	1		1
1		1	1		1
1		1	2		2
1		1	2		2
1		1	3		1
1		1	3		1
2		2	1		1
2		2	1		1
2	→	2	2	→	2
2		2	2		2
2		2	3		1
2		2	3		1
3		1	1		1
3		1	1		1
3		1	2		2
3		1	2		2
3		1	3		1
3		1	3		1

Table 4
Optimal
$2^2 3^3$ in 18 Runs

1	1	1	2	2
1	1	2	3	2
1	1	3	1	2
1	1	3	2	3
1	2	1	1	3
1	2	1	3	1
1	2	2	1	3
1	2	2	2	1
1	2	3	3	1
2	1	1	2	1
2	1	1	3	3
2	1	2	1	1
2	1	2	3	3
2	1	3	1	1
2	2	1	1	2
2	2	2	2	2
2	2	3	2	3
2	2	3	3	2

raft, one can tell that the first factor is balanced (equal support on the left and on the right) as is the second (equal support in the front and the back). The design is also orthogonal (equal support in all four corners). Making a design that supports your research conclusions is like making a raft for use in water with alligators. You want good support no matter on which portion of the raft you find yourself. Similarly, you want good support for your research and good information about all of your product attributes and attribute levels.

Now compare the raft in Figure 3 to the one shown in Figure 4. The Figure 4 raft corresponds to the two-level factors in the design shown in Table 2. This design has 18 runs, and since 18 cannot be divided by 2×2, a design that is both balanced and orthogonal is not possible. Clearly, this design is not balanced in either factor. There are twelve blocks on the left and only six on the right, and there are twelve blocks on the back and only six on the front. This design *is* however orthogonal, because the corner frequencies are proportional. These two factors can be made from 2 three-level factors in the L_{18} design, which has up to 7 three-level factors. See Table 3. The three-level factors are all orthogonal, and recoding levels, replacing 3 with 1, preserves that orthogonality at the cost of decreased efficiency and a horrendous lack of balance. See Table 3 on page 250 for the information and variance matrices for the Figure 4 raft.

Finally, compare the raft in Figure 4 to the one shown in Figure 5. Both of these correspond to designs with two-level factors in 18 runs. The Figure 5 raft corresponds to a design that is balanced. There are nine blocks on the left and nine on the right, and there are nine blocks on the back and nine on the front. The design is not however orthogonal since the corner frequencies are 4, 5, 4, and 5, which are not equal or even proportional. Ideally, you would like a raft such as the one in Figure 3, which corresponds to a design that is both orthogonal and balanced. However, to have both two or more three-level and two or more two-level factors, you need at least 36 runs. More precisely, you need a multiple of 36 runs (36, 72, 108, and so on). In 18 runs, you can make an optimal design (that is, optimal relative to all other designs in 18 runs), such as the one in Table 4 and Figure 5, that provides good support under all corners but not perfectly equal support. See Tables 3 and 4 in the next chapter on pages 251 and 250 for the information and variance matrices for the Figure 4 and 5 rafts.

On which raft would you rather walk? The Figure 3 and Figure 5 rafts are going to be reasonably stable. The Figure 3 raft is in fact optimal, given exactly 16 Styrofoam blocks, and the Figure 5 raft is also optimal, given exactly 18 Styrofoam blocks. The Figure 4 raft might be fine if you stay in the back left corner, but take one step, and you have little support. An experimental design provides support for your research just as a raft provides support for a person crossing a river. In both raft and design terms, the problem is one of stability and support. In design terms, part of your results are not stable due to a lack of information about the front right combination in your factorial design. How confident can you be in your results when you have so little information about some of your product attribute levels?

The Table 4 design (Figure 5 raft) brings to mind the story of the cup containing exactly one half cup of water. The optimist sees the cup as half full, and the pessimist sees it as half empty. In the design, the optimist sees a little extra support in the back left and front right corners. The pessimist sees a little less support in the front left and back right corners. Either way, all available resources (design points) are optimally allocated to maximize efficiency and stability. What you would really like is both balance and orthogonality. However, you cannot get both in 18 runs, because 2×2 does not divide 18. Still, you can do pretty well. Like the line in the song, "You can't always get what you want, but if you try sometimes, you just might find, you get what you need" (Jagger and Richards 1969). What you want is orthogonality and balance. What you need is good stability. Efficient designs can give you what you need even when what you want is impossible.

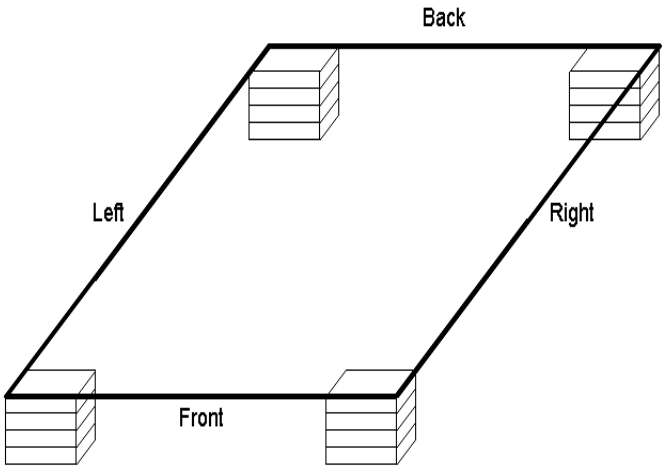


Figure 3 16 Runs, Orthogonal and Balanced

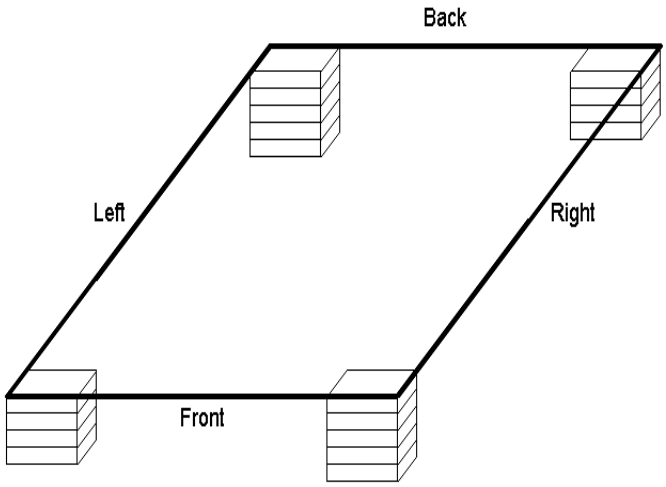


Figure 5 18 Runs, Balanced and Almost Orthogonal

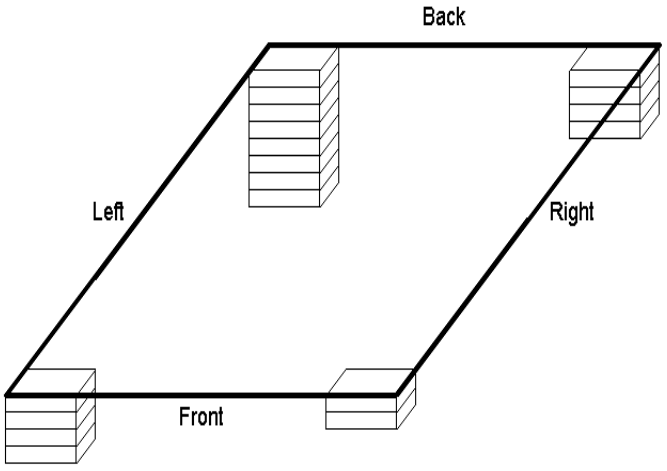
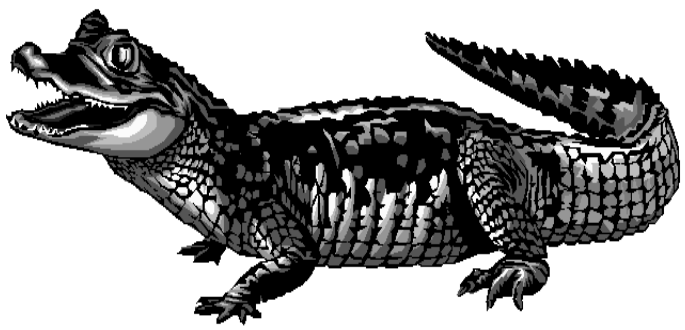


Figure 4 18 Runs, Orthogonal but not Balanced



The Table 2 and Figure 4 design might seem like just a "straw man," something that we build up just so that we can knock it down. However, this design was widely used in the past, and in fact, in spite of the fact that its deficiencies have been known for over 16 years (Kuhfeld, Tobias, and Garratt 1994), it is *still* used in some sources as a text-book example of a good design. In fact, it is a text-book example of how *not* to make designs. It is an example of what can happen when you choose orthogonality as the be-all-end-all design criterion, ignoring both balance and statistical efficiency. It is also an example of what can happen when you construct designs from a small, inferior, and incomplete catalog instead of using a comprehensive designer. Even among orthogonal designs, it is not optimal (see pages 249–251). If we achieve perfect orthogonality *and* balance, our design is optimal and has maximum efficiency. The key consideration is that maximizing statistical efficiency minimizes the variability of our parameter estimates, and that is what we want to achieve. Recall that for a linear model, the variance-covariance matrix of the vector of parameter estimates is proportional to $(\mathbf{X}'\mathbf{X})^{-1}$. Maximizing efficiency minimizes those variances, covariances, and hence standard errors. These designs are discussed in more detail, including an examination of their variance matrices, starting on page 249.

How we choose our design, our $\mathbf{X}$ values, affects the variability of our parameter estimates. Previously, we talked about eigenvalues and the variance matrix, which provides a mathematical representation of the idea that we choose our $\mathbf{X}$ values so that our parameter estimates have small standard errors. Now, we will discuss this less mathematically. Imagine that we are going to construct a very simple experiment. We are interested in investigating the purchase interest of a product as a function of its price. We design an experiment with two prices, \$1.49 and \$1.50 and ask people to rate how interested they are in the products at those two prices. We plot the results with price on the horizontal axis and purchase interest on the vertical axis. We find that the price effect is minimal. See Figure 6. Now imagine that the line is a ruler and the two dots are your fingers. Your fingers are the design points providing support for your research. Your fingers are close together because in our research design, we chose two prices that are close together. Furthermore, imagine that there is a small amount of error in your data, that is error in the reported purchase interest, which is in the vertical direction. To envision this, move your fingers up and down, just a little bit. What happens to your slope and intercept as you do this?* They vary a lot! This is not a function of your data; it is a function of your design being inefficient because you did not adequately sample a reasonable price range.

Next, let's design a similar experiment, but this time with prices of \$0.99 and \$1.99. See Figure 7. Imagine again that the line is a ruler and the two dots are your fingers, but this time they are farther apart. Again, move your fingers up and down, just a little bit. What happens to your slope and intercept as you do this? Not very much; they change a little bit. The standard errors for Figure 6 are much greater than the standard errors for Figure 7. **How you choose your design points affects the stability of your parameter estimates. This is the same lesson that the mathematics involving $(\mathbf{X}'\mathbf{X})^{-1}$ gives you. You want to choose your X's so that efficiency is maximized and the variability of your parameter estimates is minimized.** This example does not imply, however, that you should pick prices such as \$0.01 and \$1,000,000,000. Your design levels need to make sense for the product.

Conjoint, Linear Model, and Choice Designs

Consider a simple example of three brands each at two prices. We always use linear-model theory to guide us in creating designs for a full-profile conjoint studies. Usually we pick orthogonal arrays for

*I encourage you to actually try this and see what happens! It is a great physical demonstration showing that you you choose $\mathbf{X}$ affects the stability of the parameter estimates.

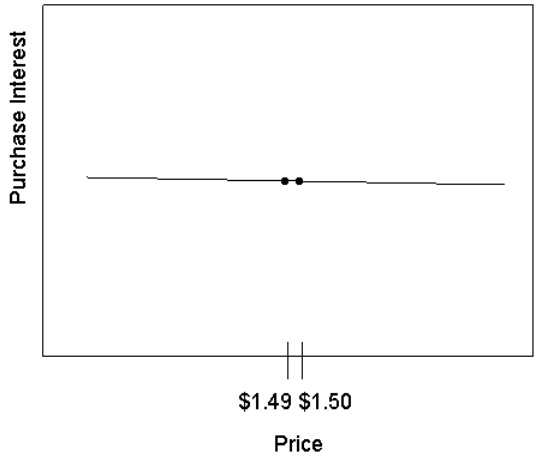


Figure 6 Prices Close Together

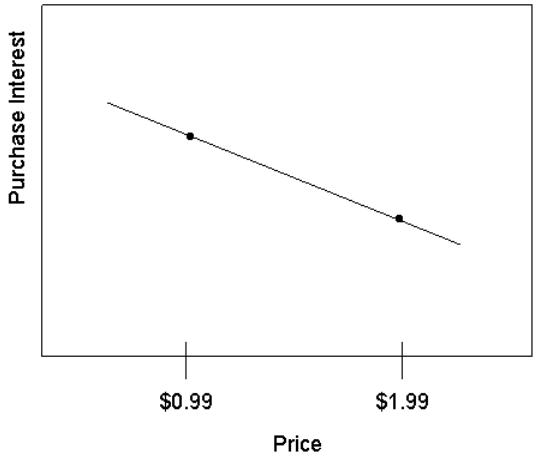


Figure 7 Prices Farther Apart

conjoint studies. For choice modeling, the process is somewhat different. We sometimes use linear-model theory to create a "linear arrangement of a choice design" (or *linear arrangement* for short) from which we then construct a true *choice design* to use in a discrete choice study. The linear arrangement does not correspond to a real and useful linear model. Rather the linear arrangement of a choice design simply provides a convenient way to generate and evaluate choice designs for certain problems such as problems with alternative-specific effects. An example of a conjoint and a linear arrangement of a choice design are as follows:

Full-Profile
Conjoint Design

Brand	Price
1	1.99
1	2.99
2	1.99
2	2.99
3	1.99
3	2.99

Linear Arrangement
of a Choice Design

Brand 1 Price	Brand 2 Price	Brand3 Price
1.99	1.99	1.99
1.99	2.99	2.99
2.99	1.99	2.99
2.99	2.99	1.99

This conjoint design has two factors, brand and price, and six runs or product profiles. Subjects are shown each combination, such as brand 1 at $1.99 and are asked to report purchase interest through either a rating (for example, on a 1 to 9 scale) or a ranking of the six profiles.

The linear arrangement of the choice design for a pricing study with three brands has three factors (Brand 1 Price, Brand 2 Price, and Brand 3 Price) and one row for each choice set. More generally, the linear arrangement has one factor for each attribute of each alternative (or brand), and brand is not a factor in the linear arrangement. Each brand is a "bin" into which its factors are collected. Subjects see these sets of products and report which one they would choose (and implicitly, which ones they

Table 5

Linear Arrangement

1	2	3
1.99	1.99	1.99
1.99	2.99	2.99
2.99	1.99	2.99
2.99	2.99	1.99

Table 6

Choice Design

Set	Brand	Price
1	1	1.99
	2	1.99
	3	1.99
2	1	1.99
	2	2.99
	3	2.99
3	1	2.99
	2	1.99
	3	2.99
4	1	2.99
	2	2.99
	3	1.99

Table 7

Choice Design Coding

	Brand Effects			Brand by Price		
Set	1	2	3	1	2	3
1	1	0	0	1.99	0	0
	0	1	0	0	1.99	0
	0	0	1	0	0	1.99
2	1	0	0	1.99	0	0
	0	1	0	0	2.99	0
	0	0	1	0	0	2.99
3	1	0	0	2.99	0	0
	0	1	0	0	1.99	0
	0	0	1	0	0	2.99
4	1	0	0	2.99	0	0
	0	1	0	0	2.99	0
	0	0	1	0	0	1.99

would not choose). However, before we fit the choice model, we need to construct a true choice design from the linear arrangement and code the choice design. See Tables 5, 6, and 7.

The linear arrangement has one row per choice set. The choice design has three rows for each choice set, one for each alternative. The linear arrangement and the choice design contain different arrangements of the exact same information. In the linear arrangement, brand is a bin into which its factors are collected (in this case one factor per brand). In the choice design, brand and price are both factors, because the design has been rearranged from one row per choice set to one row per alternative per choice set. For this problem, with only one attribute per brand, the first row of the choice design matrix corresponds to the first value in the linear arrangement, Brand 1 at $1.99. The second row of the choice design matrix corresponds to the second value in the linear arrangement, Brand 2 at $1.99. The third row of the choice design matrix corresponds to the third value in the linear arrangement, Brand 3 at $1.99, and so on.

A design is coded by replacing each factor with one more columns of *indicator variables* (which are often referred to as "dummy variables") or other codings. In this example, a brand factor is replaced by the three binary variables. We go through how to construct and code linear and choice designs many times in the examples using a number of different codings. For now, just notice that the conjoint design is different from the linear arrangement of the choice design, which is different from the customary arrangement of the choice design. They are not even the same size! You can use the `%MktEx` macro to make the linear arrangement, the `%MktRoll` macro to convert it into a choice design, the `%ChoicEff` macro to evaluate the choice design, and the TRANSREG procedure to code the choice design for analysis. Alternatively, you can use the `%ChoicEff` macro to construct and evaluate the choice design directly and never go through the linear arrangement phase. Both approaches are discussed in detail in this chapter and this book.

Table 8
The First Six Choice Sets (Out of 36 Total)

Linear Arrangement of a Choice Design									Choice Design			
Brand 1			Brand 2			Brand 3						
x1	x2	x3	x4	x5	x6	x7	x8	x9		x1	x2	x3
16 oz.	0.89	Twist Up	24 oz.	0.89	Twist Off	20 oz.	0.99	Pop Up	1	16 oz.	0.89	Twist Up
									2	24 oz.	0.89	Twist Off
									3	20 oz.	0.99	Pop Up
20 oz.	0.99	Pop Up	24 oz.	0.89	Twist Up	20 oz.	0.89	Twist Off	1	20 oz.	0.99	Pop Up
									2	24 oz.	0.89	Twist Up
									3	20 oz.	0.89	Twist Off
20 oz.	0.89	Twist Up	20 oz.	0.99	Twist Off	16 oz.	0.89	Twist Off	1	20 oz.	0.89	Twist Up
									2	20 oz.	0.99	Twist Off
									3	16 oz.	0.89	Twist Off
20 oz.	0.89	Twist Up	16 oz.	0.99	Twist Up	24 oz.	0.99	Pop Up	1	20 oz.	0.89	Twist Up
									2	16 oz.	0.99	Twist Up
									3	24 oz.	0.99	Pop Up
16 oz.	0.89	Twist Off	24 oz.	0.99	Pop Up	16 oz.	0.99	Twist Off	1	16 oz.	0.89	Twist Off
									2	24 oz.	0.99	Pop Up
									3	16 oz.	0.99	Twist Off
24 oz.	0.99	Twist Off	16 oz.	0.89	Twist Off	16 oz.	0.89	Pop Up	1	24 oz.	0.99	Twist Off
									2	16 oz.	0.89	Twist Off
									3	16 oz.	0.89	Pop Up

The effects that are labeled in Table 7 as "Brand by Price" are called *alternative-specific effects*. They are coded so that the price effect can be different for each alternative or brand.

A slightly more involved illustration of the differences between the linear and final version of a choice design is shown in Table 8. This example has three brands and three alternatives, one per brand. The product is sports beverages, and they are available in three sizes, at two prices with three different types of tops including a top that pops straight up for drinking, an ordinary twist off top, and cap that twists up for drinking without coming off.

The linear arrangement has one row per choice set. The full choice design has 36 choice sets. There is one factor for each attribute of each alternative. This experiment has three alternatives, one for each of three brands, and three attributes per alternative. The first goal is to make a linear arrangement of a choice design where each attribute, both within and between alternatives, is orthogonal and balanced, or at least very nearly so. Brand is the bin into which the linear factors are collected, and it becomes an actual attribute in the choice design. The right partition of the table shows the choice design. The `x1` attribute in the choice design is made from `x1`, `x4`, and `x7`, in the linear arrangement. These are the three size factors. Similarly, `x2` is made from `x2`, `x5`, and `x8`, in the linear arrangement. These are the three price factors. Finally, `x3` is made from the three top factors, `x3`, `x6`, and `x9`. This information is conveyed in the following table:

	x1	x2	x3
Brand 1	x1	x2	x3
Brand 2	x4	x5	x6
Brand 3	x7	x8	x9

This table provides the rules for constructing the choice design from the linear arrangement. We call this the *key* to constructing the choice design.

Blocking the Choice Design

The sports beverage example from Table 8 has 36 choice sets. Many choice designs are even larger. Even 36 choice sets might be too many judgments for one subject to make. Often larger designs are broken up into subsets or *blocks*. The number of blocks depends on the number of choice sets and the complexity of the choice task. For example, 36 choice sets might be small enough that no blocking is necessary, or instead, they can be divided into 2 blocks of size 18, 3 blocks of size 12, 4 blocks of size 9, 6 blocks of size 6, 9 blocks of size 4, 12 blocks of size 3, 18 blocks of size 2, or even 36 blocks of size 1. Technically, subjects *should* each see exactly one choice set. Showing subjects more than one choice set is economical, and in practice, most researchers almost always show multiple choice sets to each subject. The number of sets shown does not change the expected utilities, however, it does affect the covariance structure. Sometimes, attributes are highly correlated within blocks, particularly with small block sizes, but that is not a problem as long as they are not highly correlated over the entire design.

Efficiency of a Choice Design

All of the efficiency theory discussed so far concerns linear models ($\hat{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}}$). In linear models, the parameter estimates $\hat{\boldsymbol{\beta}}$ have variances proportional to $(\mathbf{X}'\mathbf{X})^{-1}$. A choice model has the following form

$$p(c_i|C) = \frac{\exp(U(c_i))}{\sum_{j=1}^{m}\exp(U(c_j))} = \frac{\exp(\mathbf{x}_i\boldsymbol{\beta})}{\sum_{j=1}^{m}\exp(\mathbf{x}_j\boldsymbol{\beta})}$$

The probability that an individual will choose one of the m alternatives, c_i, from choice set C is a nonlinear function of $\mathbf{x}_i$, the vector of coded attributes, and $\boldsymbol{\beta}$, the vector of unknown parameters. The variances of the parameter estimates in the discrete choice multinomial logit model are given by

$$V(\hat{\beta}) = \left[\Sigma_{k=1}^{n} N \left[\frac{\Sigma_{j=1}^{m}\exp(x_j'\beta)x_j x_j'}{\Sigma_{j=1}^{m}\exp(x_j'\beta)} - \frac{(\Sigma_{j=1}^{m}\exp(x_j'\beta)x_j)(\Sigma_{j=1}^{m}\exp(x_j'\beta)x_j)'}{(\Sigma_{j=1}^{m}\exp(x_j'\beta))^2}\right]\right]^{-1}$$

with

m — brands
n — choice sets
N — people
x_j — the attributes of the *jth* alternative of the *kth* choice set

In the choice model, ideally we would like to pick $\mathbf{x}$'s that make this variance matrix "small." Unfortunately, we cannot do this unless we know $\boldsymbol{\beta}$, and if we knew $\boldsymbol{\beta}$, we would not need to do the experiment. However, in the chair example on pages 556–579, we see how to make an efficient choice design when we are willing to make assumptions about $\boldsymbol{\beta}$ other than $\boldsymbol{\beta} = \mathbf{0}$.

Because we do not know $\boldsymbol{\beta}$, we often create experimental designs for choice models using efficiency criteria for linear models. We make a good design for a linear model by picking $\mathbf{x}$'s that minimize a function of $(\mathbf{X}'\mathbf{X})^{-1}$ and then convert our linear arrangement into a choice design. Certain assumptions must be made before applying ordinary general-linear-model theory to problems in marketing research. The usual goal in linear modeling is to estimate parameters and test hypotheses about those parameters. Typically, independence and normality are assumed. In full-profile conjoint study, each subject rates all products and separate ordinary-least-squares analyses are run for each subject. This is not a standard general linear model; in particular, observations are not independent and normality cannot be assumed. Discrete choice models, which are nonlinear, are even more removed from the general linear model.

Marketing researchers often make the critical assumption that designs that are good for general linear models are also good designs for conjoint and discrete choice models. We also make this assumption. We assume that an efficient design for a linear model is a good design for the multinomial logit model used in discrete choice studies. We assume that if we create the linear arrangement (one row per choice set and all of the attributes of all of the alternatives comprise that row), and if we strive for linear-model efficiency (near balance and orthogonality), then we will have a good design for measuring the utility of each alternative and the contributions of the factors to that utility. When we construct choice designs in this way, our designs have two nice properties. 1) Each attribute level occurs equally often (or at least nearly equally often) for each attribute of each alternative across all choice sets. 2) Each attribute is independent of every other attribute (or at least nearly independent), both those in the same alternative and those in all of the other alternatives. The design techniques discussed in this book, based on the assumption that linear arrangement efficiency is a good surrogate for choice design goodness, have been used quite successfully in the field for many years.

In some of the examples, we use the `%MktEx` macro to create a linear arrangement, from which we construct our final choice design. This seems to be a good, safe strategy. It is a good strategy because it makes designs where all attributes, both within and between alternatives, are orthogonal or at least nearly so. It is safe in the sense that you have enough choice sets and collect the right information so that very complex models, including models with alternative-specific effects, availability effects, and cross-effects, can be fit. However, it is good to remember that when you run the `%MktEx` macro and you get an efficiency value, it corresponds to the linear arrangement (which does not correspond directly to any real model of interest), not the efficiency of the design in the context of the choice model. The linear model efficiency is a surrogate for the criterion of interest, the efficiency of the choice design, which is unknowable unless you know the parameters. Other examples optimize choice design efficiency based on an assumed parameter vector.

Coding, Efficiency, Balance, and Orthogonality

This section discusses coding in the context of linear models. However, the coding schemes used in this section apply equally to choice models. The next section builds on the materials discussed in this section and discusses coding in the context of a choice model.

Coding is the process or replacing our design factors by the set of indicator or coded variables that are actually used when the model is fit. We mention on page 63 that we use a special orthogonal coding of $\mathbf{X}$ (specifically, the standardized orthogonal contrast coding) when computing design efficiency. This section shows that coding and other codings. Even if you gloss over the mathematical details, this section is informative, because it provides insights into coding and the meaning of 100% efficiency and less than 100% efficient designs.

Table 9 displays the nonorthogonal less-than-full-rank *binary* or *indicator* codings for two-level through five-level factors. This is also known as "GLM coding" since it is the coding that is used by default in PROC GLM. It is requested with PROC TRANSREG as follows: `class(x / zero=none)`. In other procedures (such as PROC PHREG and PROC LOGISTIC), it is requested by specifying `param=glm`. There is one column for each level, and the coding contains a 1 when the level matches the column and a zero otherwise. We use these codings in many places throughout the examples.

Table 10 displays the nonorthogonal full-rank binary (or indicator or *reference cell* codings for two-level through five-level factors. This coding is like the full-rank coding shown previously, except that the column corresponding to the *reference level* has been dropped. It is requested with PROC TRANSREG as follows: `class(x)`. In other procedures (such as PROC PHREG and PROC LOGISTIC), it is requested by specifying `param=reference`. Frequently, the reference level is the last level, but it can be any level. We use these codings in many places throughout the examples.

Table 11 displays the nonorthogonal *effects coding* or *deviations from means coding* for two-level* through five-level factors. The effects coding differs from the full-rank binary coding in that the former always has a –1 to indicate the reference level. It is requested with PROC TRANSREG as follows: `class(x / effects)`. In other procedures (such as PROC PHREG and PROC LOGISTIC), it is requested by specifying `param=effect`. We use these codings in many places throughout the examples.

Table 12 displays the *orthogonal contrast coding* for two-level through five-level factors. They are the same as the orthogonal codings that are discussed in detail next, except that this version has not been scaled. Hence, all values are integers. It is requested with PROC TRANSREG as follows: `class(x / orthogonal)`. This coding is not available in most other procedures. Notice that the codings for each level form a contrast—the *ith* level versus all of the preceding levels and the last level.

Table 13 displays the *standardized orthogonal contrast coding*, for two-level through five-level factors, that the `%MktEx` macro and PROC OPTEX use internally. It is requested with PROC TRANSREG as follows: `class(x / standorth)` (or you can instead specify `class(x / ortheffect)`). In other procedures (such as PROC PHREG and PROC LOGISTIC), it is requested by specifying `param=ortheffect`. Notice that the sum of squares for the orthogonal coding of the two-level factor is 2. For both columns of the three-level factor, the sums of squares are 3; for the three columns of the four-level factor, the sums of squares are all 4; and for the four columns of the five-level factor, the sums of squares are all 5. For example, in the last column of the five-level factor, the sum of squares is: $-0.50^2 + -0.50^2 + -0.50^2 + 2^2 - 0.50^2 = 5$. Also notice that each column within a factor is orthogonal to all of the other columns—the sum of cross products is zero. For example, in the last two columns of the five-level factor, $-0.65 \times -0.5 + -0.65 \times -0.5 + 1.94 \times -.05 + 0 \times 2 + -0.65 \times -0.5 = 0$.

*The two-level effects coding is orthogonal, but the three-level and beyond codings are not.

Table 9
Nonorthogonal Less-Than-Full-Rank Binary or Indicator Coding

Two-Level

a	1	0
b	0	1

Three-Level

a	1	0	0
b	0	1	0
c	0	0	1

Four-Level

a	1	0	0	0
b	0	1	0	0
c	0	0	1	0
d	0	0	0	1

Five-Level

a	1	0	0	0	0
b	0	1	0	0	0
c	0	0	1	0	0
d	0	0	0	1	0
e	0	0	0	0	1

Table 10
Nonorthogonal Full-Rank Binary, Indicator, or Reference Cell Coding

Two-Level

a	1
b	0

Three-Level

a	1	0
b	0	1
c	0	0

Four-Level

a	1	0	0
b	0	1	0
c	0	0	1
d	0	0	0

Five-Level

a	1	0	0	0
b	0	1	0	0
c	0	0	1	0
d	0	0	0	1
e	0	0	0	0

Table 11
Nonorthogonal Effects or Deviations from Means Coding

Two-Level

a	1
b	-1

Three-Level

a	1	0
b	0	1
c	-1	-1

Four-Level

a	1	0	0
b	0	1	0
c	0	0	1
d	-1	-1	-1

Five-Level

a	1	0	0	0
b	0	1	0	0
c	0	0	1	0
d	0	0	0	1
e	-1	-1	-1	-1

Table 12
Orthogonal Contrast Coding

Two-Level

a	1
b	-1

Three-Level

a	1	-1
b	0	2
c	-1	-1

Four-Level

a	1	-1	-1
b	0	2	-1
c	0	0	3
d	-1	-1	-1

Five-Level

a	1	-1	-1	-1
b	0	2	-1	-1
c	0	0	3	-1
d	0	0	0	4
e	-1	-1	-1	-1

Table 13
Standardized Orthogonal Contrast Coding

Two-Level

a	1.00
b	-1.00

Three-Level

a	1.22	-0.71
b	0	1.41
c	-1.22	-0.71

Four-Level

a	1.41	-0.82	-0.58
b	0	1.63	-0.58
c	0	0	1.73
d	-1.41	-0.82	-0.58

Five-Level

a	1.58	-0.91	-0.65	-0.50
b	0	1.83	-0.65	-0.50
c	0	0	1.94	-0.50
d	0	0	0	2.00
e	-1.58	-0.91	-0.65	-0.50

Table 14

Choice Design				Coding							
Brand	x1	x2	x3	Brand 1	Brand 2	Brand 3	16 oz.	20 oz.	Price	Twist 1	Twist 2
1	16 oz.	0.89	Twist Up	1	0	0	1	0	0.89	1	0
2	24 oz.	0.89	Twist Off	0	1	0	0	0	0.89	0	1
3	20 oz.	0.99	Pop Up	0	0	1	0	1	0.99	-1	-1
1	20 oz.	0.99	Pop Up	1	0	0	0	1	0.99	-1	-1
2	24 oz.	0.89	Twist Up	0	1	0	0	0	0.89	1	0
3	20 oz.	0.89	Twist Off	0	0	1	0	1	0.89	0	1
1	20 oz.	0.89	Twist Up	1	0	0	0	1	0.89	1	0
2	20 oz.	0.99	Twist Off	0	1	0	0	1	0.99	0	1
3	16 oz.	0.89	Twist Off	0	0	1	1	0	0.89	0	1
1	20 oz.	0.89	Twist Up	1	0	0	0	1	0.89	1	0
2	16 oz.	0.99	Twist Up	0	1	0	1	0	0.99	1	0
3	24 oz.	0.99	Pop Up	0	0	1	0	0	0.99	-1	-1
1	16 oz.	0.89	Twist Off	1	0	0	1	0	0.89	0	1
2	24 oz.	0.99	Pop Up	0	1	0	0	0	0.99	-1	-1
3	16 oz.	0.99	Twist Off	0	0	1	1	0	0.99	0	1
1	24 oz.	0.99	Twist Off	1	0	0	0	0	0.99	0	1
2	16 oz.	0.89	Twist Off	0	1	0	1	0	0.89	0	1
3	16 oz.	0.89	Pop Up	0	0	1	1	0	0.89	-1	-1

These codings are explained in more detail in the *SAS/STAT User's Guide*, PROC TRANSREG, DETAILS, "ANOVA Codings" section. All SAS documentation can be accessed online at: `http://support.sas.com`.

Table 14, using the design in Table 8, shows the less-than-full-rank binary coding (brand, 3 parameters), the full-rank binary coding (size, 2 parameters), and the effects coding (top, 2 parameters). Price (1 parameter) is not coded and instead is entered as is for a linear price effect.

All of the codings discussed in this section are equivalent to each other. They are equivalent in the sense that each is a transformation of the other. If $\mathbf{X_A}$ is a coded design matrix using one coding (or mix of codings), and $\mathbf{X_B}$ is coded from the same design matrix using some other coding (or mix of codings), then there exists a transformation matrix $\mathbf{T}$ such that $\mathbf{X_A} = \mathbf{X_B T}$ and $\mathbf{X_A T^{-1}} = \mathbf{X_B}$. Analyses using the different $\mathbf{X}$ matrices will produce the same predicted values, but different parameter estimates. You can always transform one set of parameters to another. This implies that we can create a design with one coding and analyze the data with one or more different codings and get equivalent results. **We will frequently use the standardized orthogonal contrast coding when creating designs and some other coding when analyzing the data.**

Note that the two orthogonal codings are both equivalent to a Gram-Schmidt orthogonalization of the effects coding, with a subsequent scaling of the columns to get the sums of squares right. You can run the following step to see this with a five-level factor:

```
proc iml;
   x = designf((1:5)‘);          /* X = effects coding,  A = X * inv(T)     */
   call gsorth(a,t,b,x);         /* A = Gram-Schmidt orthogonalization of X */
   print "Orthogonal Contrast Coding" /
         (a * inv(diag(a[1:ncol(a),])) * diag(1:ncol(a)));
   print "Standardized Orthogonal Contrast Coding" /
         (a * sqrt(nrow(a)));
   quit;
```

Recall that our measures of linear model design efficiency are scaled to range from 0 to 100.

$$A\text{-efficiency} = 100 \times \frac{1}{N_D \text{ trace}\left((\mathbf{X}'\mathbf{X})^{-1}\right)/p}$$

$$D\text{-efficiency} = 100 \times \frac{1}{N_D \left|(\mathbf{X}'\mathbf{X})^{-1}\right|^{1/p}}$$

When computing D-efficiency or A-efficiency, we code $\mathbf{X}$ so that when the design is orthogonal and balanced, $\mathbf{X}'\mathbf{X} = N_D\mathbf{I}$ where $\mathbf{I}$ is a $p \times p$ identity matrix. When our design is orthogonal and balanced, $(\mathbf{X}'\mathbf{X})^{-1} = \frac{1}{N_D}\mathbf{I}$, and $\text{trace}\left((\mathbf{X}'\mathbf{X})^{-1}\right)/p = |(\mathbf{X}'\mathbf{X})^{-1}|^{1/p} = 1/N_D$. In this case, the two denominator terms cancel and efficiency is 100%. As the average variance increases, efficiency decreases.

The following statements show the coding of a 2×6 full-factorial design in 12 runs:

```
proc iml; /* orthogonal coding, levels must be 1, 2, ..., m */
   reset fuzz;

   start orthogcode(x);
      levels = max(x);
      xstar  = shape(x, levels - 1, nrow(x))‘;
      j = shape(1 : (levels - 1), nrow(x), levels - 1);
      r = sqrt(levels # (x / (x + 1))) # (j = xstar) -
          sqrt(levels / (j # (j + 1))) # (j > xstar | xstar = levels);
      return(r);
      finish;

   Design = (1:2)‘ @ j(6, 1, 1) || {1, 1} @ (1:6)‘;
   X = j(12, 1, 1) || orthogcode(design[,1]) || orthogcode(design[,2]);
   print design[format=1.]  ’  ’
         x[format=5.2 colname={’Int’ ’Two’ ’Six’} label=’ ’];

   XpX = x‘ * x;     print xpx[format=best5.];
   Inv = inv(xpx);   print inv[format=best5.];
   d_eff = 100 / (nrow(x) #   det(inv) ## (1 / ncol(inv)));
   a_eff = 100 / (nrow(x) # trace(inv)       / ncol(inv));
   print    ’D-efficiency =’ d_eff[format=6.2 label=’ ’]
         ’   A-efficiency =’ a_eff[format=6.2 label=’ ’];
```

The orthogonal coding function requires the factor levels to be consecutive positive integers beginning with one and ending with m for an m-level factor. Note that the IML operator `#` performs ordinary (scalar) multiplication, and `##` performs exponentiation. The results are as follows:

```
Design        Int   Two   Six

     1 1     1.00  1.00  1.73 -1.00 -0.71 -0.55 -0.45
     1 2     1.00  1.00  0.00  2.00 -0.71 -0.55 -0.45
     1 3     1.00  1.00  0.00  0.00  2.12 -0.55 -0.45
     1 4     1.00  1.00  0.00  0.00  0.00  2.19 -0.45
     1 5     1.00  1.00  0.00  0.00  0.00  0.00  2.24
     1 6     1.00  1.00 -1.73 -1.00 -0.71 -0.55 -0.45
     2 1     1.00 -1.00  1.73 -1.00 -0.71 -0.55 -0.45
     2 2     1.00 -1.00  0.00  2.00 -0.71 -0.55 -0.45
     2 3     1.00 -1.00  0.00  0.00  2.12 -0.55 -0.45
     2 4     1.00 -1.00  0.00  0.00  0.00  2.19 -0.45
     2 5     1.00 -1.00  0.00  0.00  0.00  0.00  2.24
     2 6     1.00 -1.00 -1.73 -1.00 -0.71 -0.55 -0.45

                        XpX

    12     0     0     0     0     0     0
     0    12     0     0     0     0     0
     0     0    12     0     0     0     0
     0     0     0    12     0     0     0
     0     0     0     0    12     0     0
     0     0     0     0     0    12     0
     0     0     0     0     0     0    12

                        Inv

 0.083     0     0     0     0     0     0
     0 0.083     0     0     0     0     0
     0     0 0.083     0     0     0     0
     0     0     0 0.083     0     0     0
     0     0     0     0 0.083     0     0
     0     0     0     0     0 0.083     0
     0     0     0     0     0     0 0.083

D-efficiency = 100.00    A-efficiency = 100.00
```

The following statements are a continuation of the preceding program and compute *D*-efficiency and *A*-efficiency for just a subset of the design (the first 10 rows):

```
design = design[1:10,];
x = j(10, 1, 1) || orthogcode(design[,1]) || orthogcode(design[,2]);
inv = inv(x‘ * x);
d_eff = 100 / (nrow(x) #   det(inv) ## (1 / ncol(inv)));
a_eff = 100 / (nrow(x) # trace(inv)       / ncol(inv));
print    ’D-efficiency =’ d_eff[format=6.2 label=’ ’]
      ’   A-efficiency =’ a_eff[format=6.2 label=’ ’];
quit;
```

With the full orthogonal and balanced design, $\mathbf{X}'\mathbf{X} = N_D\mathbf{I} = 12\mathbf{I}$, which means $(\mathbf{X}'\mathbf{X})^{-1} = \frac{1}{N_D}\mathbf{I} = \frac{1}{12}\mathbf{I}$, and D-efficiency = 100%. With a nonorthogonal design (for example, with the first 10 rows of the 2×6 full-factorial design), D-efficiency and A-efficiency are less than 100%. The results are as follows:

```
D-efficiency =  92.90    A-efficiency =  84.00
```

In this case, $|(\mathbf{X}'\mathbf{X})^{-1}|^{1/p}$ and trace $((\mathbf{X}'\mathbf{X})^{-1})/p$ are multiplied in the denominator of the efficiency formulas by $\frac{1}{N_D} = \frac{1}{10}$. If an orthogonal and balanced design were available for this problem, then $(\mathbf{X}'\mathbf{X})^{-1}$ would equal $\frac{1}{N_D}\mathbf{I} = \frac{1}{10}\mathbf{I}$. Since an orthogonal and balanced design is not possible (6 does not divide 10), both D-efficiency and A-efficiency are less than 100%, even with the optimal design. An orthogonal and balanced design for a main-effects model, with a variance matrix equal to $\frac{1}{N_D}\mathbf{I}$, is the standard by which 100% efficiency is gauged, even when we know such a design cannot exist. The standard is the average variance for the maximally efficient *potentially hypothetical* design, which is knowable, not the average variance for the optimal design, which for many problems we have no way of knowing.

For our purposes in this book, we only consider experimental designs with at least as many runs as parameters. A *saturated* or *tight* design has as many runs as there are parameters.[†] The number of parameters in a main-effects model is the sum of the numbers of levels of all of the factors, minus the number of factors, plus 1 for the intercept. Equivalently, since there are $m-1$ parameters in an m-level factor, the number of parameters is $1 + \sum_{j=1}^{k}(m_j - 1)$ for k factors, each with m_j levels.

If a main-effects design is orthogonal and balanced, then the design must be at least as large as the saturated design and the number of runs must be divisible by the number of levels of all the factors and by the products of the number of levels of all pairs of factors. For example, a $2 \times 2 \times 3 \times 3 \times 3$ design cannot be orthogonal and balanced unless the number of runs is divisible by 2 (twice because there are two 2's), 3 (three times because there are three 3's), $2 \times 2 = 4$ (once, because there is one pair of 2's), $2 \times 3 = 6$ (six times, two 2's times three 3's), and $3 \times 3 = 9$ (three times, three pairs of 3's).* If the design is orthogonal and balanced, then all of the divisions work without a remainder. However, all of the divisions working is a necessary but not sufficient condition for the existence of an orthogonal and balanced design. For example, 45 is divisible by 3 and $3 \times 3 = 9$, but an orthogonal and balanced saturated design 3^{22} (22 three-level factors) in 45 runs does not exist.

Coding and Reference Levels: The ZERO= Option

In this book, we do most of our coding using PROC TRANSREG and its MODEL statement. In some cases, such as full-profile conjoint analysis, we call PROC TRANSREG directly to perform the analysis. In other cases, such as coding the design before fitting a choice model, we call PROC TRANSREG directly to perform the coding, but we use PROC PHREG for the analysis. In still other cases, such as with the `%ChoiceEff` macro, we specify PROC TRANSREG syntax, but do not call the procedure directly. The macro calls the procedure for us. No matter how it gets called, we often need to control the reference level or suppress the creation of a reference level for a less-than-full-rank coding. This

[†]This definition is in the context of a linear model. In a choice model, substitute number of choice sets times the number of alternatives minus one for the number of runs.

*In practice, we never have to do any of these calculations ourselves, since they are done for us by the `%MktRuns` macro.

section describes the syntax involved in the PROC TRANSREG `zero=` option, which is used to control the reference level. In this section, only the relevant MODEL statement fragments are provided rather than the full statement. This section is important because this option is used a lot in this book, and while it is not a difficult option, you might find it confusing if you do not have the background provided here.

By default, when coding, an indicator variable is not created for the last level and hence a coefficient is not computed for that level. This is called *reference-cell coding*. For example, by default, `x` with values 1, 2, and 3 is coded as follows:

`x`	Coding	
1	1	0
2	0	1
3	0	0

The level '3' is the reference level. You could instead create an indicator variable for all levels as follows:

`x`	Coding		
1	1	0	0
2	0	1	0
3	0	0	1

However, in many modeling situations, such as ANOVA models, regression models with intercepts, and choice models, the coefficient for the indicator variable for the last level would be zero, because that last column is redundant given all of the columns that come before. The name of the option, `zero=`, comes from the idea that this option lets you specify the level that will have a structural zero coefficient. There are several ways that we use the zero= option:

`zero=first`
specifies the first level as the reference level for all factors in the `class` specification in which it is applied. Coded variables are created for all but the first level for each factor.

`zero=last`
specifies the last level as the reference level for all factors in the `class` specification in which it is applied. Coded variables are created for all but the last level for each factor. This is the default.

`zero=none`
does not create a reference level for any of the factors in the `class` specification in which it is applied. Indicator variables are created for all levels of all factors.

`zero=sum`
does not create a reference level for any of the factors in the `class` specification in which it is applied. Indicator variables are created for all levels of all factors. The parameter estimates are constrained to sum to zero. This option is useful for full-profile conjoint analysis but not for choice modeling. This is because it can only be used when PROC TRANSREG is doing the analysis, not when PROC TRANSREG is just doing the coding.

`zero=`*formatted-value-list*
specifies the level corresponding to the formatted value as the reference level for all factors in the `class` specification in which it is applied. Coded variables are created for all but the specified level for each factor. The first formatted value applies to the first factor, the second formatted value applies to the second factor, and so on. When the formatted value list is shorter than the factor list, the default `zero=last` is used for the remaining factors. The formatted values must appear in quotes.

The remainder of this section illustrates properties of the `zero=` option. We assume that `x1-x6` are all three level factors with levels 1, 2, and 3. Also assume that the first three rows of the design are as follows:

x1	x2	x3	x4	x5	x6
1	1	1	1	1	1
2	2	2	2	2	2
3	3	3	3	3	3

Note that you cannot mix `zero=none`, `zero=first`, or `zero=last` with `zero=`*formatted-value-list* within a single `class` specification. However, you can use different options if you use multiple `class` specifications, as is the following example:

```
class(x1 x2 / zero=none) class(x3 x4 / zero=first) class(x5 x6 / zero='2' '3')
```

The first three rows of the coded design are as follows:

x11	x12	x13	x21	x22	x23	x32	x33	x42	x43	x51	x53	x61	x62
1	0	0	1	0	0	0	0	0	0	1	0	1	0
0	1	0	0	1	0	1	0	1	0	0	0	0	1
0	0	1	0	0	1	0	1	0	1	0	1	0	0

In this coded design, the columns are labeled by the name of the factor (from `x1-x6`) and formatted value of the relevant level (1, 2, or 3). For example, `x11` is the indicator variable for the first level of `x1`.

With `zero=`*formatted-value-list*, you can achieve the same effect as `zero=first` by specifying the first formatted value and the same effect as `zero=last` by specifying the last formatted value. Additionally, you can achieve the same effect as `zero=none` by specifying a nonexistent level such as blank. The following specification is equivalent to the preceding specification and produces the same coded design:

```
class(x1-x6 / zero=' '  ' '  '1'  '1'  '2'  '3')
```

PROC TRANSREG warns you when you specify a nonexistent reference level as follows:

```
WARNING: Reference level ZERO='' was not found for variable x1.
```

This warning can be safely ignored because the specification was deliberate.

Sometimes, it is useful to specify `zero=none` for the first factor and use the default `zero=last` for the remaining factors. The following illustrates one way of doing this:

```
class(x1 / zero=none) class(x2-x6)
```

The first three rows of the coded design are as follows:

x11	x12	x13	x21	x22	x31	x32	x41	x42	x51	x52	x61	x62
1	0	0	1	0	1	0	1	0	1	0	1	0
0	1	0	0	1	0	1	0	1	0	1	0	1
0	0	1	0	0	0	0	0	0	0	0	0	0

The following illustrates another way of doing this same thing:

```
class(x1-x6 / zero=' ')
```

The results are the same as before. Note that the nonexistent blank level applies to `x1`. Since the `zero=` list is exhausted, the reference level for the remaining factors, `x2-x6`, is the default last level.

In choice modeling, the brand factor might have a level labeled as 'None' for no purchase or none of the above. You can specify it in quotes in the `zero=` option as you would any other level. The following illustrates this:

```
class(Brand Price / zero='None')
```

This creates indicator variables for all but the 'None' level of `Brand` and for all but the last level of `Price`. Note that this is quite different from the following:

```
class(Brand Price / zero=None)
```

This specification creates indicator variables for every level of both factors.

The next section discusses coding and design efficiency. The section after that gives some more detail about the `zero=` option and its effects on coding.

Coding and the Efficiency of a Choice Design

The previous sections discuss several types of coding including reference cell coding, less-than-full-rank coding, effects coding, and the standardized orthogonal contrast coding. This section discusses their use in choice modeling. This section is important because it provides you with some context you can use to evaluate the goodness of your choice design.

Before you use a choice design, you should code it, compute the variance-covariance matrix of the parameter estimates, and check the design efficiency. You can do this with the `%ChoiceEff` macro. It uses PROC TRANSREG to do the coding, and you specify the PROC TRANSREG syntax and options that control the coding. When you code the design properly (at least for some designs), you can get a relative *D*-efficiency on a 0 to 100 scale. This is done by using a standardized orthogonal contrast coding and scaling the observed *D*-efficiency relative to the *D*-efficiency for a possibly hypothetical optimal design. In a linear model, the variance matrix for the potentially hypothetical optimal design is $\frac{1}{N_D}\mathbf{I}$ where N_D is the number or runs in the linear arrangement. Similarly, in a choice model, the variance matrix for the potentially hypothetical optimal design is in some cases $\frac{1}{N_S}\mathbf{I}$ where N_S is the number of choice sets. When a choice design is made from a linear arrangement using the method shown previously in this chapter, $N_D = N_S$.

To illustrate, consider a choice design with 3 three-level attributes, three alternatives, and three choice sets. This next example constructs an optimal generic choice design for this specification. Later sections discuss syntax and designs like this one in more detail. This section concentrates on results—specifically design efficiency and the variances of the parameter estimates. This design is constructed and evaluated as follows:

```
%mktex(3 ** 4,                      /* set number and factor levels         */
       n=9)                         /* num of sets times num of alts (3x3)  */

%mktlab(data=design,                /* design from MktEx                    */
        vars=Set x1-x3)             /* new variable names                   */
```

```
%choiceff(data=final,               /* candidate set of choice sets         */
          init=final(keep=set),     /* select these sets from candidates    */
          model=class(x1-x3 / sta), /* model with stdzd orthogonal coding   */
          nsets=3,                  /* 3 choice sets                        */
          nalts=3,                  /* 3 alternatives per set               */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */

proc print data=bestcov label;      /* covariance matrix from ChoicEff      */
   title 'Variance-Covariance Matrix';
   id __label;
   label __label = '00'x;           /* hex null suppress label header       */
   var x:;
   run;

title;
```

The `sta`* (short for `standorth`) option in the `model=` option is new with SAS 9.2 and requests a standardized orthogonal contrast coding. We use it whenever possible, because in some situations, it gives us a relative *D*-efficiency in the range 0 to 100. That is, for certain optimal designs such as this one, relative *D*-efficiency is 100 when the standardized orthogonal contrast coding is used. For other optimal designs, such as those with constraints such as a constant alternative, the maximum relative *D*-efficiency is less than 100 with this coding. The precise maximum is hard to know in general when there are constraints. With other codings, the raw *D*-efficiency can be any nonnegative value. In some cases, it can even be greater than 100.

A subset of the results are as follows:

Final Results

Design	1
Choice Sets	3
Alternatives	3
Parameters	6
Maximum Parameters	6
D-Efficiency	3.0000
Relative D-Eff	100.0000
D-Error	0.3333
1 / Choice Sets	0.3333

*This option is first available with SAS 9.2. It will not be recognized, and it will cause an error in earlier SAS releases.

n	Variable Name	Label	Variance	DF	Standard Error
1	x11	x1 1	0.33333	1	0.57735
2	x12	x1 2	0.33333	1	0.57735
3	x21	x2 1	0.33333	1	0.57735
4	x22	x2 2	0.33333	1	0.57735
5	x31	x3 1	0.33333	1	0.57735
6	x32	x3 2	0.33333	1	0.57735
				==	
				6	

Variance-Covariance Matrix

	x1 1	x1 2	x2 1	x2 2	x3 1	x3 2
x1 1	0.33333	0.00000	0.00000	0.00000	0.00000	0.00000
x1 2	0.00000	0.33333	0.00000	0.00000	0.00000	0.00000
x2 1	0.00000	0.00000	0.33333	0.00000	0.00000	0.00000
x2 2	0.00000	0.00000	0.00000	0.33333	0.00000	0.00000
x3 1	0.00000	0.00000	0.00000	0.00000	0.33333	0.00000
x3 2	0.00000	0.00000	0.00000	0.00000	0.00000	0.33333

Starting with the last table, the variance-covariance matrix is $\frac{1}{3}\mathbf{I}$. The 3 in the denominator comes from the fact that there are 3 choice sets. For reference, the last line in the first table displays one over the number of choice sets. Above that, it displays *D*-error (1/3), which is the inverse of *D*-efficiency (3, the number of choice sets). Relative *D*-efficiency is equal to *D*-efficiency divided by the optimal value (the number of choice sets) and multiplied by 100. In an optimal design such as this one, relative *D*-efficiency is 100. The middle table displays the variances, which are the diagonal values from the variance-covariance matrix, and the standard errors, which are the square roots of the variances. With the standardized orthogonal contrast coding used here, an optimal design has all zeros on the off diagonals and $1/N_S$ on the diagonal.

It is also good to compare the number of parameters in the model with the maximum number of parameters that could be estimated with the number of choice sets and alternatives in this experiment. These are shown in the first table. In this case, both of these numbers are the same (6) showing that the design is saturated. Unless you have an optimal design such as this one, you will usually not want the number of parameters to be this close to the maximum. The maximum value is the number of choice sets times the number of alternatives minus one: $3(3-1) = 6$.

In practice, most of our designs are not optimal like this one. However, you can still use the standardized orthogonal contrast coding to see how much bigger the variances are and use that information to guide your design construction decisions. To illustrate, consider the next example, which creates a random choice design. This is *not* a recommended strategy; it is just done to show how a less-than-optimal design compares to the optimal design. The following steps make and evaluate a random design:

```
data final;                          /* random design                    */
   do Set = 1 to 3;                  /* 3 choice sets                    */
      do Alt = 1 to 3;               /* 3 alternatives                   */
         x1 = ceil(3 * uniform(151));/* random levels for each attr      */
         x2 = ceil(3 * uniform(151));
         x3 = ceil(3 * uniform(151));
         output;
         end;
      end;
   run;

%choiceff(data=final,               /* candidate set of choice sets     */
          init=final(keep=set),     /* select these sets from candidates */
          model=class(x1-x3 / sta), /* model with stdzd orthogonal coding */
          nsets=3,                  /* 3 choice sets                    */
          nalts=3,                  /* 3 alternatives per set           */
          options=relative,         /* display relative D-efficiency    */
          beta=zero)                /* assumed beta vector, Ho: b=0     */

proc print data=bestcov label;      /* covariance matrix from ChoicEff  */
   title 'Variance-Covariance Matrix';
   id __label;
   label __label = '00'x;           /* hex null suppress label header   */
   var x:;
   run;

title;
```

The same tables as in the previous example are as follows:

Final Results

Design	1
Choice Sets	3
Alternatives	3
Parameters	6
Maximum Parameters	6
D-Efficiency	1.0000
Relative D-Eff	33.3333
D-Error	1.0000
1 / Choice Sets	0.3333

n	Variable Name	Label	Variance	DF	Standard Error
1	x11	x1 1	1.00000	1	1.00000
2	x12	x1 2	2.33333	1	1.52753
3	x21	x2 1	2.00000	1	1.41421
4	x22	x2 2	2.66667	1	1.63299
5	x31	x3 1	3.00000	1	1.73205
6	x32	x3 2	1.00000	1	1.00000
				==	
				6	

Variance-Covariance Matrix

	x1 1	x1 2	x2 1	x2 2	x3 1	x3 2
x1 1	1.00000	0.00000	1.00000	1.15470	-1.00000	-0.57735
x1 2	0.00000	2.33333	-0.00000	1.00000	0.57735	0.33333
x2 1	1.00000	-0.00000	2.00000	1.73205	-0.50000	-0.86603
x2 2	1.15470	1.00000	1.73205	2.66667	-0.86603	-0.83333
x3 1	-1.00000	0.57735	-0.50000	-0.86603	3.00000	0.57735
x3 2	-0.57735	0.33333	-0.86603	-0.83333	0.57735	1.00000

Now our variances range from 1 to 3 instead of all being 1/3. The design is 33.3333% as efficient as the optimal design. Even if we did not have the results from the optimal design for reference, we can see from the table that one over the number of choice sets = 0.3333, so our variances are way bigger than we would expect.

Even if you plan on using a different coding for the analysis, it is good to evaluate the design using the standardized orthogonal contrast coding and see how large the variances are relative to the optimal value. Note, however, that in many situations, we do not know what the optimal variance is. One over the number of choice sets might be too small when the design is more complicated than an optimal generic choice design. To illustrate, again consider the same problem, but this time we will force the third alternative to be constant (all levels 2). The following steps create and evaluate the design:

```
%mktex(3 ** 3,                      /* just the factor levels              */
       n=27)                        /* number of candidate alts            */

%mktlab(data=design,                /* design from MktEx                   */
        int=f1-f3)                  /* flag which alt can go where, 3 alts */

data final;                         /* all candidates go to alt 1 and 2    */
   set final;                       /* x1=2 x2=2 x3=2 also goes to alt 3   */
   f3 = (x1 eq 2 and x2 eq 2 and x3 eq 2);
   run;
```

```
%choiceff(data=final,               /* candidate set of alternatives        */
          model=class(x1-x3 / sta), /* model with stdzd orthogonal coding   */
          seed=205,                 /* random number seed                   */
          nsets=3,                  /* 3 choice sets                        */
          flags=f1-f3,              /* flag which of the 3 alts go where    */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */

proc print data=bestcov label;      /* covariance matrix from ChoicEff      */
   title 'Variance-Covariance Matrix';
   id __label;
   label __label = '00'x;           /* hex null suppress label header       */
   var x:;
   run;

title;
```

The same tables as in the previous examples are as follows:

Final Results

Design	1
Choice Sets	3
Alternatives	3
Parameters	6
Maximum Parameters	6
D-Efficiency	1.5874
Relative D-Eff	52.9134
D-Error	0.6300
1 / Choice Sets	0.3333

n	Variable Name	Label	Variance	DF	Standard Error
1	x11	x1 1	1.75000	1	1.32288
2	x12	x1 2	1.08333	1	1.04083
3	x21	x2 1	0.50000	1	0.70711
4	x22	x2 2	0.66667	1	0.81650
5	x31	x3 1	0.50000	1	0.70711
6	x32	x3 2	0.66667	1	0.81650
				==	
				6	

Variance-Covariance Matrix

	x1 1	x1 2	x2 1	x2 2	x3 1	x3 2
x1 1	1.75000	-0.28868	-0.12500	0.21651	0.12500	0.21651
x1 2	-0.28868	1.08333	-0.07217	-0.37500	0.07217	-0.37500
x2 1	-0.12500	-0.07217	0.50000	-0.14434	0.00000	0.14434
x2 2	0.21651	-0.37500	-0.14434	0.66667	-0.14434	-0.16667
x3 1	0.12500	0.07217	0.00000	-0.14434	0.50000	0.14434
x3 2	0.21651	-0.37500	0.14434	-0.16667	0.14434	0.66667

This is a small and easy problem for the %ChoicEff macro, and it is given all possible candidates from which to work. Hence, it has almost certainly found the optimal design for this specification. However, relative *D*-efficiency is 52.9134% and the variances are all larger than one over the number of choice sets.*

These results show that for this small design, it appears that the maximum *D*-efficiency is 1.5874. If you know the maximum possible *D*-efficiency (or even have a guess), you can use it as a scaling factor for relative *D*-efficiency instead of using the default (the number of choice sets). The following step specifies the maximum *D*-Efficiency in the rscale= (relative efficiency scaling factor) option:

```
%choiceff(data=final,               /* candidate set of alternatives        */
          model=class(x1-x3 / sta), /* model with stdzd orthogonal coding   */
          seed=205,                 /* random number seed                   */
          nsets=3,                  /* 3 choice sets                        */
          flags=f1-f3,              /* flag which of the 3 alts go where    */
          rscale=1.5874,            /* scale using previous D-efficiency    */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */

proc print data=bestcov label;      /* covariance matrix from ChoicEff      */
   title 'Variance-Covariance Matrix';
   id __label;
   label __label = '00'x;           /* hex null suppress label header       */
   var x:;
   run;

title;
```

The same tables as in the previous examples are as follows:

*If you run this step repeatedly with different seeds, you will get the same relative *D*-efficiency, but the individual variances will change as the %ChoicEff finds different but equivalent designs.

Final Results

Design	1
Choice Sets	3
Alternatives	3
Parameters	6
Maximum Parameters	6
D-Efficiency	1.5874
Relative D-Eff	100.0001
D-Error	0.6300
1 / Choice Sets	0.3333

n	Variable Name	Label	Variance	DF	Standard Error
1	x11	x1 1	1.75000	1	1.32288
2	x12	x1 2	1.08333	1	1.04083
3	x21	x2 1	0.50000	1	0.70711
4	x22	x2 2	0.66667	1	0.81650
5	x31	x3 1	0.50000	1	0.70711
6	x32	x3 2	0.66667	1	0.81650
				==	
				6	

Variance-Covariance Matrix

	x1 1	x1 2	x2 1	x2 2	x3 1	x3 2
x1 1	1.75000	-0.28868	-0.12500	0.21651	0.12500	0.21651
x1 2	-0.28868	1.08333	-0.07217	-0.37500	0.07217	-0.37500
x2 1	-0.12500	-0.07217	0.50000	-0.14434	0.00000	0.14434
x2 2	0.21651	-0.37500	-0.14434	0.66667	-0.14434	-0.16667
x3 1	0.12500	0.07217	0.00000	-0.14434	0.50000	0.14434
x3 2	0.21651	-0.37500	0.14434	-0.16667	0.14434	0.66667

Now, relative *D*-efficiency is approximately 100. It would be exactly 100 if it there were no rounding error in the *D*-efficiency displayed in the first table. The variances and covariances are unchanged.

In summary, the `%ChoicEff` macro gives you options and context to help you evaluate your choice designs. In some simple situations, the maximum *D*-efficiency is clear, and it can be used to scale the current design *D*-efficiency to get a relative *D*-efficiency on a 0 to 100 scale. When the maximum is not clear, you can instead specify your own scale factor.

Orthogonal Coding and the ZERO=' ' Option

The `zero=` option was explained in a preceding section. There is one more aspect of `zero=' '` versus `zero=none` usage that needs to be explained. The `zero=none` option is designed to transform binary (0,1) reference cell coding (one indicator variable for every level except one) into a cell-means coding (one indicator variable for every level). It is not designed for use with the effects (also known as deviations from means) coding (specified with `effects`, `eff`, `deviations`, `dev`), or the orthogonal codings (specified with `orthogonal`, `ort`, `standorth`, `sta`). However, you an use `zero=' '` to specify a cell-means-style coding for just the first factor in a `class` specification with any of those options. This lets you specify these codings within groups. This is probably used most often when evaluating a choice design that has alternative-specific effects. To illustrate, consider the following data set:

```
data x;
   input Brand $ x1-x2;
   datalines;
A 1 1
A 1 2
A 2 1
A 2 2
B 1 1
B 1 2
B 2 1
B 2 2
C 1 1
C 1 2
C 2 1
C 2 2
;
```

The following PROC TRANSREG step codes this design using the standardized orthogonal contrast coding for alternative-specific effects within brand:

```
proc transreg design data=x;
   model class(brand / lprefix=0)
         class(brand * x1 brand * x2 / sta zero=' ' lprefix=0 2 2);
   output out=coded(drop=_: in:) separators='' ' ';
   run;

proc print label noobs; run;
```

Note that `zero=' '` applies to only the first factor in that `class` specification (`Brand`), and it applies to it every place that it is used in that `class` specification. The results are as follows:

A	B	A x11	B x11	C x11	A x21	B x21	C x21	Brand	x1	x2
1	0	1	0	0	1	0	0	A	1	1
1	0	1	0	0	-1	0	0	A	1	2
1	0	-1	0	0	1	0	0	A	2	1
1	0	-1	0	0	-1	0	0	A	2	2
0	1	0	1	0	0	1	0	B	1	1
0	1	0	1	0	0	-1	0	B	1	2
0	1	0	-1	0	0	1	0	B	2	1
0	1	0	-1	0	0	-1	0	B	2	2
0	0	0	0	1	0	0	1	C	1	1
0	0	0	0	1	0	0	-1	C	1	2
0	0	0	0	-1	0	0	1	C	2	1
0	0	0	0	-1	0	0	-1	C	2	2

The first 8 columns contain the coded design, and the last 3 contain the input factors (`class` variables). The specification `class(brand / lprefix=0)` creates the first two columns, labeled `A` and `B`, which are indicator variables for the first two brands. The third brand, `C`, corresponds to the reference level. For all three brands, both `x1` and `x2` are coded with a standardized orthogonal contrast coding within brand. Since both `x1` and `x2` have only two levels, there is only one coded variable for each factor for each brand. The interaction between `Brand` and `x1` creates the design columns labeled as `A x11`, `B x11`, and `C x11`, and the interaction between `Brand` and `x2` creates the design columns labeled as `A x21`, `B x21`, and `C x21`.

You can better understand why the columns `A x11` through `C x21` are coded as they are by examining the main effects that go into creating these interaction terms. The following steps create and display both the main effects and interactions:

```
proc transreg design data=x;
   model class(brand | x1 brand | x2 / sta zero=' ' lprefix=0 2 2);
   output out=coded separators='' ' ';
   run;

proc print label noobs;
   var BrandA BrandB BrandC x11 x21 BrandAx11 BrandBx11 BrandCx11
       BrandAx21 BrandBx21 BrandCx21;
   run;
```

The results are as follows:

A	B	C	x11	x21	A x11	B x11	C x11	A x21	B x21	C x21
1	0	0	1	1	1	0	0	1	0	0
1	0	0	1	-1	1	0	0	-1	0	0
1	0	0	-1	1	-1	0	0	1	0	0
1	0	0	-1	-1	-1	0	0	-1	0	0
0	1	0	1	1	0	1	0	0	1	0
0	1	0	1	-1	0	1	0	0	-1	0
0	1	0	-1	1	0	-1	0	0	1	0
0	1	0	-1	-1	0	-1	0	0	-1	0
0	0	1	1	1	0	0	1	0	0	1
0	0	1	1	-1	0	0	1	0	0	-1
0	0	1	-1	1	0	0	-1	0	0	1
0	0	1	-1	-1	0	0	-1	0	0	-1

The columns labeled `A x11` through `C x21` are the same as we saw before. `A x11` is the element-wise product of `A` and `x11`, `B x11` is the element-wise product of `B` and `x11`, `C x11` is the element-wise product of `C` and `x11`, `A x21` is the element-wise product of `A` and `x21`, `B x21` is the element-wise product of `B` and `x21`, and `C x21` is the element-wise product of `C` and `x21`. More information about the standardized orthogonal contrast coding, along with the use of `zero=' '`, can be found in the example starting on page 858.

Orthogonally Coding Price and Other Quantitative Attributes

For inherently quantitative factors such as price, you might want to use different strategies for coding during the analysis instead of using indicator variables or effects coding. When we create a design with a quantitative factor such as price, we do not have to do anything special. The orthogonal coding what we use to make qualitative factors is just as applicable when the factor are quantitative. See page 251 for more information. However, for analysis, we might want a different coding than the binary or effects coding. For example, imagine a choice experiment involving SUV's with price as an attribute and with levels of $27,500, $30,000, and $32,500. You probably will not code them as is and just add these prices directly to the model, because these values are considerably larger than the other values in your coded factors, which usually consist of values such as –1, 0, and 1. You might believe that choice is not a linear function of price; it might be nonlinear or quadratic. Hence, you might think about adding a price-squared term, but squaring values this large is almost certain to cause problems with collinearity. When you are dealing with factors such as this, you are usually better off recoding them in a "nicer" way. The following table shows some of the steps in the recoding:

Price	Centered Price			Divide By Increment			Square		
27,500	$27,500 - 30,000$	$=$	$-2,500$	$-2,500/2,500$	$=$	-1	-1^2	$=$	1
30,000	$30,000 - 30,000$	$=$	0	$0/2,500$	$=$	0	0^2	$=$	0
32,500	$32,500 - 30,000$	$=$	$2,500$	$2,500/2,500$	$=$	1	1^2	$=$	1

The second column shows the results of centering the values—subtracting the mean price of $30,000. The third column shows the results of dividing the centered values by the increment between values,

2,500. The fourth column shows the square of the third column. These last two columns make much better linear and quadratic price terms than the original price and the original price squared, however, we can do better still. The next table shows the final steps and the full, orthogonal, quadratic coding.

Coding			Centered Quadratic					Multiply Through					Orthogonal Coding		
1	−1	1	1	−	2/3	=	1/3	3	×	1/3	=	1	1	−1	1
1	0	0	0	−	2/3	=	−2/3	3	×	−2/3	=	−2	1	0	−2
1	1	1	1	−	2/3	=	1/3	3	×	1/3	=	1	1	1	1

The first coding consists of an intercept, a linear term, and a quadratic term. Notice that the sum of the quadratic term is not zero, so the quadratic term is not orthogonal to the intercept. We can correct this by centering (subtracting the mean which is 2/3). After centering, all three columns are orthogonal. We can make the coding nicer still by multiplying the quadratic term by 3 to get rid of the fractions. The full orthogonal coding is shown in the last set of columns. Note, however, that only the last two columns are used. The intercept is just there to more clearly show that all columns are orthogonal. This orthogonal coding works for any three-level quantitative factor with equal intervals between adjacent levels.

For four equally-spaced levels, and with less detail, the linear and quadratic coding is shown in the last two columns of the following table:

Price	Center	Divide	Integers	Square	Center	Smallest Integers	Coding	
27500	−3750	−1.5	−3	9	4	1	−3	1
30000	−1250	−0.5	−1	1	−4	−1	−1	−1
32500	1250	0.5	1	1	−4	−1	1	−1
35000	3750	1.5	3	9	4	1	3	1

The Number of Factor Levels

The number of levels of the factors can affect design efficiency. Since two points define a line, it is inefficient to use more than two points to model a linear function. When a quadratic function is used (x and x^2 are included in the model), three points are needed—the two extremes and the midpoint. Similarly, four points are needed for a cubic function. More levels are needed when the functional form is unknown. Extra levels let you examine complicated nonlinear functions, with a cost of decreased efficiency for the simpler functions. When you assume that the function is linear, experimental points should not be spread throughout the range of experimentation.

We are often tempted to have more levels than we really need, particularly for factors such as price. If you expect to model a quadratic price function, you only need three price points. It might make sense to have one or two more price points so that you can test for departures from the quadratic model, but you do not want more than that. You probably would never be interested in a price function more complicated than a cubic function. Creating a design with many price points and then fitting a low-order price function reduces efficiency at analysis time. The more factors you have with more than two or three levels, the harder it is usually going to be to find an orthogonal and balanced design or even a close approximation.

There are times, however, when you can reasonably create factors with more levels than you really need. Say you have a design with two-level and four-level factors and you want to create quadratic price effects, which would require three evenly-spaced levels. Say you also want the ability to test for departures from a quadratic model. You could use a strategy that begins with an eight-level price factor. Then you can recode it as follows: (1 2 3 4 5 6 7 8) → (1 2 3 4 5 1 3 5). Notice that you end up with twice as many points at the minimum, middle, and maximum positions as in the second and fourth positions. This gives you good efficiency for the quadratic effect and some information about higher-order effects. Furthermore, there are many designs with mixtures of (2, 4, and 8)-level factors in 64 runs, which you can easily block. In contrast, you need 400 runs ($4 \times 4 \times 5 \times 5$) before you can find a design such as $2^2 4^2 5^2$ in an orthogonal array. If you are assigning levels with a format, you can assign levels and do the recoding all at the same time as in the following example:

```
proc format;
   value price  1 = $2.89 2 = $2.99 3 = $3.09 4 = $3.19 5 = $3.29
                6 = $2.89           7 = $3.09           8 = $3.29;
   run;
```

Randomization

Randomization is the process of sorting the rows of a design into a random order and randomly reassigning all of the factor levels. See page 57 for an example of randomization. Some designs must be randomized before you use them. Full-factorial designs, fraction-factorial designs, and orthogonal arrays all need to be randomized. These designs must be randomized for subject-effect reasons not for statistical reasons. Randomization does not change a design's efficiency, orthogonality, or balance. Hence, statistically, it does not matter. The problem with many full-factorial designs, fraction-factorial designs, and orthogonal arrays is they have some rows have a recognizable pattern of levels. Most typically, there are two problems with the original design. Often, the rows are sorted (or the levels change in some other predictable way), and often, one row is constant. Typically, when a row is constant, the first row consists entirely of the first factor level. The `%MktEx` macro automatically randomizes designs and stores the randomized design in the `outr=` data set.

Designs created through a coordinate exchange or candidate set search do not need to be randomized, since the construction algorithm has a random component. However, randomizing these designs is fine as long as there are no restrictions on the design. Randomization occurs after the design is constructed with the restrictions imposed. If you restrict the design, say by preventing a certain level of one factor from appearing with a certain level of another factor, and then randomize, the restriction will quite likely be violated in the randomized design.

Typically, designs for linear models, such as those used for conjoint models, are randomized. Linear model designs that are later used to make choice designs are also typically randomized. However, once a design is in choice-design form, it should not be randomized. For example, exchanging rows could in many cases destroy the integrity of the design. At best, it could decrease the efficiency of the choice design. Randomization is typically not required for choice designs because often, some randomization has occurred earlier in their construction.

Random Number Seeds

Our designs are created in many ways, but virtually every design we create uses a random number stream as part of the process. In some cases, the initial design is random, then a computerized search is used to improve it. In other cases, an orthogonal array is used, but first it is randomized (the rows are sorted into a random order and the factor levels are randomly reassigned). There are many other ways in which random number streams are used in making designs. All of the SAS design macros that have a random component have a `seed=` option to control the random number seed. The random number seed is an integer in the range 1 to 2,147,483,646. This seed is used to provide a starting point for the random number stream. Note, however, that the seed itself is not a part of the stream. If you do not specify a seed, then a seed is generated for you and displayed in the SAS log. The default seed is a (nonobvious) function of the date and the time. Most of the examples in this book use an explicitly set random number seed. The few exceptions are cases where orthogonal arrays or full-factorial designs (without randomization) are generated, and the seed does not matter.

Explicitly specifying the random number seed is a good programming practice. If you do not specify a seed, and you are creating a design with a random component, then you probably will get different results each time you run the macro. You will certainly get different results unless your design is very small. Even with only a single two-level factor, you will only get the same results 50% of the time. With different seeds, you should expect efficiencies that are similar, but you should not expect them to always be the same.

It is important to not only specify the seed, but it is also important that you save your design for later use in the analysis. If by chance you make a design and then install new macros, get a new computer, or update the operating system or SAS release, even knowing the seed might not be enough to reproduce the same design. Algorithms with random components are not guaranteed to always produce the same results when things change. Imagine an ant climbing a large sand dune, each time stepping only on higher grains of sand. Then imagine a light breeze slightly shifting the sand. A second ant will quite likely find a slightly different path even if she starts from the same place as the first ant.

Sometimes, when you run one of the design macros, you might get results that are undesirable for some reason. Perhaps there are duplicates but you did not specify `options=nodups`. Perhaps one factor is slightly less balanced than you would like. There are many other things that could happen. Sometimes changing the random number seed is enough to make the undesirable results go away. Other times you need to take more aggressive approaches such as specifying new options.

Duplicates

It is sometimes the case that an optimal experimental design will have duplicates. For a linear arrangement or conjoint design, this means duplicate runs or profiles. In a choice design, this can mean duplicate choice sets or duplicate profiles within a choice set. Consider the following orthogonal array in 12 runs:

1	1	1	2	1	1	2	1	2	2	2
1	1	2	1	1	2	1	2	2	2	1
1	1	2	1	2	2	2	1	1	1	2
1	2	1	1	2	1	2	2	2	1	1
1	2	1	2	2	2	1	1	1	2	1
1	2	2	2	1	1	1	2	1	1	2
2	1	1	1	2	1	1	2	1	2	2
2	1	1	2	1	2	2	2	1	1	1
2	1	2	2	2	1	1	1	2	1	1
2	2	1	1	1	2	1	1	2	1	2
2	2	2	1	1	1	2	1	1	2	1
2	2	2	2	2	2	2	2	2	2	2

This is an optimal design with 11 factors and with no duplicates. Any subset of columns is also an optimal design but with a reduced number of factors. However, if you select only the first four columns, then the second and third rows are duplicates. Often, researchers will want to avoid designs with duplicates. The reasons will typically have more to do with worries about what the subjects will think if given the same task twice or what the client or brand manager will think. That is, the reasons to worry about duplicates are based on human concerns not statistical concerns. Duplicates do not pose any problem from a statistical or design efficiency point of view. In fact, the maximum *D*-efficiency for the design 2^4 in 12 runs with duplicates is 100 and without duplicates it is 97.6719. Still, human concerns *are* important in making a design, so often researchers try to avoid duplicates.

You can use the `%MktDups` macro to check your linear or choice design for duplicates. In practice, duplicates do not occur very often. When they occur, sometimes something as simple as changing the random number seed is sufficient to avoid duplicates. When that fails, you can ensure that duplicates are not created with macro options. You can specify `options=nodups` with both the `%MktEx` and `%ChoicEff` macros. The `nodups` specification is not the default because it imposes time-consuming restrictions on the algorithm that are rarely needed. See the documentation for the `%MktDups` macro beginning on page 1004 for more information about duplicates. In addition, there are numerous examples of using this macro throughout this book including in this chapter on the following pages: 147, 174, 198, and 206.

Orthogonal Arrays and Difference Schemes

This section provides some details about how certain orthogonal arrays are constructed. This section also discusses difference schemes which are fundamental building blocks in orthogonal array and optimal generic choice design construction. This section is optional and can be skipped by all but the most interested of readers. The next section starts on page 101. This section begins by illustrating how

the first (that is, smallest) five orthogonal arrays are constructed. They were previously shown on page 59. These designs require the easiest of orthogonal array construction methods. The point of this discussion is not to fully explain how orthogonal arrays are constructed or even fully explain how these orthogonal arrays are constructed. Rather, the goal is just to provide the tiniest glimpse into the methods that the `%MktEx` macro uses to construct orthogonal arrays, to illustrate some of their beauty, and to show some of the ways they are connected.

The five designs that were previously displayed on page 59 are now displayed with a different set of integers representing the levels. The following representation is more natural from a design construction point of view:

2^3			$2^1 3^1$		2^7							$2^4 4^1$					3^4			
1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
-1	1	-1	0	1	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1	0	1	2	1
1	-1	-1	0	2	1	-1	-1	1	1	-1	-1	1	1	-1	-1	2	0	2	1	2
-1	-1	1	1	0	-1	-1	1	1	-1	-1	1	1	-1	-1	1	3	1	1	1	0
			1	1	1	1	1	-1	-1	-1	-1	-1	-1	-1	-1	0	1	2	0	1
			1	2	-1	1	-1	-1	1	-1	1	-1	1	-1	1	1	1	0	2	2
					1	-1	-1	-1	-1	1	1	-1	-1	1	1	2	2	2	2	0
					-1	-1	1	-1	1	1	-1	-1	1	1	-1	3	2	0	1	1
																	2	1	0	2

We will consider the second design first. It is a full-factorial design, and hence it is trivially constructed by making all combinations of the levels.

Next, consider the following matrix:

$$\mathbf{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

This matrix is called a Hadamard matrix of order 2. In an experiment with two treatments, the first column represents the intercept or grand mean, and the second column represents a contrast between the two group means. Next, consider the Kronecker product of $\mathbf{H}_2$ with itself:

$$\mathbf{H}_4 = \mathbf{H}_2 \otimes \mathbf{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} h_{11}\mathbf{H}_2 & h_{12}\mathbf{H}_2 \\ h_{21}\mathbf{H}_2 & h_{22}\mathbf{H}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{H}_2 & \mathbf{H}_2 \\ \mathbf{H}_2 & -\mathbf{H}_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

The last three columns of $\mathbf{H}_4$ form the design 2^3 in four runs and the first column provides an intercept. $\mathbf{H}_4$ also matches the first four rows of the two-level factors in $2^4 4^1$ in eight runs, and $-\mathbf{H}_4$ matches the last four rows. The last column of $2^4 4^1$ consists of two repetitions of (0, 1, 2, 3).

Next, consider the Kronecker product of $\mathbf{H}_2$ with $\mathbf{H}_4$:

$$\mathbf{H}_8 = \mathbf{H}_2 \otimes \mathbf{H}_4 = \begin{bmatrix} \mathbf{H}_4 & \mathbf{H}_4 \\ \mathbf{H}_4 & -\mathbf{H}_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix}$$

The last seven columns of $\mathbf{H}_8$ form the design 2^7 in eight runs and the first column provides an intercept.

Next, consider the following matrix product:

$$\mathbf{D}_3 = \begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} \begin{bmatrix} 0 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 2 & 1 \end{bmatrix}$$

Note that arithmetic in this case is field arithmetic, not real number arithmetic, so all results are mod 3. More is said about this later in this section. In this case, the only result different from real number arithmetic is $2 \times 2 = 1$ (which comes from $2 \times 2 \bmod 3 = 4 \bmod 3 = 1$, where $n \bmod m$ returns the remainder after dividing n by m). $\mathbf{D}_3$ matches the first three rows and columns of 3^4 in nine runs. The full first three columns are as follows:

$$\begin{bmatrix} 0 \\ 1 \\ 2 \end{bmatrix} \oplus \mathbf{D}_3 = \begin{bmatrix} 0 + \mathbf{D}_3 \\ 1 + \mathbf{D}_3 \\ 2 + \mathbf{D}_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 2 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 0 \\ 1 & 0 & 2 \\ 2 & 2 & 2 \\ 2 & 0 & 1 \\ 2 & 1 & 0 \end{bmatrix}$$

The $\oplus$ operator denotes an operation like the Kronecker product but elements are added rather than multiplied. Again, arithmetic is mod 3 (hence, $2 + 1 = 0$, $2 + 2 = 1$). The last column of 3^4 consists of three repetitions of (0, 1, 2). The matrix $\mathbf{D}_3$ is called a *difference scheme*, and this construction method is called "developing a difference scheme."

An orthogonal array $p^1 m^q$ in $p \times m$ runs is made by developing a difference scheme (Wangand Wu 1991). A difference scheme is a matrix that is a "building block" used in the construction of many orthogonal arrays. It is called a difference scheme because if you subtract any two columns, all differences occur equally often. Note that like field addition, subtraction in a field is quite different from subtraction in the real number system. Here, arithmetic operations are in a Galois or abelian field. Explaining this fully is beyond the scope of this discussion, but we will provide an example. Specifically, we consider the case where $p = m = q = 5$. The following tables show the addition, subtraction, multiplication, and inversion tables that are used in a Galois field of order 5 (GF(5)):

Addition

	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Subtraction

	0	1	2	3	4
0	0	4	3	2	1
1	1	0	4	3	2
2	2	1	0	4	3
3	3	2	1	0	4
4	4	3	2	1	0

Multiplication

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Inverse

0	
1	1
2	3
3	2
4	4

These tables are used when constructing factors with five levels (0 1 2 3 4). In this case, since the order, 5, is a prime number, the rules for addition and multiplication follow the rules for integer arithmetic mod 5. For example, $4 + 4 \bmod 5 = 8 \bmod 5 = 3$ and $4 \times 4 \bmod 5 = 16 \bmod 5 = 1$. These results can also be seen by accessing the row 4, column 4 entries of the addition and multiplication tables. The rules for subtraction can easily be derived from the rules for addition, and the rules for inversion can easily be derived from the rules for multiplication. For example, since $4 + 3 = 2$ in GF(5), then $4 = 2 - 3$, and since $3 \times 2 = 1$, then 2 is the inverse of 3. Note that in many cases, the rules for field arithmetic are not this simple. In some cases, such as when the order of the field is a power of a prime (4, 8, 9 ...) or a composite number that contains a power of a prime (12, 18, ...), the rules are much more complicated, and modulo m arithmetic does not work.

Let $\ell_5' = [0\ 1\ 2\ 3\ 4]$ be a row vector with the field elements and $\mathbf{1}_5$ be a column vector with 5 ones. In GF(5), the multiplication table is a 5×5 difference scheme, $\mathbf{D}_5 = \ell_5 \ell_5'$ (where $\ell_5 \ell_5'$ arithmetic, of course, occurs in GF(5)). You can verify that if you subtract every column from every other column, the five elements in ℓ_5 all occur exactly once in all of the difference vectors. The following matrix is the orthogonal array 5^6 in 25 runs:

$$\left[\ \mathbf{1}_5 \otimes \ell_5 \quad \ell_5 \oplus \mathbf{D}_5\ \right] = \begin{bmatrix} \ell_5 & 0 + \mathbf{D}_5 \\ \ell_5 & 1 + \mathbf{D}_5 \\ \ell_5 & 2 + \mathbf{D}_5 \\ \ell_5 & 3 + \mathbf{D}_5 \\ \ell_5 & 4 + \mathbf{D}_5 \end{bmatrix}$$

This matrix is partitioned vertically into five blocks of five rows and horizontally into a column followed by a set of five columns. For each new row block, the difference scheme is shifted by adding 1 to the previous matrix. The resulting orthogonal array is shown in Table 15 on the left, and the generic choice design made by sorting this orthogonal array on the first factor is shown on the right.

This is the same orthogonal array that `%MktEx` produces except that by default, `%MktEx` uses one-based integers instead of a zero base. For a generic choice design, the difference scheme provides levels for the first alternative, and all other alternatives are made from the previous alternative by adding 1 in the appropriate field.

You can use the `%MktEx` macro to make difference schemes by selecting just the right rows and columns of a design. The following steps create and display a 6×6 difference scheme of order 3:

```
%mktex(3 ** 6 6,                    /* factors, difference scheme in 3 ** 6 */
       n=18,                        /* number of runs in full design        */
       options=nosort,              /* do not sort the design               */
       levels=0)                    /* make levels 0, 1, 2 (not 1, 2, 3)    */

proc print data=design(obs=6) noobs; var x1-x6; run;
```

Table 15

Orthogonal Array						Generic Choice Design						
						Set	Attributes					
0	0	0	0	0	0	0	0	0	0	0	0	
1	0	1	2	3	4	0	1	1	1	1	1	
2	0	2	4	1	3	0	2	2	2	2	2	
3	0	3	1	4	2	0	3	3	3	3	3	
4	0	4	3	2	1	0	4	4	4	4	4	
0	1	1	1	1	1	1	0	1	2	3	4	
1	1	2	3	4	0	1	1	2	3	4	0	
2	1	3	0	2	4	1	2	3	4	0	1	
3	1	4	2	0	3	1	3	4	0	1	2	
4	1	0	4	3	2	1	4	0	1	2	3	
0	2	2	2	2	2	2	0	2	4	1	3	
1	2	3	4	0	1	2	1	3	0	2	4	
2	2	4	1	3	0	2	2	4	1	3	0	
3	2	0	3	1	4	2	3	0	2	4	1	
4	2	1	0	4	3	2	4	1	3	0	2	
0	3	3	3	3	3	3	0	3	1	4	2	
1	3	4	0	1	2	3	1	4	2	0	3	
2	3	0	2	4	1	3	2	0	3	1	4	
3	3	1	4	2	0	3	3	1	4	2	0	
4	3	2	1	0	4	3	4	2	0	3	1	
0	4	4	4	4	4	4	0	4	3	2	1	
1	4	0	1	2	3	4	1	0	4	3	2	
2	4	1	3	0	2	4	2	1	0	4	3	
3	4	2	0	3	1	4	3	2	1	0	4	
4	4	3	2	1	0	4	4	3	2	1	0	

The difference scheme is as follows:

```
 x1    x2    x3    x4    x5    x6

 0     0     0     0     0     0
 0     0     1     1     2     2
 0     1     0     2     2     1
 0     1     2     0     1     2
 0     2     1     2     1     0
 0     2     2     1     0     1
```

The difference scheme is the matrix $\mathbf{D}_6$ from the orthogonal array $3^6 6^1$ in 18 runs which is created by `%MktEx` as follows:

$$\left[\begin{array}{cc} \ell_3 \oplus \mathbf{D}_6 & \mathbf{1}_3 \otimes \ell_6 \end{array}\right] = \left[\begin{array}{cc} 0+\mathbf{D}_6 & \ell_6 \\ 1+\mathbf{D}_6 & \ell_6 \\ 2+\mathbf{D}_6 & \ell_6 \end{array}\right]$$

While it is easy to make an $m \times m$ difference scheme of order m when m is prime using mod m arithmetic, it is hard to make a difference scheme of order $2m \times 2m$ order m and most larger difference schemes without software such as the `%MktEx` macro.

You can easily verify that this difference scheme, using each row (plus one) as a first alternative, and cyclic shifting for the other alternatives, makes the optimal choice design on page 109. You could construct the same design directly from the difference scheme as follows:

```
data choice(keep=set x1-x6);
   Set = _n_;
   set design(obs=6);
   array x[6];
   do i = 1 to 6; x[i] + 1; end;
   output;
   do i = 1 to 6; x[i] = mod(x[i], 3) + 1; end;
   output;
   do i = 1 to 6; x[i] = mod(x[i], 3) + 1; end;
   output;
   run;

proc print; id set; by set; run;
```

The design is follows:

Set	x1	x2	x3	x4	x5	x6
1	1	1	1	1	1	1
	2	2	2	2	2	2
	3	3	3	3	3	3
2	1	1	2	2	3	3
	2	2	3	3	1	1
	3	3	1	1	2	2
3	1	2	1	3	3	2
	2	3	2	1	1	3
	3	1	3	2	2	1
4	1	2	3	1	2	3
	2	3	1	2	3	1
	3	1	2	3	1	2

```
5       1       3       2       3       2       1
        2       1       3       1       3       2
        3       2       1       2       1       3

6       1       3       3       2       1       2
        2       1       1       3       2       3
        3       2       2       1       3       1
```

Canonical Correlations

We use canonical correlations to evaluate nonorthogonal designs and the extent to which factors are correlated or are not independent. To illustrate, consider the following design with four three-level factors in 9 runs:

Linear Arrangement

x1	x2	x3	x4
1	1	1	1
1	2	3	3
1	3	2	2
2	1	2	3
2	2	1	2
2	3	3	1
3	1	3	2
3	2	2	1
3	3	1	3

Coded Linear Arrangement

x1			x2			x3			x4		
1	2	3	1	2	3	1	2	3	1	2	3
1	0	0	1	0	0	1	0	0	1	0	0
1	0	0	0	1	0	0	0	1	0	0	1
1	0	0	0	0	1	0	1	0	0	1	0
0	1	0	1	0	0	0	1	0	0	0	1
0	1	0	0	1	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0	1	1	0	0
0	0	1	1	0	0	0	0	1	0	1	0
0	0	1	0	1	0	0	1	0	1	0	0
0	0	1	0	0	1	1	0	0	0	0	1

Each three-level factor can be coded with three columns that contain the less-than-full-rank binary coding (see page 73). A factor can be recoded by applying a coefficient vector $\boldsymbol{\alpha}' = (\alpha_1\ \alpha_2\ \alpha_3)$ or $\boldsymbol{\beta}' = (\beta_1\ \beta_2\ \beta_3)$ to a coded factor to create a single column. In other words, the original coding of (1 2 3) can be replaced with arbitrary $(\alpha_1\ \alpha_2\ \alpha_3)$ or $(\beta_1\ \beta_2\ \beta_3)$. If two factors are orthogonal, then for all choices of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, the simple correlation between recoded columns is zero. A *canonical correlation* shows the maximum correlation between two recoded factors that can be obtained with the optimal $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$. This design, 3^4 in 9 runs is orthogonal so for all pairs of factors and all choices of $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, the simple correlation between recoded factors is zero. The canonical correlation between a factor and itself is 1.0.

For nonorthogonal designs and designs with interactions, the canonical-correlation matrix is not a substitute for looking at the variance matrix discussed on pages 351, 425, and 1058. It just provides a quick and more-compact picture of the correlations between the factors. The variance matrix is sensitive to the actual model specified and the actual coding. The canonical-correlation matrix just tells you if there is some correlation between the main effects. A matrix of canonical correlations provides a useful picture of the orthogonality or lack of orthogonality in a design. For example, the following canonical-correlation matrix from the vacation example on page 350, shows a design with 16 factors that are mostly orthogonal:

Canonical Correlation Matrix

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15	x16
x1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
x4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
x5	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
x6	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
x7	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
x8	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
x9	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
x10	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
x11	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
x12	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
x13	0	0	0	0	0	0	0	0	0	0	0	0	1	0.25	0.25	0
x14	0	0	0	0	0	0	0	0	0	0	0	0	0.25	1	0.25	0
x15	0	0	0	0	0	0	0	0	0	0	0	0	0.25	0.25	1	0
x16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

However, `x13-x15` are not orthogonal to each other. Still, with $r^2 = 0.25^2 = 0.0625$, these factors are nearly independent.

Optimal Generic Choice Designs

In some situations, particularly for certain generic choice experiments, we can make optimal choice designs under the assumption that $\boldsymbol{\beta} = 0$. The idea of optimal generic choice designs started with the work of Don Anderson (Bunch, Louviere, and Anderson 1996) who introduced the idea of creating these designs by shifting alternatives in orthogonal arrays. We approach optimal generic designs only slightly differently—from the point of view of difference scheme development and orthogonal array selection. The approach I discuss here grew from numerous discussions that I have had with Don Anderson over the years. Optimal generic choice designs are discussed extensively (much more extensively than here) by Street and Burgess (2007) who provide a great deal of theory.

Our goal in this section is to construct optimal generic designs and get a report of the design's efficiency on a 0 to 100 scale like we get with linear model designs. A generic choice experiment is one that does not have any brands. The alternatives are simply bundles of attributes. For example, a manufacturer of any electronic product might construct a choice study with potential variations on a new product to see which attributes are the most important drivers of choice. Consider a study that involves 4 two-level factors and four choice sets, each with two alternatives. The following tables display an the optimal generic choice design and show how it is constructed:

Optimal Generic Choice Design

1	1	1	1
2	2	2	2
1	1	2	2
2	2	1	1
1	2	2	1
2	1	1	2
1	2	1	2
2	1	2	1

Fractional Factorial 2^3 in 4 Runs With Intercept

1	1	1	1
1	1	2	2
1	2	2	1
1	2	1	2

Shifted Fractional Factorial

2	2	2	2
2	2	1	1
2	1	1	2
2	1	2	1

Orthogonal Array $4^1 2^4$ in 8 Runs

1	1	1	1	1
2	1	1	2	2
3	1	2	2	1
4	1	2	1	2
1	2	2	2	2
2	2	2	1	1
3	2	1	1	2
4	2	1	2	1

Sorted Orthogonal Array

1	1	1	1	1
1	2	2	2	2
2	1	1	2	2
2	2	2	1	1
3	1	2	2	1
3	2	1	1	2
4	1	2	1	2
4	2	1	2	1

The fractional-factorial design consists of 3 two-level factors in 4 runs and an intercept (the customary column of ones). The shifted design consists of 3 two-level factors in 4 runs along with a column of twos (the intercept plus 1) instead of the customary column of ones. The final table is a mixed orthogonal array with 1 four-level factor and 4 two-level factors in eight runs. Using the notation discussed in the section beginning on page 95, this design is constructed by adding 1 to the following orthogonal array:

$$\left[\begin{array}{cc} \mathbf{1}_2 \otimes \ell_4 & \ell_2 \oplus \mathbf{D}_4 \end{array}\right] = \left[\begin{array}{cc} \ell_4 & 0 + \mathbf{D}_4 \\ \ell_4 & 1 + \mathbf{D}_4 \end{array}\right]$$

The first fractional-factorial design exactly matches the two-level factors in the first half of the third fractional-factorial design, and the second table exactly matches the two-level factors in the second half of the fractional-factorial design in the third table. Sorting this design on the four-level factor and using the four-level factor as the choice set number yields the optimal generic choice design.

The optimal generic choice design is constructed by creating a fractional-factorial design with an intercept and using it to make the first alternative of each choice set. The second alternative is made from the first by shifting or cycling through the levels (changing 1 to 2 and 2 to 1). The first alternative is shown in the fractional-factorial table, and the second alternative is shown in shifted fractional-factorial table. The plan for the second alternative is a different fractional-factorial plan. Alternatively, equivalently, and more clearly, this design can be made from the orthogonal array $4^1 2^4$ in 8 runs by using the four-level factor as the choice set number. Note that the optimal generic choice design never shows two alternatives with the same levels of any factor. For this reason, some researchers do not use them and consider this class of designs to be more of academic and combinatorial interest than of practical significance.

A randomized version of this design (where the first choice set will not consist of constant attributes within each alternative) is constructed and evaluated as follows:

```
%mktex(4 2 ** 4,                    /* choice set number and attr levels    */
       n=8)                         /* 8 runs - 4 sets, two alts each       */

%mktlab(data=randomized,            /* randomized design                    */
        vars=Set x1-x4)             /* var names for set var and for attrs  */

proc sort; by set; run;

proc print; by set; id set; run;

%choiceff(data=final,               /* candidate set of choice sets         */
          init=final(keep=set),     /* select these sets from candidates    */
          model=class(x1-x4 / sta), /* model with stdzd orthogonal coding   */
          nsets=4,                  /* 4 choice sets                        */
          nalts=2,                  /* 2 alternatives per set               */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

The following boxes show an optimal generic choice design with 9 three-level attributes, with three alternatives, and nine choice sets, each in a separate box:

1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3

1	2	3	3	1	2	2	3	1
2	3	1	1	2	3	3	1	2
3	1	2	2	3	1	1	2	3

1	1	1	2	2	2	3	3	3
2	2	2	3	3	3	1	1	1
3	3	3	1	1	1	2	2	2

1	3	2	1	3	2	1	3	2
2	1	3	2	1	3	2	1	3
3	2	1	3	2	1	3	2	1

1	1	1	3	3	3	2	2	2
2	2	2	1	1	1	3	3	3
3	3	3	2	2	2	1	1	1

1	3	2	2	1	3	3	2	1
2	1	3	3	2	1	1	3	2
3	2	1	1	3	2	2	1	3

1	2	3	1	2	3	1	2	3
2	3	1	2	3	1	2	3	1
3	1	2	3	1	2	3	1	2

1	3	2	3	2	1	2	1	3
2	1	3	1	3	2	3	2	1
3	2	1	2	1	3	1	3	2

1	2	3	2	3	1	3	1	2
2	3	1	3	1	2	1	2	3
3	1	2	1	2	3	2	3	1

It is made from the orthogonal design $3^9 9^1$ in 27 runs by using the nine-level factor as the choice set number. Notice that each alternative is made from the previous alternative by adding one to the previous level, mod 3.[†] Similarly, the first alternative is made from the third alternative by adding one to the previous level, mod 3. A randomized version of this design (where the first choice set will not consist of constant attributes within each alternative) is constructed and evaluated as follows:

[†]More precisely, since these numbers are based on one instead of zero, the operation is: $(x \bmod 3) + 1$.

```
%mktex(9 3 ** 9,                    /* choice set number and attr levels    */
       n=27)                        /* 27 runs - 9 sets, 3 alts each        */

%mktlab(data=randomized,            /* randomized design                    */
        vars=Set x1-x9)             /* var names for set var and for attrs  */

proc sort; by set; run;

proc print; by set; id set; run;

%choiceff(data=final,               /* candidate set of choice sets         */
          init=final(keep=set),     /* select these sets from candidates    */
          model=class(x1-x9 / sta), /* model with stdzd orthogonal coding   */
          nsets=9,                  /* 9 choice sets                        */
          nalts=3,                  /* 3 alternatives per set               */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

An optimal generic choice design with 8 four-level attributes, with four alternatives, and eight choice sets, each in a separate box is shown as follows:

1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4

1	3	2	1	4	2	3	4
2	4	1	2	3	1	4	3
3	1	4	3	2	4	1	2
4	2	3	4	1	3	2	1

1	1	3	4	2	2	4	3
2	2	4	3	1	1	3	4
3	3	1	2	4	4	2	1
4	4	2	1	3	3	1	2

1	3	4	4	3	1	2	2
2	4	3	3	4	2	1	1
3	1	2	2	1	3	4	4
4	2	1	1	2	4	3	3

1	2	2	3	3	4	4	1
2	1	1	4	4	3	3	2
3	4	4	1	1	2	2	3
4	3	3	2	2	1	1	4

1	4	1	3	2	3	2	4
2	3	2	4	1	4	1	3
3	2	3	1	4	1	4	2
4	1	4	2	3	2	3	1

1	2	4	2	4	3	1	3
2	1	3	1	3	4	2	4
3	4	2	4	2	1	3	1
4	3	1	3	1	2	4	2

1	4	3	2	1	4	3	2
2	3	4	1	2	3	4	1
3	2	1	4	3	2	1	4
4	1	2	3	4	1	2	3

It is made from the fractional-factorial design $4^8 8$ in 32 runs by using the eight-level factor as the choice set number. Notice that every attribute has all four levels in each factor. With four-level factors, the rules that are used to make orthogonal arrays are more complicated than the mod 3 addition that is used with three-level factors, so you do not get the same pattern of shifted results that we saw previously. A randomized version of this design (where the first choice set will not consist of constant attributes within each alternative) is constructed and evaluated as follows:

```
%mktex(8 4 ** 8,                    /* choice set number and attr levels    */
       n=32)                        /* 32 runs - 8 sets, four alts each     */

%mktlab(data=randomized,            /* randomized design                    */
        vars=Set x1-x8)             /* var names for set var and for attrs  */

proc sort; by set; run;

proc print; by set; id set; run;

%choiceff(data=final,               /* candidate set of choice sets         */
          init=final(keep=set),     /* select these sets from candidates    */
          model=class(x1-x8 / sta), /* model with stdzd orthogonal coding   */
          nsets=8,                  /* 8 choice sets                        */
          nalts=4,                  /* 4 alternatives per set               */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

If you need a generic choice design and you do not have the level of symmetry shown in these examples (all m-level factors with m alternatives) then you can use the `%ChoicEff` macro to find an efficient generic design using the methods shown on page 198 and in the chair example on page 556. Also see the documentation for the `%ChoicEff` macro beginning on page 806 for examples of generic design construction.

An interesting class of optimal generic designs can be constructed for experiments with p choice sets and m-level factors with m alternatives when there is an orthogonal array $p^1 m^q$ in $p \times m$ runs where $q \leq p$. We can process the design catalog from the `%MktOrth` macro to find these as follows:

```
%mktorth(maxn=100,                  /* output up to 100 runs                */
         options=parent)            /* just list the parent designs         */

data x;
   set mktdeslev;
   array x[50];
   gotone = 0;
   do p = 50 to 1 by -1 until(gotone);
      if x[p] eq 1 then gotone = 1;
      end;
   if not gotone then do;
      p = sqrt(n);
      if abs(p * p - n) < 1e-8 then if x[p] then gotone = 1;
      end;
```

```
      if gotone then do;
         m = n / p;
         if m eq p then x[m] + -1;
         q = x[m];
         design = compbl(put(m, 5.) || ' ** ' || put(q, 5.));
         From   = compbl(put(p, 5.) || ' ** 1 ' ||
                         trim(design) || ', n=' || put(n, 5. -L));
         if (n le 10 and q > 1) or (n gt 10 and q > 2) then output;
         end;
      run;

   proc sort;
      by n p descending q;
      run;

   data list(keep=sets alts design from);
      length Sets Alts 8;
      set x;
      by n p;
      if first.p;
      sets = p;
      alts = n / p;
      run;

   proc sort;
      by sets descending alts;
      run;

   proc print; by sets; id sets; run;
```

The final `if` statement in the first DATA step filters out designs that technically meet this definition, but are mostly uninteresting. Specifically, a number of designs have exactly 2 two-level factors in varying numbers of choice sets. These are filtered out by the last clause of the `if` statement. A few of the smaller designs that work are as follows:

Sets	Alts	Design	From
2	2	2 ** 2	2 ** 1 2 ** 2, n=4
3	3	3 ** 3	3 ** 1 3 ** 3, n=9
4	4	4 ** 4	4 ** 1 4 ** 4, n=16
	2	2 ** 4	4 ** 1 2 ** 4, n=8
5	5	5 ** 5	5 ** 1 5 ** 5, n=25
6	3	3 ** 6	6 ** 1 3 ** 6, n=18
7	7	7 ** 7	7 ** 1 7 ** 7, n=49

```
 8       8     8 ** 8     8 ** 1 8 ** 8, n=64
         4     4 ** 8     8 ** 1 4 ** 8, n=32
         2     2 ** 8     8 ** 1 2 ** 8, n=16

 9       9     9 ** 9     9 ** 1 9 ** 9, n=81
         3     3 ** 9     9 ** 1 3 ** 9, n=27

10      10     10 ** 3    10 ** 1 10 ** 3, n=100
         5     5 ** 10    10 ** 1 5 ** 10, n=50

12       6     6 ** 6     12 ** 1 6 ** 6, n=72
         4     4 ** 12    12 ** 1 4 ** 12, n=48
         3     3 ** 12    12 ** 1 3 ** 12, n=36
         2     2 ** 12    12 ** 1 2 ** 12, n=24

14       7     7 ** 14    14 ** 1 7 ** 14, n=98

15       5     5 ** 8     15 ** 1 5 ** 8, n=75
         3     3 ** 9     15 ** 1 3 ** 9, n=45

16       4     4 ** 16    16 ** 1 4 ** 16, n=64
         2     2 ** 16    16 ** 1 2 ** 16, n=32

18       3     3 ** 18    18 ** 1 3 ** 18, n=54

20       5     5 ** 20    20 ** 1 5 ** 20, n=100
         4     4 ** 10    20 ** 1 4 ** 10, n=80
         2     2 ** 20    20 ** 1 2 ** 20, n=40

21       3     3 ** 12    21 ** 1 3 ** 12, n=63

24       4     4 ** 20    24 ** 1 4 ** 20, n=96
         3     3 ** 24    24 ** 1 3 ** 24, n=72
         2     2 ** 24    24 ** 1 2 ** 24, n=48

27       3     3 ** 27    27 ** 1 3 ** 27, n=81

28       2     2 ** 28    28 ** 1 2 ** 28, n=56

30       3     3 ** 30    30 ** 1 3 ** 30, n=90

32       2     2 ** 32    32 ** 1 2 ** 32, n=64

33       3     3 ** 13    33 ** 1 3 ** 13, n=99

36       2     2 ** 36    36 ** 1 2 ** 36, n=72

40       2     2 ** 40    40 ** 1 2 ** 40, n=80

44       2     2 ** 44    44 ** 1 2 ** 44, n=88
```

```
48       2     2 ** 48    48 ** 1 2 ** 48, n=96
```

It is important to note that this is not the complete list of small orthogonal arrays that can be developed into optimal generic choice designs. Rather these are the most interesting ones that work in the symmetric case, that is, where all levels of all factors are the same. More is said about this later in this section.

For a specification of p, q, and m, assuming the orthogonal array $p^1 m^q$ in pm runs exists, the following steps make an optimal generic choice design and use the `%ChoicEff` macro to evaluate the results:

```
%let p = 6;                          /* p - number of choice sets           */
%let m = 3;                          /* m-level factors                     */
%let q = &p;                         /* q - number of factors               */

%mktex(&p &m ** &q,                  /* choice set number and attr levels   */
       n=&p * &m)                    /* p * m runs - p sets, m alts each    */

%mktlab(data=design,                 /* orthogonal array                    */
        vars=Set x1-x&q)             /* var names for set var and for attrs */

proc print; id set; by set; run;

%choiceff(data=final,                /* candidate set of choice sets        */
          init=final(keep=set),      /* select these sets from candidates   */
          model=class(x1-x&q / sta),/* model with stdzd orthogonal coding   */
          nsets=&p,                  /* &p choice sets                      */
          nalts=&m,                  /* &m alternatives per set             */
          options=relative,          /* display relative D-efficiency       */
          beta=zero)                 /* assumed beta vector, Ho: b=0        */
```

The `sta` (short for `standorth`) option in the `model=` option is new with SAS 9.2 and requests a standardized orthogonal contrast coding. You must specify this coding if you want to see relative D-efficiency on a 0 to 100 scale. Relative D-efficiency is not displayed by default since it is only meaningful for a limited class of designs such as these designs. You must request it with `options=relative`. Relative D-efficiency is explained in more detail in starting on page 81 and in this example as more results are presented. The following example has $m = 3$ and $p = q = 6$ choice sets:

Set	x1	x2	x3	x4	x5	x6
1	1	1	1	1	1	1
	2	2	2	2	2	2
	3	3	3	3	3	3
2	1	1	2	2	3	3
	2	2	3	3	1	1
	3	3	1	1	2	2

```
3       1       2       1       3       3       2
        2       3       2       1       1       3
        3       1       3       2       2       1

4       1       2       3       1       2       3
        2       3       1       2       3       1
        3       1       2       3       1       2

5       1       3       2       3       2       1
        2       1       3       1       3       2
        3       2       1       2       1       3

6       1       3       3       2       1       2
        2       1       1       3       2       3
        3       2       2       1       3       1
```

The results summary table that follows the iteration history is as follows:

```
                Final Results

Design                       1
Choice Sets                  6
Alternatives                 3
Parameters                  12
Maximum Parameters          12
D-Efficiency            6.0000
Relative D-Eff        100.0000
D-Error                 0.1667
1 / Choice Sets         0.1667
```

This table shows that *D*-Efficiency is 6. *D*-Error is $1/6 \approx 0.1667$, since *D*-Error is always the inverse of the *D*-Efficiency. It shows that a design with a relative *D*-efficiency of 100 was found. It also shows that this design is saturated—the number of parameters and the maximum number of parameters are the same. The parameters names and their variances under the null hypothesis that $\boldsymbol{\beta} = \mathbf{0}$ are as follows:

n	Variable Name	Label	Variance	DF	Standard Error
1	x11	x1 1	0.16667	1	0.40825
2	x12	x1 2	0.16667	1	0.40825
3	x21	x2 1	0.16667	1	0.40825
4	x22	x2 2	0.16667	1	0.40825
5	x31	x3 1	0.16667	1	0.40825
6	x32	x3 2	0.16667	1	0.40825
7	x41	x4 1	0.16667	1	0.40825
8	x42	x4 2	0.16667	1	0.40825
9	x51	x5 1	0.16667	1	0.40825
10	x52	x5 2	0.16667	1	0.40825
11	x61	x6 1	0.16667	1	0.40825
12	x62	x6 2	0.16667	1	0.40825
				==	
				12	

Since this optimal design and the standardized orthogonal contrast coding is used, the variances are all exactly the same as the *D*-Error, and *D*-Error equals one over the number of choice sets. For nonoptimal designs and with this coding, you would expect that one or more of the variances to exceed one over the number of choice sets. The variances are the diagonal of the covariance matrix. You can examine the covariance matrix for the parameters for the choice model under the assumption that $\boldsymbol{\beta} = 0$ as follows:

```
proc format;
   value zer -1e-12 - 1e-12 = ' 0   ';
   run;

proc print data=bestcov label;
   id __label;
   label __label = '00'x;
   var x:;
   format _numeric_ zer5.2;
   run;
```

The format displays values very close to zero as precisely zero to make a better display.

The results are as follows:

	x1 1	x1 2	x2 1	x2 2	x3 1	x3 2	x4 1	x4 2	x5 1	x5 2	x6 1	x6 2
x1 1	0.17	0	0	0	0	0	0	0	0	0	0	0
x1 2	0	0.17	0	0	0	0	0	0	0	0	0	0
x2 1	0	0	0.17	0	0	0	0	0	0	0	0	0
x2 2	0	0	0	0.17	0	0	0	0	0	0	0	0
x3 1	0	0	0	0	0.17	0	0	0	0	0	0	0
x3 2	0	0	0	0	0	0.17	0	0	0	0	0	0
x4 1	0	0	0	0	0	0	0.17	0	0	0	0	0
x4 2	0	0	0	0	0	0	0	0.17	0	0	0	0
x5 1	0	0	0	0	0	0	0	0	0.17	0	0	0
x5 2	0	0	0	0	0	0	0	0	0	0.17	0	0
x6 1	0	0	0	0	0	0	0	0	0	0	0.17	0
x6 2	0	0	0	0	0	0	0	0	0	0	0	0.17

The covariance matrix equals $p^{-1}\mathbf{I} = \frac{1}{6}\mathbf{I}$. With an optimal generic design such as this, the covariance matrix is diagonal, and each diagonal value is one over the number of choice sets. Hence, the minimum *D*-error is one over the number of choice sets, and the maximum *D*-efficiency is the number of choice sets. The number of choice sets is used to scale *D*-efficiency to get the relative *D*-efficiency.

The preceding discussion has concerned symmetric designs. A design is said to be *symmetric* when all of the attributes have the same number of levels. When at least one attribute has a different number of levels from at least one other attribute, the design is asymmetric. Designs that are optimal, generic, and asymmetric can also be constructed from orthogonal arrays. For example, using the orthogonal array $2^{27}3^{11}6^{1}12^{1}$ in 72 runs, you can construct an optimal generic choice design with 12 choice sets, 6 alternatives, for attributes $2^{27}3^{11}6^{1}$ as follows:

```
%mktex(12 2 ** 27 3 ** 11 6,        /* Set and factor levels               */
       n=72)                        /* num of sets times num of alts (12x6) */

%mktlab(data=design,                /* design from MktEx                   */
        vars=Set x1-x39)            /* new variable names                  */

%choiceff(data=final,               /* candidate set of choice sets        */
          init=final(keep=set),     /* select these sets from candidates   */
          model=class(x1-x39 / sta),/* model with stdzd orthogonal coding  */
          nsets=12,                 /* 12 choice sets                      */
          nalts=6,                  /* 6 alternatives per set              */
          options=relative,         /* display relative D-efficiency       */
          beta=zero)                /* assumed beta vector, Ho: b=0        */

proc print; by set; id set; var x:; run;
```

The last part of the results and the first part of the design are as follows:

Final Results

Design	1
Choice Sets	12
Alternatives	6
Parameters	54
Maximum Parameters	60
D-Efficiency	12.0000
Relative D-Eff	100.0000
D-Error	0.0833
1 / Choice Sets	0.0833

n	Variable Name	Label	Variance	DF	Standard Error
1	x11	x1 1	0.083333	1	0.28868
2	x21	x2 1	0.083333	1	0.28868
3	x31	x3 1	0.083333	1	0.28868
4	x41	x4 1	0.083333	1	0.28868
5	x51	x5 1	0.083333	1	0.28868
6	x61	x6 1	0.083333	1	0.28868
7	x71	x7 1	0.083333	1	0.28868
8	x81	x8 1	0.083333	1	0.28868
9	x91	x9 1	0.083333	1	0.28868
10	x101	x10 1	0.083333	1	0.28868
11	x111	x11 1	0.083333	1	0.28868
12	x121	x12 1	0.083333	1	0.28868
13	x131	x13 1	0.083333	1	0.28868
14	x141	x14 1	0.083333	1	0.28868
15	x151	x15 1	0.083333	1	0.28868
16	x161	x16 1	0.083333	1	0.28868
17	x171	x17 1	0.083333	1	0.28868
18	x181	x18 1	0.083333	1	0.28868
19	x191	x19 1	0.083333	1	0.28868
20	x201	x20 1	0.083333	1	0.28868
21	x211	x21 1	0.083333	1	0.28868
22	x221	x22 1	0.083333	1	0.28868
23	x231	x23 1	0.083333	1	0.28868
24	x241	x24 1	0.083333	1	0.28868
25	x251	x25 1	0.083333	1	0.28868
26	x261	x26 1	0.083333	1	0.28868
27	x271	x27 1	0.083333	1	0.28868
28	x281	x28 1	0.083333	1	0.28868
29	x282	x28 2	0.083333	1	0.28868
30	x291	x29 1	0.083333	1	0.28868
31	x292	x29 2	0.083333	1	0.28868

32	x301	x30	1	0.083333	1	0.28868
33	x302	x30	2	0.083333	1	0.28868
34	x311	x31	1	0.083333	1	0.28868
35	x312	x31	2	0.083333	1	0.28868
36	x321	x32	1	0.083333	1	0.28868
37	x322	x32	2	0.083333	1	0.28868
38	x331	x33	1	0.083333	1	0.28868
39	x332	x33	2	0.083333	1	0.28868
40	x341	x34	1	0.083333	1	0.28868
41	x342	x34	2	0.083333	1	0.28868
42	x351	x35	1	0.083333	1	0.28868
43	x352	x35	2	0.083333	1	0.28868
44	x361	x36	1	0.083333	1	0.28868
45	x362	x36	2	0.083333	1	0.28868
46	x371	x37	1	0.083333	1	0.28868
47	x372	x37	2	0.083333	1	0.28868
48	x381	x38	1	0.083333	1	0.28868
49	x382	x38	2	0.083333	1	0.28868
50	x391	x39	1	0.083333	1	0.28868
51	x392	x39	2	0.083333	1	0.28868
52	x393	x39	3	0.083333	1	0.28868
53	x394	x39	4	0.083333	1	0.28868
54	x395	x39	5	0.083333	1	0.28868
					==	
					54	

S										x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
e	x	x	x	x	x	x	x	x	x	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3
t	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1	2	1	2	2	2	1	1	1	1	2	1	1	2	1	2	2	2	1	1	2	1	2	2	2	1	1	3	3	3	3	3	3	3	3	3	3	6
	1	2	2	2	1	1	1	2	1	1	1	2	1	1	2	1	2	2	1	2	1	1	2	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	5
	2	1	1	1	2	2	2	1	2	2	2	1	2	2	1	2	1	1	2	1	2	2	1	2	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2
	2	1	2	1	1	1	2	2	2	2	1	2	2	1	2	1	1	1	2	2	1	2	1	1	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	4
2	1	2	1	1	1	2	2	2	1	2	1	1	1	2	2	2	1	2	2	1	2	1	1	1	2	2	2	1	1	1	2	2	3	3	2	2	3	3	1
	1	2	2	1	2	1	1	1	2	2	1	1	1	2	2	2	1	2	2	2	1	2	2	1	2	1	1	2	2	2	3	3	1	1	3	3	1	1	2
	1	2	2	2	1	1	1	2	1	2	2	2	1	1	1	2	1	1	2	1	1	2	1	2	2	2	1	3	3	3	1	1	2	2	1	1	2	2	6
	2	1	1	1	2	2	2	1	2	1	1	1	2	2	2	1	2	2	1	2	2	1	2	1	1	1	2	3	3	3	1	1	2	2	1	1	2	2	3
	2	1	1	2	1	2	2	2	1	1	2	2	2	1	1	1	2	1	1	1	2	1	1	2	1	2	2	2	2	2	3	3	1	1	3	3	1	1	5
	2	1	2	2	2	1	1	1	2	1	2	2	2	1	1	1	2	1	1	2	1	2	2	2	1	1	1	1	1	1	2	2	3	3	2	2	3	3	4

```
 3 1 1 2 1 2 2 2 1 1 1 2 1 2 2 2 1 1 1 2 1 1 2 1 2 2 2 1 2 3 3 2 2 3 3 1 1 1 1 5
   1 1 2 2 2 1 2 2 1 2 1 2 1 1 1 2 2 2 2 1 2 2 1 2 1 1 1 3 1 1 3 3 1 1 2 2 2 2 3
   1 2 2 2 1 1 1 2 1 1 2 1 2 2 2 1 1 1 2 1 2 2 2 1 1 1 2 1 2 2 1 1 2 2 3 3 3 3 4
   2 1 1 1 2 2 2 1 2 2 1 2 1 1 1 2 2 2 1 2 1 1 1 2 2 2 1 1 2 2 1 1 2 2 3 3 3 3 1
   2 2 1 1 1 2 1 1 2 1 2 1 2 2 2 1 1 1 1 2 1 1 2 1 2 2 2 3 1 1 3 3 1 1 2 2 2 2 6
   2 2 1 2 1 1 1 2 2 2 1 2 1 1 1 2 2 2 1 2 2 1 2 1 1 1 2 2 3 3 2 2 3 3 1 1 1 1 2
```

The covariance matrix is not displayed because of its size. However, we can get a summary of its values as follows:

```
proc iml;
   use bestcov(keep=x:); read all into x;
   x = round(shape(x,1)‘, 1e-12);
   create veccov from x; append from x;
   quit;

proc freq; run;
```

PROC IML is used to turn the matrix into a vector and round the values. Then PROC FREQ is used to summarize the results. The results are as follows:

The FREQ Procedure

COL1	Frequency	Percent	Cumulative Frequency	Cumulative Percent
0	2862	98.15	2862	98.15
0.0833333333	54	1.85	2916	100.00

There are 54 values (the number of parameters) equal to $1/12 \approx 0.0833333333$ (the 54 diagonal values), and the rest (the off-diagonal values) are all zero.

Block Designs

This section discusses balanced incomplete block designs (BIBDs), unbalanced block designs, and incomplete block designs. These are useful in MaxDiff experiments (see page 225) and for making partial profile designs (see page 207). We will begin with a more familiar factorial design example and then show how it is related to a block design. We can use the `%MktEx` macro to make an efficient factorial design with a six-level and a four-level factor in 12 runs as follows:

```
%mktex(6 4,                        /* factor levels                    */
       n=12,                       /* 12 runs                          */
       seed=513,                   /* random number seed               */
       options=nohistory)          /* do not print iteration history   */
```

The results are as follows:

The OPTEX Procedure

Class Level Information

Class	Levels	Values
x1	6	1 2 3 4 5 6
x2	4	1 2 3 4

Design Number	D-Efficiency	A-Efficiency	G-Efficiency	Average Prediction Standard Error
1	87.3580	75.0000	100.0000	0.8660

The following step displays the design:

```
proc print; run;
```

The results are as follows:

Obs	x1	x2
1	1	2
2	1	3
3	2	1
4	2	2
5	3	3
6	3	4
7	4	2
8	4	4
9	5	1
10	5	3
11	6	1
12	6	4

We can transpose this design and display the results as follows:

```
proc transpose data=design out=bd(drop=x1 _:) prefix=b; by x1; run;

proc print; run;
```

The results are as follows:

Obs	b1	b2
1	2	3
2	1	2
3	3	4
4	2	4
5	1	3
6	1	4

In this representation of the design, only the values that were in `x2` are displayed, and the values that were in `x1` are implicit. They are displayed in the row numbers, the column labeled "Obs" that PROC PRINT displays. This is an example of a block design. The design has $b = 6$ rows or blocks (and the first factor of our factorial design had 6 levels). It has $t = 4$ different values displayed (and the second factor of our factorial design had 4 levels). It has $k = 2$ columns (the $n = kb = 12$ values in `x2` divided by $b = 6$ blocks equals $k = 2$ columns). Notice that each of the $t = 4$ treatments occurs exactly $r = 3$ times in the design. Also notice that each of the $4(4-1)/2 = 6$ pairs of treatments ((1,2), (1,3), (1,4), (2,3), (2,4), (3,4)) occurs exactly $\lambda = 1$ time. These last two properties mean that this block design is a balanced incomplete block design. It is incomplete in the sense that each block has only a subset of the treatments. In contrast, the following design is complete since each treatment appears in every block:

1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

While the `%MktEx` macro can easily find small BIBDs like the one shown in this example, it does not provide the optimal approach to finding BIBDs. It does not use an optimal algorithm, nor does it provide for optimal formatting, processing, or display of the results. Instead, we will use the `%MktBIBD` macro as follows:

```
%mktbibd(b=6,                    /* 6 blocks                          */
         t=4,                    /* 4 treatments                      */
         k=2,                    /* 2 treatments in each block        */
         seed=350)               /* random number seed                */
```

The results are as follows:

Block Design Efficiency Criterion	100.0000
Number of Treatments, t	4
Block Size, k	2
Number of Blocks, b	6
Treatment Frequency	3
Pairwise Frequency	1
Total Sample Size	12
Positional Frequencies Optimized?	Yes

Treatment by Treatment Frequencies

	1	2	3	4
1	3	1	1	1
2		3	1	1
3			3	1
4				3

Treatment by Position Frequencies

	1	2
1	1	2
2	1	2
3	2	1
4	2	1

Balanced Incomplete Block Design

x1	x2
3	2
1	4
4	3
3	1
2	1
4	2

The first line of the output tells us that the design is 100% efficient. This means a BIBD was found. Previously, `%MktEx` found an equivalent design and reported that it was 87.358% efficient. The discrepancy is due to these two values being computed relative to different criteria. The `%MktEx` macro is reporting that relative to the variance matrix that an orthogonal array would have, this design is 87.358% efficient. An orthogonal array cannot exist for the design $6^1 4^1$ in 12 runs since 6×4 does not divide 12. Hence, relative to a hypothetical orthogonal array, these designs are 87.358% efficient, but relative to a BIBD they are 100% efficient.

The first matrix of results shows that each of the 4 treatments occurs 3 times, and each pair of treatments occurs once. The second matrix shows us that the first two treatments both occur in the first position once and in the second position twice. The opposite pattern occurs for the second two treatments. In some BIBDs, every treatment appears in every position the same number of times. In others, such as this one, that is not possible. The positions are optimized after the BIBD is found by the `%MktBIBD` macro. The `%MktEx` macro has no such facility. The last matrix is the BIBD.

The following step creates a block design with $t = 5$ treatments shown in $b = 5$ blocks of size $k = 2$:

```
%mktbibd(b=5,                    /* 5 blocks                          */
         t=5,                    /* 5 treatments                      */
         k=2,                    /* 2 treatments in each block        */
         seed=420)               /* random number seed                */
```

The results are as follows:

```
Block Design Efficiency Criterion        89.4427
Number of Treatments, t                        5
Block Size, k                                  2
Number of Blocks, b                            5
Average Treatment Frequency                    2
Average Pairwise Frequency                   0.5
Total Sample Size                             10
Positional Frequencies Optimized?            Yes

       Treatment by Treatment Frequencies

              1  2  3  4  5

         1  2  0  1  1  0
         2     2  1  0  1
         3        2  0  0
         4           2  1
         5              2

       Treatment by Position Frequencies

                 1  2

              1  1  1
              2  1  1
              3  1  1
              4  1  1
              5  1  1

               Design

              x1    x2

               5     4
               1     3
               2     5
               4     1
               3     2
```

This is an unbalanced block design. Each treatment occurs the same number of times (twice), but the pairwise frequencies are not equal, so it is not a BIBD. You can also see this by examining the block design efficiency criterion. It is less than 100, so the design is not a BIBD. The treatment by position frequencies, however, are perfect. For many purposes in marketing research, an unbalanced block design is adequate. However, we might be reluctant, depending on our purposes, to use a design where some treatments are never paired with other treatments. Unbalanced block designs occur for many specifications in which a BIBD is not possible.

Next, we will try the same specification again, but this time requesting one fewer block. The following step creates the block design:

```
%mktbibd(b=4,                    /* 4 blocks                          */
         t=5,                    /* 5 treatments                      */
         k=2,                    /* 2 treatments in each block        */
         seed=420)               /* random number seed                */
```

The results are as follows:

Block Design Efficiency Criterion	74.7674
Number of Treatments, t	5
Block Size, k	2
Number of Blocks, b	4
Average Treatment Frequency	1.6
Average Pairwise Frequency	0.4
Total Sample Size	8
Positional Frequencies Optimized?	Yes

Treatment by Treatment Frequencies

	1	2	3	4	5
1	2	1	1	0	0
2		1	0	0	0
3			2	0	1
4				1	1
5					2

Treatment by Position Frequencies

	1	2
1	1	1
2	1	0
3	1	1
4	0	1
5	1	1

Design

x1	x2
1	3
2	1
3	5
5	4

This is an incomplete block design. Each treatment does not occur the same number of times. Usually, we would prefer to have a BIBD or an unbalanced block design.

We will use BIBD's (and unbalanced block designs) in two ways in this book. First, they can provide one of the components for constructing a certain class of optimal partial-profile designs (Chrzan and Elrod 1995). In these partial-profile designs, there are t attributes, shown in b blocks of choice sets, where k attributes vary in each block, while the remaining $t - k$ attributes are held constant. That is the subject of the example starting on page 207. Second, they are used for MaxDiff designs. In a MaxDiff study (Louviere 1991, Finn and Louviere 1992), there are t attributes shown in b sets of size k. That is the subject of the example starting on page 225.

You can find the sizes in which a BIBD might be available for ranges of t, b, and k, using the `%MktBSize` macro as in the following example:

```
%mktbsize(t=5,                    /* 5 treatments                          */
          k=2 to 3,               /* 2 or 3 treatments per block           */
          b=2 to 20)              /* between 2 and 20 blocks               */
```

The results of this step are as follows:

t Number of Treatments	k Block Size	b Number of Blocks	r Treatment Frequency	Lambda Pairwise Frequencies	n Total Sample Size
5	2	10	4	1	20
5	3	10	6	3	30

You can see the unbalanced block designs as follows:

```
%mktbsize(t=5,                    /* 5 treatments                          */
          k=2 to 3,               /* 2 or 3 treatments per block           */
          b=2 to 20,              /* between 2 and 20 blocks               */
          options=ubd)            /* also show unbalanced block designs    */
```

The results of this step are as follows:

t Number of Treatments	k Block Size	b Number of Blocks	r Treatment Frequency	Lambda Pairwise Frequencies	n Total Sample Size
5	2	5	2	0.5	10
5	3	5	3	1.5	15

Note that these designs are precisely half the size of the designs that were listed previously. By default, the %MktBSize macro will not report designs whose sizes are multiples of other reported designs. You can specify maxreps=2 to get all of the designs as follows:

```
%mktbsize(t=5,                    /* 5 treatments                          */
          k=2 to 3,               /* 2 or 3 treatments per block           */
          b=2 to 20,              /* between 2 and 20 blocks               */
          options=ubd,            /* also show unbalanced block designs    */
          maxreps=2)              /* allow 1 or 2 replications             */
```

The results of this step are as follows:

t Number of Treatments	k Block Size	b Number of Blocks	r Treatment Frequency	Lambda Pairwise Frequencies	n Total Sample Size	Number of Replications
5	2	5	2	0.5	10	1
5	2	10	4	1	20	2
5	3	5	3	1.5	15	1
5	3	10	6	3	30	2

The %MktBSize macro reports on sizes that meet necessary but not sufficient criteria for the existence of BIBDs. When $r = b \times k/t$ and $l = r \times (k-1)/(t-1)$ are integers, and $k = t$ and $b \geq t$, then a complete block design might be possible. When $r = b \times k/t$ and $l = r \times (k-1)/(t-1)$ are integers, and $k < t$ and $b \geq t$, then a balanced incomplete block design might be possible. When r is an integer, then an unbalanced block design is possible. The %MktBIBD macro will not always find a BIBD even when one is known to exist. However, it usually works quite well in finding BIBDs for small specifications and at least a highly efficient block design for larger specifications.

Both macros have two sets of options for specifying b (b=b and nsets=b), t (t=t and nattrs=t), and k (k=k and setsize=k). When you specify t=t, you get the output shown previously. The b=b, t=t, and k=k options use a notation that is common for statistical applications. However, when you specify nattrs=t) the output of both the %MktBIBD and %MktBSize macros will use the word "Attribute" rather than "Treatment" and "Set" rather than "Block". The following step uses the alternative option names:

```
%mktbsize(nattrs=5,               /* 5 attributes                          */
          setsize=2 to 3,         /* 2 or 3 attributes per set             */
          nsets=2 to 20)          /* between 2 and 20 sets                 */
```

The results of this step are as follows:

t Number of Attributes	k Set Size	b Number of Sets	r Attribute Frequency	Lambda Pairwise Frequencies	n Total Sample Size
5	2	10	4	1	20
5	3	10	6	3	30

The Process of Designing a Choice Experiment

It is important that you understand a number of things in this chapter before you design your first choice experiment. Most of this chapter is fairly straight-forward, but without a clear understanding of it, you will no doubt get confused when you actually design an experiment. You should go back and review this chapter if you are not completely comfortable with the meaning of any of these terms: linear arrangement, choice design, generic choice design, factors, attributes, alternatives, choice sets, orthogonality, balance, and efficiency. In particular, the meaning of *linear arrangement* and *choice design* (pages 67–71) and the relationship between the two is fundamental. These two design layouts are the source of a great deal of confusion when many people start out. Make sure that you understand them. You do not have to understand the formula for the variance matrix for a choice model, the orthogonal coding, or the formulas for efficiency. However, you should be comfortable with the idea of the average variability of the parameter estimates and how it is related to efficiency.

This section lists the steps in designing a choice experiment. The next section illustrates these steps with several simple examples. You should work through the simple examples in the next section before consulting the more complex examples in the discrete choice chapter on page 285.

The first step in designing a choice experiment involves determining:

- Is this a generic study (no brands) or a branded study? Branded studies have a label for each alternative that conveys meaning beyond ordinary attributes. Brand names are the most common example. See pages 127, 166, 188, and 302 for examples of studies with brands. The destinations in the vacation example (pages 339 and 410) also act like brands. In a generic study, the alternatives are simply bundles of attributes. See pages 198, 102, and 556 for examples of generic designs. Also see the documentation for the `%ChoicEff` macro beginning on page 806 for examples of generic design construction.
- If it is branded, what are the brands?
- How many alternatives?
- Is there a constant (none, no purchase, delay purchase, or stick with my regular brand) alternative?
- What are the attributes of all of the alternatives, and what are their levels?
- Are any of the attributes generic? In other words, are there attributes that you expect to behave the same way across all alternatives?

- Are any of the attributes alternative-specific? In other words, are there attributes that you expect to behave differently across all alternatives (brand by attribute interactions)?
- Are there any restrictions, within alternatives, within choice sets, or across choice sets?

Step 1. Write down all of the attributes of all of your alternatives and their levels. See pages 339, 410, 469, and 556 for examples.

Step 2. Is this a generic study (such as the chair example on page 556) or a branded example (such as the vacation examples on pages 339 and 410 and the food example on page 469)?

Step 3. If this is a branded study:

- Use the `%MktRuns` macro to suggest the number of choice sets. See page 1159 for documentation. See the following pages for examples of using this macro in this chapter: 128 and 188. Also see the following pages for examples of using this macro in the discrete choice chapter: 340, 411, 415, 482, and 483. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 895, 905, 916, 919, 1057, 1159, 1160, 1161, 1162, and 1165.
- Use the `%MktEx` macro to make a linear arrangement of a choice design. See page 1017 for documentation. See the following pages for examples of using this macro in this chapter: 129 and 190. Also see the following pages for examples of using this macro in the discrete choice chapter: 304, 304, 320, 333, 343, 352, 413, 415, 416, 417, 422, 425, 472, 479, 485, 488, and 491. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 804, 808, 815, 817, 819, 858, 878, 881, 896, 903, 905, 917, 919, 920, 921, 923, 924, 926, 926, 927, 929, 952, 979, 979, 1002, 1005, 1006, 1008, 1009, 1012, 1017, 1018, 1026, 1027, 1028, 1029, 1030, 1030, 1058, 1062, 1067, 1067, 1068, 1069, 1073, 1076, 1079, 1080, 1082, 1093, 1093, 1093, 1096, 1098, 1099, 1102, 1103, 1135, 1137, 1142, 1142, 1142, 1143, 1145, 1150, 1154, 1154, and 1155.
- Use the `%MktEval` macro to evaluate the linear arrangement. See page 1012 for documentation. See page 130 for an example of using this macro in this chapter. Also see the following pages for examples of using this macro in the discrete choice chapter: 306, 308, 349, 353, 413, 423, 480, 485, 489, 491, 493, 538, 588, and 591. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 959, 1012, 1073, and 1099.
- Display and check the linear arrangement. See page 305 for an example.
- Use the `%MktKey` and `%MktRoll` macros to make a choice design from the linear arrangement. See page 1153 for documentation on the `%MktRoll` macro and page 1090 for documentation on the `%MktKey` macro. See the following pages for examples of using the `%MktRoll` macro in this chapter: 134 and 192. Also see the following pages for examples of using this macro in the discrete choice chapter: 312, 320, 357, 387, 429, 505, 546, 556, 575, 607, 617, 628, and 636. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 819, 878, 881, 898, 917, 929, 982, 1005, 1008, 1085, 1154, 1155, 1156, and 1156. See the following pages for examples of using the `%MktKey` macro in this chapter: 133 and 192. Also see the following pages for examples of using this macro in the discrete choice chapter: 356, 546, 556, 575, 607, 617, 628, and 636. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 897, 1085, 1090, 1090, 1090, and 1153.

Step 4. If this is a generic study:

- Use the `%MktRuns` macro to suggest a size for the candidate design. See page 1159 for documentation. See page 199 for an example of using this macro in this chapter. Also see page 557 for an example of using this macro in the discrete choice chapter. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 895, 905, 916, 919, 1057, 1159, 1160, 1161, 1162, and 1165.

- Use the `%MktEx` macro to make a candidate design. See page 1017 for documentation. See the following pages for examples of using this macro in this chapter: 81, 85, 166, 200, 109, 112, and 98. Also see the following pages for examples of using this macro in the discrete choice chapter: 556, 558, 564, 567, 570, and 575. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 804, 808, 815, 817, 819, 858, 878, 881, 896, 903, 905, 917, 919, 920, 921, 923, 924, 926, 926, 927, 929, 952, 979, 979, 1002, 1005, 1006, 1008, 1009, 1012, 1017, 1018, 1026, 1027, 1028, 1029, 1030, 1030, 1058, 1062, 1067, 1067, 1068, 1069, 1073, 1076, 1079, 1080, 1082, 1093, 1093, 1093, 1096, 1098, 1099, 1102, 1103, 1135, 1137, 1142, 1142, 1142, 1143, 1145, 1150, 1154, 1154, and 1155.

- Use the `%MktLab` macro to add alternative flags. See page 1093 for documentation. See the following pages for examples of using this macro in this chapter: 85 and 201. Also see the following pages for examples of using this macro in the discrete choice chapter: 564, and 567. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 814, 815, 817, 858, 921, 926, 952, 979, 1006, 1062, 1093, 1094, 1095, 1096, 1098, 1099, 1102, and 1103.

- Display and check the candidate design. See page 201 for an example.

- Use the `%ChoicEff` macro to find an efficient choice design. See page 806 for documentation. See the following pages for examples of using this macro in this chapter: 87, 170, 193, and 203. Also see the following pages for examples of using this macro in the discrete choice chapter: 320, 559, 564, 567, 570, 570, 574, 576, 607, 618, 628, 632, and 636. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 809, 816, 817, 819, 859, 862, 863, 878, 880, 882, 883, 887, 891, 899, 901, 908, 913, 917, 919, 920, 921, 924, 926, 926, 927, 929, 952, 982, 1005, 1006, 1145, and 1150.

- Display and check the choice design. See page 205 for an example.

- Go back and try the `%MktEx` step with other size choice sets (unless you are using a small, full-factorial candidate set). Stop when you feel comfortable with the results.

Step 5. Continue processing the design:

- Display and check the choice design. See page 134 for an example.

- Assign formats and labels. See page 136 for an example.

- Display and check the choice design. See page 136 for an example.

- Use the `%ChoicEff` macro to evaluate the design. See page 806 for documentation. See the following pages for examples of using this macro in this chapter: 81, 83, 137, 140, 142, 109, and 112. Also see the following pages for examples of using this macro in the discrete choice chapter: 313, 317, 322, 360, 365, 366, 430, 508, 509, 542, 570, 574, 597, 599, 645, 650, 654, 656, 659, and 662. In addition, see the following pages for examples of using this macro in the macro

documentation chapter: 809, 816, 817, 819, 859, 862, 863, 878, 880, 882, 883, 887, 891, 899, 901, 908, 913, 917, 919, 920, 921, 924, 926, 926, 927, 929, 952, 982, 1005, 1006, 1145, and 1150.

- Use the `%MktDups` macro to check for duplicate choice sets. See page 1004 for documentation. See the following pages for examples of using this macro in this chapter: 147, 174, 198, and 206. Also see the following pages for examples of using this macro in the discrete choice chapter: 319, 368, 519, 564, 564, 567, 570, 576, 597, 607, 617, 628, 636, 645, 650, 654, 656, 659, and 662. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 809, 817, 819, 1005, 1006, 1008, 1009, and 1010.

- For larger designs, you might need to block the design. See page 979 for documentation. Also see the following pages for examples of using this macro in the discrete choice chapter: 426 and 497. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 979, 982, 1098, and 1099. Also see the following pages for examples of using this macro in the discrete choice chapter: 641, 642, and 660. Alternatively, with the linear arrangement, you can sometimes just add a blocking factor directly to the linear arrangement. See page 979 for an example.

Step 6. Collect data and process the design:

- Display or otherwise generate the choice tasks, and then collect and enter the data. See page 147 for an example.

- Use the `%MktMerge` macro to merge the data and the design. See page 1125 for documentation. See the following pages for examples of using this macro in this chapter: 149 and 176. Also see the following pages for examples of using this macro in the discrete choice chapter: 325, 371, 387, 437, 522, and 529. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 1125, 1125, and 1126.

- Display part of the data and design and check the results. See page 149 for an example.

- Optionally, particularly for large data sets, you can aggregate the data set using PROC SUMMARY. See page 522 for an example.

- Use the TRANSREG procedure to code the design. See the following pages for examples of using this procedure: 150, 176, 327, 372, 378, 380, 383, 388, 438, 447, 449, 452, 460, 460, 462, 464, 523, 528, 530, and 528.

- Display part of the coded design and check the results. See page 150 for an example.

- Use the `%PHChoice` macro to customize the output. See page 1173 for documentation. See the following pages for examples of using this macro in this chapter: 152 and 187. Also see the following pages for examples of using this macro in the discrete choice chapter: 287 and 288. In addition, see the following pages for examples of using this macro in the macro documentation chapter: 1173, 1173, and 1177.

- Use the PHREG procedure to fit the multinomial logit model. See pages 152, 176, 295, 298, 329, 375, 378, 380, 385, 390, 440, 442, 447, 449, 457, 462, 464, 524, 528, 532, 550, and 552.

There are many variations not covered in this simple outline. See the examples in the discrete choice chapter (pages 285–663) for many other possibilities.

Overview of the Examples

The next six sections show how to create small and simple choice experiments from start to finish. Each example illustrates one of the basic approaches to making a choice design. These examples are simple. In contrast, the examples on pages 285 through 663 tend to be much more involved and have many more nuances. These introductory examples show the basic steps in the context of a design with no complications. Note, however, that each of these examples is longer than it needs to be because each displays extra information to help you better understand how choice designs are created and how they work. Also note that steps that are common to all examples are explained in more detail in the earlier examples than in the later examples, so please read all of them to get a full understanding of the process. Understanding these introductory examples will help you with the more involved examples that come later. The first example directly uses the `%MktEx` macro to find a design. The other examples use a combination of the `%MktEx` macro, the `%ChoicEff` macro, and in some cases, other macros.

Example 1: Orthogonal and Balanced Factors, the Linear Arrangement Approach

In this example, we use the `%MktEx` macro to find a linear arrangement of a choice design, then we convert it into choice design format. You should use this approach when you want all of the attributes of all of the alternatives to be balanced and orthogonal (as in this example) or at least nearly so (as in most real-life examples that are more complicated). This approach lets you fit complicated models including models with alternative-specific effects without specifying in advance the exact nature of the model or parameters.

The product is breakfast bars, and there are three brands, Branolicious, Brantopia, and Brantasia.* The choice sets consist of three brands and a constant (no purchase) alternative. Each brand has two attributes, a four-level attribute for price and a two-level attribute for the number of bars per box. The prices are \$2.89, \$2.99, \$3.09, and \$3.19, and the sizes are 6 count and 8 count. We can make a choice design by first making a design that is optimal for a hypothetical linear model that has factors for all of the attributes of all of the alternatives. The linear arrangement consists of the six factors, which are shown organized by brand and also organized by attribute. There is only one set of attributes, however that set is shown in two different ways. The attributes are as follows:

Factors Organized By Brand

Linear Factor Name	Levels	Brand	Choice Design Attribute
x1	4 levels	Branolicious	Price
x2	2 levels	Branolicious	Count
x3	4 levels	Brantopia	Price
x4	2 levels	Brantopia	Count
x5	4 levels	Brantasia	Price
x6	2 levels	Brantasia	Count

Factors Organized By Attribute

Linear Factor Name	Levels	Brand	Choice Design Attribute
x1	4 levels	Branolicious	Price
x3	4 levels	Brantopia	Price
x5	4 levels	Brantasia	Price
x2	2 levels	Branolicious	Count
x4	2 levels	Brantopia	Count
x6	2 levels	Brantasia	Count

*Real studies, of course, use real brands. Since we have not collected real data, we cannot use real brand names. We picked these silly names so no one would confuse our artificial data with real data.

We need a factorial design with 6 factors: Branolicious Price, Branolicious Count, Brantopia Price, Brantopia Count, Brantasia Price, and Brantasia Count. From it, we make a choice design with three attributes, brand, count, and price. We can use the `%MktRuns` macro as follows to suggest the number of choice sets:

```
title 'Cereal Bars';

%mktruns(4 2  4 2  4 2)     /* factor level list for all attrs and alts     */
```

The input to the macro is the number of levels of all of the factors (that is, all of the attributes of all of the alternatives). The output from the macro is as follows:

Cereal Bars

Design Summary

Number of Levels	Frequency
2	3
4	3

Cereal Bars

Saturated = 13
Full Factorial = 512

Some Reasonable Design Sizes	Violations	Cannot Be Divided By
16 *	0	
32 *	0	
24	3	16
20	12	8 16
28	12	8 16
14	18	4 8 16
18	18	4 8 16
22	18	4 8 16
26	18	4 8 16
30	18	4 8 16
13 S	21	2 4 8 16

* - 100% Efficient design can be made with the MktEx macro.
S - Saturated Design - The smallest design that can be made.
Note that the saturated design is not one of the
recommended designs for this problem. It is shown
to provide some context for the recommended sizes.

Cereal Bars

n	Design		Reference
16	2 ** 6	4 ** 3	Fractional-Factorial
16	2 ** 3	4 ** 4	Fractional-Factorial
32	2 ** 22	4 ** 3	Fractional-Factorial
32	2 ** 19	4 ** 4	Fractional-Factorial
32	2 ** 16	4 ** 5	Fractional-Factorial
32	2 ** 15	4 ** 3 8 ** 1	Fractional-Factorial
32	2 ** 13	4 ** 6	Fractional-Factorial
32	2 ** 12	4 ** 4 8 ** 1	Fractional-Factorial
32	2 ** 10	4 ** 7	Fractional-Factorial
32	2 ** 9	4 ** 5 8 ** 1	Fractional-Factorial
32	2 ** 7	4 ** 8	Fractional-Factorial
32	2 ** 6	4 ** 6 8 ** 1	Fractional-Factorial
32	2 ** 4	4 ** 9	Fractional-Factorial
32	2 ** 3	4 ** 7 8 ** 1	Fractional-Factorial

The output tells us that there are 3 two-level factors and 3 four-level factors. The saturated design has 13 runs or rows, so we need at least 13 choice sets with this approach. The full-factorial design has 512 runs, so there are a maximum of 512 possible choice sets. The `%MktRuns` macro suggests 16 as its first choice because 16 meets necessary but not sufficient conditions for the existence of an orthogonal array. Sixteen can be divided by 2 (we have two-level factors), 4 (we have four-level factors), 2×2 (we have more than one two-level factor), 4×4 (we have more than one four-level factor), and 2×4 (we have both two-level factors and four-level factors). The number of choice sets must be divisible by all of these if the design is going to be orthogonal and balanced. Thirty-two meets these conditions as well. However, 16 is a more reasonable number of judgments for people to make, and the other suggestions (24, 20, 28, 14, 18, 22, 26, 30) all cannot be divided by at least one of the relevant numbers. For this example, the macro only considers sizes up to 32. By default, the macro stops considering larger sizes when it finds a perfect size (in this case 32) that is twice as big as another perfect size (16). Sixteen choice sets is ideal for this example. The necessary conditions are sufficient in this case, and there is an orthogonal-array that we can use. The last part of the output lists the orthogonal arrays that `%MktEx` knows how to make that work for our specification.

We use the `%MktEx` macro as follows to get our factorial design as follows:

```
%mktex(4 2  4 2  4 2,       /* factor level list for all attrs and alts     */
       n=16,                /* number of choice sets                        */
       seed=17)             /* random number seed                           */
```

The macro accepts a factor-level list like the `%MktRuns` list along with the number of runs or choice sets. We specify a random number seed so that we always get the same design if we rerun the `%MktEx` macro.

The results are as follows:

Cereal Bars

Algorithm Search History

Design	Row,Col	Current D-Efficiency	Best D-Efficiency	Notes
1	Start	100.0000	100.0000	Tab
1	End	100.0000		

Cereal Bars

The OPTEX Procedure

Class Level Information

Class	Levels	Values
x1	4	1 2 3 4
x2	2	1 2
x3	4	1 2 3 4
x4	2	1 2
x5	4	1 2 3 4
x6	2	1 2

Cereal Bars

Design Number	D-Efficiency	A-Efficiency	G-Efficiency	Average Prediction Standard Error
1	100.0000	100.0000	100.0000	0.9014

The %MktEx macro found a 100% efficient, orthogonal and balanced design with 3 two-level factors and 3 four-level factors, just as the %MktRuns macro told us it would. The levels are all positive integers, starting with 1 and continuing up to the number of levels. The note in the algorithm search history of "Tab" on a line that displays 100% efficiency shows that the design was directly constructed from the %MktEx macro's table or catalog of orthogonal designs.

Next, we examine some of the properties of the design and display it. This step is not necessary since we have a 100% efficient design. However, we go through it here to better see the properties of a 100% efficient design. The %MktEval macro tells us which factors are orthogonal and which are correlated. It also tells us how often each level occurs, how often each pair of levels occurs across pairs of factors, and how often each run or choice set occurs. The following steps evaluate and display the design:

```
title2 'Examine Correlations and Frequencies';

%mkteval(data=randomized)   /* evaluate randomized design                    */

title2 'Examine Design';
proc print data=randomized; run;
```

The first part of the output is as follows:

Cereal Bars
Examine Correlations and Frequencies
Canonical Correlations Between the Factors
There are 0 Canonical Correlations Greater Than 0.316

	x1	x2	x3	x4	x5	x6
x1	1	0	0	0	0	0
x2	0	1	0	0	0	0
x3	0	0	1	0	0	0
x4	0	0	0	1	0	0
x5	0	0	0	0	1	0
x6	0	0	0	0	0	1

All canonical correlations off the diagonal are zero, which tells us that the design is orthogonal—that every factor is uncorrelated with every other factor.

The next part of the output is as follows:

Cereal Bars
Examine Correlations and Frequencies
Summary of Frequencies
There are 0 Canonical Correlations Greater Than 0.316

Frequencies

x1	4 4 4 4
x2	8 8
x3	4 4 4 4
x4	8 8
x5	4 4 4 4
x6	8 8
x1 x2	2 2 2 2 2 2 2 2
x1 x3	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
x1 x4	2 2 2 2 2 2 2 2
x1 x5	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
x1 x6	2 2 2 2 2 2 2 2

```
x2 x3    2 2 2 2 2 2 2 2
x2 x4    4 4 4 4
x2 x5    2 2 2 2 2 2 2 2
x2 x6    4 4 4 4
x3 x4    2 2 2 2 2 2 2 2
x3 x5    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
x3 x6    2 2 2 2 2 2 2 2
x4 x5    2 2 2 2 2 2 2 2
x4 x6    4 4 4 4
x5 x6    2 2 2 2 2 2 2 2
N-Way    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

This is a very compact presentation of output from PROC SUMMARY. The one-way, two-way, and n-way frequencies show us how often each level, pair of levels, and choice set occurs. It tells us that each level occurs equally often, (4 times in the four-level factors and 8 times in the two-level factors), and each pair of levels occurs equally often. The n-way frequencies tell us that every choice set occurs only once in the design—there are no duplicate choice sets.

The randomized design is follows:

Cereal Bars
Examine Design

Obs	x1	x2	x3	x4	x5	x6
1	1	1	2	1	2	1
2	4	1	2	2	4	2
3	3	2	2	1	3	2
4	3	1	4	1	4	1
5	2	2	2	2	1	1
6	4	1	1	1	1	1
7	3	2	1	2	2	1
8	1	2	3	2	4	1
9	2	1	4	2	2	2
10	1	2	4	1	1	2
11	4	2	4	2	3	1
12	2	1	3	1	3	1
13	2	2	1	1	4	2
14	1	1	1	2	3	2
15	3	1	3	2	1	2
16	4	2	3	1	2	2

It has 3 four-level factors with levels 1, 2, 3, 4, and 3 two-level factors with levels 1 and 2. It has 16 rows since there are 16 choice sets. The levels and rows are not sorted (that is, the design is randomized), so this linear arrangement is in a good form to use to make the choice design.

Next, we need to make a choice design from our linear arrangement. This involves taking levels for alternatives, which are next to each other in the linear arrangement, and moving them on top of each

other to form the choice design. We specify the rules for doing this in a SAS data set. We call this data set the key to constructing the choice design. Before we describe how this data set is created and what it means, let's look at the key data set to see where we are going. The following output displays the key data set for this example:

Obs	Brand	Price	Count
1	Branolicious	x1	x2
2	Brantopia	x3	x4
3	Brantasia	x5	x6
4	None		

We need to specify that the brands are Branolicious, Brantopia, Brantasia, and None. We need to specify that the Branolicious Price is made from x1, the Branolicious Count is made from x2, the Brantopia Price is made from x3, the Brantopia Count is made from x4, the Brantasia Price is made from x5, and the Brantasia Count is made from x6. We also need to specify that the None alternative is not made from any of the attributes. The variables in this data set correspond to the attributes in the choice design, and the values correspond to the brands and to the factorial design factors. The `%MktKey` macro gives us the linear arrangement factor names that we can copy and paste into this data set. For many designs (particularly larger designs), this macro makes it easy to construct the design key. The following step creates the names:

```
%mktkey(3 2)                /* x1-x6 (since 3*2=6) in 3 rows and 2 columns  */
```

The results are as follows:

x1	x2
x1	x2
x3	x4
x5	x6

The names `x1-x6` are arranged into three rows (the first value of "3 2") and two columns (the second value of "3 2") for pasting into the key data set. The following step creates the key data set:

```
title2 'Create the Choice Design Key';

data key;
   input
Brand $ 1-12    Price $    Count $; datalines;
Branolicious    x1         x2
Brantopia       x3         x4
Brantasia       x5         x6
None            .          .
;
```

Note that when reading missing or blank character data with list input in a DATA step, as we do here, you can use a period for the blank values. SAS automatically translates them into blanks.

The %MktRoll macro processes the linear arrangement using the information in the key data set to make the choice design as follows:

```
title2 'Create Choice Design from Linear Arrangement';

%mktroll(design=randomized,  /* input randomized linear arrangement        */
         key=key,            /* rules for making choice design             */
         alt=brand,          /* brand or alternative label var             */
         out=cerealdes)      /* output choice design                       */

proc print; id set; by set; run;
```

The choice design contains the variable Set along with the variable names and brands from the key data set. The information from the linear arrangement is all stored in the right places. The Brand variable contains literal names, and it is named in the alt= option, which designates the alternative name (often brand) attribute. The remaining variables contain factor names from the linear data set. The choice design is as follows:

Cereal Bars
Create Choice Design from Linear Arrangement

Set	Brand	Price	Count
1	Branolicious	1	1
	Brantopia	2	1
	Brantasia	2	1
	None	.	.
2	Branolicious	4	1
	Brantopia	2	2
	Brantasia	4	2
	None	.	.
3	Branolicious	3	2
	Brantopia	2	1
	Brantasia	3	2
	None	.	.
4	Branolicious	3	1
	Brantopia	4	1
	Brantasia	4	1
	None	.	.
5	Branolicious	2	2
	Brantopia	2	2
	Brantasia	1	1
	None	.	.

```
 6    Branolicious     4       1
      Brantopia        1       1
      Brantasia        1       1
      None             .       .

 7    Branolicious     3       2
      Brantopia        1       2
      Brantasia        2       1
      None             .       .

 8    Branolicious     1       2
      Brantopia        3       2
      Brantasia        4       1
      None             .       .

 9    Branolicious     2       1
      Brantopia        4       2
      Brantasia        2       2
      None             .       .

10    Branolicious     1       2
      Brantopia        4       1
      Brantasia        1       2
      None             .       .

11    Branolicious     4       2
      Brantopia        4       2
      Brantasia        3       1
      None             .       .

12    Branolicious     2       1
      Brantopia        3       1
      Brantasia        3       1
      None             .       .

13    Branolicious     2       2
      Brantopia        1       1
      Brantasia        4       2
      None             .       .

14    Branolicious     1       1
      Brantopia        1       2
      Brantasia        3       2
      None             .       .

15    Branolicious     3       1
      Brantopia        3       2
      Brantasia        1       2
      None             .       .
```

16	Branolicious	4	2
	Brantopia	3	1
	Brantasia	2	2
	None	.	.

These following steps assign formats to the levels and display the choice sets:

```
title2 'Final Choice Design';

proc format;
   value price 1 = $2.89 2 = $2.99 3 = $3.09 4 = $3.19 . = ' ';
   value count 1 = 'Six Bars' 2 = 'Eight Bars'  . = ' ';
   run;

data sasuser.cerealdes;
   set cerealdes;
   format price price. count count.;
   run;

proc print data=sasuser.cerealdes(obs=16);
   by set;   id set;
   run;
```

In the interest of space, only the first four choice sets are displayed. The design is stored in a permanent SAS data set so it is available at analysis time. The first four choice sets are as follows:

Cereal Bars
Final Choice Design

Set	Brand	Price	Count
1	Branolicious	$2.89	Six Bars
	Brantopia	$2.99	Six Bars
	Brantasia	$2.99	Six Bars
	None		
2	Branolicious	$3.19	Six Bars
	Brantopia	$2.99	Eight Bars
	Brantasia	$3.19	Eight Bars
	None		
3	Branolicious	$3.09	Eight Bars
	Brantopia	$2.99	Six Bars
	Brantasia	$3.09	Eight Bars
	None		

```
                    4     Branolicious    $3.09    Six Bars
                          Brantopia       $3.19    Six Bars
                          Brantasia       $3.19    Six Bars
                          None
```

The following step evaluates the goodness of the design for a choice model using the `%ChoicEff` macro:

```
title2 'Evaluate Design';

%choiceff(data=sasuser.cerealdes,          /* candidate choice sets         */
          init=sasuser.cerealdes(keep=set),/* select these sets from cands  */
          intiter=0,                       /* eval without internal iters   */
          model=class(brand price count),  /* model, ref cell coding        */
          nalts=4,                         /* number of alternatives        */
          nsets=16,                        /* number of choice sets         */
          beta=zero)                       /* assumed beta vector, Ho: b=0  */
```

The `%ChoicEff` macro constructs the covariance matrix of the specified choice model parameters and displays the variances and standard errors. It also displays the *D*-efficiency and other information. Here, we are using the `%ChoicEff` macro to evaluate a design. It can also be used to search for efficient choice designs. When we evaluate a design, we need to provide the design in the `data=` specification. Usually, you use the `data=` option to specify the candidate set to be searched. In some sense, the `data=` design is a candidate set in this context as well, and we use the `init=` option to specify how the final design is constructed from the candidate set. We do this by bringing in just the choice set numbers in the initial design. This is accomplished with the `init=sasuser.cerealdes(keep=set)` specification. Then the `%ChoicEff` macro selects just the specified candidate choice sets (in this case all of them) and uses them as the initial design. The `intiter=0` option specifies no internal iterations, so the design is evaluated but no attempt is made to improve upon it. Other options include a specification of the number of alternatives, the number of choice sets, and the assumed beta vector. You can specify a list of parameter values or `beta=zero` for all zeros.

The first part of the output is as follows:

```
                              Cereal Bars
                            Evaluate Design

n    Name                 Beta    Label

1    BrandBranolicious      0     Brand Branolicious
2    BrandBrantasia         0     Brand Brantasia
3    BrandBrantopia         0     Brand Brantopia
4    Price_2_89             0     Price $2.89
5    Price_2_99             0     Price $2.99
6    Price_3_09             0     Price $3.09
7    CountSix_Bars          0     Count Six Bars
```

This table provides a list of the generated names for all of the parameters, the specified beta value for each, and the generated label. Whenever you specify a list of betas, you need to use this table to ensure that the right betas are assigned to the right parameters.

The next part of the output is as follows:

```
                          Cereal Bars
                        Evaluate Design

Design   Iteration  D-Efficiency       D-Error
-------------------------------------------------
    1        0           1.93756       0.51611
```

This part of the output contains the iteration history table. Since `intiter=0` was specified, this contains only a report of the efficiency of the initial design, which is labeled as iteration 0.

The following results contain the last output tables, which are what we are most interested in seeing:

```
                          Cereal Bars
                        Evaluate Design

                         Final Results

                Design                 1
                Choice Sets           16
                Alternatives           4
                Parameters             7
                Maximum Parameters    48
                D-Efficiency      1.9376
                D-Error           0.5161

                          Cereal Bars
                        Evaluate Design

                                                                    Standard
n    Variable Name       Label                  Variance    DF      Error

1    BrandBranolicious   Brand Branolicious      0.94444     1     0.97183
2    BrandBrantasia      Brand Brantasia         0.94444     1     0.97183
3    BrandBrantopia      Brand Brantopia         0.94444     1     0.97183
4    Price_2_89          Price $2.89             0.88889     1     0.94281
5    Price_2_99          Price $2.99             0.88889     1     0.94281
6    Price_3_09          Price $3.09             0.88889     1     0.94281
7    CountSix_Bars       Count Six Bars          0.44444     1     0.66667
                                                            ==
                                                             7
```

We see three parameters for brand (4 alternatives including None minus 1), three for price $(4-1)$, one for count $(2-1)$. All are estimable, and all have reasonable standard errors. With 16 choice sets and 4 alternatives, we can estimate at most $16 \times (4-1) = 48$ parameters. Note that with the `%ChoicEff` macro and with many choice designs and coding schemes, *D*-efficiency is not reported on a 0 to 100 scale as it is in the `%MktEx` macro and linear model designs. This is because the range over which *D*-efficiency can vary is less clear with some choice designs. However, later in this example, we will come close. Also see page 102 for examples of choice designs with *D*-efficiency scaled to the 0 to 100 range. For now, we know the following:

- *D*-efficiency is 1.9376 on a scale of 0 to unknown.
- All parameters are estimable and the design is more than big enough to estimate all of our parameters.
- The variances (and standard errors) are constant within each attribute, which is usually a good sign.
- The variances (and standard errors) are all of a similar magnitude, which is usually a good sign. When the variances vary a lot (which is hard to quantify, but varying by a factor of over 100 or 1000 is certainly enough to make you worry) it is usually a sign of a problem with the design. Often it is a sign of too few choice sets to precisely estimate all of the parameters.

This pattern of variances and standard errors suggests (but certainly does not prove) that this is a good design. We can run the macro again and use the standardized orthogonal contrast coding to get a better evaluation this design. However, before we do that, it is instructive to look at the covariance matrix of the parameter estimates. The following steps display this matrix, which is automatically output to a SAS data set called `bestcov`:

```
proc format;
   value zer -1e-12 - 1e-12 = ' 0   ';
   run;

proc print data=bestcov label;
   id __label;
   label __label = '00'x;
   var BrandBranolicious -- CountSix_Bars;
   format _numeric_ zer5.2;
   run;
```

The format simply displays values very close to zero, both above and below zero, as precisely zero to make a better display.

The results are as follows:

Cereal Bars
Evaluate Design

	Brand Branolicious	Brand Brantasia	Brand Brantopia	Price $2.89	Price $2.99	Price $3.09	Count Six Bars
Brand Branolicious	0.94	0.69	0.69	-0.44	-0.44	-0.44	-0.22
Brand Brantasia	0.69	0.94	0.69	-0.44	-0.44	-0.44	-0.22
Brand Brantopia	0.69	0.69	0.94	-0.44	-0.44	-0.44	-0.22
Price $2.89	-0.44	-0.44	-0.44	0.89	0.44	0.44	0
Price $2.99	-0.44	-0.44	-0.44	0.44	0.89	0.44	0
Price $3.09	-0.44	-0.44	-0.44	0.44	0.44	0.89	0
Count Six Bars	-0.22	-0.22	-0.22	0	0	0	0.44

You can see that the diagonal of the covariance matrix contains the variances that are reported by the `%ChoicEff` macro. The off-diagonal elements show the covariances. The variances and covariances depend on how the design is coded. Here, the default reference-cell coding is used. Other coding schemes will create quite different results. The standardized orthogonal contrast coding is of particular interest when you evaluate designs. See page 73. The `sta`* (short for `standorth`) option in the `model=` option requests a standardized orthogonal contrast coding. The following steps generate and display the covariance matrix with the standardized orthogonal contrast coding:

```
%choiceff(data=sasuser.cerealdes,          /* candidate choice sets         */
          init=sasuser.cerealdes(keep=set),/* select these sets from cands  */
          intiter=0,                       /* eval without internal iters   */
                                           /* model with stdz orthog coding */
          model=class(brand price count / sta),
          nalts=4,                         /* number of alternatives        */
          nsets=16,                        /* number of choice sets         */
          options=relative,                /* display relative D-efficiency */
          beta=zero)                       /* assumed beta vector, Ho: b=0  */

proc print data=bestcov label;
   id __label;
   label __label = '00'x;                  /* hex null suppress label header*/
   var BrandBranolicious -- CountSix_Bars;
   format _numeric_ zer5.2;
   run;
```

*This option is first available with SAS 9.2. It will not be recognized, and it will cause an error in earlier SAS releases.

Some of the results are as follows:

Cereal Bars
Evaluate Design

Final Results

Design	1
Choice Sets	16
Alternatives	4
Parameters	7
Maximum Parameters	48
D-Efficiency	11.5168
Relative D-Eff	71.9802
D-Error	0.0868
1 / Choice Sets	0.0625

Cereal Bars
Evaluate Design

n	Variable Name	Label	Variance	DF	Standard Error
1	BrandBranolicious	Brand Branolicious	0.06250	1	0.25000
2	BrandBrantasia	Brand Brantasia	0.06250	1	0.25000
3	BrandBrantopia	Brand Brantopia	0.06250	1	0.25000
4	Price_2_89	Price $2.89	0.11111	1	0.33333
5	Price_2_99	Price $2.99	0.11111	1	0.33333
6	Price_3_09	Price $3.09	0.11111	1	0.33333
7	CountSix_Bars	Count Six Bars	0.11111	1	0.33333
				==	
				7	

Cereal Bars
Evaluate Design

	Brand Branolicious	Brand Brantasia	Brand Brantopia	Price $2.89	Price $2.99	Price $3.09	Count Six Bars
Brand Branolicious	0.06	0	0	0	0	0	0
Brand Brantasia	0	0.06	0	0	0	0	0
Brand Brantopia	0	0	0.06	0	0	0	0
Price $2.89	0	0	0	0.11	0	0	0
Price $2.99	0	0	0	0	0.11	0	0
Price $3.09	0	0	0	0	0	0.11	0
Count Six Bars	0	0	0	0	0	0	0.11

With the proper coding, the covariances are all zero. The variances for the brand effects are the inverse of the number of choice sets (1/16 = 0.06250).* The four-level brand attribute works perfectly with four-alternative choice sets. The variances for the other parameters are larger. Relative *D*-efficiency is 71.9802. Relative *D*-efficiency is based on a 0 to 100 scale. Note, however, that this relative *D*-efficiency of 71.9802 is a pessimistic statement of the goodness of this design, since *D*-efficiency is measured relative to a hypothetical optimal design that does not have the constraint of a constant alternative. The variances of the parameter estimates are more important when there is a constant alternative than the measure of relative *D*-efficiency.

The preceding steps all used a main-effects model. Alternatively, you could fit separate price and count effects for each brand. These are alternative-specific effects and consist of brand by attribute interactions. The following steps provide an example:

```
title2 'Evaluate Design for Alternative-Specific Model';

%choiceff(data=sasuser.cerealdes,          /* candidate choice sets         */
          init=sasuser.cerealdes(keep=set),/* select these sets from cands  */
          intiter=0,                       /* eval without internal iters   */
                                           /* alternative-specific model    */
                                           /* stdzd orthogonal coding       */
          model=class(brand brand*price brand*count / sta) /
                cprefix=0                  /* lpr=0 labels from just levels */
                lprefix=0,                 /* cpr=0 names from just levels  */
          nalts=4,                         /* number of alternatives        */
          nsets=16,                        /* number of choice sets         */
          options=relative,                /* display relative D-efficiency */
          beta=zero)                       /* assumed beta vector, Ho: b=0  */
```

Now, brand effects are requested as well as brand by price and brand by count interactions. The `cprefix=0` option is specified so that variable names are constructed just from the attribute levels using zero characters of the attribute (or class) variable names. Similarly, the `lprefix=0` option is specified so that variable labels are constructed just from the attribute levels using zero characters of the attribute (or class) variable names or labels. This is because we do not need to see names such as "Brand" or "Price" in our names and labels to understand them. The following results contain the last two output tables, which are what we are most interested in seeing:

*This comparison is only valid when the standardized orthogonal contrast coding is used.

Cereal Bars
Evaluate Design for Alternative-Specific Model

Final Results

Design	1
Choice Sets	16
Alternatives	4
Parameters	15
Maximum Parameters	48
D-Efficiency	8.7825
Relative D-Eff	54.8908
D-Error	0.1139
1 / Choice Sets	0.0625

Cereal Bars
Evaluate Design for Alternative-Specific Model

n	Variable Name	Label	Variance	DF	Standard Error
1	Branolicious	Branolicious	0.06250	1	0.25000
2	Brantasia	Brantasia	0.06250	1	0.25000
3	Brantopia	Brantopia	0.06250	1	0.25000
4	Branolicious_2_89	Branolicious * $2.89	0.25000	1	0.50000
5	Branolicious_2_99	Branolicious * $2.99	0.25000	1	0.50000
6	Branolicious_3_09	Branolicious * $3.09	0.25000	1	0.50000
7	Brantasia_2_89	Brantasia * $2.89	0.13889	1	0.37268
8	Brantasia_2_99	Brantasia * $2.99	0.13889	1	0.37268
9	Brantasia_3_09	Brantasia * $3.09	0.13889	1	0.37268
10	Brantopia_2_89	Brantopia * $2.89	0.11111	1	0.33333
11	Brantopia_2_99	Brantopia * $2.99	0.11111	1	0.33333
12	Brantopia_3_09	Brantopia * $3.09	0.11111	1	0.33333
13	BranoliciousSix_Bars	Branolicious * Six Bars	0.25000	1	0.50000
14	BrantasiaSix_Bars	Brantasia * Six Bars	0.13889	1	0.37268
15	BrantopiaSix_Bars	Brantopia * Six Bars	0.11111	1	0.33333
				==	
				15	

Now, there are 15 parameters as opposed to the 7 we had previously. There are $(4-1) = 3$ for brand, $(4-1) \times (4-1) = 9$ for the alternative-specific price effects and $(4-1) \times (2-1) = 3$ for the alternative-specific count effects. With 16 choice sets and 4 alternatives, we can estimate at most $16 \times (4-1) = 48$ parameters, so we are still nowhere close to trying to estimate the maximum number of parameters. This design looks good for an alternative-specific effects model. All parameters are estimable, and the variances look reasonable (not overly large relative to 1/16).

You can use the following steps to display the covariance matrix, which is large, in a series of panels:

```
%macro printcov(vars);
   proc print data=bestcov label;
      id __label;
      label __label = '00'x;
      var &vars;
      format _numeric_ zer5.2;
      run;
   %mend;

%printcov(Branolicious Brantasia Brantopia)
%printcov(Branolicious_2_89 Branolicious_2_99 Branolicious_3_09)
%printcov(Brantasia_2_89 Brantasia_2_99 Brantasia_3_09)
%printcov(Brantopia_2_89 Brantopia_2_99 Brantopia_3_09)
%printcov(BranoliciousSix_Bars BrantasiaSix_Bars BrantopiaSix_Bars)
```

The results are as follows:

Cereal Bars
Evaluate Design for Alternative-Specific Model

	Branolicious	Brantasia	Brantopia
Branolicious	0.06	0	0
Brantasia	0	0.06	0
Brantopia	0	0	0.06
Branolicious * $2.89	0	0	0
Branolicious * $2.99	0	0	0
Branolicious * $3.09	0	0	0
Brantasia * $2.89	0	0	0
Brantasia * $2.99	0	0	0
Brantasia * $3.09	0	0	0
Brantopia * $2.89	0	0	0
Brantopia * $2.99	0	0	0
Brantopia * $3.09	0	0	0
Branolicious * Six Bars	0	0	0
Brantasia * Six Bars	0	0	0
Brantopia * Six Bars	0	0	0

Cereal Bars
Evaluate Design for Alternative-Specific Model

	Branolicious * $2.89	Branolicious * $2.99	Branolicious * $3.09
Branolicious	0	0	0
Brantasia	0	0	0
Brantopia	0	0	0
Branolicious * $2.89	0.25	0	0
Branolicious * $2.99	0	0.25	0
Branolicious * $3.09	0	0	0.25
Brantasia * $2.89	0.10	0	0
Brantasia * $2.99	0	0.10	0
Brantasia * $3.09	0	0	0.10
Brantopia * $2.89	0.07	0	0
Brantopia * $2.99	0	0.07	0
Brantopia * $3.09	0	0	0.07
Branolicious * Six Bars	0	0	0
Brantasia * Six Bars	0	0	0
Brantopia * Six Bars	0	0	0

Cereal Bars
Evaluate Design for Alternative-Specific Model

	Brantasia * $2.89	Brantasia * $2.99	Brantasia * $3.09
Branolicious	0	0	0
Brantasia	0	0	0
Brantopia	0	0	0
Branolicious * $2.89	0.10	0	0
Branolicious * $2.99	0	0.10	0
Branolicious * $3.09	0	0	0.10
Brantasia * $2.89	0.14	0	0
Brantasia * $2.99	0	0.14	0
Brantasia * $3.09	0	0	0.14
Brantopia * $2.89	0.04	0	0
Brantopia * $2.99	0	0.04	0
Brantopia * $3.09	0	0	0.04
Branolicious * Six Bars	0	0	0
Brantasia * Six Bars	0	0	0
Brantopia * Six Bars	0	0	0

Cereal Bars
Evaluate Design for Alternative-Specific Model

	Brantopia * $2.89	Brantopia * $2.99	Brantopia * $3.09
Branolicious	0	0	0
Brantasia	0	0	0
Brantopia	0	0	0
Branolicious * $2.89	0.07	0	0
Branolicious * $2.99	0	0.07	0
Branolicious * $3.09	0	0	0.07
Brantasia * $2.89	0.04	0	0
Brantasia * $2.99	0	0.04	0
Brantasia * $3.09	0	0	0.04
Brantopia * $2.89	0.11	0	0
Brantopia * $2.99	0	0.11	0
Brantopia * $3.09	0	0	0.11
Branolicious * Six Bars	0	0	0
Brantasia * Six Bars	0	0	0
Brantopia * Six Bars	0	0	0

Cereal Bars
Evaluate Design for Alternative-Specific Model

	Branolicious * Six Bars	Brantasia * Six Bars	Brantopia * Six Bars
Branolicious	0	0	0
Brantasia	0	0	0
Brantopia	0	0	0
Branolicious * $2.89	0	0	0
Branolicious * $2.99	0	0	0
Branolicious * $3.09	0	0	0
Brantasia * $2.89	0	0	0
Brantasia * $2.99	0	0	0
Brantasia * $3.09	0	0	0
Brantopia * $2.89	0	0	0
Brantopia * $2.99	0	0	0
Brantopia * $3.09	0	0	0
Branolicious * Six Bars	0.25	0.10	0.07
Brantasia * Six Bars	0.10	0.14	0.04
Brantopia * Six Bars	0.07	0.04	0.11

There are some nonzero but small covariances off the diagonal.

There is one more test that could be run before a design is used. The `%MktDups` macro in the following step checks the design to see if any choice sets are duplicates of any other choice sets:

```
%mktdups(branded,                          /* a design with brands          */
         data=sasuser.cerealdes,           /* the input design to evaluate  */
         factors=brand price count,        /* factors in the design         */
         nalts=4)                          /* number of alternatives        */
```

The first parameter is a positional parameter. There is no *key-word*= preceding its value, and it must always be specified. We specify that this is a branded design as opposed to a generic design (bundles of attributes with no brands). We also specify the input SAS data set, the factors (attributes) in the design, and the number of alternatives. The results are as follows:

```
Design:          Branded
Factors:         brand price count
                 Brand
                 Count Price
Duplicate Sets:  0
```

The first line of the table tells us that this is a branded design. The second line tells us the factors as specified in the `factors=` option. These are followed by the actual variable names for the factors. The last line reports the number of duplicates. In this case, there are no duplicate choice sets. (Of course, we already knew that from the *n*-way frequencies in the `%MktEval` output.) If there had been duplicate choice sets, then changing the random number seed will sometimes help. Sometimes, changing other aspects of the design or the approach for making the design will help.

Next, the questionnaire is designed. Two sample choice sets are as follows:

Branolicious	Brantopia	Brantasia	No Purchase
$2.89	$2.99	$2.99	
Six Bars	Six Bars	Six Bars	

Branolicious	Brantopia	Brantasia	No Purchase
$3.19	$2.99	$3.19	
Six Bars	Eight Bars	Eight Bars	

In practice, data collection is usually much more elaborate than this. It might involve art work or photographs, and the choice sets might be presented and the data might be collected through personal interview or over the Web. However the choice sets are presented and the data are collected, the essential ingredients remain the same. Subjects are shown sets of alternatives and are asked to make a choice, then they go on to the next set. Each subject sees all 16 choice sets and chooses one alternative from each. The data for each subject consist of 16 integers in the range 1 to 4 showing which alternative was chosen.

The data are collected and entered into a SAS data set as follows:

```
title2 'Read Data';

data results;
   input Subject (r1-r16) (1.);
   datalines;
  1 1331132331312213
  2 3231322131312233
  3 1233332111132233
  4 1211232111313233
  5 1233122111312233
  6 3231323131212313
  7 3231232131332333
  8 3233332131322233
  9 1223332111333233
 10 1332132111233233
 11 1233222211312333
 12 1221332111213233
 13 1231332131133233
 14 3211333211313233
 15 3313332111122233
 16 3321123231331223
 17 3223332231312233
 18 3211223311112233
 19 1232332111132233
 20 1213233111312413
 21 1333232131212233
 22 3321322111122231
 23 3231122131312133
 24 1232132111311333
 25 3113332431213233
 26 3213132141331233
 27 3221132111312233
 28 3222333131313231
 29 1221332131312231
 30 3233332111212233
 31 1221332111342233
 32 2233232111111211
 33 2332332131211231
 34 2221132211312411
 35 1232233111332233
 36 1231333131322333
 37 1231332111331333
 38 1223132211233331
 39 1321232131211231
 40 1223132331321233
;
```

There is one row for each subject containing the number of the chosen alternatives for each of the 16 choice sets.

The `%MktMerge` macro in the following step merges the data and the design and creates the dependent variable:

```
title2 'Merge Data and Design';

%mktmerge(design=sasuser.cerealdes, /* input design                          */
          data=results,             /* input data set                       */
          out=res2,                 /* output data set with design and data */
          nsets=16,                 /* number of choice sets                */
          nalts=4,                  /* number of alternatives               */
          setvars=r1-r16)           /* variables with the chosen alt nums   */
```

The `design=` input data set has one row for each alternative of each choice set. The `data=` input data set has one row for each subject. The `out=` data set has one row for each alternative of each choice set for each subject (in this case, there are $4 \times 16 \times 40 = 2560$ rows). The following step displays the first four choice sets for the first subject:

```
title2 'Design and Data Both';

proc print data=res2(obs=16);
   by set subject;   id set subject;
   run;
```

The first four choice sets for the first subject are as follows:

Cereal Bars
Design and Data Both

Set	Subject	Brand	Price	Count	c
1	1	Branolicious	$2.89	Six Bars	1
		Brantopia	$2.99	Six Bars	2
		Brantasia	$2.99	Six Bars	2
		None			2
2	1	Branolicious	$3.19	Six Bars	2
		Brantopia	$2.99	Eight Bars	2
		Brantasia	$3.19	Eight Bars	1
		None			2
3	1	Branolicious	$3.09	Eight Bars	2
		Brantopia	$2.99	Six Bars	2
		Brantasia	$3.09	Eight Bars	1
		None			2

```
 4        1       Branolicious    $3.09    Six Bars      1
                  Brantopia       $3.19    Six Bars      2
                  Brantasia       $3.19    Six Bars      2
                  None                                   2
```

The dependent variable is `c`. A 1 in `c` indicates first choice, and a 2 indicates the alternatives that were not chosen (second or subsequent choices).

This following step codes the design for analysis:

```
title2 'Code the Independent Variables';

proc transreg design norestoremissing data=res2;
   model class(brand price count);
   id subject set c;
   output out=coded(drop=_type_ _name_ intercept) lprefix=0;
   run;
```

We will typically use PROC TRANSREG for coding because it has a series of options that are useful for coding choice models. This step does not do any analysis; it just codes. This is because the `design` option specifies that only coding is to be done. **The `norestoremissing` option creates zeros in the indicator variables for `class` variables with missing values instead of by default replacing the zeros with missings.** You will need to use the `norestoremissing` option whenever there is a constant or None alternative that is indicated in whole or in part by missing values. The `data=` option names the design to code. The `model` statement names the product attributes. Since no options are specified in the `class` specification, the default reference-cell coding is used.* The `id` statement names the other variables that we will need for analysis. The `output` statement creates and `out=coded` data set with the coded design, drops a few variables that we do not need, and uses the `lprefix=0` option to get labels for the parameters from just the levels and not the input variable names and labels.

The following steps display the coded results for the first subject for the first four choice sets:

```
proc print data=coded(obs=16) label;
   title3 'ID Information and the Dependent Variable';
   format price price. count count.;
   var Brand Price Count Subject Set c;
   by set subject;   id set subject;
   run;

proc print data=coded(obs=16) label;
   title3 'ID Information and the Coding of Brand';
   format price price. count count.;
   var brandbranolicious brandbrantasia brandbrantopia brand;
   by set subject;   id set subject;
   run;
```

*Note that there is no problem with using the standardized orthogonal contrast coding to make the design and using reference cell, effects, or any other coding when you use the design. *D*-efficiency guarantees that you will get equivalent results if you change codings.

```
proc print data=coded(obs=16) label;
   title3 'ID Information and the Coding of Price and Count';
   format price price. count count.;
   var Price_2_89 Price_2_99 Price_3_09 CountSix_Bars Price Count;
   by set subject;   id set subject;
   run;
```

The coded design for the first four choice sets is shown in the following three panels:

Cereal Bars
Code the Independent Variables
ID Information and the Dependent Variable

Set	Subject	Brand	Price	Count	Subject	Set	c
1	1	Branolicious	$2.89	Six Bars	1	1	1
		Brantopia	$2.99	Six Bars	1	1	2
		Brantasia	$2.99	Six Bars	1	1	2
		None			1	1	2
2	1	Branolicious	$3.19	Six Bars	1	2	2
		Brantopia	$2.99	Eight Bars	1	2	2
		Brantasia	$3.19	Eight Bars	1	2	1
		None			1	2	2
3	1	Branolicious	$3.09	Eight Bars	1	3	2
		Brantopia	$2.99	Six Bars	1	3	2
		Brantasia	$3.09	Eight Bars	1	3	1
		None			1	3	2
4	1	Branolicious	$3.09	Six Bars	1	4	1
		Brantopia	$3.19	Six Bars	1	4	2
		Brantasia	$3.19	Six Bars	1	4	2
		None			1	4	2

Cereal Bars
Code the Independent Variables
ID Information and the Coding of Brand

Set	Subject	Branolicious	Brantasia	Brantopia	Brand
1	1	1	0	0	Branolicious
		0	0	1	Brantopia
		0	1	0	Brantasia
		0	0	0	None
2	1	1	0	0	Branolicious
		0	0	1	Brantopia
		0	1	0	Brantasia
		0	0	0	None

3	1	1	0	0	Branolicious
		0	0	1	Brantopia
		0	1	0	Brantasia
		0	0	0	None
4	1	1	0	0	Branolicious
		0	0	1	Brantopia
		0	1	0	Brantasia
		0	0	0	None

Cereal Bars
Code the Independent Variables
ID Information and the Coding of Price and Count

Set	Subject	$2.89	$2.99	$3.09	Six Bars	Price	Count
1	1	1	0	0	1	$2.89	Six Bars
		0	1	0	1	$2.99	Six Bars
		0	1	0	1	$2.99	Six Bars
		0	0	0	0		
2	1	0	0	0	1	$3.19	Six Bars
		0	1	0	0	$2.99	Eight Bars
		0	0	0	0	$3.19	Eight Bars
		0	0	0	0		
3	1	0	0	1	0	$3.09	Eight Bars
		0	1	0	1	$2.99	Six Bars
		0	0	1	0	$3.09	Eight Bars
		0	0	0	0		
4	1	0	0	1	1	$3.09	Six Bars
		0	0	0	1	$3.19	Six Bars
		0	0	0	1	$3.19	Six Bars
		0	0	0	0		

The following steps fit the choice model:

```
%phchoice(on)                        /* customize PHREG for a choice model   */

title2 'Multinomial Logit Discrete Choice Model';

proc phreg data=coded brief;
   model c*c(2) = &_trgind / ties=breslow;
   strata subject set;
   run;

%phchoice(off)                       /* restore PHREG to a survival PROC     */
```

Notice that we use the `%PHChoice` macro to customize the output from PROC PHREG so that it looks more like discrete choice output and less like survival analysis output. The choice model is a special case of a survival-analysis model. The `brief` option is used to get a brief summary of the pattern of chosen and not chosen alternatives. This is very useful for checking data entry. Before the model equals sign, the first mention of `c` indicates the chosen alternative and the second mention of `c` indicates the alternatives that were not chosen. The list in parentheses indicates that values of 2 or greater were not chosen. When we set `c` as we did in this example (1 means first choice and 2 means unobserved second or subsequent choices), we will always specify `c*c(2)` before the equal sign as our response specification. A macro variable that PROC TRANSREG creates is specified after the equal sign. This macro variable is always called `&_trgind` and it contains the list of coded variables. The list of variables PROC TRANSREG creates will change for every study, but you can always use the macro variable `&_trgind` to get the list. The `ties=breslow` option specifies the likelihood function that we want for the multinomial logit discrete choice model. Each subject and set combination makes a contribution to the likelihood function, so those variables are specified in the `strata` statement. The results are as follows:

Cereal Bars
Multinomial Logit Discrete Choice Model

The PHREG Procedure

Model Information

Data Set	WORK.CODED
Dependent Variable	c
Censoring Variable	c
Censoring Value(s)	2
Ties Handling	BRESLOW

Number of Observations Read	2560
Number of Observations Used	2560

Summary of Subjects, Sets, and Chosen and Unchosen Alternatives

Pattern	Number of Choices	Number of Alternatives	Chosen Alternatives	Not Chosen
1	640	4	1	3

Convergence Status

Convergence criterion (GCONV=1E-8) satisfied.

Model Fit Statistics

Criterion	Without Covariates	With Covariates
-2 LOG L	1774.457	1142.630
AIC	1774.457	1156.630
SBC	1774.457	1187.860

Testing Global Null Hypothesis: BETA=0

Test	Chi-Square	DF	Pr > ChiSq
Likelihood Ratio	631.8271	7	<.0001
Score	518.1014	7	<.0001
Wald	275.0965	7	<.0001

Cereal Bars
Multinomial Logit Discrete Choice Model

The PHREG Procedure

Multinomial Logit Parameter Estimates

	DF	Parameter Estimate	Standard Error	Chi-Square	Pr > ChiSq
Branolicious	1	2.64506	0.47268	31.3142	<.0001
Brantasia	1	2.94600	0.47200	38.9571	<.0001
Brantopia	1	2.44876	0.47416	26.6706	<.0001
$2.89	1	2.69907	0.20307	176.6557	<.0001
$2.99	1	1.72036	0.17746	93.9845	<.0001
$3.09	1	0.76407	0.17437	19.2008	<.0001
Six Bars	1	-0.54645	0.11899	21.0912	<.0001

Notice near the top of the output that there was one pattern of results. There were 640 times (16 choice sets times 40 people) that four alternatives were presented and one was chosen. This table, which was produced by the `brief` option, provides a check on the data entry. Usually, the number of alternatives is the same in all choice sets, as it is here. Multiple patterns would mean a data entry error had occurred. The "Multinomial Logit Parameter Estimates" table is of primary interest. All of the part-worth utilities (parameter estimates) are significant, and the clearest pattern in the results is that the lower prices have the highest utility (the larger parameter estimates).

The following steps are not necessary, but they show some of the details about how the parameters are interpreted and how they can be used to find the utility of each combination. Recall that the choice model has the following form

$$p(c_i|C) = \frac{\exp(U(c_i))}{\sum_{j=1}^{m} \exp(U(c_j))} = \frac{\exp(\mathbf{x}_i\boldsymbol{\beta})}{\sum_{j=1}^{m} \exp(\mathbf{x}_j\boldsymbol{\beta})}$$

The following steps create the predicted utility of each alternative, $\widehat{U(c_i)}$, from each alternative's attributes, $\mathbf{x}_j$, and the vector of parameter estimates, $\hat{\boldsymbol{\beta}}$ (and additionally the element-wise products of $\mathbf{x}_j$, and $\hat{\boldsymbol{\beta}}$):

```
proc sort data=coded nodupkeys out=combos(drop=subject -- c);
   by brand price count;
   run;

data utils(drop=i);
   set combos;
   array b[7] _temporary_ (2.7 2.3 2.9 2.9 1.7 0.7 -1.2);
   array x[7] brandbranolicious -- countsix_bars;
   u = 0;
   do i = 1 to 7;
      x[i] = b[i] * x[i];
      u + x[i];
      end;
   run;

proc print label noobs split='-';
   title2 'Part-Worth Utility Report';
   label BrandBranolicious = 'Bran-olic-ious-'
         BrandBrantasia    = 'Bran-tas -ia  -'
         BrandBrantopia    = 'Bran-top -ia  -';
   id u;
   run;
```

The first step sorts the coded design data set by brand, price, and count, the three attributes, and deletes all duplicates. This creates 25 combinations: 3 brands times 4 prices times 2 counts plus one constant alternative. The DATA step multiplies each indicator variable in the resulting data set by its associated parameter estimate or part-worth utility. It also sums the appropriate part-worth utilities across all of the attributes and stores the result in the variable `u`. The final step displays the results, which are as follows:

Cereal Bars
Part-Worth Utility Report

u	Bran olic ious	Bran tas ia	Bran top ia	$2.89	$2.99	$3.09	Six Bars	Brand	Price	Count
4.4	2.7	0.0	0.0	2.9	0.0	0.0	-1.2	Branolicious	$2.89	Six Bars
5.6	2.7	0.0	0.0	2.9	0.0	0.0	0.0	Branolicious	$2.89	Eight Bars
3.2	2.7	0.0	0.0	0.0	1.7	0.0	-1.2	Branolicious	$2.99	Six Bars
4.4	2.7	0.0	0.0	0.0	1.7	0.0	0.0	Branolicious	$2.99	Eight Bars
2.2	2.7	0.0	0.0	0.0	0.0	0.7	-1.2	Branolicious	$3.09	Six Bars
3.4	2.7	0.0	0.0	0.0	0.0	0.7	0.0	Branolicious	$3.09	Eight Bars

```
1.5  2.7  0.0  0.0  0.0   0.0   0.0  -1.2 Branolicious $3.19 Six Bars
2.7  2.7  0.0  0.0  0.0   0.0   0.0   0.0 Branolicious $3.19 Eight Bars
4.0  0.0  2.3  0.0  2.9   0.0   0.0  -1.2 Brantasia    $2.89 Six Bars
5.2  0.0  2.3  0.0  2.9   0.0   0.0   0.0 Brantasia    $2.89 Eight Bars
2.8  0.0  2.3  0.0  0.0   1.7   0.0  -1.2 Brantasia    $2.99 Six Bars
4.0  0.0  2.3  0.0  0.0   1.7   0.0   0.0 Brantasia    $2.99 Eight Bars
1.8  0.0  2.3  0.0  0.0   0.0   0.7  -1.2 Brantasia    $3.09 Six Bars
3.0  0.0  2.3  0.0  0.0   0.0   0.7   0.0 Brantasia    $3.09 Eight Bars
1.1  0.0  2.3  0.0  0.0   0.0   0.0  -1.2 Brantasia    $3.19 Six Bars
2.3  0.0  2.3  0.0  0.0   0.0   0.0   0.0 Brantasia    $3.19 Eight Bars
4.6  0.0  0.0  2.9  2.9   0.0   0.0  -1.2 Brantopia    $2.89 Six Bars
5.8  0.0  0.0  2.9  2.9   0.0   0.0   0.0 Brantopia    $2.89 Eight Bars
3.4  0.0  0.0  2.9  0.0   1.7   0.0  -1.2 Brantopia    $2.99 Six Bars
4.6  0.0  0.0  2.9  0.0   1.7   0.0   0.0 Brantopia    $2.99 Eight Bars
2.4  0.0  0.0  2.9  0.0   0.0   0.7  -1.2 Brantopia    $3.09 Six Bars
3.6  0.0  0.0  2.9  0.0   0.0   0.7   0.0 Brantopia    $3.09 Eight Bars
1.7  0.0  0.0  2.9  0.0   0.0   0.0  -1.2 Brantopia    $3.19 Six Bars
2.9  0.0  0.0  2.9  0.0   0.0   0.0   0.0 Brantopia    $3.19 Eight Bars
0.0  0.0  0.0  0.0  0.0   0.0   0.0   0.0 None              .          .
```

At the bottom of the output, you can see that the utility for the constant alternative is 0, and the coded variables are such that it cannot possibly be anything else. The utilities for all of the other alternatives are computed relative to the constant. If the constant alternative were the most preferred, all of the other utilities would be negative. In this case, the constant alternative is least preferred and the other utilities are positive. Each of the 25 different combinations has a different pattern of indicator variables and parameters. "None" is the reference level for brand, and it has a zero part-worth utility for brand. The other brands have positive part-worth utilities relative to "None". The reference level for price is $3.19, and it has a zero part-worth utility. The other prices have positive part-worth utilities relative to $3.19 that increase as price decreases. Eight bars is the reference level for count, and it has a zero part-worth utility. The other count (6 bars) has negative part-worth utility relative to 8 bars. Other coding schemes would produce different but equivalent patterns of results.

In summary, this example shows the basic steps in designing, processing, and analyzing a choice experiment using the approach that creates the design directly using the `%MktEx` macro. The remaining examples illustrate other approaches that are also commonly used. Also, pages 285 through 663 have many more examples, much greater detail, and show how to use other tools.

Example 2: The Linear Arrangement Approach with Restrictions

In this example, we create a design for the same study as in the previous example. The product line is the same three fictitious breakfast bars with the same attributes as before. However, this time, rather than making a design in which all of the attributes of all of the alternatives are balanced and orthogonal, we impose restrictions on the design. This is not a full example, rather it simply shows how to modify the design-creation steps in the first example to impose restrictions. We will restrict the design to avoid choice sets where attributes are constant. That is, we want to select just the choice sets where neither price nor count is constant within a choice set. We use the `%MktEx` macro as follows

to get a restricted factorial design for this problem as follows:

```
%macro res;
   if x1 = x3 & x1 = x5 then bad = 1;
   if x2 = x4 & x2 = x6 then bad = bad + 1;
   %mend;

%mktex(4 2  4 2  4 2,       /* factor level list for all attrs and alts     */
       n=16,                /* number of choice sets                        */
       restrictions=res,    /* name of the restrictions macro               */
       options=resrep,      /* detailed report on restrictions              */
       seed=17)             /* random number seed                           */
```

Restrictions are written with PROC IML statements embedded in a macro. You provide the `%MktEx` macro with the name of the restrictions macro by using the `restrictions=`*macro-name* option. In the restrictions macro, you compute an IML scalar called `bad` that quantifies the badness of the design. In this case, since the restrictions are entirely within choice set, you can just quantify the badness of one choice set at at time by evaluating the values in the scalars `x1-x6`, which correspond to the six attributes. If it is easier to use indexing to write the restrictions, you can instead use the vector `x`, where `x[1]` = `x1`, ..., `x[6]` = `x6`. Furthermore, while it is not illustrated in this example, the levels of the attributes for the entire design are stored in a matrix called `xmat`, and you can use these values as well as the values in `x` to impose restrictions across choice sets. The scalar `bad` is automatically initialized to zero by the `%MktEx` macro. In this example, `bad` is set to 1 if the `Price` variable (which is made from `x1`, `x3`, and `x5`) is constant. The scalar `bad` is incremented by 1 if the `Count` variable (which is made from `x2`, `x4`, and `x6`) is constant. The `if` statements use the Boolean syntax of PROC IML. We must use the following IML logical operators, which do not have all of the same syntactical alternatives as DATA step operators:

Specify	For	Do Not Specify
=	equals	EQ
∧ = or ¬ =	not equals	NE
<	less than	LT
<=	less than or equal to	LE
>	greater than	GT
>=	greater than or equal to	GE
&	and	AND
|	or	OR
∧ or ¬	not	NOT
a <= b & b <= c	range check	a <= b <= c

Your macro can look at several scalars, along with a vector and a matrix in quantifying badness, and it must store its results in `bad`. The following names are available:

`i` – is a scalar that contains the number of the row currently being changed or evaluated. If you are writing restrictions that use the variable `i`, you almost certainly should specify `options=nosort`.

`try` – is a scalar similar to `i`, which contains the number of the row currently being changed. However, `try`, starts at zero and is incremented for each row, but it is only set back to zero when a new design starts, not when `%MktEx` reaches the last row. Use `i` as a matrix index and `try` to evaluate how far `%MktEx` is into the process of constructing the design.

`x` – is a row vector of factor levels for row `i` that always contains integer values beginning with 1 and continuing on to the number of levels for each factor. These values are always one-based, even if `levels=` is specified.

`x1` is the same as `x[1]`, `x2` is the same as `x[2]`, and so on.

`j1` – is a scalar that contains the number of the column currently being changed. In the steps where the badness macro is called once per row, `j1` = 1.

`j2` – is a scalar that contains the number of the other column currently being changed (along with `j1`) with `exchange=2`. Both `j1` and `j2` are defined when the `exchange=` value is greater than or equal to two. This scalar will not exist with `exchange=1`. In the steps where the badness macro is called once per row, `j1` = `j21` = 1.

`j3` – is a scalar that contains the number of the third column currently being changed (along with `j1` and `j2`) with `exchange=3` and larger `exchange=` values. This scalar will not exist with `exchange=1` and `exchange=2`. If and only if the `exchange=`value is greater than 3, there will be a `j4` and so on. In the steps where the badness macro is called once per row, `j1` = `j2` = `j3` = 1.

`xmat` – is the entire `x` matrix. Note that the *ith* row of `xmat` is often different from `x` since `x` contains information about the exchanges being considered, whereas `xmat` contains the current design.

`bad` – results: 0 – fine, or the number of violations of restrictions. You can make this value large or small, and you can use integers or real numbers. However, the values should always be nonnegative. When there are multiple sources of design badness, it is sometimes good to scale the different sources on different scales so that they do not trade off against each other. For example, for the first source, you might multiply the number of violations by 1000, by 100 for another source, by 10 for another source, by 1 for another source, and even sometimes by 0.1 or 0.01 for another source. The final badness is the sum of `bad`, `__pbad` (when it exists), and `__bbad` (when it exists). The scalars `__pbad` and `__bbad` are explained next.

`__pbad` – is the badness from the `partial=` option. When `partial=` is not specified, this scalar does not exist. Your macro can weight this value, typically by multiplying it times a constant, to differentially weight the contributors to badness, e.g.: `__pbad = __pbad * 10`.

`__bbad` – is the badness from the `balance=` option. When `balance=` is not specified, this scalar does not exist. Your macro can weight this value, typically by multiplying it times a constant, to differentially weight the contributors to badness, e.g.: `__bbad = __bbad * 100`.

Do not use these names (other than `bad`) for intermediate values!

Other than that, you can create intermediate variables without worrying about conflicts with the names in the macro. The levels of the factors for one row of the experimental design are stored in a vector `x`, and the first level is always 1, the second always 2, and so on. All restrictions must be defined in terms of `x[j]` (or alternatively, `x1`, `x2`, ..., and perhaps the other matrices).

One other option is specified in this example, and that is `options=resrep`. It is a good idea to specify this option with restrictions until you are sure that your restrictions macro is correct. It provides detailed information about the swaps that the `%MktEx` macro performs and its success in imposing restrictions.

Part of the iteration history table, with a lot of `options=resrep` information and other information deleted, is as follows:

Cereal Bars

Algorithm Search History

Design	Row,Col	Current D-Efficiency	Best D-Efficiency	Notes
1	Start	91.5504		Can
1	1	91.5504		0 Violations
1	2	91.5504		0 Violations
.				
.				
.				
1	16	91.5504		0 Violations
1	1	91.5504		0 Violations
1	2 1	91.5504	91.5504	Conforms
1	End	91.5504		
2	Start	100.0000		Tab
2	1	95.2509		0 Violations
2	2	95.2509		0 Violations
.				
.				
.				
2	14 1	87.8163		Conforms
2	End	89.5326		
.				
.				
.				
12	Start	61.3114		Ran,Mut,Ann
12	1	61.3115		0 Violations
12	2	65.3098		0 Violations
.				
.				
.				
12	16 1	79.4843		Conforms
12	End	89.1152		
.				
.				
.				

21	Start		53.6255	Ran,Mut,Ann
21	1		60.0902	0 Violations
21	2		60.0902	0 Violations
.				
.				
.				
21	16	1	81.3052	Conforms
21	End		89.5758	

NOTE: Performing 1000 searches of 360 candidates.

Cereal Bars

Design Search History

Design	Row,Col		Current D-Efficiency	Best D-Efficiency	Notes
0	Initial		91.5504	91.5504	Ini
1	Start		91.5504		Can
1	1		91.5504		0 Violations
1	2		91.5504		0 Violations
1	3		91.5504		0 Violations
.					
.					
.					
1	16		91.5504		0 Violations
1	1		91.5504		0 Violations
1	2	1	91.5504	91.5504	Conforms
1	End		91.5504		

Cereal Bars

Design Refinement History

Design	Row,Col	Current D-Efficiency	Best D-Efficiency	Notes
0	Initial	91.5504	91.5504	Ini
1	Start	85.9351		Pre,Mut,Ann
1	1	85.9351		0 Violations
1	2	85.9351		0 Violations
1	3	87.0050		0 Violations
.				
.				
.				
1	16 1	91.5504	91.5504	
1	2 1	91.5504	91.5504	
1	End	91.5504		
.				
.				
.				
9	Start	91.5504		Pre,Mut,Ann
9	1	91.5504		0 Violations
9	2	91.5504		0 Violations
.				
.				
.				
9	16	91.5504		0 Violations
9	1	91.5504		0 Violations
9	2 1	91.5504	91.5504	Conforms
9	End	87.4793		

NOTE: Stopping since it appears that no improvement is possible.

The OPTEX Procedure

Class Level Information

Class	Levels	Values
x1	4	1 2 3 4
x2	2	1 2
x3	4	1 2 3 4
x4	2	1 2
x5	4	1 2 3 4
x6	2	1 2

Design Number	D-Efficiency	A-Efficiency	G-Efficiency	Average Prediction Standard Error
1	91.5504	80.7068	93.8194	0.9014

The iteration history table for the %MktEx macro is described in detail in the discrete choice chapter starting on page 285. In this example, just note a few things. The %MktEx macro is successful in making the design conform to all restrictions. In all cases, it reports "0 Violations" of the restrictions. In some cases, a design with 100% *D*-efficiency is replaced by a design with a lower *D*-efficiency as the restrictions are imposed. The final *D*-efficiency is 91.5504. Designs with this same *D*-efficiency are repeatedly found, which often indicates that an optimal design was found.

The following steps create the choice design from the linear arrangement and display the results:

```
title2 'Create the Choice Design Key';

data key;
   input
Brand $ 1-12    Price $    Count $; datalines;
Branolicious    x1         x2
Brantopia       x3         x4
Brantasia       x5         x6
None            .          .
;

title2 'Create Choice Design from Linear Arrangement';

proc format;
   value price 1 = $2.89 2 = $2.99 3 = $3.09 4 = $3.19 . = ' ';
   value count 1 = 'Six Bars' 2 = 'Eight Bars'  . = ' ';
   run;

%mktroll(design=randomized,  /* input randomized linear arrangement         */
         key=key,            /* rules for making choice design              */
         alt=brand,          /* brand or alternative label var              */
         out=cerealdes)      /* output choice design                        */

title2;
proc print; format price price. count count.; id set; by set; run;
```

The choice design, with restrictions, is as follows:

Cereal Bars

Set	Brand	Price	Count
1	Branolicious	$3.09	Six Bars
	Brantopia	$2.89	Six Bars
	Brantasia	$3.09	Eight Bars
	None		
2	Branolicious	$2.89	Six Bars
	Brantopia	$2.99	Eight Bars
	Brantasia	$3.19	Six Bars
	None		
3	Branolicious	$2.99	Eight Bars
	Brantopia	$2.99	Eight Bars
	Brantasia	$3.09	Six Bars
	None		
4	Branolicious	$2.89	Eight Bars
	Brantopia	$2.89	Six Bars
	Brantasia	$2.99	Eight Bars
	None		
5	Branolicious	$3.19	Eight Bars
	Brantopia	$3.19	Six Bars
	Brantasia	$3.09	Six Bars
	None		
6	Branolicious	$3.09	Six Bars
	Brantopia	$3.19	Eight Bars
	Brantasia	$2.99	Eight Bars
	None		
7	Branolicious	$2.89	Six Bars
	Brantopia	$3.09	Eight Bars
	Brantasia	$3.09	Eight Bars
	None		
8	Branolicious	$3.19	Six Bars
	Brantopia	$2.99	Six Bars
	Brantasia	$2.99	Eight Bars
	None		
9	Branolicious	$3.09	Eight Bars
	Brantopia	$2.99	Six Bars
	Brantasia	$2.89	Eight Bars
	None		

```
10    Branolicious    $3.19    Eight Bars
      Brantopia       $2.89    Eight Bars
      Brantasia       $3.19    Six Bars
      None

11    Branolicious    $2.89    Eight Bars
      Brantopia       $3.19    Six Bars
      Brantasia       $2.89    Six Bars
      None

12    Branolicious    $3.09    Eight Bars
      Brantopia       $3.09    Six Bars
      Brantasia       $2.99    Six Bars
      None

13    Branolicious    $3.19    Six Bars
      Brantopia       $3.09    Eight Bars
      Brantasia       $2.89    Eight Bars
      None

14    Branolicious    $2.99    Six Bars
      Brantopia       $3.19    Six Bars
      Brantasia       $3.19    Eight Bars
      None

15    Branolicious    $2.99    Six Bars
      Brantopia       $2.89    Eight Bars
      Brantasia       $2.89    Six Bars
      None

16    Branolicious    $2.99    Eight Bars
      Brantopia       $3.09    Six Bars
      Brantasia       $3.19    Eight Bars
      None
```

This example provides an illustration of a restrictions macro in a context that is simple enough that it is easy to write the restrictions macro and nothing is likely to go wrong. Many other uses of restrictions are not this simple. The discrete choice chapter starting on page 285 and the `%MktEx` macro documentation starting on page 1017 provide more information about restricting designs with the `%MktEx` macro. There are a few things to always keep in mind when writing restrictions:

- Ensure that you are specifying a set of restrictions that are possible. It is not uncommon for people to write a set of restrictions that cannot possibly be satisfied (for example, `bad = (a >= b) + (b >= a);` although most errors in restriction specifications are not nearly this obvious).
- You can specify restrictions that prohibit certain combinations of attributes from appearing together. You can ask for interactions involving those attributes. However, if you do both, you might find that the *D*-efficiency of your design is zero. Depending on what you specify, you might not be able to have both estimable interactions and restrictions. In order for an interaction parameter to be estimable, certain combinations of levels must appear in the design. If you prohibit

those combinations from occurring, the parameter cannot be estimated.

- Badness must be quantified. When there are multiple sources of badness, badness must be quantified as granularly as you can. If you simply set `bad` to zero when all is fine and nonzero when all is not fine, you might not be giving `%MktEx` enough information to impose restrictions. You have to quantify badness in a way that tells `%MktEx` when it makes a change that is closer to the desired result even when it does not achieve the desired result. Without that information, `%MktEx` does not know when it is moving in the right direction. For example, if your badness criterion forces some attributes to be constant, your badness criterion should reflect how far the varying attributes are from constant when too few attributes are constant.

- When there are multiple independent sources of badness, you might have to differentially weight the sources (e.g. give one part a weight of 1, another a weight of 10, another a weight of 50, and so on). It might not matter which part gets what weight. Often, the important thing is to provide some differential weighting so that sources of badness never trade off against one another. For example, if all sources get an implicit weight of 1, then every time `%MktEx` makes progress on one source it might make the other source worse. Differential weighting can help prevent this.

- Always use `options=resrep` until you are sure you are creating the design correctly. The information that is provided in the iteration history table with `options=resrep` is often very helpful in diagnosing problems with the restrictions macro.

- It is good practice to use `options=quickr` at first with restrictions, unless you have a very small problem like the one in this example. This makes the macro run faster and just tries to make a single design. Use this option to minimize run time until you are sure your restrictions macro is correct, then remove it and let the macro run longer to make the final design. There are many other options to minimize run time for complicated designs while you are checking your code.

- Use the `out=design` data set with restrictions and do not randomize the design. Randomization can destroy the structure imposed by the restrictions.

- Do not change the values of `i`, `try`, `x`, `x1 - x`*n*, `j1`, `j2`, `j3`, `xmat`, `__pbad`, or `__bbad` in your restrictions macro. Do not use these names for any intermediate matrices. Be particularly careful to avoid using `i` as an index in a `do` loop. Use `j` or `k` instead. Do not ignore warnings that you are changing macros in your restrictions macro.

- Restrictions are written in PROC IML, the interactive matrix language, which has a rich variety of matrix and vector operations. Do not use long series of scalar operations when simple vector or matrix operations are available. The following two macros are equivalent, although the first is obviously easier to construct than the second:

```
%macro bad; * The easy way with vectors;
   c1 = sum(x[, 1:32] = 1);
   c2 = sum(x[,33:64] = 1);
   bad = abs(6 - c1) + abs(6 - c2);
   %mend;
```

```
%macro bad; * The hard way with scalars;
   c1 = (x1  = 1) + (x2  = 1) + (x3  = 1) + (x4  = 1) + (x5  = 1) + (x6  = 1)
      + (x7  = 1) + (x8  = 1) + (x9  = 1) + (x10 = 1) + (x11 = 1) + (x12 = 1)
      + (x13 = 1) + (x14 = 1) + (x15 = 1) + (x16 = 1) + (x17 = 1) + (x18 = 1)
      + (x19 = 1) + (x20 = 1) + (x21 = 1) + (x22 = 1) + (x23 = 1) + (x24 = 1)
      + (x25 = 1) + (x26 = 1) + (x27 = 1) + (x28 = 1) + (x29 = 1) + (x30 = 1)
      + (x31 = 1) + (x32 = 1);
   c2 = (x33 = 1) + (x34 = 1) + (x35 = 1) + (x36 = 1) + (x37 = 1) + (x38 = 1)
      + (x39 = 1) + (x40 = 1) + (x41 = 1) + (x42 = 1) + (x43 = 1) + (x44 = 1)
      + (x45 = 1) + (x46 = 1) + (x47 = 1) + (x48 = 1) + (x49 = 1) + (x50 = 1)
      + (x51 = 1) + (x52 = 1) + (x53 = 1) + (x54 = 1) + (x55 = 1) + (x56 = 1)
      + (x57 = 1) + (x58 = 1) + (x59 = 1) + (x60 = 1) + (x61 = 1) + (x62 = 1)
      + (x63 = 1) + (x64 = 1);
   bad = abs(6 - c1) + abs(6 - c2);
   %mend;
```

Both count how many times the first 32 attributes are different from 1 and how many times the second 32 attributes are different from 1. The first macro does so by comparing a vector with a scalar resulting in a vector of ones when the comparison is true and zeros when the comparison is false. Both create a sum of a series of zeros and ones (falses and trues). Both increase badness when there are not 6 ones in the first 32 attributes and 6 ones in the second 32 attributes.

Example 3, Searching a Candidate Set of Alternatives

In this example, we create a design for the same study as in the previous example. The product line is the same three fictitious breakfast bars with the same attributes as before. However, this time, rather than making a design in which all of the attributes of all of the alternatives are balanced and orthogonal, we instead make a design that is efficient for a choice model under the null hypothesis $\boldsymbol{\beta} = \mathbf{0}$ and with a specific model specification. The design is constructed from a candidate set of alternatives. See page 127 for more information about the brands and levels.

Our design consists of sets of alternatives, just like the design in the previous example (see page 134). In each set, the first alternative always consists of one of the 8 combinations for Branolicious (4 prices $\times$ 2 sizes). Similarly, the second alternative always consists of one of the 8 combinations for Brantopia, and the third alternative always consists of one of the 8 combinations for Brantasia. There is only one possibility for the constant or "None" alternative. We construct a candidate set that consists of four types of candidates, one type for each of the four alternative brands, and then we use the `%ChoicEff` macro to build a design from those candidates.

First, we use `%MktEx` to make the full set of price and size combinations that are needed to make the candidate design. The `%MktEx` specification makes a 4×2 experimental design in 8 runs. If this example had been substantially larger, we might instead make an orthogonal array with a subset of the combinations instead of a full-factorial design. We use the `%MktLab` macro to provide the names of the attributes. The original (not the randomized) design from the `%MktEx` macro is used as input. (Since all combinations are being used, there is no need to randomize.) The following steps make the design:

```
title 'Cereal Bars';

%mktex(4 2,                          /* all attribute levels                */
       n=8)                          /* number of candidate alternatives    */

%mktlab(data=design,                 /* original design from MktEx          */
        vars=Price Count)            /* new variable names                  */

proc print; run;
```

The results are as follows:

Cereal Bars

Obs	Price	Count
1	1	1
2	1	2
3	2	1
4	2	2
5	3	1
6	3	2
7	4	1
8	4	2

From this design, we can create the full list of candidates as follows:

```
data cand;
   length Brand $ 12;
   retain Price Count . f1-f4 0;

   if _n_ = 1 then do;
      brand = 'None        '; f4 = 1; output; f4 = 0;    /* brand 4 (None)  */
      end;

   set final;

   brand = 'Branolicious';    f1 = 1; output; f1 = 0;    /* brand 1         */
   brand = 'Brantasia   ';    f2 = 1; output; f2 = 0;    /* brand 2         */
   brand = 'Brantopia   ';    f3 = 1; output; f3 = 0;    /* brand 3         */
   run;

proc print; run;
```

This DATA step reads each row of the full-factorial design and processes it. However, before it executes the `set` statement for the first time, on the first pass through the DATA step (when `_n_ = 1`), it writes out a candidate for the None alternative and flags it by changing `f4` to 1. Note that initially, `f1-f4` are all initialized to zero in the `retain` statement. Then `f4` is set back to zero after the row is written to the SAS data set. When each of the 8 price and size combinations is read in, three are written out,

one for each brand. When a Branolicious alternative is created, `f1` is set to 1 and `f2`, `f3`, and `f4` are zero. When a Brantasia alternative is created, `f2` is set to 1 and `f1`, `f3`, and `f4` are zero. When a Brantopia alternative is created, `f3` is set to 1 and `f1`, `f2`, and `f4` are zero. The results are as follows:

Obs	Brand	Price	Count	f1	f2	f3	f4
1	None	.	.	0	0	0	1
2	Branolicious	1	1	1	0	0	0
3	Brantasia	1	1	0	1	0	0
4	Brantopia	1	1	0	0	1	0
5	Branolicious	1	2	1	0	0	0
6	Brantasia	1	2	0	1	0	0
7	Brantopia	1	2	0	0	1	0
8	Branolicious	2	1	1	0	0	0
9	Brantasia	2	1	0	1	0	0
10	Brantopia	2	1	0	0	1	0
11	Branolicious	2	2	1	0	0	0
12	Brantasia	2	2	0	1	0	0
13	Brantopia	2	2	0	0	1	0
14	Branolicious	3	1	1	0	0	0
15	Brantasia	3	1	0	1	0	0
16	Brantopia	3	1	0	0	1	0
17	Branolicious	3	2	1	0	0	0
18	Brantasia	3	2	0	1	0	0
19	Brantopia	3	2	0	0	1	0
20	Branolicious	4	1	1	0	0	0
21	Brantasia	4	1	0	1	0	0
22	Brantopia	4	1	0	0	1	0
23	Branolicious	4	2	1	0	0	0
24	Brantasia	4	2	0	1	0	0
25	Brantopia	4	2	0	0	1	0

The result is a data set with $3 \times 8 + 1 = 25$ candidates. The flag variables, `f1-f4`, designate the first alternative (`f1` = 1), second alternative (`f2` = 1), third alternative (`f3` = 1), and fourth alternative (`f4` = 1). In this study, exactly one variable in the `f1-f4` list is equal to 1 at any one time. If this were a generic study with no brands and the same attribute levels for each alternative, it would be possible to use the same candidate for multiple alternatives. The results might look clearer when sorted by brand. Sorting is not necessary. It just permits a clearer display of the structure of the data set. Formats are also provided. The following steps process and display the candidate set:

```
proc format;
   value price 1 = $2.89 2 = $2.99 3 = $3.09 4 = $3.19 . = ' ';
   value count 1 = 'Six Bars' 2 = 'Eight Bars'  . = ' ';
   run;
```

```
proc sort;
   by brand price count;
   format price price. count count.;
   run;

proc print label; run;
```

The results, sorted by brand, are as follows:

Cereal Bars

Obs	Brand	Price	Count	f1	f2	f3	f4
1	Branolicious	$2.89	Six Bars	1	0	0	0
2	Branolicious	$2.89	Eight Bars	1	0	0	0
3	Branolicious	$2.99	Six Bars	1	0	0	0
4	Branolicious	$2.99	Eight Bars	1	0	0	0
5	Branolicious	$3.09	Six Bars	1	0	0	0
6	Branolicious	$3.09	Eight Bars	1	0	0	0
7	Branolicious	$3.19	Six Bars	1	0	0	0
8	Branolicious	$3.19	Eight Bars	1	0	0	0
9	Brantasia	$2.89	Six Bars	0	1	0	0
10	Brantasia	$2.89	Eight Bars	0	1	0	0
11	Brantasia	$2.99	Six Bars	0	1	0	0
12	Brantasia	$2.99	Eight Bars	0	1	0	0
13	Brantasia	$3.09	Six Bars	0	1	0	0
14	Brantasia	$3.09	Eight Bars	0	1	0	0
15	Brantasia	$3.19	Six Bars	0	1	0	0
16	Brantasia	$3.19	Eight Bars	0	1	0	0
17	Brantopia	$2.89	Six Bars	0	0	1	0
18	Brantopia	$2.89	Eight Bars	0	0	1	0
19	Brantopia	$2.99	Six Bars	0	0	1	0
20	Brantopia	$2.99	Eight Bars	0	0	1	0
21	Brantopia	$3.09	Six Bars	0	0	1	0
22	Brantopia	$3.09	Eight Bars	0	0	1	0
23	Brantopia	$3.19	Six Bars	0	0	1	0
24	Brantopia	$3.19	Eight Bars	0	0	1	0
25	None			0	0	0	1

The candidate set consists of eight rows for the first alternative, flagged by `f1` = 1 (and `f2` = `f3` = `f4` = 0). The second group is flagged by `f2` = 1, and so on. The constant alternative is flagged by `f4` = 1. The design is created from the candidates using the `%ChoicEff` macro as follows:

```
%choiceff(data=cand,                /* candidate set of alternatives        */
          bestout=sasuser.cerealdes,/* choice design permanently stored     */
                                    /* model with stdz orthogonal coding    */
          model=class(brand price count / sta),
          maxiter=10,               /* maximum number of designs to make    */
          flags=f1-f4,              /* flag which alt can go where, 4 alts  */
          nsets=16,                 /* number of choice sets                */
          seed=306,                 /* random number seed                   */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

The `data=` option names the data set of candidates. The best design is stored in a SAS data set named in the `bestout=` option. In the `%ChoicEff` macro, all of the designs go to the `out=` data set, so typically you want to use the `bestout=` data set instead. By default, this data set is called `best`. Here we create a permanent SAS data set so that it is around at analysis time. The main-effects model is specified in the `model=` option, and a standardized orthogonal coding is used. When you have a candidate set of alternatives, as we have here, then you need to specify the flag variables in the `flags=` option. Otherwise, when you have a candidate set of choice sets, you specify the `nalts=` option. Note that the `%ChoicEff` macro uses the number of variables in the `flags=` list to set the number of alternatives. The `beta=zero` option specifies the assumed parameter vector. We specify a random number seed so that we always get the same design if we rerun the `%ChoicEff` macro. Some of the results are as follows:

Cereal Bars

Final Results

Design	4
Choice Sets	16
Alternatives	4
Parameters	7
Maximum Parameters	48
D-Efficiency	12.9142
Relative D-Eff	80.7140
D-Error	0.0774
1 / Choice Sets	0.0625

Cereal Bars 16

n	Variable Name	Label	Variance	DF	Standard Error
1	BrandBranolicious	Brand Branolicious	0.062500	1	0.25000
2	BrandBrantasia	Brand Brantasia	0.062500	1	0.25000
3	BrandBrantopia	Brand Brantopia	0.062500	1	0.25000
4	Price_2_89	Price $2.89	0.090933	1	0.30155
5	Price_2_99	Price $2.99	0.090980	1	0.30163
6	Price_3_09	Price $3.09	0.090909	1	0.30151
7	CountSix_Bars	Count Six Bars	0.091003	1	0.30167
				==	
				7	

This table shows the variances and standard errors under the null-hypothesis assumption $\boldsymbol{\beta} = \mathbf{0}$. We see three parameters for brand (4 alternatives including None minus 1), three for price $(4-1)$, one for count $(2-1)$. With 16 choice sets and 4 alternatives, we can estimate at most $16 \times (4-1) = 48$ parameters. All are estimable, and all have reasonable standard errors. The variances for the brand effects are the inverse of the number of choice sets $(1/16 = 0.06250)$.* The other variances are bigger. These results look good.

The following step displays the choice sets:

```
proc print data=sasuser.cerealdes;
   by set;
   id set;
   var brand -- count;
   run;
```

The choice sets are as follows:

Cereal Bars

Set	Brand	Price	Count
1	Branolicious	$3.19	Six Bars
	Brantasia	$2.89	Six Bars
	Brantopia	$2.99	Eight Bars
	None		
2	Branolicious	$2.89	Eight Bars
	Brantasia	$3.19	Six Bars
	Brantopia	$2.99	Eight Bars
	None		

*This comparison is only valid when the standardized orthogonal contrast coding is used.

3	Branolicious	$3.19	Six Bars
	Brantasia	$3.09	Eight Bars
	Brantopia	$2.99	Six Bars
	None		
4	Branolicious	$2.99	Eight Bars
	Brantasia	$3.19	Six Bars
	Brantopia	$2.89	Six Bars
	None		
5	Branolicious	$3.09	Six Bars
	Brantasia	$2.99	Eight Bars
	Brantopia	$2.89	Eight Bars
	None		
6	Branolicious	$3.09	Eight Bars
	Brantasia	$3.19	Eight Bars
	Brantopia	$2.89	Six Bars
	None		
7	Branolicious	$2.99	Eight Bars
	Brantasia	$2.89	Six Bars
	Brantopia	$3.09	Six Bars
	None		
8	Branolicious	$2.89	Eight Bars
	Brantasia	$3.09	Six Bars
	Brantopia	$2.99	Six Bars
	None		
9	Branolicious	$3.09	Six Bars
	Brantasia	$2.89	Eight Bars
	Brantopia	$3.19	Eight Bars
	None		
10	Branolicious	$2.89	Eight Bars
	Brantasia	$2.99	Six Bars
	Brantopia	$3.19	Eight Bars
	None		
11	Branolicious	$3.19	Eight Bars
	Brantasia	$3.09	Eight Bars
	Brantopia	$2.89	Six Bars
	None		
12	Branolicious	$3.09	Eight Bars
	Brantasia	$2.99	Six Bars
	Brantopia	$3.19	Eight Bars
	None		

```
13    Branolicious    $2.89    Six Bars
      Brantasia       $3.19    Six Bars
      Brantopia       $3.09    Eight Bars
      None

14    Branolicious    $2.99    Six Bars
      Brantasia       $2.89    Eight Bars
      Brantopia       $3.09    Six Bars
      None

15    Branolicious    $3.19    Six Bars
      Brantasia       $2.99    Eight Bars
      Brantopia       $3.09    Six Bars
      None

16    Branolicious    $2.99    Six Bars
      Brantasia       $3.09    Eight Bars
      Brantopia       $3.19    Eight Bars
      None
```

It is instructive to look at the covariance matrix of the parameter estimates. The following steps display this matrix, which is automatically output to a SAS data set called `bestcov`:

```
proc format;
   value zer -1e-12 - 1e-12 = ' 0   ';
   run;

proc print data=bestcov label;
   id __label;
   label __label = '00'x;
   var BrandBranolicious -- CountSix_Bars;
   format _numeric_ zer5.2;
   run;
```

The format displays values very close to zero as precisely zero to make a better display. The results are as follows:

Cereal Bars

	Brand Branolicious	Brand Brantasia	Brand Brantopia	Price $2.89	Price $2.99	Price $3.09	Count Six Bars
Brand Branolicious	0.06	0	0	0	0	0	0
Brand Brantasia	0	0.06	0	0	0	0	0
Brand Brantopia	0	0	0.06	0	0	0	0
Price $2.89	0	0	0	0.09	0.00	0	0.00
Price $2.99	0	0	0	0.00	0.09	0	0.00
Price $3.09	0	0	0	0	0	0.09	0
Count Six Bars	0	0	0	0.00	0.00	0	0.09

There are some nonzero covariances between the price and count attributes.

There is one more test that should be run before a design is used. In the following step, the `%MktDups` macro checks the design to see if any choice sets are duplicates of any other choice sets:

```
%mktdups(branded,                          /* a design with brands          */
         data=sasuser.cerealdes,           /* the input design to evaluate  */
         factors=brand price count,        /* factors in the design         */
         nalts=4)                          /* number of alternatives        */
```

The results are as follows:

```
Design:          Branded
Factors:         brand price count
                 Brand
                 Count Price
Duplicate Sets:  0
```

The first line of the table tells us that this is a branded design as opposed to a generic design (bundles of attributes with no brands). The second line tells us the factors as specified in the `factors=` option. These are followed by the actual variable names for the factors. The last line reports the number of duplicates. In this case, there are no duplicate choice sets. If there had been duplicate choice sets, then changing the random number seed might help. Sometimes, changing other aspects of the design or the approach for making the design helps.

The questionnaire is designed and data are collected and entered, as they were in the previous example. The data are entered in the following step:

```
title2 'Read Data';

data results;
   input Subject (r1-r16) (1.);
   datalines;
  1 3131312123331232
  2 3131332121331222
  3 3133331121321222
  4 3111232121321223
  5 3131312122311222
  6 3131333121213232
  7 3131311132331332
  8 3131331121321223
  9 3121332122331222
 10 3131132123321222
 11 3131312121321223
 12 3121332122311232
 13 3131331122321222
 14 3131311121331223
 15 3131311121121223
 16 3121333221321223
 17 3121332131311222
 18 3131233121331222
 19 3131332121131232
 20 3131231121311432
 21 2133232121331223
 22 3121332121121221
 23 3131332121331223
 24 3133312131311323
 25 3131332431311232
 26 3131312143331222
 27 3121112133311223
 28 3123331121321221
 29 3121312122311231
 30 3131332121311233
 31 3121332121341222
 32 2131231122331231
 33 2131312122231221
 34 2121332321311421
 35 3133311121331233
 36 3131333121321323
 37 3131312131331223
 38 3121312331331221
 39 3121312121221231
 40 3123232121321232
;
```

The %MktMerge macro merges the data and the design and creates the dependent variable as follows:

```
title2 'Merge Data and Design';

%mktmerge(design=sasuser.cerealdes, /* input design                          */
          data=results,             /* input data set                       */
          out=res2,                 /* output data set with design and data */
          nsets=16,                 /* number of choice sets                */
          nalts=4,                  /* number of alternatives               */
          setvars=r1-r16)           /* variables with the chosen alt nums   */
```

The coding and analysis are the same as in the previous example and are as follows:

```
title2 'Code the Independent Variables';

proc transreg design norestoremissing data=res2;
   model class(brand price count);
   id subject set c;
   output out=coded(drop=_type_ _name_ intercept) lprefix=0;
   run;

%phchoice(on)                       /* customize PHREG for a choice model   */

title2 'Multinomial Logit Discrete Choice Model';

proc phreg data=coded brief;
   model c*c(2) = &_trgind / ties=breslow;
   strata subject set;
   run;

%phchoice(off)                      /* restore PHREG to a survival PROC     */
```

The parameter estimate table is as follows:

Cereal Bars
Multinomial Logit Discrete Choice Model

The PHREG Procedure

Multinomial Logit Parameter Estimates

	DF	Parameter Estimate	Standard Error	Chi-Square	Pr > ChiSq
Branolicious	1	2.70313	0.48143	31.5257	<.0001
Brantasia	1	2.32653	0.48515	22.9970	<.0001
Brantopia	1	2.88085	0.47575	36.6674	<.0001
$2.89	1	2.92298	0.20046	212.6164	<.0001
$2.99	1	1.71761	0.18827	83.2303	<.0001
$3.09	1	0.66151	0.19513	11.4922	0.0007
Six Bars	1	-1.21635	0.11703	108.0246	<.0001

Example 4, Searching a Candidate Set of Alternatives with Restrictions

In this example, we create a design for the same study as the previous example. The product line is the same three fictitious breakfast bars with the same attributes as before. Like last time, we create a candidate set of alternatives and make a design that is efficient for a choice model under the null hypothesis $\boldsymbol{\beta} = \mathbf{0}$ and with a specific model specification. However, this time, we place restrictions on how the candidates can come into the design. This example uses options in the `%ChoicEff` macro that first became available with this edition of the macros and book.

We will begin by creating a candidate set of alternatives in exactly the same way that we did it in the preceding example:

```
title 'Cereal Bars';

%mktex(4 2,                          /* all attribute levels                */
       n=8)                          /* number of candidate alternatives    */

%mktlab(data=design,                 /* original design from MktEx          */
        vars=Price Count)            /* new variable names                  */

data cand;
   length Brand $ 12;
   retain Price Count . f1-f4 0;

   if _n_ = 1 then do;
      brand = 'None        '; f4 = 1; output; f4 = 0;    /* brand 4 (None)  */
      end;

   set final;

   brand = 'Branolicious';    f1 = 1; output; f1 = 0;    /* brand 1         */
   brand = 'Brantasia   ';    f2 = 1; output; f2 = 0;    /* brand 2         */
   brand = 'Brantopia   ';    f3 = 1; output; f3 = 0;    /* brand 3         */
   run;

proc format;
   value price 1 = $2.89 2 = $2.99 3 = $3.09 4 = $3.19 . = ' ';
   value count 1 = 'Six Bars' 2 = 'Eight Bars'  . = ' ';
   run;

proc sort;
   by brand price count;
   format price price. count count.;
   run;

proc print; run;
```

Sorting by brand is not necessary. It is just done here to more clearly show the structure of the candidate set. The candidate set consists of eight rows for the first alternative, flagged by `f1` = 1 (and `f2` = `f3` = `f4` = 0). The second group is flagged by `f2` = 1 and the rest zero, and so on. The constant alternative is flagged by `f4` = 1. The candidate set is as follows:

Cereal Bars

Obs	Brand	Price	Count	f1	f2	f3	f4
1	Branolicious	$2.89	Six Bars	1	0	0	0
2	Branolicious	$2.89	Eight Bars	1	0	0	0
3	Branolicious	$2.99	Six Bars	1	0	0	0
4	Branolicious	$2.99	Eight Bars	1	0	0	0
5	Branolicious	$3.09	Six Bars	1	0	0	0
6	Branolicious	$3.09	Eight Bars	1	0	0	0
7	Branolicious	$3.19	Six Bars	1	0	0	0
8	Branolicious	$3.19	Eight Bars	1	0	0	0
9	Brantasia	$2.89	Six Bars	0	1	0	0
10	Brantasia	$2.89	Eight Bars	0	1	0	0
11	Brantasia	$2.99	Six Bars	0	1	0	0
12	Brantasia	$2.99	Eight Bars	0	1	0	0
13	Brantasia	$3.09	Six Bars	0	1	0	0
14	Brantasia	$3.09	Eight Bars	0	1	0	0
15	Brantasia	$3.19	Six Bars	0	1	0	0
16	Brantasia	$3.19	Eight Bars	0	1	0	0
17	Brantopia	$2.89	Six Bars	0	0	1	0
18	Brantopia	$2.89	Eight Bars	0	0	1	0
19	Brantopia	$2.99	Six Bars	0	0	1	0
20	Brantopia	$2.99	Eight Bars	0	0	1	0
21	Brantopia	$3.09	Six Bars	0	0	1	0
22	Brantopia	$3.09	Eight Bars	0	0	1	0
23	Brantopia	$3.19	Six Bars	0	0	1	0
24	Brantopia	$3.19	Eight Bars	0	0	1	0
25	None			0	0	0	1

We will restrict the design to avoid choice sets where attributes are constant. That is, you want to select just the choice sets where neither price nor count is constant within a choice set (outside the constant alternative). This is accomplished with the following steps:

```
%macro res;
   if x[1,1] = x[2,1] & x[1,1] = x[3,1] then bad = 1;
   if x[1,2] = x[2,2] & x[1,2] = x[3,2] then bad = bad + 1;
   %mend;

%choiceff(data=cand,                /* candidate set of alternatives        */
          bestout=sasuser.cerealdes,/* choice design permanently stored     */
                                    /* model with stdz orthogonal coding    */
          model=class(brand price count / sta),
          maxiter=10,               /* maximum number of designs to make    */
          flags=f1-f4,              /* flag which alt can go where, 4 alts  */
          nsets=16,                 /* number of choice sets                */
          seed=306,                 /* random number seed                   */
          options=relative          /* display relative D-efficiency        */
                  resrep,           /* detailed report on restrictions      */
          restrictions=res,         /* name of the restrictions macro       */
          resvars=price count,      /* vars used in defining restrictions   */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

Two new options are used in the `%ChoicEff` macro. The `resvars=` option lists the variables that are used to form restrictions. These variables must be numeric. Values of these variables are stored in a matrix called `x` for each choice set as it is being processed, and you can use these values to ensure that each choice set meets within choice set restrictions. Furthermore, while it is not illustrated in this example, the levels of the attributes for the entire design are stored in a matrix called `xmat`, and you can use these values as well as the values in `x` to impose restrictions across choice sets.

Restrictions are written with PROC IML statements embedded in a macro. You specify the name of the macro by using the `restrictions=` option. In the restrictions macro, you compute an IML scalar called `bad` that quantifies the badness of the design. In this case, since the restrictions are entirely within choice set, you can just quantify the badness of one choice set at at time by evaluating the values in `x`. The matrix `x` has four rows (since there are four alternatives) and two columns (since there are two variables listed in the `resvars=` option. The scalar `bad` is automatically initialized to zero by the `%ChoicEff` macro. In this example, `bad` is set to 1 if the `Price` variable is constant in a choice set, `bad` is incremented by 1 if the `Count` variable is constant in a choice set. The `if` statements use the Boolean syntax of PROC IML. We must use the following IML logical operators, which do not have all of the same syntactical alternatives as DATA step operators:

Specify	For	Do Not Specify
=	equals	EQ
∧ = or ¬ =	not equals	NE
<	less than	LT
<=	less than or equal to	LE
>	greater than	GT
>=	greater than or equal to	GE
&	and	AND
|	or	OR
∧ or ¬	not	NOT
`a <= b & b <= c`	range check	`a <= b <= c`

One other option is specified in this example, and that is `options=resrep`. It is a good idea to specify this option with restrictions until you are sure that your restrictions macro is correct. It provides detailed information about the swaps that the `%ChoicEff` macro performs and its success in imposing restrictions.

Part of the iteration history table, with a lot of `options=resrep` information and other information deleted, is as follows:

```
                              Cereal Bars

            Design   Iteration  D-Efficiency       D-Error
            ------------------------------------------------
                 1       0          11.33805 *     0.08820
at     1    1 swapped in     1      11.34298 bad =       0
at     1    2 swapped in     4      11.34298 bad =       0
at     1    2 swapped in     6      11.35439 bad =       0
at     1    3 swapped in     7      11.46812 bad =       0
at     1    4 swapped in     1      11.46812 bad =       0
.
.
.
at    16    4 swapped in     1      12.87489 bad =       0

at     1    3 swapped in     8      12.87590 bad =       0
.
.
.
at     1    4 swapped in     1      12.87590 bad =       0
at    15    4 swapped in     1      12.90851 bad =       0
                         2          12.90851 *     0.07747

            Design   Iteration  D-Efficiency       D-Error
            ------------------------------------------------
                 2       0          11.26776       0.08875
at     1    1 swapped in     1      11.46996 bad =       0
.
.
.
at     1    2 swapped in     5      11.46996 bad =       0
at    16    4 swapped in     1      12.87774 bad =       0
                         1          12.87774       0.07765
at     1    1 swapped in     1      12.87774 bad =       0
.
.
.
at    10    4 swapped in     1      12.89796 bad =       0
                         2          12.89796       0.07753
.
.
.
```

```
            Design   Iteration  D-Efficiency       D-Error
            ------------------------------------------------
                10       0          10.41643       0.09600
at     1    1 swapped in     1      10.64012 bad =       0
.
.
.
at    16    4 swapped in     1      12.85431 bad =       0
                         1          12.85431       0.07779
at     1    1 swapped in     1      12.85431 bad =       0
.
.
.
at     1    2 swapped in     4      12.85431 bad =       0
at    16    4 swapped in     1      12.91615 bad =       0
                        2           12.91615 *     0.07742
```

The first line shows the *D*-efficiency of the first initial design. The next row, and all other rows that begin with "`at`", are produced by `options=resrep` and report the set and alternative that is being swapped, the number of the candidate alternative that is being swapped in (in this case it is candidate number within candidate type), the *D*-efficiency, and the badness after the swap. In this problem, you can see that the `%ChoicEff` macro has no problem minimizing the badness to zero. That is not always the case depending on both the design requirements and how you pose the restrictions. Among rows that do not begin with "`at`", an asterisk is used to indicate places where *D*-efficiency is greater than any previously reported value.

The final results are as follows:

```
                        Cereal Bars

                      Final Results

                Design                 10
                Choice Sets            16
                Alternatives            4
                Parameters              7
                Maximum Parameters     48
                D-Efficiency      12.9161
                Relative D-Eff    80.7259
                D-Error            0.0774
                1 / Choice Sets    0.0625
```

n	Variable Name	Label	Variance	DF	Error
1	BrandBranolicious	Brand Branolicious	0.062500	1	0.25000
2	BrandBrantasia	Brand Brantasia	0.062500	1	0.25000
3	BrandBrantopia	Brand Brantopia	0.062500	1	0.25000
4	Price_2_89	Price $2.89	0.090909	1	0.30151
5	Price_2_99	Price $2.99	0.090909	1	0.30151
6	Price_3_09	Price $3.09	0.090909	1	0.30151
7	CountSix_Bars	Count Six Bars	0.090909	1	0.30151
				==	
				7	

It is instructive to look at the covariance matrix of the parameter estimates. The following steps display this matrix, which is automatically output to a SAS data set called `bestcov`:

```
proc format;
   value zer -1e-12 - 1e-12 = ' 0   ';
   run;

proc print data=bestcov label;
   id __label;
   label __label = '00'x;
   var BrandBranolicious -- CountSix_Bars;
   format _numeric_ zer5.2;
   run;
```

The results are as follows:

Cereal Bars

	Brand Branolicious	Brand Brantasia	Brand Brantopia	Price $2.89	Price $2.99	Price $3.09	Count Six Bars
Brand Branolicious	0.06	0	0	0	0	0	0
Brand Brantasia	0	0.06	0	0	0	0	0
Brand Brantopia	0	0	0.06	0	0	0	0
Price $2.89	0	0	0	0.09	0	0	0
Price $2.99	0	0	0	0	0.09	0	0
Price $3.09	0	0	0	0	0	0.09	0
Count Six Bars	0	0	0	0	0	0	0.09

With a diagonal covariance matrix, this design appears optimal for this problem. If this design is optimal, you might wonder why relative D-efficiency is not 100%. The variances for both price and count are larger than the variances for brand. A relative D-efficiency of 100% is based on a hypothetical optimal design where every factor can achieve the minimum variance like the brand attribute. Here, we have two- and three-level factors in a choice set with four alternatives, so perfect balance and hence perfect D-efficiency is not possible. Furthermore, we have a constant alternative, which places another

constraint on the maximum *D*-efficiency.

The following step prints the design:

```
proc print data=sasuser.cerealdes; id set; by set; var brand price count; run;
```

The design is as follows:

Cereal Bars

Set	Brand	Price	Count
1	Branolicious	$2.89	Six Bars
	Brantasia	$2.99	Eight Bars
	Brantopia	$3.09	Six Bars
	None		
2	Branolicious	$2.89	Six Bars
	Brantasia	$3.19	Six Bars
	Brantopia	$2.99	Eight Bars
3	Branolicious	$3.19	Eight Bars
	Brantasia	$2.99	Eight Bars
	Brantopia	$2.89	Six Bars
	None		
4	Branolicious	$3.19	Six Bars
	Brantasia	$3.09	Six Bars
	Brantopia	$2.99	Eight Bars
	None		
5	Branolicious	$3.09	Eight Bars
	Brantasia	$2.89	Eight Bars
	Brantopia	$2.99	Six Bars
	None		
6	Branolicious	$3.09	Six Bars
	Brantasia	$2.89	Eight Bars
	Brantopia	$3.19	Eight Bars
	None		
7	Branolicious	$3.09	Eight Bars
	Brantasia	$3.19	Six Bars
	Brantopia	$2.99	Six Bars
	None		
8	Branolicious	$2.99	Eight Bars
	Brantasia	$3.09	Six Bars
	Brantopia	$2.89	Eight Bars
	None		

9	Branolicious	$3.19	Eight Bars
	Brantasia	$3.09	Six Bars
	Brantopia	$2.89	Six Bars
	None		
10	Branolicious	$2.89	Six Bars
	Brantasia	$3.19	Eight Bars
	Brantopia	$3.09	Eight Bars
	None		
11	Branolicious	$2.99	Six Bars
	Brantasia	$3.09	Eight Bars
	Brantopia	$3.19	Eight Bars
	None		
12	Branolicious	$3.19	Six Bars
	Brantasia	$2.89	Eight Bars
	Brantopia	$3.09	Six Bars
	None		
13	Branolicious	$2.99	Six Bars
	Brantasia	$2.89	Eight Bars
	Brantopia	$3.19	Eight Bars
	None		
14	Branolicious	$3.09	Eight Bars
	Brantasia	$2.99	Six Bars
	Brantopia	$2.89	Six Bars
	None		
15	Branolicious	$2.89	Eight Bars
	Brantasia	$2.99	Six Bars
	Brantopia	$3.19	Six Bars
	None		
16	Branolicious	$2.99	Eight Bars
	Brantasia	$3.19	Six Bars
	Brantopia	$3.09	Eight Bars
	None		

You can see that there are no constant attributes.

There is one more test that should be run before a design is used. In the following step, the `%MktDups` macro checks the design to see if any choice sets are duplicates of any other choice sets:

```
%mktdups(branded,                          /* a design with brands          */
         data=sasuser.cerealdes,           /* the input design to evaluate  */
         factors=brand price count,        /* factors in the design         */
         nalts=4)                          /* number of alternatives        */
```

The results are as follows:

```
Design:          Branded
Factors:         brand price count
                 Brand
                 Count Price
Duplicate Sets:  0
```

You can see that there are no duplicate choice sets.

The questionnaire is designed and data are collected and entered, as they were in the previous example. The data are entered in the following step:

```
title2 'Read Data';

data results;
   input Subject (r1-r16) (1.);
   datalines;
  1 2133321333322311
  2 3133221131222211
  3 2133321131122311
  4 1113221131322311
  5 2133321133322311
  6 2133221131222311
  7 2133221132322311
  8 2133221131322311
  9 1123221112222311
 10 1133123133222211
 11 3133221331322111
 12 1123221113223311
 13 2231221132123321
 14 2133221331222321
 15 2133321131122111
 16 2123221331222321
 17 2123223331332311
 18 2133223311132121
 19 1133221131122311
 20 2133221131222411
 21 2133221131222321
 22 2123221111122321
 23 2133223131222111
 24 1133321111332111
 25 3133321431222131
 26 2133221143223113
 27 2123131113322311
 28 3322221131322311
 29 2123221133322311
 30 2133221131222311
```

```
 31 1121221111242311
 32 2133221133122311
 33 2133321133222311
 34 2123221331322411
 35 1132221111322311
 36 1133323131322121
 37 1333221131322311
 38 1123321311222311
 39 1123223131222111
 40 2123221331222313
;
```

The %MktMerge macro merges the data and the design and creates the dependent variable as follows:

```
title2 'Merge Data and Design';

%mktmerge(design=sasuser.cerealdes, /* input design                          */
          data=results,             /* input data set                       */
          out=res2,                 /* output data set with design and data */
          nsets=16,                 /* number of choice sets                */
          nalts=4,                  /* number of alternatives               */
          setvars=r1-r16)           /* variables with the chosen alt nums   */
```

The coding and analysis are the same as in the previous example and are as follows:

```
title2 'Code the Independent Variables';

proc transreg design norestoremissing data=res2;
   model class(brand price count);
   id subject set c;
   output out=coded(drop=_type_ _name_ intercept) lprefix=0;
   run;

%phchoice(on)                       /* customize PHREG for a choice model   */

title2 'Multinomial Logit Discrete Choice Model';

proc phreg data=coded brief;
   model c*c(2) = &_trgind / ties=breslow;
   strata subject set;
   run;

%phchoice(off)                      /* restore PHREG to a survival PROC     */
```

The parameter estimate table is as follows:

Multinomial Logit Parameter Estimates

	DF	Parameter Estimate	Standard Error	Chi-Square	Pr > ChiSq
Branolicious	1	2.88724	0.48197	35.8866	<.0001
Brantasia	1	2.43226	0.49040	24.5995	<.0001
Brantopia	1	2.41739	0.48561	24.7815	<.0001
$2.89	1	2.91685	0.19962	213.5135	<.0001
$2.99	1	1.73771	0.19894	76.2957	<.0001
$3.09	1	0.69198	0.21241	10.6125	0.0011
Six Bars	1	-1.28398	0.12586	104.0674	<.0001

These results are similar to what we saw in the previous example.

Example 5, Searching a Candidate Set of Choice Sets

In this example, we create a design for the same study as the previous example. The product line is the same three fictitious breakfast bars with the same attributes as before. However, this time, rather than making a design from candidate alternatives, we create a design from candidate choice sets. Usually, you would not use this approach, but you could use it when there are restrictions that prevent certain alternatives from appearing with other alternatives. You provide candidate choice sets to the `%ChoiceEff` macro that conform to the restrictions. The candidate alternative approach in the previous example will usually be superior to the candidate choice set approach in this example since the former can consider more possible designs using smaller candidate sets. However, for very small problems such as this one, the two approaches should perform similarly.

We will begin like we began the first example, by making a design that is *D*-efficient for a linear model and then converting it to a choice design. However, this time, that design will form a candidate set and not the final design. We can use the `%MktRuns` macro as follows to suggest the number of choice sets in the candidate set. The input to the macro is the number of levels of all of the factors (that is, all of the attributes of all of the alternatives). The following step runs the macro:

```
title 'Cereal Bars';

%mktruns(4 2  4 2  4 2)     /* factor level list for all attrs and alts     */
```

The results are as follows:

Cereal Bars

Design Summary

Number of Levels	Frequency
2	3
4	3

Cereal Bars

Saturated = 13
Full Factorial = 512

Some Reasonable Design Sizes	Violations	Cannot Be Divided By
16 *	0	
32 *	0	
24	3	16
20	12	8 16
28	12	8 16
14	18	4 8 16
18	18	4 8 16
22	18	4 8 16
26	18	4 8 16
30	18	4 8 16
13 S	21	2 4 8 16

* - 100% Efficient design can be made with the MktEx macro.
S - Saturated Design - The smallest design that can be made.
Note that the saturated design is not one of the
recommended designs for this problem. It is shown
to provide some context for the recommended sizes.

Cereal Bars

n	Design		Reference
16	2 ** 6	4 ** 3	Fractional-Factorial
16	2 ** 3	4 ** 4	Fractional-Factorial
32	2 ** 22	4 ** 3	Fractional-Factorial
32	2 ** 19	4 ** 4	Fractional-Factorial
32	2 ** 16	4 ** 5	Fractional-Factorial
32	2 ** 15	4 ** 3 8 ** 1	Fractional-Factorial
32	2 ** 13	4 ** 6	Fractional-Factorial
32	2 ** 12	4 ** 4 8 ** 1	Fractional-Factorial
32	2 ** 10	4 ** 7	Fractional-Factorial
32	2 ** 9	4 ** 5 8 ** 1	Fractional-Factorial
32	2 ** 7	4 ** 8	Fractional-Factorial
32	2 ** 6	4 ** 6 8 ** 1	Fractional-Factorial
32	2 ** 4	4 ** 9	Fractional-Factorial
32	2 ** 3	4 ** 7 8 ** 1	Fractional-Factorial

Since the goal is to make a candidate set of choice sets not the final design, we can pick a much larger number of choice sets than we used in the first example. The `%MktRuns` macro lists orthogonal arrays that the `%MktEx` macro knows how to make. In some cases, these might provide good candidate designs. However, in this case, the full-factorial design, at 512 runs, is not too large for this problem. We use the `%MktEx` macro as follows to create the full-factorial design:

```
%mktex(4 2  4 2  4 2,       /* factor level list for all attrs and alts     */
       n=512,               /* number of candidate choice sets              */
       out=full,            /* output data set with full factorial          */
       seed=17)             /* random number seed                           */
```

The results are as follows:

Cereal Bars

Algorithm Search History

Design	Row,Col	Current D-Efficiency	Best D-Efficiency	Notes
1	Start	100.0000	100.0000	Tab
1	End	100.0000		

Cereal Bars

The OPTEX Procedure

Class Level Information

Class	Levels	Values
x1	4	1 2 3 4
x2	2	1 2
x3	4	1 2 3 4
x4	2	1 2
x5	4	1 2 3 4
x6	2	1 2

Cereal Bars

Design Number	D-Efficiency	A-Efficiency	G-Efficiency	Average Prediction Standard Error
1	100.0000	100.0000	100.0000	0.1593

The full-factorial design has 100% *D*-efficiency.

Say that you want to restrict the design to avoid choice sets where attributes are constant. That is, you want to select just the choice sets where neither price nor count is constant within a choice set. The following step subsets the design by deleting choice sets with constant alternatives:

```
data design;
   set full;
   if x1 eq x3 and x3 eq x5 then delete;  /* delete constant price          */
   if x2 eq x4 and x4 eq x6 then delete;  /* delete constant count          */
   run;
```

Note that it is easier to impose restrictions across alternatives at this stage (when the design is arranged with one row per choice set) rather than later (when the design is arranged with one row per alternative per choice set). Now, restrictions can be imposed by examining the variables in one row at a time rather than looking across multiple rows. After running this step, the candidate design has 360 choice sets. You can also specify restrictions directly in the `%MktEx` macro. This is illustrated extensively throughout this book, but it is beyond the scope of this chapter. For this problem, the DATA step approach is superior, since the goal is to eliminate unsuitable candidates from a full-factorial design. Usually, the restrictions are not this simple.

Like the linear arrangement example, we can use the `%MktKey` macro and the `%MktRoll` macro to create the key to converting the linear arrangement into a choice design.

The following steps create and display the key data set:

```
%mktkey(3 2)         /* x1-x6 (since 3*2=6) in 3 rows and 2 columns        */

data key;
   input Brand $ 1-12 Price $ Count $;
   datalines;
Branolicious    x1         x2
Brantopia       x3         x4
Brantasia       x5         x6
None            .          .
;

proc print; run;
```

The results are as follows:

Cereal Bars

Obs	Brand	Price	Count
1	Branolicious	x1	x2
2	Brantopia	x3	x4
3	Brantasia	x5	x6
4	None		

The following steps create the candidate set of choice sets from the linear arrangement using the rules specified in the `key=key` data set, create and set the formats, and display the first four candidate choice sets:

```
%mktroll(design=design,      /* input linear candidate design              */
         key=key,            /* rules for making choice design             */
         alt=brand,          /* brand or alternative label var             */
         out=cand)           /* output candidate choice design             */

proc format;
   value price 1 = $2.89 2 = $2.99 3 = $3.09 4 = $3.19 . = ' ';
   value count 1 = 'Six Bars' 2 = 'Eight Bars'  . = ' ';
   run;

data cand;
   set cand;
   format price price. count count.;
   run;

proc print data=cand(obs=16); id set; by set; run;
```

The first four choice sets are as follows:

Cereal Bars

Set	Brand	Price	Count
1	Branolicious	$2.89	Six Bars
	Brantopia	$2.89	Six Bars
	Brantasia	$2.99	Eight Bars
	None		
2	Branolicious	$2.89	Six Bars
	Brantopia	$2.89	Six Bars
	Brantasia	$3.09	Eight Bars
	None		
3	Branolicious	$2.89	Six Bars
	Brantopia	$2.89	Six Bars
	Brantasia	$3.19	Eight Bars
	None		
4	Branolicious	$2.89	Six Bars
	Brantopia	$2.89	Eight Bars
	Brantasia	$2.99	Six Bars
	None		

You can see that in these choice sets that all of the attributes vary. The full candidate data set contains 1440 observations, from the 360 candidates each with 4 alternatives. Next, like the example with a candidate set of alternatives, the `%ChoicEff` macro is run to create a design that is efficient under the null hypothesis $\boldsymbol{\beta} = \mathbf{0}$ and with a main-effects model specification. However, this time there are no flag variables or `flags=` option to indicate which alternatives go where, since the candidate set contains choice sets not alternatives. Instead, the `nalts=` option is used to specify the number of alternatives in each choice set. The `%ChoicEff` macro assumes that each block of `nalts=4` alternatives forms one candidate choice set. Again, a standardized orthogonal contrast coding is used. By default, the best design found is stored in the `outbest=best` SAS data set. The following step creates the design:

```
%choiceff(data=cand,                    /* candidate set of choice sets   */
                                        /* model with stdz orthog coding  */
          model=class(brand price count / sta) /
                cprefix=0               /* lpr=0 labels from just levels  */
                lprefix=0,              /* cpr=0 names from just levels   */
          nsets=16,                     /* number of choice sets          */
          seed=145,                     /* random number seed             */
          nalts=4,                      /* number of alternatives         */
          options=relative,             /* display relative D-efficiency  */
          beta=zero)                    /* assumed beta vector            */
```

The results are as follows:

Cereal Bars

n	Name	Beta	Label
1	Branolicious	0	Branolicious
2	Brantasia	0	Brantasia
3	Brantopia	0	Brantopia
4	_2_89	0	$2.89
5	_2_99	0	$2.99
6	_3_09	0	$3.09
7	Six_Bars	0	Six Bars

Design	Iteration	D-Efficiency	D-Error
1	0	11.19099 *	0.08936
	1	12.91424 *	0.07743
	2	12.91424	0.07743

Design	Iteration	D-Efficiency	D-Error
2	0	11.50095	0.08695
	1	12.88929	0.07758
	2	12.91042	0.07746

Cereal Bars

Final Results

Design	1
Choice Sets	16
Alternatives	4
Parameters	7
Maximum Parameters	48
D-Efficiency	12.9142
Relative D-Eff	80.7140
D-Error	0.0774
1 / Choice Sets	0.0625

Cereal Bars

n	Variable Name	Label	Variance	DF	Standard Error
1	Branolicious	Branolicious	0.062500	1	0.25000
2	Brantasia	Brantasia	0.062500	1	0.25000
3	Brantopia	Brantopia	0.062500	1	0.25000
4	_2_89	$2.89	0.090933	1	0.30155
5	_2_99	$2.99	0.090917	1	0.30152
6	_3_09	$3.09	0.090972	1	0.30162
7	Six_Bars	Six Bars	0.091003	1	0.30167
				==	
				7	

In the iteration history, an asterisk is used to indicate places where *D*-efficiency is greater than any previously reported value.

We see three parameters for brand (4 alternatives including None minus 1), three for price $(4-1)$, one for count $(2-1)$. All are estimable, and all have reasonable standard errors. The variances for the brand effects are the inverse of the number of choice sets ($1/16 = 0.06250$). The other variances are bigger. These results look good.

One thing to note is that the *D*-efficiency is 12.9142, which is the same as we saw when we used a candidate set of alternatives. This might be surprising for two reasons. First, this design is restricted but the previous design was not. Second, this design was created from a candidate set of choice sets rather than a candidate set of alternatives. The latter approach usually provides more freedom than the former and with a smaller candidate set. If you look at the design from the previous example on page 171, you will see that it conforms to our restrictions even though they were not formally imposed. Designs without constant attributes within choice sets tend to be more efficient than designs with constant attributes. Hence, in this case, the restrictions did not have any effect. However, paring down the candidate set by eliminating less-than-optimal candidates made it easier for the `%ChoicEff` macro to find a good design. Still, even the full-factorial at 512 choice sets is small enough that the `%ChoicEff` macro has no trouble searching it, particularly with as few attributes and alternatives as are in this problem.

The following step displays the first four choice sets:

```
proc print data=best(obs=16);
   by notsorted set;
   id set;
   var brand -- count;
   run;
```

The first four choice sets are as follows:

Cereal Bars

Set	Brand	Price	Count
140	Branolicious	$2.99	Eight Bars
	Brantopia	$2.89	Six Bars
	Brantasia	$3.09	Six Bars
	None		
115	Branolicious	$2.99	Six Bars
	Brantopia	$3.09	Six Bars
	Brantasia	$3.19	Eight Bars
	None		
19	Branolicious	$2.89	Six Bars
	Brantopia	$2.99	Eight Bars
	Brantasia	$3.09	Eight Bars
	None		
164	Branolicious	$2.99	Eight Bars
	Brantopia	$3.09	Six Bars
	Brantasia	$3.19	Eight Bars
	None		

With this approach, the choice set number refers to the original choice set numbers in the candidate set. Usually, they are in a random order. You can assign consecutive choice set numbers as follows:

```
data sasuser.choice;
   set best(keep=brand price count);
   retain Set 1;
   output;
   if brand = 'None' then set + 1;
   run;

proc print data=sasuser.choice(obs=16);
   by set;
   id set;
   var brand -- count;
   run;
```

The DATA step also stores the final design in a permanent SAS data set so that it is around at analysis time.

The results are as follows:

Cereal Bars

Set	Brand	Price	Count
1	Branolicious	$2.99	Eight Bars
	Brantopia	$2.89	Six Bars
	Brantasia	$3.09	Six Bars
	None		
2	Branolicious	$2.99	Six Bars
	Brantopia	$3.09	Six Bars
	Brantasia	$3.19	Eight Bars
	None		
3	Branolicious	$2.89	Six Bars
	Brantopia	$2.99	Eight Bars
	Brantasia	$3.09	Eight Bars
	None		
4	Branolicious	$2.99	Eight Bars
	Brantopia	$3.09	Six Bars
	Brantasia	$3.19	Eight Bars
	None		

It is instructive to look at the covariance matrix of the parameter estimates. The following steps display this matrix, which is automatically output to a SAS data set called `bestcov`:

```
proc format;
   value zer -1e-12 - 1e-12 = ' 0   ';
   run;

proc print data=bestcov label;
   id __label;
   label __label = '00'x;
   var Branolicious -- Six_Bars;
   format _numeric_ zer5.2;
   run;
```

The results are as follows:

Cereal Bars

	Branolicious	Brantasia	Brantopia	$2.89	$2.99	$3.09	Six Bars
Branolicious	0.06	0	0	0	0	0	0
Brantasia	0	0.06	0	0	0	0	0
Brantopia	0	0	0.06	0	0	0	0
$2.89	0	0	0	0.09	-0.00	-0.00	0.00
$2.99	0	0	0	-0.00	0.09	0.00	-0.00
$3.09	0	0	0	-0.00	0.00	0.09	-0.00
Six Bars	0	0	0	0.00	-0.00	-0.00	0.09

Like before, there are some nonzero covariances between the price and count attributes.

There is one more test that should be run before a design is used. In the following step, the %MktDups macro checks the design to see if any choice sets are duplicates of any other choice sets:

```
%mktdups(branded,                          /* a design with brands           */
         data=sasuser.choice,              /* the input design to evaluate   */
         factors=brand price count,        /* factors in the design          */
         nalts=4)                          /* number of alternatives         */
```

The results are as follows:

```
Design:          Branded
Factors:         brand price count
                 Brand
                 Count Price
Duplicate Sets:  0
```

There are no duplicate choice sets. Collecting and analyzing data is no different than was illustrated previously, so it will not be shown here.

Example 6, A Generic Choice Design

In this example, we create a design for a purely generic experiment—an experiment involving no brands, just bundles of attributes. The design is efficient for a choice model under the null hypothesis $\boldsymbol{\beta} = \mathbf{0}$ and a main-effects model. The design is constructed from a candidate set of alternatives using the %ChoicEff macro. Imagine that the manufacturer is interested in better understanding choices for cereal bars independent of brand. Like before, they are interested in price and count, but now they are also interested in the number of calories and whether consumers are influenced by claims such as "naturally cholesterol free."

Design Factors and Levels

Factor Name	Attribute	Number of Levels	Levels
x1	Price	4	$2.89, $2.99, $3.09, $3.19
x2	Count	2	Six, Eight
x3	Calories	3	90, 110, 130
x4	Cholesterol	2	Cholesterol Free, No Claim

For this experiment, we will use the `%MktEx` macro to make a candidate set of alternatives and then use the `%ChoicEff` macro to create a choice design from the candidate alternatives. First, we can use the `%MktRuns` macro to suggest sizes for the candidate set. We have four factors since there are four attributes with 4, 2, 3, and 2 levels. We use the `%MktRuns` macro as follows:

```
%mktruns(4 2 3 2)                  /* factor level list for one alternative */
```

The results are as follows:

Cereal Bars

Design Summary

Number of Levels	Frequency
2	2
3	1
4	1

Cereal Bars

Saturated = 8
Full Factorial = 48

Some Reasonable Design Sizes		Violations	Cannot Be Divided By
24	*	0	
48	*	0	
12		2	8
36		2	8
8	S	4	3 6 12
16		4	3 6 12
32		4	3 6 12
40		4	3 6 12
18		5	4 8 12
30		5	4 8 12

* - 100% Efficient design can be made with the MktEx macro.
S - Saturated Design - The smallest design that can be made.

Cereal Bars

n	Design	Reference
24	2 ** 13 3 ** 1 4 ** 1	Orthogonal Array
48	2 ** 37 3 ** 1 4 ** 1	Orthogonal Array
48	2 ** 34 3 ** 1 4 ** 2	Orthogonal Array
48	2 ** 31 3 ** 1 4 ** 3	Orthogonal Array
48	2 ** 28 3 ** 1 4 ** 4	Orthogonal Array
48	2 ** 25 3 ** 1 4 ** 5	Orthogonal Array
48	2 ** 22 3 ** 1 4 ** 6	Orthogonal Array
48	2 ** 19 3 ** 1 4 ** 7	Orthogonal Array
48	2 ** 16 3 ** 1 4 ** 8	Orthogonal Array
48	2 ** 13 3 ** 1 4 ** 9	Orthogonal Array
48	2 ** 10 3 ** 1 4 ** 10	Orthogonal Array
48	2 ** 7 3 ** 1 4 ** 11	Orthogonal Array
48	2 ** 4 3 ** 1 4 ** 12	Orthogonal Array

This approach results in much smaller designs compared to the linear arrangement approach since you are only creating factors for one alternative at a time instead of factors for all of the attributes of all of the alternatives. The `%MktRuns` macro lists orthogonal arrays that the `%MktEx` macro knows how to make. In some cases, these might provide good candidate designs. However, in this case, 48 runs is sufficiently small that we can use the full-factorial design as a candidate set. The `%MktEx` macro can be used as follows to make the candidate alternatives:

```
%mktex(4 2 3 2,            /* factor level list for one alternative        */
       n=48)               /* number of candidate alternatives             */
```

The results are as follows:

Cereal Bars

Algorithm Search History

Design	Row,Col	Current D-Efficiency	Best D-Efficiency	Notes
1	Start	100.0000	100.0000	Tab
1	End	100.0000		

Cereal Bars

The OPTEX Procedure

Class Level Information

Class	Levels	Values
x1	4	1 2 3 4
x2	2	1 2
x3	3	1 2 3
x4	2	1 2

Cereal Bars

Design Number	D-Efficiency	A-Efficiency	G-Efficiency	Average Prediction Standard Error
1	100.0000	100.0000	100.0000	0.4082

The full-factorial design has 100% *D*-efficiency. This design, like all designs that come out of the `%MktEx` macro, has factor names `x1`, `x2`, and so on, and levels of 1, 2, and so on. The next steps do several things. The first step specifies formats for the levels of the attributes. The `%MktLab` macro step assigns meaningful variable names and the formats. In this design, we will have three alternatives, so the `%MktLab` macro creates three new variables, `f1-f3` with the `int=` or intercept option. The values of these three variables are all ones. They are used as flags to indicate that every candidate can appear in every alternative. The following steps create the candidate design:

```
proc format;
   value price 1 = $2.89 2 = $2.99 3 = $3.09 4 = $3.19;
   value count 1 = 'Six Bars' 2 = 'Eight Bars';
   value cal   1 = '90 Calories' 2 = '110 Calories' 3 = '130 Calories';
   value chol  1 = 'Cholesterol Free' 2 = 'No Claim';
   run;

%mktlab(data=design,           /* input data set                           */
        vars=Price Count Calories Cholesterol, /* new attribute names      */
        int=f1-f3,             /* create 3 columns of 1's in f1-f3         */
        out=final,             /* output design                            */
                               /* add a format statement for the attributes */
        stmts=format price price. count count. calories cal. cholesterol chol.)
```

The following step displays the candidates:

```
proc print; run;
```

The results are as follows:

Cereal Bars

Obs	f1	f2	f3	Price	Count	Calories	Cholesterol
1	1	1	1	$2.89	Six Bars	90 Calories	Cholesterol Free
2	1	1	1	$2.89	Six Bars	90 Calories	No Claim
3	1	1	1	$2.89	Six Bars	110 Calories	Cholesterol Free
4	1	1	1	$2.89	Six Bars	110 Calories	No Claim
5	1	1	1	$2.89	Six Bars	130 Calories	Cholesterol Free
6	1	1	1	$2.89	Six Bars	130 Calories	No Claim
7	1	1	1	$2.89	Eight Bars	90 Calories	Cholesterol Free
8	1	1	1	$2.89	Eight Bars	90 Calories	No Claim
9	1	1	1	$2.89	Eight Bars	110 Calories	Cholesterol Free
10	1	1	1	$2.89	Eight Bars	110 Calories	No Claim
11	1	1	1	$2.89	Eight Bars	130 Calories	Cholesterol Free
12	1	1	1	$2.89	Eight Bars	130 Calories	No Claim
13	1	1	1	$2.99	Six Bars	90 Calories	Cholesterol Free
14	1	1	1	$2.99	Six Bars	90 Calories	No Claim
15	1	1	1	$2.99	Six Bars	110 Calories	Cholesterol Free
16	1	1	1	$2.99	Six Bars	110 Calories	No Claim
17	1	1	1	$2.99	Six Bars	130 Calories	Cholesterol Free
18	1	1	1	$2.99	Six Bars	130 Calories	No Claim
19	1	1	1	$2.99	Eight Bars	90 Calories	Cholesterol Free
20	1	1	1	$2.99	Eight Bars	90 Calories	No Claim
21	1	1	1	$2.99	Eight Bars	110 Calories	Cholesterol Free
22	1	1	1	$2.99	Eight Bars	110 Calories	No Claim
23	1	1	1	$2.99	Eight Bars	130 Calories	Cholesterol Free
24	1	1	1	$2.99	Eight Bars	130 Calories	No Claim
25	1	1	1	$3.09	Six Bars	90 Calories	Cholesterol Free
26	1	1	1	$3.09	Six Bars	90 Calories	No Claim
27	1	1	1	$3.09	Six Bars	110 Calories	Cholesterol Free
28	1	1	1	$3.09	Six Bars	110 Calories	No Claim
29	1	1	1	$3.09	Six Bars	130 Calories	Cholesterol Free
30	1	1	1	$3.09	Six Bars	130 Calories	No Claim
31	1	1	1	$3.09	Eight Bars	90 Calories	Cholesterol Free
32	1	1	1	$3.09	Eight Bars	90 Calories	No Claim
33	1	1	1	$3.09	Eight Bars	110 Calories	Cholesterol Free
34	1	1	1	$3.09	Eight Bars	110 Calories	No Claim
35	1	1	1	$3.09	Eight Bars	130 Calories	Cholesterol Free
36	1	1	1	$3.09	Eight Bars	130 Calories	No Claim
37	1	1	1	$3.19	Six Bars	90 Calories	Cholesterol Free
38	1	1	1	$3.19	Six Bars	90 Calories	No Claim
39	1	1	1	$3.19	Six Bars	110 Calories	Cholesterol Free
40	1	1	1	$3.19	Six Bars	110 Calories	No Claim

41	1	1	1	$3.19	Six Bars	130 Calories	Cholesterol Free
42	1	1	1	$3.19	Six Bars	130 Calories	No Claim
43	1	1	1	$3.19	Eight Bars	90 Calories	Cholesterol Free
44	1	1	1	$3.19	Eight Bars	90 Calories	No Claim
45	1	1	1	$3.19	Eight Bars	110 Calories	Cholesterol Free
46	1	1	1	$3.19	Eight Bars	110 Calories	No Claim
47	1	1	1	$3.19	Eight Bars	130 Calories	Cholesterol Free
48	1	1	1	$3.19	Eight Bars	130 Calories	No Claim

The candidate design is structured so that every candidate alternative can appear anywhere in the final design. You can see this by looking at the flag variables, `f1-f3`. When `f1` = 1, then the candidate can be used in the first alternative; when `f2` = 1, then the candidate can be used in the second alternative; and when `f3` = 1, then the candidate can be used in the third alternative. This candidate design is then input to the `%ChoicEff` macro as follows:

```
%choiceff(data=final,                  /* candidate set of alternatives      */
          bestout=sasuser.cerealdes,   /* choice design permanently stored   */
                                       /* model with stdz orthog coding      */
          model=class(price count calories cholesterol / sta) /
                cprefix=0              /* lpr=0 labels from just levels      */
                lprefix=0,             /* cpr=0 names from just levels       */
          nsets=9,                     /* number of choice sets to make      */
          seed=145,                    /* random number seed                 */
          flags=f1-f3,                 /* flag which alt can go where, 3 alts*/
          options=relative,            /* display relative D-efficiency      */
          beta=zero)                   /* assumed beta vector                */
```

The `%ChoicEff` step creates a generic choice design with 9 choice sets and 3 alternatives. The model specification specifies a main-effects model and the standardized orthogonal contrast coding. The `cprefix=0` option is specified so that variable names are constructed just from the attribute levels using zero characters of the attribute (or class) variable names. Similarly, the `lprefix=0` option is specified so that variable labels are constructed just from the attribute levels using zero characters of the attribute (or class) variable names or labels. The results are as follows:

Cereal Bars

n	Name	Beta	Label
1	_2_89	0	$2.89
2	_2_99	0	$2.99
3	_3_09	0	$3.09
4	Six_Bars	0	Six Bars
5	_90_Calories	0	90 Calories
6	_110_Calories	0	110 Calories
7	Cholesterol_Free	0	Cholesterol Free

Design	Iteration	D-Efficiency	D-Error
1	0	0	.
	1	7.79175 *	0.12834
	2	8.15348 *	0.12265
	3	8.17776 *	0.12228

Design	Iteration	D-Efficiency	D-Error
2	0	4.44502	0.22497
	1	8.03861	0.12440
	2	8.18335 *	0.12220
	3	8.20929 *	0.12181

Cereal Bars

Final Results

Design	2
Choice Sets	9
Alternatives	3
Parameters	7
Maximum Parameters	18
D-Efficiency	8.2093
Relative D-Eff	91.2143
D-Error	0.1218
1 / Choice Sets	0.1111

Cereal Bars

n	Variable Name	Label	Variance	DF	Standard Error
1	_2_89	$2.89	0.13420	1	0.36633
2	_2_99	$2.99	0.12509	1	0.35368
3	_3_09	$3.09	0.12331	1	0.35115
4	Six_Bars	Six Bars	0.12539	1	0.35410
5	_90_Calories	90 Calories	0.11310	1	0.33630
6	_110_Calories	110 Calories	0.11310	1	0.33630
7	Cholesterol_Free	Cholesterol Free	0.12622	1	0.35528
				==	
				7	

The first table lists the parameters and their assumed values (all zero). The next two tables show the iteration history. The results for iteration 0 are the results for the initial random selection of alternatives. It is often the case that the *D*-efficiency for the initial random design is zero, but with iteration, the efficiency increases. The final two tables provide information about the design specification, the final *D*-efficiency, and the variances and standard errors. We see three parameters for brand (4 alternatives including None minus 1), three for price $(4-1)$, one for count $(2-1)$. All are estimable, and

all have reasonable standard errors. The best you can hope for with three level factors is a variance of $1/9 \approx 0.11111$. This design looks good. The standard errors are all similar and close to the minimum, and all of the parameters can be estimated. The following step displays the design:

```
proc print data=sasuser.cerealdes;
   var price -- cholesterol;
   id set; by set;
   run;
```

The results are as follows:

Set	Price	Count	Calories	Cholesterol
1	$3.19	Eight Bars	90 Calories	No Claim
	$2.89	Six Bars	110 Calories	No Claim
	$3.09	Six Bars	130 Calories	Cholesterol Free
2	$3.19	Eight Bars	130 Calories	No Claim
	$2.99	Six Bars	90 Calories	Cholesterol Free
	$2.89	Six Bars	110 Calories	Cholesterol Free
3	$3.19	Six Bars	110 Calories	Cholesterol Free
	$2.89	Eight Bars	130 Calories	No Claim
	$3.09	Eight Bars	90 Calories	Cholesterol Free
4	$3.19	Eight Bars	110 Calories	Cholesterol Free
	$2.99	Six Bars	130 Calories	No Claim
	$3.09	Six Bars	90 Calories	Cholesterol Free
5	$2.99	Eight Bars	130 Calories	Cholesterol Free
	$2.89	Six Bars	90 Calories	Cholesterol Free
	$3.09	Eight Bars	110 Calories	No Claim
6	$2.89	Eight Bars	130 Calories	Cholesterol Free
	$2.99	Eight Bars	110 Calories	Cholesterol Free
	$3.19	Six Bars	90 Calories	No Claim
7	$3.09	Eight Bars	130 Calories	Cholesterol Free
	$2.89	Eight Bars	90 Calories	No Claim
	$2.99	Six Bars	110 Calories	No Claim
8	$3.19	Six Bars	130 Calories	Cholesterol Free
	$2.99	Eight Bars	90 Calories	No Claim
	$3.09	Eight Bars	110 Calories	No Claim
9	$2.99	Eight Bars	90 Calories	Cholesterol Free
	$3.09	Six Bars	130 Calories	No Claim
	$2.89	Eight Bars	110 Calories	Cholesterol Free

The following step is not necessary, but it is instructive to look at the covariance matrix of the parameter estimates. The following steps display this matrix, which is automatically output to a SAS data set called `bestcov`:

```
proc format;
   value zer -1e-12 - 1e-12 = ' 0   ';
   run;

proc print data=bestcov label;
   id __label;
   label __label = '00'x;
   var _2_89 -- Cholesterol_Free;
   format _numeric_ zer5.2;
   run;
```

The results are as follows:

Cereal Bars

	$2.89	$2.99	$3.09	Six Bars	90 Calories	110 Calories	Cholesterol Free
$2.89	0.13	0.01	0.01	0.00	0.00	-0.01	-0.01
$2.99	0.01	0.13	0.00	0.00	-0.01	0.01	-0.00
$3.09	0.01	0.00	0.12	0.00	0.01	0.01	-0.01
Six Bars	0.00	0.00	0.00	0.13	0.00	0.00	-0.01
90 Calories	0.00	-0.01	0.01	0.00	0.11	0.00	-0.00
110 Calories	-0.01	0.01	0.01	0.00	0.00	0.11	-0.00
Cholesterol Free	-0.01	-0.00	-0.01	-0.01	-0.00	-0.00	0.13

There are some nonzero covariances between the price and count attributes.

There is one more test that should be run before a design is used. In the following step, the `%MktDups` macro checks the design to see if any choice sets are duplicates of any other choice sets:

```
%mktdups(generic,                   /* generic design (no brands)          */
         data=sasuser.cerealdes,    /* the input design to evaluate        */
                                    /* factors in the design               */
         factors=price count calories cholesterol,
         nalts=3)                   /* number of alternatives              */
```

The results are as follows:

```
Design:          Generic
Factors:         price count calories cholesterol
                 Calories Cholesterol Count Price
Sets w Dup Alts: 0
Duplicate Sets:  0
```

The results contain a line that we did not see with branded designs. With generic designs, the macro also checks for duplicate alternatives within choice sets. In this case, there are no duplicate choice sets and no duplicate alternatives within choice sets.

Collecting and analyzing data is no different than was illustrated previously, so it will not be shown here.

Example 7, A Partial-Profile Choice Experiment

This example is like the previous example in the sense that we will create a design for a generic experiment—an experiment involving no brands, just bundles of attributes. We will create a design that is optimal for a main-effects choice model under the null hypothesis $\boldsymbol{\beta} = \mathbf{0}$. We will continue to work with cereal bars. This time, the manufacturer is interested in knowing people's preferences for ingredients of high-end bars. The goal is to construct an experiment with 13 attributes, all described by the presence or absence of the following ingredients:

Almonds
Apple
Banana Chips
Brown Sugar
Cashews
Chocolate
Coconut
Cranberries
Hazel Nuts
Peanuts
Pecans
Raisins
Walnuts

Since it might be difficult for people to compare that many ingredients simultaneously, we will only vary subsets of the attributes at any one time. This kind of experiment is called a partial-profile choice experiment (Chrzan and Elrod 1995), and the experimental design is made from a balanced incomplete block design (BIBD) and an orthogonal array. See page 115 for more about BIBDs. See page 58 for more about orthogonal arrays. We need to construct a design where our $t = 13$ attributes can be shown in b blocks of choice sets where k attributes vary in each block. The next task is to determine values for b and k. We can use the `%MktBSize` macro for this. It tells us sizes in which a BIBD or unbalanced block design might be possible. The following step sets the number of attributes to 13 and asks for sizes in the default range of 2 to 500 blocks of choice sets with a size in the range 3 to 8:

```
title 'Cereal Bars';

%mktbsize(nattrs=13,                /* 13 attributes                        */
          setsize=3 to 8,           /* try set sizes in range 3 to 8        */
          options=ubd)              /* consider unbalanced designs too      */
```

The results can contain BIBDs (λ, the pairwise frequency is an integer) and also unbalanced block designs (since `options=ubd` was specified) where λ is not an integer. The results are as follows:

Cereal Bars

t Number of Attributes	k Set Size	b Number of Sets	r Attribute Frequency	Lambda Pairwise Frequencies	n Total Sample Size
13	3	13	3	0.5	39
13	4	13	4	1	52
13	5	13	5	1.67	65
13	6	13	6	2.5	78
13	7	13	7	3.5	91
13	8	13	8	4.67	104

A BIBD might be possible with 13 blocks of 13 attributes shown 4 at a time. Four seems like a good value to pick for the set size for several reasons. Four seems good because it will work well with an orthogonal array that we could use. The array $4^1 2^4$ in 8 runs can provide the two-level factors that we need to make the partial-profile design. The orthogonal array must be a p^k subset of an array $p^k s^1$ in $p \times s$ runs with $k \leq s$. See page 1145 for more information. Now, returning to the block design, 4 is also good because that leads to a BIBD ($\lambda = 1$, an integer) which is preferable to an unbalanced block design (λ not an integer). However, in fact, four is not good, because it leads to undesirable choice sets (not statistically inefficient, but undesirable for other reasons). To better understand this, let's consider the following potential choice sets with $k = 4$:

Set	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13
1	.	1	2	.	.	.	1	.	2	.	.	.	.
	.	2	1	.	.	.	2	.	1	.	.	.	.
2	.	1	1	.	.	.	1	.	1	.	.	.	.
	.	2	2	.	.	.	2	.	2	.	.	.	.
3	.	2	1	.	.	.	1	.	1	.	.	.	.
	.	1	2	.	.	.	2	.	2	.	.	.	.
4	.	1	1	.	.	.	1	.	2	.	.	.	.
	.	2	2	.	.	.	2	.	1	.	.	.	.

For clarity, these are displayed with 2 for present, 1 for absent, and missing (.) for not varied in this set. Choice set one looks good. The trade off is between attributes 3 and 9 versus 2 and 7 (banana chips and hazel nuts versus apple and coconut). Unfortunately, whenever this kind of choice set appears (two attributes versus two attributes) another type of choice set will also appear, such as the one shown in choice set 2. It pairs four attributes present versus none present. This in no way diminishes statistical design optimality, but it might diminish realism. Do you really want a series of choice sets comparing a plain bar with one with lots of extras? Perhaps; perhaps not. The alternative with $k = 4$ is the kind of choice set shown in sets 3 and 4. They compare one attribute with three others. You will get either

a bunch of sets like 1 and 2 or a bunch like 3 and 4 depending on the random number seed you use. In general, there is not going to be a way for you to know how things will work out with your design until you try a particular combination and then carefully evaluate your design to see if it looks okay.

We will try something different. We will try $k = 6$ here and see how that works. Setting $k = 6$ has the potential to lead to comparisons of 3 attributes with 3 other attributes, 2 with 4, and sometimes perhaps 1 with 5. We can create the design as follows:

```
%mktbibd(nattrs=13,                /* 13 attributes                    */
         setsize=6,                /* vary 6 at a time                 */
         b=13,                     /* create 13 blocks of choice sets  */
         seed=289)                 /* random number seed               */
```

The results are as follows:

Cereal Bars

Block Design Efficiency Criterion	99.8492
Number of Attributes, t	13
Set Size, k	6
Number of Sets, b	13
Average Attribute Frequency	6
Average Pairwise Frequency	2.5
Total Sample Size	78
Positional Frequencies Optimized?	Yes

Attribute by Attribute Frequencies

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	3	3	3	2	2	2	3	2	3	2	2
2		6	3	2	2	2	2	3	2	3	2	3	3
3			6	3	2	3	3	2	2	2	2	2	3
4				6	3	2	3	3	2	2	2	3	2
5					6	2	2	2	3	3	3	2	3
6						6	2	3	3	3	3	2	3
7							6	2	3	3	3	3	2
8								6	3	3	3	2	2
9									6	2	2	3	2
10										6	2	3	2
11											6	2	3
12												6	3
13													6

Attribute by Position Frequencies

	1	2	3	4	5	6
1	1	1	1	1	1	1
2	1	1	1	1	1	1
3	1	1	1	1	1	1
4	1	1	1	1	1	1
5	1	1	1	1	1	1
6	1	1	1	1	1	1
7	1	1	1	1	1	1
8	1	1	1	1	1	1
9	1	1	1	1	1	1
10	1	1	1	1	1	1
11	1	1	1	1	1	1
12	1	1	1	1	1	1
13	1	1	1	1	1	1

Cereal Bars

Design

x1	x2	x3	x4	x5	x6
1	10	5	13	11	6
13	2	3	12	10	5
9	1	11	8	5	2
12	5	13	4	9	7
2	7	9	10	12	1
10	11	4	5	7	8
8	4	12	6	2	10
3	6	7	2	13	11
5	9	6	3	1	4
6	3	10	7	8	9
11	13	8	9	6	12
7	12	1	11	4	3
4	8	2	1	3	13

The less than 100% efficiency shows that a BIBD was not found, as do the nonconstant attribute by attribute frequencies (2's and 3's). However, the constant attribute frequencies (6) show that an unbalanced block design was found. The attribute by position frequencies are perfect. Every attribute appears in every position exactly once. Note, however, that they are *not* important for partial-profile designs (they are important for MaxDiff designs). You can make the macro run faster by specifying `positer=0` so that it will not try to optimize positional frequencies. You can also make it run faster by asking for fewer PROC OPTEX iterations. By default, the macro is trying to find multiple optimal designs so it can pick the one that is best in terms of position frequencies. We can run the macro again as follows to find a design much faster:

```
%mktbibd(nattrs=13,                /* 13 attributes                          */
         setsize=6,                /* vary 6 at a time                       */
         b=13,                     /* create 13 blocks of choice sets        */
         seed=289,                 /* random number seed                     */
         positer=0,                /* do not optimize position frequencies */
         optiter=100)              /* only 100 PROC OPTEX iterations         */
```

The results are as follows:

Cereal Bars

Block Design Efficiency Criterion	99.8492
Number of Attributes, t	13
Set Size, k	6
Number of Sets, b	13
Average Attribute Frequency	6
Average Pairwise Frequency	2.5
Total Sample Size	78
Positional Frequencies Optimized?	No

Attribute by Attribute Frequencies

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	6	3	3	2	3	3	3	2	3	2	2	2	2
2		6	2	2	2	3	2	3	2	3	2	3	3
3			6	2	3	2	3	3	2	3	2	3	2
4				6	2	2	3	3	3	3	2	3	3
5					6	2	2	3	2	3	3	2	3
6						6	3	3	2	2	3	3	2
7							6	2	2	2	3	2	3
8								6	3	2	2	2	2
9									6	3	3	3	2
10										6	3	2	2
11											6	2	3
12												6	3
13													6

Attribute by Position Frequencies

	1	2	3	4	5	6
1	6	0	0	0	0	0
2	3	3	0	0	0	0
3	3	1	2	0	0	0
4	1	4	1	0	0	0
5	0	3	2	1	0	0
6	0	2	2	2	0	0
7	0	0	4	0	2	0
8	0	0	2	2	1	1
9	0	0	0	5	1	0
10	0	0	0	2	3	1
11	0	0	0	1	3	2
12	0	0	0	0	3	3
13	0	0	0	0	0	6

Cereal Bars

Design

x1	x2	x3	x4	x5	x6
3	5	6	8	10	11
3	4	7	8	9	10
2	4	6	10	12	13
1	2	4	6	7	8
1	2	3	9	10	12
1	3	5	6	7	12
1	4	5	9	10	11
2	6	8	9	11	12
3	4	7	11	12	13
1	6	7	9	11	13
4	5	8	9	12	13
2	5	7	10	11	13
1	2	3	5	8	13

Our efficiency is the same, and this step ran on the order of a few seconds (compared to on the order of a minute for the previous step). The first row of the design specifies that in the first block of choice sets, attributes 2, 5, 6, 8, 10, and 11 will vary (Apple, Cashews, Chocolate, Cranberries, Peanuts, and Pecans) while the others stay constant.

Next, we will need an orthogonal array with 6 two-level attributes. The next biggest power of 2 (because we have two-level factors) greater than 6 (because we have 6 of them) is 8. We will need to divide our orthogonal array into two blocks of size 8. More is said about this later. Also see page 1145.

The following steps make the orthogonal array, combine it with the BIBD using the %MktPPro macro, and evaluate the resulting choice design:

```
%mktex(8 2 ** 6,                    /* 1 eight-level and 6 two-level factors*/
       n=16,                        /* 16 runs                              */
       seed=382)                    /* random number seed                   */

proc sort data=randomized           /* sort randomized data set             */
          out=randes(drop=x1);      /* do not need 8-level factor any more  */
   by x2 x1;                        /* must sort by x2 then x1.  Really!    */
   run;

%mktppro(ibd=bibd,                  /* input block design                   */
         design=randes)             /* input orthogonal array               */

%choiceff(data=chdes,               /* candidate set of choice sets         */
          init=chdes,               /* initial design                       */
          initvars=x1-x13,          /* factors in the initial design        */
          model=class(x1-x13 / sta),/* model with stdz orthogonal coding    */
          nsets=104,                /* number of choice sets                */
          nalts=2,                  /* number of alternatives               */
          rscale=                   /* relative D-efficiency scale factor   */
          %sysevalf(104 * 6 / 13),  /* 6 of 13 attrs in 104 sets vary       */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

The design has 104 choice sets ($b = 13$ times $s = 8$). The top half ($s = 8$) rows of the orthogonal array form the first alternatives, and the second half form the second. The last part of the output from the %ChoicEff macro is as follows:

Cereal Bars

Final Results

Design	1
Choice Sets	104
Alternatives	2
Parameters	13
Maximum Parameters	104
D-Efficiency	48.0000
Relative D-Eff	100.0000
D-Error	0.0208
1 / Choice Sets	0.009615

Cereal Bars

n	Variable Name	Label	Variance	DF	Standard Error
1	x11	x1 1	0.020833	1	0.14434
2	x21	x2 1	0.020833	1	0.14434
3	x31	x3 1	0.020833	1	0.14434
4	x41	x4 1	0.020833	1	0.14434
5	x51	x5 1	0.020833	1	0.14434
6	x61	x6 1	0.020833	1	0.14434
7	x71	x7 1	0.020833	1	0.14434
8	x81	x8 1	0.020833	1	0.14434
9	x91	x9 1	0.020833	1	0.14434
10	x101	x10 1	0.020833	1	0.14434
11	x111	x11 1	0.020833	1	0.14434
12	x121	x12 1	0.020833	1	0.14434
13	x131	x13 1	0.020833	1	0.14434
				==	
				13	

The macro reports a relative *D*-efficiency of 100. If we had not specified the `rscale=` option, the relative *D*-efficiency would have been 46.1538. This should be compared to the maximum possible relative *D*-efficiency, which is $100(k/t) = 100(6/13) = 46.1538$. The design is $(6 / 13)th$ as efficient as a generic design with all 13 attributes simultaneously varying, so it is optimal for this partial-profile experiment. When there are no restrictions, you expect *D*-efficiency to equal the number of choice sets. We are getting $(6 / 13)th$ as much information as that. Hence, for an optimal partial-profile design with 6 of 13 attributes varying in 104 choice sets, we expect a *D*-efficiency of $(6 / 13)th$ of 104. Specifying that value in the `rscale=` option gives us a relative *D*-efficiency that is relative to an optimal partial-profile design, which is in fact what we have. Note that the expression `%sysevalf(104 * 6 / 13)` is evaluated by the macro processor before the macro is invoked, and the resulting number is passed to the `%ChoiceEff` macro. Since the result is not an integer, we cannot use the `%eval` function.

The first 16 choice sets are displayed as follows:

```
proc print data=chdes; id set; by set; where set le 16; run;
```

The results are as follows:

Cereal Bars

Set	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13
1	1	1	1	1	2	2	1	2	1	2	2	1	1
	1	1	2	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	2	1	1	2	1	1
	1	1	2	1	2	2	1	1	1	2	1	1	1

3	1	1	1	1	1	1	1	2	1	2	1	1	1
	1	1	2	1	2	2	1	1	1	1	2	1	1
4	1	1	1	1	1	2	1	1	1	2	1	1	1
	1	1	2	1	2	1	1	2	1	1	2	1	1
5	1	1	1	1	2	1	1	1	1	2	2	1	1
	1	1	2	1	1	2	1	2	1	1	1	1	1
6	1	1	1	1	1	2	1	1	1	1	2	1	1
	1	1	2	1	2	1	1	2	1	2	1	1	1
7	1	1	1	1	2	2	1	2	1	1	1	1	1
	1	1	2	1	1	1	1	1	1	2	2	1	1
8	1	1	1	1	2	1	1	1	1	1	1	1	1
	1	1	2	1	1	2	1	2	1	2	2	1	1
9	1	1	1	2	1	1	2	2	2	2	1	1	1
	1	1	2	1	1	1	1	1	1	1	1	1	1
10	1	1	1	1	1	1	1	2	1	2	1	1	1
	1	1	2	2	1	1	2	1	2	1	1	1	1
11	1	1	1	1	1	1	1	2	2	1	1	1	1
	1	1	2	2	1	1	2	1	1	2	1	1	1
12	1	1	1	1	1	1	2	1	2	1	1	1	1
	1	1	2	2	1	1	1	2	1	2	1	1	1
13	1	1	1	2	1	1	1	1	2	2	1	1	1
	1	1	2	1	1	1	2	2	1	1	1	1	1
14	1	1	1	1	1	1	2	1	1	2	1	1	1
	1	1	2	2	1	1	1	2	2	1	1	1	1
15	1	1	1	2	1	1	2	2	1	1	1	1	1
	1	1	2	1	1	1	1	1	2	2	1	1	1
16	1	1	1	2	1	1	1	1	1	1	1	1	1
	1	1	2	1	1	1	2	2	2	2	1	1	1

When an attribute is all 1's in a set, then that attribute does not vary. Exactly six attributes vary in each set. The first 8 choice sets are constructed from the orthogonal array and the first row of the BIBD, the next 8 choice sets are constructed from the orthogonal array and the second row of the BIBD, and so on. In the first 8 choice sets, attributes 3, 5, 6, 8, 10 and 11 vary, and in the second block, it is 3, 4, 7, 8, 9 and 10.

You can display the orthogonal array as follows:

```
proc print noobs data=randes; run;
```

The results are as follows:

Cereal Bars

x2	x3	x4	x5	x6	x7
1	2	2	2	2	2
1	1	1	2	1	2
1	1	1	2	2	1
1	1	2	1	2	1
1	2	1	1	2	2
1	1	2	1	1	2
1	2	2	2	1	1
1	2	1	1	1	1
2	1	1	1	1	1
2	2	2	1	2	1
2	2	2	1	1	2
2	2	1	2	1	2
2	1	2	2	1	1
2	2	1	2	2	1
2	1	1	1	2	2
2	1	2	2	2	2

This matrix has two blocks corresponding to `x2` = 1 and `x2` = 2. These are called difference schemes (although the full difference schemes have 8 columns not just these 6). Understanding this is not critical, but you can go to page 115 for more information. What is important to understand is that the seemingly odd sorting of the randomized design by `x2` and then by `x1` and the dropping of `x1` is required. It guarantees that the orthogonal array has this layout of stacked difference schemes. The randomized design is used since it is much less likely to consist of rows that are constant (usually all ones) than the original design that is stored by default in the `out=design` data set. Orthogonal arrays that can be used to make optimal partial-profile designs include: $2^4 4^1$ in 8 runs, $2^8 8^1$ in 16 runs, 4^5 in 16 runs, $3^6 6^1$ in 18 runs, $2^{12} 12^1$ in 24 runs, 5^6 in 25 runs, $3^9 9^1$ in 27 runs, $2^{16} 16^1$ in 32 runs, $4^8 8^1$ in 32 runs, $3^{12} 12^1$ in 36 runs, $2^{20} 20^1$ in 40 runs, $3^9 15^1$ in 45 runs, $2^{24} 24^1$ in 48 runs, $4^{12} 12^1$ in 48 runs, 7^8 in 49 runs, $5^{10} 10^1$ in 50 runs, $3^{18} 18^1$ in 54 runs, $2^{28} 28^1$ in 56 runs, $3^{12} 21^1$ in 63 runs, $2^{32} 32^1$ in 64 runs, $4^{16} 16^1$ in 64 runs, and so on.

Next, the attributes that vary from the first four choice sets are displayed:

Cereal Bars

Set	x3	x5	x6	x8	x10	x11
1	1	2	2	2	2	2
	2	1	1	1	1	1
2	1	1	1	2	1	2
	2	2	2	1	2	1
3	1	1	1	2	2	1
	2	2	2	1	1	2
4	1	1	2	1	2	1
	2	2	1	2	1	2
5	1	2	1	1	2	2
	2	1	2	2	1	1
6	1	1	2	1	1	2
	2	2	1	2	2	1
7	1	2	2	2	1	1
	2	1	1	1	2	2
8	1	2	1	1	1	1
	2	1	2	2	2	2

Each set consists of one row from the top block of the orthogonal array and its corresponding row from the bottom block, stored in the locations dictated by the BIBD.

At 104 choice sets, most researchers would consider this design to be too large for one person to evaluate, so it could be blocked into 8 subdesigns of size 13 as follows:

```
%mktblock(data=chdes,              /* input choice design to block        */
          out=sasuser.chdes,       /* output blocked choice design        */
                                   /* stored in permanent SAS data set    */
          nalts=2,                 /* two alternatives                    */
          nblocks=8,               /* eight blocks                        */
          factors=x1-x13,          /* 13 attributes, x1-x13               */
          print=design,            /* print the blocked design (only)     */
          seed=472)                /* random number seed                  */
```

A sample of the resulting partial-profile design is as follows:

Cereal Bars

Block	Set	Alt	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13
1	1	1	1	1	1	1	1	1	1	2	1	1	2	1	1
		2	1	1	2	1	2	2	1	1	1	2	1	1	1
.															
.															
.															
1	13	1	1	2	1	1	1	1	1	2	1	1	1	1	2
		2	2	1	2	1	2	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	2	2	1	2	1	1	1	1	1
		2	1	1	2	1	1	1	1	1	1	2	2	1	1
.															
.															
.															
2	13	1	1	1	2	1	1	1	1	2	1	1	1	1	1
		2	2	2	1	1	2	1	1	1	1	1	1	1	2
3	1	1	1	1	1	1	2	1	1	1	1	2	2	1	1
		2	1	1	2	1	1	2	1	2	1	1	1	1	1
.															
.															
.															
3	13	1	1	2	1	1	1	1	1	1	1	1	1	1	1
		2	2	1	2	1	2	1	1	2	1	1	1	1	2
.															
.															
.															
8	1	1	1	1	1	1	1	2	1	1	1	1	2	1	1
		2	1	1	2	1	2	1	1	2	1	2	1	1	1
.															
.															
.															
8	13	1	1	1	1	1	2	1	1	1	1	1	1	1	2
		2	2	2	2	1	1	1	1	2	1	1	1	1	1

The following DATA step displays a summary of the design using the actual attribute levels:

```
data _null_;
   array alts[13] $ 12 _temporary_ ('Almonds' 'Apple' 'Banana Chips'
                   'Brown Sugar' 'Cashews' 'Chocolate' 'Coconut' 'Cranberries'
                   'Hazel Nuts' 'Peanuts' 'Pecans' 'Raisins' 'Walnuts');
   array x[13];                              /* 13 design factors              */
   set sasuser.chdes;                        /* read each alternative          */
   if alt eq 1 then put / block 1. set 3. +1 @; /* write block and set num  */
   else put '<vs> ' @;                       /* print '<vs>' to separate alts  */
   c = 0;                                    /* do not print a comma yet       */
   do j = 1 to 13;                           /* loop over all 13 attrs         */
      if x[j] eq 2 then do;                  /* if this one is shown           */
         if c then put +(-1) ', ' @;         /* print comma if not on 1st attr */
         put alts[j] @;                      /* print attr value               */
         c = 1;                              /* not on first term so do commas */
         end;
      end;
   run;
```

The first `array` statement creates a temporary array (no individual variable names are created or stored) of character variables (by virtue of the "$") of length 12 and initializes them to the attribute labels. The `put` statement prints lines, and lines produced by `put` statements that end in "`@`" are held so that the next `put` statement can add to it. The first `put` statement begins with a slash, which clears the previous line and starts a new line. The first block of choice sets is as follows:

```
1  1 Cranberries, Pecans <vs> Banana Chips, Cashews, Chocolate, Peanuts
1  2 Brown Sugar <vs> Banana Chips, Coconut, Cranberries, Hazel Nuts, Peanuts
1  3 Chocolate, Raisins <vs> Apple, Brown Sugar, Peanuts, Walnuts
1  4 Chocolate, Coconut <vs> Almonds, Apple, Brown Sugar, Cranberries
1  5 Hazel Nuts, Peanuts <vs> Almonds, Apple, Banana Chips, Raisins
1  6 Banana Chips, Coconut, Raisins <vs> Almonds, Cashews, Chocolate
1  7 Brown Sugar, Cashews, Hazel Nuts, Peanuts, Pecans <vs> Almonds
1  8 Cranberries, Raisins <vs> Apple, Chocolate, Hazel Nuts, Pecans
1  9 Pecans, Walnuts <vs> Banana Chips, Brown Sugar, Coconut, Raisins
1 10 Chocolate, Coconut, Hazel Nuts <vs> Almonds, Pecans, Walnuts
1 11 Cashews <vs> Brown Sugar, Cranberries, Hazel Nuts, Raisins, Walnuts
1 12 Peanuts, Walnuts <vs> Apple, Cashews, Coconut, Pecans
1 13 Apple, Cranberries, Walnuts <vs> Almonds, Banana Chips, Cashews
```

Obviously, this is a crude portrayal of the choice sets, but all the information is there. It is important to look to see if this design looks reasonable for your purposes. The presentation of the design to the subjects is much more involved than this, but the essential elements are here. Subjects are presented with two alternatives and asked to pick one of the two.

The data are collected and read into a SAS data set as follows:

```
data chdata;
   input Block Sub (c1-c13) (1.) @@;
   datalines;
 1  1 1212221112122 1  2 2222221112122 1  3 2221222112122 1  4 2211221112121
 1  5 2212222112212 1  6 2212222122122 1  7 2222221112122 1  8 2122222112122
 1  9 2122221112122 1 10 2212221112122 1 11 1222222112122 1 12 2221222112122
 1 13 1212221112212 1 14 2112222122122 1 15 2212221112222 1 16 2212221112122
 1 17 2212221112122 1 18 2212221112122 1 19 2222221112222 1 20 2212221112122
 2  1 1212112212122 2  2 1112222212122 2  3 1212222112122 2  4 1112222212122
 2  5 1221112122122 2  6 1112212122122 2  7 1212112212122 2  8 1212212212122
 2  9 1112222222122 2 10 1211122212122 2 11 1112222222122 2 12 1212112222112
 2 13 1112222212122 2 14 1212122222122 2 15 1222222222122 2 16 1212122222112
 2 17 1212212212122 2 18 1112122212122 2 19 1112222222122 2 20 1112122212122
 3  1 1221211112222 3  2 1222221111222 3  3 2121211112222 3  4 1221211112222
 3  5 1222212122222 3  6 1122211121222 3  7 1222211112222 3  8 1121211112222
 3  9 1222211112222 3 10 1221211111222 3 11 2121211111222 3 12 1121221112222
 3 13 1122211112222 3 14 1122211122222 3 15 1121212111222 3 16 1121211122222
 3 17 1122211112222 3 18 1122211112222 3 19 1122211122222 3 20 1112211121222
 4  1 2211212112212 4  2 2111212122212 4  3 2111212222212 4  4 2111211122212
 4  5 2111222122212 4  6 2122212112212 4  7 2221212122221 4  8 2111212121222
 4  9 2121212111212 4 10 2122212121211 4 11 2121212111211 4 12 2111212122212
 4 13 2111212111211 4 14 2221212122212 4 15 2211211122212 4 16 2111212111212
 4 17 2211212121221 4 18 2121212112212 4 19 2112212111212 4 20 2111212121221
 5  1 2112122112112 5  2 1212122122211 5  3 2222122111111 5  4 2212222121211
 5  5 2222122112111 5  6 2122122212112 5  7 1212122111112 5  8 2222222122111
 5  9 1222222112111 5 10 2212122111111 5 11 2211221122111 5 12 2211121111111
 5 13 2212222221212 5 14 2222222112111 5 15 2222222121111 5 16 2222222111111
 5 17 2222122122211 5 18 1212122112111 5 19 2212222122111 5 20 2212222212111
 6  1 1222221212112 6  2 1222221212122 6  3 1222121212212 6  4 1222221212212
 6  5 1222221211112 6  6 1222221212112 6  7 1222221112122 6  8 1222221212112
 6  9 1221221211112 6 10 1222121222112 6 11 1222221211122 6 12 1221221212212
 6 13 1222121222212 6 14 1222221211112 6 15 1221121212112 6 16 1222221211212
 6 17 1222121212212 6 18 1221221212112 6 19 1212121212212 6 20 1222221212112
 7  1 2112222221211 7  2 1122222222111 7  3 1112222222111 7  4 2112222222111
 7  5 2112222222112 7  6 2112222222111 7  7 2112222222111 7  8 1112222221211
 7  9 2112222122111 7 10 1122222222211 7 11 2112222222211 7 12 2112222222121
 7 13 2112222222211 7 14 2112222222211 7 15 1112222222111 7 16 1112122222111
 7 17 2112222222111 7 18 2112222222211 7 19 2112222212211 7 20 1112222222211
 8  1 2211112221122 8  2 2212212222121 8  3 2212212221121 8  4 2212212121122
 8  5 2211122221121 8  6 2212222121121 8  7 2221122221111 8  8 2211212221122
 8  9 2211212221121 8 10 2211112221111 8 11 2211212221121 8 12 2212222221121
 8 13 2211112221122 8 14 2111222221122 8 15 2212222221121 8 16 2212222221121
 8 17 2222222221121 8 18 2111112121122 8 19 2111212221122 8 20 2112212222121
;
```

The data consist of a block number, a subject number, and then 13 choices, one for each of the 13 sets within each block. In the interest of space, data from four subjects appear on a single line. The

following steps merge the data and the design and do the analysis:

```
%mktmerge(design=sasuser.chdes,     /* input final blocked choice design    */
          data=chdata,              /* input choice data                    */
          out=desdata,              /* output design and data               */
          blocks=block,             /* the blocking variable is block       */
          nsets=13,                 /* 13 choice sets per subject           */
          nalts=2,                  /* 2 alternatives in each set           */
          setvars=c1-c13)           /* the choices for each subject vars    */

%phchoice(on)                       /* customize PHREG for a choice model   */

proc phreg brief data=desdata;      /* provide brief summary of strata      */
   ods output parameterestimates=pe;/* output parameter estimates           */
   class x1-x13 / ref=first;        /* name all as class vars, '1' ref level*/
   model c*c(2) = x1-x13;           /* 1 - chosen, 2 - not chosen           */
                                    /* x1-x13 are independent vars          */
   label x1  = 'Almonds'            /* set of descriptive labels            */
         x2  = 'Apple'
         x3  = 'Banana Chips'
         x4  = 'Brown Sugar'
         x5  = 'Cashews'
         x6  = 'Chocolate'
         x7  = 'Coconut'
         x8  = 'Cranberries'
         x9  = 'Hazel Nuts'
         x10 = 'Peanuts'
         x11 = 'Pecans'
         x12 = 'Raisins'
         x13 = 'Walnuts';
   strata block sub set;            /* set within subject within block      */
   run;                             /* identify each choice set             */

proc sort data=pe;                  /* process the parameter estimates      */
   by descending estimate;          /* table by sorting by estimate         */
   run;

data pe;                            /* also get rid of the '2' level        */
   set pe;                          /* in the label                         */
   substr(label, length(label)) = ' ';
   run;

proc print label;                   /* print estimates with largest first   */
   id label;
   label label = '00'x;
   var df -- probchisq;
   run;

%phchoice(off)                      /* restore PHREG to a survival PROC     */
```

The results are as follows:

Cereal Bars

The PHREG Procedure

Model Information

Data Set	WORK.DESDATA
Dependent Variable	c
Censoring Variable	c
Censoring Value(s)	2
Ties Handling	BRESLOW

Number of Observations Read	4160
Number of Observations Used	4160

Class Level Information

Class	Value	Design Variables
x1	1	0
	2	1
x2	1	0
	2	1
x3	1	0
	2	1
x4	1	0
	2	1
x5	1	0
	2	1
x6	1	0
	2	1
x7	1	0
	2	1
x8	1	0
	2	1
x9	1	0
	2	1
x10	1	0
	2	1

x11	1	0
	2	1
x12	1	0
	2	1
x13	1	0
	2	1

Cereal Bars

The PHREG Procedure

Summary of Subjects, Sets, and Chosen and Unchosen Alternatives

Pattern	Number of Choices	Number of Alternatives	Chosen Alternatives	Not Chosen
1	2080	2	1	1

Convergence Status

Convergence criterion (GCONV=1E-8) satisfied.

Model Fit Statistics

Criterion	Without Covariates	With Covariates
-2 LOG L	2883.492	1414.992
AIC	2883.492	1440.992
SBC	2883.492	1514.314

Testing Global Null Hypothesis: BETA=0

Test	Chi-Square	DF	Pr > ChiSq
Likelihood Ratio	1468.4999	13	<.0001
Score	1083.7833	13	<.0001
Wald	551.9476	13	<.0001

Type 3 Tests

Effect	DF	Wald Chi-Square	Pr > ChiSq
x1	1	201.2893	<.0001
x2	1	3.2314	0.0722
x3	1	12.5575	0.0004
x4	1	65.4783	<.0001
x5	1	394.3608	<.0001
x6	1	0.9942	0.3187
x7	1	17.9353	<.0001
x8	1	183.3954	<.0001
x9	1	132.2685	<.0001
x10	1	20.2881	<.0001
x11	1	46.2902	<.0001
x12	1	123.3861	<.0001
x13	1	92.4112	<.0001

Cereal Bars

The PHREG Procedure

Multinomial Logit Parameter Estimates

	DF	Parameter Estimate	Standard Error	Chi-Square	Pr > ChiSq
Almonds 2	1	1.67575	0.11811	201.2893	<.0001
Apple 2	1	-0.18096	0.10067	3.2314	0.0722
Banana Chips 2	1	0.36271	0.10235	12.5575	0.0004
Brown Sugar 2	1	-0.79392	0.09811	65.4783	<.0001
Cashews 2	1	2.95195	0.14865	394.3608	<.0001
Chocolate 2	1	-0.10184	0.10213	0.9942	0.3187
Coconut 2	1	-0.41696	0.09845	17.9353	<.0001
Cranberries 2	1	1.56238	0.11537	183.3954	<.0001
Hazel Nuts 2	1	-1.23998	0.10782	132.2685	<.0001
Peanuts 2	1	0.46061	0.10226	20.2881	<.0001
Pecans 2	1	0.72250	0.10619	46.2902	<.0001
Raisins 2	1	1.13299	0.10200	123.3861	<.0001
Walnuts 2	1	1.01308	0.10539	92.4112	<.0001

Cereal Bars

	DF	Parameter Estimate	Standard Error	Chi-Square	Pr > ChiSq
Cashews	1	2.95195	0.14865	394.3608	<.0001
Almonds	1	1.67575	0.11811	201.2893	<.0001
Cranberries	1	1.56238	0.11537	183.3954	<.0001
Raisins	1	1.13299	0.10200	123.3861	<.0001
Walnuts	1	1.01308	0.10539	92.4112	<.0001
Pecans	1	0.72250	0.10619	46.2902	<.0001
Peanuts	1	0.46061	0.10226	20.2881	<.0001
Banana Chips	1	0.36271	0.10235	12.5575	0.0004
Chocolate	1	-0.10184	0.10213	0.9942	0.3187
Apple	1	-0.18096	0.10067	3.2314	0.0722
Coconut	1	-0.41696	0.09845	17.9353	<.0001
Brown Sugar	1	-0.79392	0.09811	65.4783	<.0001
Hazel Nuts	1	-1.23998	0.10782	132.2685	<.0001

The number of observations read and used is 4160. This is 20 subjects times 8 blocks times 13 sets per block, times 2 alternatives. Also, the data consist of 2080 (4160 divided by 2 alternatives) choice sets where two alternatives were presented, one was chosen, and one was not chosen. The parameter estimate table is displayed twice. Once in the original order, the order of the attributes, and once sorted by the parameter estimates. Cashews are most preferred, and Hazel Nuts are least preferred.

Example 8, A MaxDiff Choice Experiment

This example is like the previous example in that we will use the same ingredients of high-end cereal bars as before. These ingredients are as follows:

Almonds
Apple
Banana Chips
Brown Sugar
Cashews
Chocolate
Coconut
Cranberries
Hazel Nuts
Peanuts
Pecans
Raisins
Walnuts

This time, we will show subjects sets of attributes (or ingredients) and ask them to pick the one they like the best and the one they like the least. This is called a MaxDiff or best-worst study (Louviere 1991, Finn and Louviere 1992). We will use a balanced incomplete block design. The $t = 13$ attributes are shown in b sets of size k.

You can find the sizes in which a BIBD might be available for ranges of t, b, and k, using the %MktBSize macro as in the following example:

```
title 'Cereal Bars';

%mktbsize(nattrs=13,                /* 13 total attributes                 */
          setsize=2 to 6,           /* show between 2 and 6 at once        */
          nsets=2 to 40,            /* make between 2 and 40 choice sets   */
          options=ubd,              /* consider unbalanced designs         */
          maxreps=5)                /* permit multiple replications, which */
                                    /* will show us some BIBDs that might  */
                                    /* not be otherwise listed with        */
                                    /* options=ubd                         */
```

The results of this step are as follows:

Cereal Bars

t Number of Attributes	k Set Size	b Number of Sets	r Attribute Frequency	Lambda Pairwise Frequencies	n Total Sample Size	Number of Replications
13	2	13	2	0.17	26	1
13	2	26	4	0.33	52	2
13	2	39	6	0.5	78	3
13	3	13	3	0.5	39	1
13	3	26	6	1	78	2
13	3	39	9	1.5	117	3
13	4	13	4	1	52	1
13	4	26	8	2	104	2
13	4	39	12	3	156	3
13	5	13	5	1.67	65	1
13	5	26	10	3.33	130	2
13	5	39	15	5	195	3
13	6	13	6	2.5	78	1
13	6	26	12	5	156	2
13	6	39	18	7.5	234	3

A BIBD might be possible with all four values of k, the set size shown. We could do a small pilot study with 13 choice sets and 4 attributes shown. The following step constructs the BIBD:

```
%mktbibd(out=sasuser.bibd,          /* output BIBD                         */
         nattrs=13,                 /* 13 total attributes                 */
         setsize=4,                 /* show 4 in each set                  */
         nsets=13,                  /* 13 choice sets                      */
         seed=93)                   /* random number seed                  */
```

The results are as follows:

Cereal Bars

Block Design Efficiency Criterion	100.0000
Number of Attributes, t	13
Set Size, k	4
Number of Sets, b	13
Attribute Frequency	4
Pairwise Frequency	1
Total Sample Size	52
Positional Frequencies Optimized?	Yes

Attribute by Attribute Frequencies

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	4	1	1	1	1	1	1	1	1	1	1	1	1
2		4	1	1	1	1	1	1	1	1	1	1	1
3			4	1	1	1	1	1	1	1	1	1	1
4				4	1	1	1	1	1	1	1	1	1
5					4	1	1	1	1	1	1	1	1
6						4	1	1	1	1	1	1	1
7							4	1	1	1	1	1	1
8								4	1	1	1	1	1
9									4	1	1	1	1
10										4	1	1	1
11											4	1	1
12												4	1
13													4

Attribute by Position Frequencies

	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1
5	1	1	1	1
6	1	1	1	1
7	1	1	1	1
8	1	1	1	1
9	1	1	1	1
10	1	1	1	1
11	1	1	1	1
12	1	1	1	1
13	1	1	1	1

Cereal Bars
Balanced Incomplete Block Design

x1	x2	x3	x4
3	11	1	10
11	4	7	6
4	9	13	3
8	10	6	9
7	13	10	12
5	1	9	7
10	2	5	4
13	5	8	11
1	6	2	13
12	8	4	1
2	7	3	8
9	12	11	2
6	3	12	5

A BIBD was found, and every attribute is shown with every other attribute exactly once. Furthermore, the positional frequencies are perfect. It is important to vary the positions in which the attributes are displayed.

The following step displays the BIBD, but with the actual attribute names rather than numbers:

```
data _null_;
   array alts[13] $ 12 _temporary_ ('Almonds' 'Apple' 'Banana Chips'
                   'Brown Sugar' 'Cashews' 'Chocolate' 'Coconut' 'Cranberries'
                   'Hazel Nuts' 'Peanuts' 'Pecans' 'Raisins' 'Walnuts');
   set sasuser.bibd;                /* read design                          */
   put alts[x1] +(-1) ', '          /* print each attr, comma separated     */
       alts[x2] +(-1) ', ' alts[x3] +(-1) ', ' alts[x4];
   run;
```

The results are as follows:

```
Banana Chips, Pecans, Almonds, Peanuts
Pecans, Brown Sugar, Coconut, Chocolate
Brown Sugar, Hazel Nuts, Walnuts, Banana Chips
Cranberries, Peanuts, Chocolate, Hazel Nuts
Coconut, Walnuts, Peanuts, Raisins
Cashews, Almonds, Hazel Nuts, Coconut
Peanuts, Apple, Cashews, Brown Sugar
Walnuts, Cashews, Cranberries, Pecans
Almonds, Chocolate, Apple, Walnuts
Raisins, Cranberries, Brown Sugar, Almonds
Apple, Coconut, Banana Chips, Cranberries
Hazel Nuts, Raisins, Pecans, Apple
Chocolate, Banana Chips, Raisins, Cashews
```

Obviously, this is a crude portrayal of the choice sets, but all the information is there. It is important to look to see if this design looks reasonable for your purposes. The presentation of the design to the subjects is much more involved than this, but the essential elements are here. Subjects are presented with four ingredients and asked to pick the one they like the best and the one they like the least.

The data are entered into a SAS data set as follows:

```
data bwdata;
   input (x1-x26) (1.);
   datalines;
34133132412334244323323143
34143214411431314314422141
23143414421432344221422142
31123213421332231241432143
34133214421434242124233142
34314232211334211423432141
24434114312334344243122141
31134124322334214243213442
41123114131432241321422432
41423113341334431242424132
32423412122334244343423141
31433214431334231313122443
41133214411334244314422141
34213123431334241223212443
13134214311334211343422141
32242421341331232323122142
41124214311342244313132142
31143214411334231213411442
41423124211234244323412141
31233413211431121243424141
;
```

There are 20 rows for 20 subjects and 26 variables (13 best choices and 13 worst choices). The data alternate best then worst: `x1` is a best choice, `x2` is a worst choice, `x3` is a best choice, `x4` is a worst choice, and so on. The data all consist of integers in the range 1 to 4. These represent the positions of the chosen alternatives. There are many other ways the data could be handled. Best can come first or worst can come first, the variables can alternate or not, and the data could be positions (1-4 in this case) or the data could be attribute numbers (1-13 in this case). See the `%MktMDiff` macro on page 1105 for more information.

The data are analyzed as follows:

```
%let attrlist=Almonds,Apple,Banana Chips,Brown Sugar,Cashews,Chocolate
,Coconut,Cranberries,Hazel Nuts,Peanuts,Pecans,Raisins,Walnuts;

%phchoice(on)                          /* customize PHREG for a choice model   */

%mktmdiff(bwaltpos,                    /* data are best then worst and         */
                                       /* alternating and are the positions    */
                                       /* of the chosen attributes             */
          nattrs=13,                   /* 13 attributes                        */
          nsets=13,                    /* 13 choice sets                       */
          setsize=4,                   /* 4 attributes shown in each set       */
          attrs=attrlist,              /* list of attribute names              */
          data=bwdata,                 /* input data set with data             */
          design=sasuser.bibd)         /* input data set with BIBD             */
```

First, the `%PHChoice` macro is used to customize the output from PROC PHREG, which is called by the `%MktMDiff` macro, for the multinomial logit model. Next, the description of each attribute is stored in a macro variable with commas delimiting the individual labels. Next, the `%MktMDiff` macro is called to combine the data and the design and do the analysis. The results are as follows:

```
                                Cereal Bars
Var Order:   Best then Worst
Alternating: Variables Alternate
Data:        Positions (Not Attribute Numbers)
Best Vars:   x1 x3 x5 x7 x9 x11 x13 x15 x17 x19 x21 x23 x25
Worst Vars:  x2 x4 x6 x8 x10 x12 x14 x16 x18 x20 x22 x24 x26
Attributes:  Almonds
             Apple
             Banana Chips
             Brown Sugar
             Cashews
             Chocolate
             Coconut
             Cranberries
             Hazel Nuts
             Peanuts
             Pecans
             Raisins
             Walnuts
```

Cereal Bars

The PHREG Procedure

Model Information

Data Set	WORK.CODED
Dependent Variable	c
Censoring Variable	c
Censoring Value(s)	2
Frequency Variable	Count
Ties Handling	BRESLOW

Number of Observations Read	191
Number of Observations Used	191
Sum of Frequencies Read	2080
Sum of Frequencies Used	2080

Summary of Subjects, Sets, and Chosen and Unchosen Alternatives

Pattern	Number of Choices	Number of Alternatives	Chosen Alternatives	Not Chosen
1	26	80	20	60

Convergence Status

Convergence criterion (GCONV=1E-8) satisfied.

Model Fit Statistics

Criterion	Without Covariates	With Covariates
-2 LOG L	4557.308	4135.039
AIC	4557.308	4159.039
SBC	4557.308	4210.085

Testing Global Null Hypothesis: BETA=0

Test	Chi-Square	DF	Pr > ChiSq
Likelihood Ratio	422.2690	12	<.0001
Score	368.1231	12	<.0001
Wald	268.2494	12	<.0001

Cereal Bars
Multinomial Logit Parameter Estimates

	DF	Parameter Estimate	Standard Error	Chi-Square	Pr > ChiSq
Cashews	1	2.37882	0.35539	44.8047	<.0001
Cranberries	1	0.49982	0.29266	2.9167	0.0877
Almonds	1	0.46203	0.27989	2.7250	0.0988
Raisins	1	0.18013	0.28220	0.4074	0.5233
Walnuts	0	0	.	.	.
Pecans	1	-0.46634	0.27270	2.9245	0.0872
Peanuts	1	-0.57648	0.29073	3.9319	0.0474
Chocolate	1	-1.26584	0.28645	19.5276	<.0001
Banana Chips	1	-1.32289	0.28004	22.3153	<.0001
Apple	1	-1.34142	0.29063	21.3033	<.0001
Coconut	1	-1.83155	0.28743	40.6033	<.0001
Brown Sugar	1	-1.96331	0.29142	45.3891	<.0001
Hazel Nuts	1	-2.55326	0.29966	72.5995	<.0001

The first table provides a summary of the data and the specifications. The data alternate best then worst and are positions not attribute numbers. The best variables are the odd numbered variables, and the worst variables are the even numbered variables. Finally the attributes are listed. There were 26 choices (13 best and 13 worst) and 20 times (20 subjects) an alternative was chosen and 60 times (3 not chosen by 20 subjects) alternatives were not chosen. The final table of parameter estimates is displayed ordered in descending order of preference. Cashews are most preferred and Hazel nuts are least preferred by these subjects.

The design is arrayed so that there is one classification variable with 13 levels. The 'Walnuts' level, being the last level alphabetically, is the reference level and has a coefficient of 0. If by chance it had been the most preferred level, then all of the other coefficients would have been negative. If it had been the least preferred level, then all of the other coefficients would have been positive. This is illustrated in the following step:

```
%mktmdiff(bwaltpos,                 /* data are best then worst and         */
                                    /* alternating and are the positions    */
                                    /* of the chosen attributes             */
          nattrs=13,                /* 13 attributes                        */
          nsets=13,                 /* 13 choice sets                       */
          setsize=4,                /* 4 attributes shown in each set       */
          attrs=attrlist,           /* list of attribute names              */
          classopts=zero='Hazel Nuts',/* set the reference level            */
          data=bwdata,              /* input data set with data             */
          design=sasuser.bibd)      /* input data set with BIBD             */

%phchoice(off)                      /* restore PHREG to a survival PROC     */
```

Normally, you should not specify the `classopts=` option unless you are changing the reference level.

The last table of results is as follows:

Cereal Bars

Multinomial Logit Parameter Estimates

	DF	Parameter Estimate	Standard Error	Chi-Square	Pr > ChiSq
Cashews	1	4.93207	0.38051	168.0095	<.0001
Cranberries	1	3.05308	0.30862	97.8642	<.0001
Almonds	1	3.01529	0.31258	93.0567	<.0001
Raisins	1	2.73338	0.30734	79.0976	<.0001
Walnuts	1	2.55326	0.29966	72.5995	<.0001
Pecans	1	2.08691	0.29407	50.3637	<.0001
Peanuts	1	1.97678	0.30362	42.3879	<.0001
Chocolate	1	1.28741	0.28701	20.1202	<.0001
Banana Chips	1	1.23036	0.29306	17.6254	<.0001
Apple	1	1.21184	0.28834	17.6636	<.0001
Coconut	1	0.72170	0.27811	6.7342	0.0095
Brown Sugar	1	0.58994	0.28726	4.2175	0.0400

Now, all but the reference level is shown, and all of the estimates are positive. The difference is they have all been shifted by subtracting the original coefficient for 'Hazel Nuts'. This makes the new coefficient for 'Hazel Nuts' zero (larger by the absolute value of the original coefficient) and the rest larger by the same amount.

Conclusions

This chapter introduced some choice design terminology and ideas with some examples but without going into great detail on how to make designs and process data for analysis. The information in this chapter should provide a good foundation for all of the detailed examples in the discrete choice chapter.

Choice Design Glossary

Experimental design, choice modeling, and choice design, like all other areas, all have their own vocabularies. This section defines some of those terms. These terms are used and defined throughout this chapter and the discrete choice chapter (pages 285–663).

aliased – Two effects are confounded or aliased when they are not distinguishable from each other. Lower-order effects such as main effects or two-way interactions might be aliased with higher-order interactions in most of our designs. We estimate lower-order effects by assuming that higher-order effects are zero or negligible. See page 495.

allocation study – An allocation study is a choice study where multiple, not single choices are made. For example, in prescription drug marketing, physicians are asked questions like "For the next ten prescriptions you write for a particular condition, how many would you write for each of these drugs?" See page 535.

alternative – An alternative is one of the options available to be chosen in a choice set. An alternative might correspond to a particular brand in a branded study or just a bundle of attributes in a generic study. See page 55.

alternative-specific attribute – An alternative-specific attribute is one that is expected to interact with brand. If you expect utility to change in different ways for the different brands, then the attribute is alternative-specific. Otherwise, it is generic. In the analysis, there is a set of alternative-specific attribute parameters for each alternative. See page 55.

asymmetric design – An experimental design where not all factors have the same number of levels. At least one factor has a number of levels that is different from at least one other factor. See page 112.

attribute – An attribute is one of the characteristics of an alternative. Common attributes include price, size, and a variety of other product-specific factors. See page 55.

availability cross-effects – A design might have a varying number of alternatives. When not all alternatives are available in every choice set, availability cross-effects, might be of interest. These capture the effects of the presence/absence of one brand on the utility of another. See page 470.

balance – A design is balanced when each level occurs equally often within each factor. See page 58.

balanced incomplete block design – A balanced incomplete block design (BIBD) is a list of t treatments that appear together in b blocks. Each block contains a subset ($k < t$) of the treatments. A BIBD is commonly represented by a $b \times k$ matrix with entries ranging from 1 to t. Each of the b rows is one block. Each of the t treatments must appear the same number of times in the design, and each of the t treatments must appear with each of the other $t - 1$ treatments in exactly the same number of blocks. When the treatment frequencies are constant in a block design, but the pairwise frequencies are not constant, the design is called an *unbalanced block design*. More generally, an *incomplete block design* includes BIBDs, unbalanced block designs, and block designs where neither the treatment nor the pairwise frequencies are constant. BIBDs and unbalanced block designs are used in marketing research for MaxDiff studies (see the `%MktMDiff` macro, page 1105) and partial-profile designs (see the `%MktPPro` macro, page 1145). In a MaxDiff study, there are t attributes shown in b sets of size k. In a certain class of partial-profile designs, there are t attributes, shown in b blocks of choice sets, where k attributes vary in each block. Block designs can be constructed with the `%MktBIBD` macro (see page 963). Also see page 989 for information about when a BIBD might exist. See page 115.

binary coding – Binary coding replaces the levels of qualitative or `class` variables with binary indicator variables. Less-than-full-rank binary coding creates one binary variable for each level of the factor. Full-rank binary coding (or reference cell coding) creates one binary variable for all but one level, the reference level. See page 73.

blocking – Large choice designs need to be broken into blocks. Subjects will just see a subset of the full design. How many blocks depends on the number of choice sets and the complexity of the choice task. See pages 217 and 426.

branded design – A branded choice design has one factor that consists of a brand name or other alternative label. The vacation examples on pages 339-443 are examples of branded designs even though the labels, destinations, and not brands. The examples starting on pages 302, 468, and 444 use branded designs and actual brand names.

canonical correlation – The first canonical correlation is the maximum correlation that can occur between linear combinations of two sets of variables. We use the canonical correlation between two sets of coded `class` variables as a way of showing deviations from design orthogonality. See page 101.

choice design – A choice design has one column for every different product attribute and one row for every alternative of every choice set. In some cases, different alternatives have different attributes and different choice sets might have differing numbers of alternatives. See pages 55 and 67–71.

choice set – A choice set consists of two or more alternatives. Subjects see one or more choice sets and choose one alternative from each set. See page 55.

confounded – See "aliased."

covariance matrix – See "variance matrix."

cross-effects – A cross-effect represents the effect of one alternative on the utility of another alternative. When the IIA assumption holds, all cross-effects are zero. See page 452.

deviations from means coding – See "effects coding."

efficiency – Efficiency is a scale or measurement of the goodness of an experimental design based on the average of the eigenvalues of the variance matrix. *A*-efficiency is a function of the arithmetic mean of the eigenvalues, which is also the arithmetic mean of the variances. *D*-efficiency is a function of the geometric mean of the eigenvalues. In many cases, efficiency is scaled to a 0 to 100 scale where 0 means one or more parameters cannot be estimated and 100 means the design is perfect. See page 62.

effects coding – Effects coding (or deviations from means coding) is similar to full-rank binary coding, except that the row for the reference level is set to all –1's instead of all zeros. See page 73.

experimental design – An experimental design is a plan for running an experiment. See page 53.

factor – A factor is a column of an experimental design with two or more fixed values, or levels. In the context of conjoint and choice modeling, you could use the terms "factor" and "attribute" interchangeably. However, in this book, the term "factor" is usually used to refer to a column of a "raw" design (with columns such as `x1` and `x2`) that has not yet been processed and relabeled into the form of a conjoint or choice design. It is also used when discussing coding and other design concepts that are the same for linear model, choice, and conjoint designs. See page 54.

fractional-factorial design – A fractional-factorial design is a subset of a full-factorial design. Often, this term is used to refer to particularly "nice" fractions such as the designs created by PROC FACTEX. See page 995.

full-factorial design – A full-factorial design consists of all possible combinations of the all of the levels of all of the factors. See page 57.

generic attribute – A generic attribute is one that is not expected to interact with brand (or more generally, the attribute label). If you expect utility to change as a function of the levels of the attribute in the same way for every brand, then the attribute is generic. In contrast, if you expect utility to change in different ways for the different brands, then the attribute is alternative-specific. All attributes in generic designs are generic. In the analysis, there is one set of parameters for generic attributes, regardless of the number of alternatives. See page 55.

generic design or **generic model** – A generic design has no brands or labels for the alternatives. The alternatives are simply bundles of attributes. For example, each alternative might be a cell phone or computer all made by the same manufacturer. See page 102.

IIA – The independence of irrelevant alternatives or IIA property states that utility only depends on an alternative's own attributes. IIA means the odds of choosing alternative c_i over c_j do not depend on the other alternatives in the choice set. Departures from IIA exist when certain subsets of brands are in more direct competition and tend to draw a disproportionate amount of share from each other than from other brands. See pages 452, 459, 468, 674, and 679.

incomplete block design – A block design where treatment frequencies and pairwise frequencies are not constant, or an unbalanced block design, or a balanced incomplete block design. Usually, the term "incomplete block design" is used to refer to block designs that do not meet the stricter criteria necessary to be classified as an unbalanced block design or a balanced incomplete block design. See "balanced incomplete block design" for more information.

indicator variables – Indicator variables (or "dummy variables") are binary variables that are used to represent categorical or `class` variables in an analysis. Less precisely, this term is sometimes used to refer to other coding schemes. See page 73.

information matrix – The information matrix (for factorial design matrix $\mathbf{X}$) is $\mathbf{X}'\mathbf{X}$. See page 62.

interaction – Interactions involve two or more factors, such as a brand by price interaction. For example, in a model with interactions brand preference is different at the different prices and the price effect is different for the different brands. See page 57.

level – A level is a fixed value of a design factor. Raw designs typically start with levels that are positive or nonnegative integers. Then these levels are reassigned with actual levels such as brands or prices. See page 53.

linear arrangement – The linear arrangement of a choice design ("linear arrangement" for short) contains one row for each choice set and one column for every attribute of every alternative. However, brand or some other alternative-labeling factor, is not a factor in the linear arrangement. Rather, the brand or alternative label is a bin into which the other factors are collected. The columns are grouped, the first group contains every attribute for the first alternative, ..., and the *jth* group contains every attribute for the *jth* alternative. The linear arrangement is an intermediate step in constructing a choice design by one of the several available approaches. In the linear arrangement, all of the information for a single choice set is arrayed in a single line or row vector. You can rearrange the design from the linear arrangement to the standard choice design arrangement by moving each of the m blocks for the m alternatives below the preceding block creating a choice design with m times as many rows as previously and approximately $1/m$ times as many columns. In other words, in the linear arrangement, there is one row per choice set, and in the choice design arrangement there is one matrix with m rows per choice set. See pages 67–71.

linear design – A term used in previous editions for what is now called the linear arrangement of a choice design or "linear arrangement" for short.

main effect – A main effect is a simple effect, such as a price or brand effect. For example, in a main-effects model the brand effect is the same at the different prices and the price effect is the same for the different brands. See page 57.

MaxDiff – In a MaxDiff study, subjects are shown sets of messages or product attributes and are asked to choose the best (or most important) from each set as well as the worst (or least important). A balanced incomplete block design is used, and the data are analyzed with a choice model. See page 225.

mother logit model – The mother logit model is a model with cross-effects that can be used to test for violations of IIA. See page 452.

orthogonal – When every pair of levels occurs equally often across all pairs of factors, the design is orthogonal. Another way in which a design can be orthogonal is when the frequencies for level pairs are proportional instead of equal. See page 58.

orthogonal array – An orthogonal array is an experimental design in which all estimable effects are uncorrelated. See page 59.

orthogonal contrast coding – Orthogonal contrast coding codes all but the reference level as an orthogonal contrast between each level and the levels that come before along with the reference level. The coded values are all integers. See page 73. Also see "standardized orthogonal contrast coding."

partial-profile design – A partial-profile choice design consists of bundles of attributes where only a subset of attributes vary in each choice set. Partial-profile designs can be constructed from an orthogonal array and a balanced incomplete block design. See page 207. Alternatively, they can be constructed by creating designs with restrictions. See page 595.

random number seed – The random number seed is an integer in the range 1 to 2,147,483,646 that is used to provide a starting point for the random number stream. While our designs are not random, there is often some random process used in their creation. See page 94.

randomization – Randomization involves sorting the rows of a design into a random order and randomly reassigning all of the factor levels. See page 57.

reference cell coding – See "binary coding."

reference level – The reference level is the level of a factor that does not correspond to a binary variable in reference level (binary, or indicator variable) coding. In effects or the orthogonal codings, it is the level that corresponds to the row of –1's. The reference level is by default the last level of the factor, but you can change that with the `zero=` option. See page 73.

resolution – Resolution identifies which effects are estimable. For resolution III designs, all main effects are estimable free of each other, but some of them are confounded with two-factor interactions. For resolution IV designs, all main effects are estimable free of each other and free of all two-factor interactions, but some two-factor interactions are confounded with other two-factor interactions. For resolution V designs, all main effects and two-factor interactions are estimable free of each other. See page 58.

runs – A run is a row of an experimental design. See page 53.

seed – See "random number seed."

standardized orthogonal contrast coding – Standardized orthogonal contrast coding codes all but the reference level as an orthogonal contrast between each level and the levels that come before along with the reference level. The coded values are scaled so that the sum of squares of each column equals the number of levels. See page 73. Also see "orthogonal contrast coding."

symmetric design – An experimental design where all factors have the same number of levels. See page 112.

unbalanced block design – An incomplete block design where every treatment appears with the same frequency, but pairwise frequencies are not constant. See "balanced incomplete block design" for more information.

variance matrix – The variance matrix (for linear model design matrix $\mathbf{X}$) is proportional to $(\mathbf{X}'\mathbf{X})^{-1}$. See page 62. Also see page 71 to see the variance matrix for a choice model.

Exercises

These exercises are designed to acquaint you with the basic principles of designing a choice experiment. You might find it helpful to store each choice design in a permanent SAS data set, each with a unique name, so that you can compare the designs when you are done. You should also specify random number seeds throughout so that you can reproduce your results.

Here are some questions to ask yourself throughout. What coding should I use? Does it matter? Why? What is the range of *D*-efficiency with this coding? Is this a good candidate set size? Might I do better with a smaller candidate set? Might I do better with a larger candidate set? Might I do better with a larger design and more blocks? Might I do better with a smaller design? Should I use the randomized design? Does it matter?

Do not expect the answers to these or some of the questions below (such as some of the "Why?" questions) to always be straight-forward. There might not be a clear correct answer. Sometimes you have to make judgment calls.

If any step takes more than a minute or two of computer time, you should go back and try to simplify your approach. Typically, you should start with a very small candidate set and a small number of iterations. Make your first design for each problem as quickly as possible, even if it is not optimal. You can go back later and try more iterations or larger candidate sets.

Direct Construction of a Generic Design

1) In this exercise, you will make a generic choice design with 6 alternatives, 6 choice sets, and six-level attributes. Assume a main-effects model with $\boldsymbol{\beta} = \mathbf{0}$.

1.a) What is the maximum number of six-level attributes that you can create in a choice design with perfect 100% relative *D*-efficiency?

1.b) Make and display an optimal generic design with the maximum number of attributes determined in the previous question. How often does each level of each attribute appear with each level of each other attribute across all alternatives and choice sets?

1.c) Evaluate the *D*-efficiency of the choice design.

1.d) What are the variances? What is *D*-error? What is *D*-efficiency? How are these quantities related?

1.e) How many parameters are in the model? Why? What is the maximum number of parameters you could estimate with 6 alternatives and 6 choice sets? Why?

1.f) Make a generic choice design with 6 alternatives, 6 choice sets, and 6 six-level attributes using the same approach that you used to make the previous design. That is, just add more attributes to the code you already have. Evaluate it and its *D*-efficiency.

1.g) How many parameters are in the model? Why? How many more parameters could you add to the model?

1.h) If a design existed for this specification with 100% relative *D*-efficiency, what would the raw unscaled *D*-efficiency be?

1.i) Would you use this design in a real study? Why or why not? What might you consider changing to make a better design?

Generic Design Construction by Searching Candidate Alternatives

2) In this exercise, you will again make a generic choice design with 6 alternatives, 6 choice sets, and 6 six-level attributes, but by using a different method. Assume a main-effects model with $\boldsymbol{\beta} = \mathbf{0}$.

2.a) Construct this design by searching a candidate set of alternatives. What is the *D*-efficiency?

2.b) Try at least two other candidate sets. What sizes did you pick? Why? Which one works best?

2.c) Construct the relative *D*-efficiency of each design (including the corresponding design from exercise 1 relative to each other using the following program:

```
proc iml;
   eff = { };   /* insert list of efficiencies inside of braces:
                   example: eff = { 5.2 3.1 4.3 3.5};
                */
   label = { }; /* provide labels for each design showing the candidate
                   set sizes.  example:
                   label = {"exercise 1"   "cand = a"
                            "cand = b"     "cand = c"};
                */
   x = j(ncol(eff), ncol(eff), 0);
   do i = 1 to ncol(eff);
      do j = 1 to ncol(eff);
         x[i,j] = 100 # eff[i] / eff[j];
         end;
      end;
   print x[rowname=label colname=label];
   quit;
```

2.d) Which design is best? Why?

Generic Design Construction by Searching Candidate Choice Sets

3) In this exercise, you will again make a generic choice design with 6 alternatives, 6 choice sets, and 6 six-level attributes, but again by using a different method. Assume a main-effects model with $\boldsymbol{\beta} = \mathbf{0}$.

3.a) Construct this design by searching a candidate set of choice sets. What is the *D*-efficiency? Hint: use the `%MktEx` options `options=quickr largedesign`, `maxtime=1`. See the macro documentation chapter to answer the next question: What do these options do, and why do we use them here?

3.b) Add the *D*-efficiency from this design to your PROC IML program and re-run the program. What are the results? What is the best approach for this problem?

Symmetric Alternative-Specific Designs

4) In this exercise, you will create a choice design for a study with four brands, A, B, C, and D. Each choice set will have four alternatives, and each of the four brands will always appear in each choice set. Each brand has 4 four-level attributes. You should begin by using the `%MktEx` and `%MktRoll` macros to make a linear arrangement and convert it into a choice design.

4.a) What is the minimum number of choice sets that you need? How many would you choose? Would you block the design? If so, how many blocks would you choose? What other block sizes might you consider?

4.b) Construct and display the linear and choice designs.

4.c) Evaluate the efficiency of the choice design. Assume a main-effects model with $\boldsymbol{\beta} = \mathbf{0}$. How many parameters are in the model? Why? What is the structure of the variance matrix?

4.d) Evaluate the efficiency of the choice design. Assume an alternative-specific effects model with $\boldsymbol{\beta} = \mathbf{0}$. How many parameters are in the model? Did you get 51? Why? What is the structure of the variance matrix?

4.e) What are the variances? What is *D*-efficiency? Is this design optimal? What are its strengths? What are its weaknesses?

4.f) Again, construct a choice experiment for four brands, each with 4 four-level attributes, for an alternative-specific model with $\boldsymbol{\beta} = \mathbf{0}$. However, this time search a candidate set of alternatives. What is the structure of the variance matrix?

4.g) Try several candidate set sizes. Compare your *D*-efficiencies from the linear arrangement and the candidate set search approach using the IML program from exercise 2. Which method works best? Why?

4.h) Which approach has the "nicest" variances and covariances? What is the range of relative *D*-efficiency in this problem?

4.i) Block the best design that you found.

4.j) Again, construct a choice experiment for four brands, each with 4 four-level attributes, for a main-effects model with $\boldsymbol{\beta} = \mathbf{0}$. Search a candidate set of alternatives, but this time create only three alternatives. Each choice set will have between 1 and 3 brands, and at least one brand will be missing from each choice set.

4.k) Does brand ever appear more than once in a choice set? Why or why not?

4.l) Once again, construct a choice experiment for four brands, each with 4 four-level attributes, for an alternative-specific model with $\boldsymbol{\beta} = \mathbf{0}$. Search a candidate set of alternatives. This time, the first attribute is price. Construct a design such that the prices are as follows: Brand A's prices are 1.49, 1.99, 2.49, and 2.99; Brand B's prices are 1.99, 2.49, 2.99, and 3.49; Brand C's prices are 1.79, 2.29, 2.79, and 3.29; and Brand D's prices are 1.69, 2.19, 2.69, and 3.19. Hints: Use a DATA step to convert the `x1` factor into a price attribute with different prices for each alternative. You can use the `drop=` option in the `%ChoicEff` macro to drop extra terms from the model. What is the difference between your designs with and without the extra terms dropped?

Asymmetric Alternative-Specific Designs

5) In this exercise, you will create a choice design for a study with four brands, A, B, C, and D. Each choice set will have four alternatives, and each of the four brands will always appear in each choice set. Each brand has a four-level attribute, 2 three-level attributes, and a two-level attributes. You will begin by using the `%MktEx` and `%MktRoll` macros to make a linear arrangement and convert it into a choice design.

5.a) What is the minimum number of choice sets that you need? How many would you choose? Would you block the design? If so, how many blocks would you choose? What other block sizes might you consider?

5.b) Construct and display the linear and choice designs.

5.c) Evaluate the efficiency of the choice design. Assume a main-effects model with $\boldsymbol{\beta} = \mathbf{0}$. How many parameters are in the model? Why?

5.d) Evaluate the efficiency of the choice design. Assume an alternative-specific effects model with $\boldsymbol{\beta} = \mathbf{0}$. How many parameters are in the model? Why?

5.e) What are the variances? What is *D*-efficiency? Is this design optimal? What are its strengths? What are its weaknesses?

5.f) Again, construct a choice experiment for four brands, each with 4 four-level attributes, for an alternative-specific model with $\boldsymbol{\beta} = \mathbf{0}$. However, this time search a candidate set of alternatives.

5.g) Try several candidate set sizes. Compare your *D*-efficiencies from the linear arrangement and the candidate set search approach using the IML program from exercise 2. Which method works best? Why? What is the range of relative *D*-efficiency in this problem?

Restricted Designs

6) Construct a choice experiment for three brands, each with 4-, 3-, 3-, and 2-level attributes, for a main-effects model with $\boldsymbol{\beta} = \mathbf{0}$. Search a candidate set of choice sets, each with three alternative, one per brand. Disallow from consideration any choice set where the four-level attribute has the same level in two or more alternatives.

6.a) How many candidate choice sets would you try first? What other choices are worth a try? What is the smallest number that you could try? What is the largest number that you would try?

6.b) Try several different choice set sizes. Which works best? Why? Did the restrictions get imposed correctly?

6.c) How many parameters are in the choice model? What is the maximum number for this specification? Why? Would you feel comfortable using the best design that you found?

6.d) Construct a choice experiment for the same three brands, again with 4-, 3-, 3-, and 2-level attributes, for a main-effects model with $\boldsymbol{\beta} = \mathbf{0}$. Search a candidate set of alternatives. Disallow from consideration any choice set where the three- or four-level attributes have the same level in two or more alternatives. Write a restrictions macro without `do` loops.

6.e) Repeat exercise 6.d, but this time use 3 `do` loops and a single assignment statement with one simple Boolean expression. Do you get the same results as 6.d?

Answers appear in `http://support.sas.com/techsup/technote/mr2010c.sas`.