%ChoicEff Macro

The %ChoicEff autocall macro finds efficient experimental designs for choice experiments and evaluates choice designs. You supply a set of candidates. The macro searches the candidates for an efficient experimental design—a design in which the variances of the parameter estimates are minimized, given an assumed parameter vector $\boldsymbol{\beta}$.

If you are anxious to get started on choice designs, and you want to start immediately, then this is the right place. The examples in the examples section illustrate all of the tools that you need to make good choice designs. They do not illustrate all of the tools that you can use or always illustrate the very best methods for your particular situation, but they illustrate the minimum subset of tools you need to do virtually anything you might ever need to do in the area of choice design. If instead, you want to begin by better understanding what you are doing, be sure to read the experimental design chapter beginning on page 53.

You should also see the discrete choice chapter starting on page 285. Note though that the discrete choice chapter is an older chapter, and the approach that it emphasizes (making a choice design through the %MktEx and %MktRoll macros) is not in use as much as it once was. While that approach is important, and it can in fact create optimal designs for some problems, you can make all of the designs that you need with the more limited tool set described in this chapter where the %MktEx macro only makes candidate sets for the %ChoicEff macro. When you become a sophisticated designer of choice experiments, you will want to be facile with all of the tools in the discrete choice chapter, the experimental design chapter, and this chapter. However, when you are just starting, you might find it easer to concentrate on simply using the %ChoicEff macro with a candidate set of alternatives that you create by using the %MktEx macro.

See the following pages for examples of using this macro in the design chapter: 81, 83, 85, 87, 109, 112, 137, 140, 142, 170, 193 and 203. Also see the following pages for examples of using this macro in the discrete choice chapter: 313, 317, 320, 322, 360, 365, 366, 430, 508, 509, 542, 559, 564, 567, 570, 570, 574, 576, 597, 599, 607, 618, 628, 632, 636, 645, 650, 654, 656, 659 and 662. Additional examples appear throughout this chapter.

There are four ways that you can use the %ChoicEff macro:

- You create a candidate set of alternatives, and the macro creates a design consisting of choice sets built from the alternatives you supplied. You must designate for each candidate alternative the design alternative(s) for which it is a candidate. For a generic design, you create one list of candidate alternatives, and each candidate can be used for every alternative in the design. For a branded study with m brands, you must create a list with m types of candidate alternatives, one for each brand.

- You create a candidate set of choice sets, and the macro builds a design from the choice sets that you supplied. This approach was designed to handle restrictions across alternatives (certain alternatives may not appear with certain other alternatives) and with partial-profile designs (Chrzan and Elrod 1995). However, the candidate set of alternatives approach along with the new restrictions= option (described next) is often better than this approach. This is because for all but the smallest designs, the candidate set of choice set approach considers much smaller subsets of possible designs. Unless it is much easier for you to create a candidate set of restricted choice sets than to create a restrictions macro, you should use the restrictions= option and a candidate set of choice sets instead of a candidate set of choice sets.

- You create a candidate set of alternatives and a macro that provides restrictions on how the alternatives can be used to make the design. The `%ChoicEff` macro creates a design consisting of choice sets built from the alternatives you supplied. You must designate for each candidate alternative the design alternative(s) for which it is a candidate. For a generic design, you create one list of candidate alternatives, and each candidate can be used for every alternative in the design. For a branded study with m brands, you must create a list with m types of candidate alternatives, one for each brand. You can restrict the design in any way (within alternatives, across alternatives and within choice sets, or even across choice sets). For example, you can use the restrictions macro to prevent dominated alternatives, to force or prevent overlap in factor levels within choice set, to prevent certain levels from occurring with other levels, to force constant attributes within choice set, to control the number of constant attributes across choice sets, and so on.
- You supply a choice design, and the `%ChoicEff` macro evaluates it. The choice design might have been created by a previous run of the `%ChoicEff` macro, or by the `%MktEx` macro, or by other means.

The `%ChoicEff` macro uses a modified Fedorov candidate-set-search algorithm, just like the OPTEX procedure and the parts of the `%MktEx` macro. Typically, you use as a candidate set a full-factorial design, a fractional-factorial design, or an orthogonal array created with the `%MktEx` macro. First, the `%ChoicEff` macro either constructs a random initial design from the candidates or it uses an initial design that you specified. The macro considers swapping out every design alternative/set and replacing it with each candidate alternative/set. Swaps that increase efficiency are performed. The process of evaluating and swapping continues until efficiency stabilizes at a local maximum. This process is repeated with different initial designs, and the best design is output for use. The key differences between the `%ChoicEff` macro and the `%MktEx` macro are as follows:

- The `%ChoicEff` macro requires you to specify the true (or assumed true) parameters and it optimizes the variance matrix for a multinomial logit discrete choice model, which is a nonlinear model.
- The `%MktEx` macro optimizes the variance matrix for a linear model, which does not depend on the parameters.

Examples

The example section provides a series of examples of different ways that you can use the %ChoicEff macro.

Generic Choice Design Constructed from Candidate Alternatives

This example creates a design for a generic choice model with 3 three-level factors. First, you use the %MktEx macro to create a set of candidate alternatives, where x1-x3 are the factors. Note that the n= specification accepts expressions. The following steps make and display the candidate set:

```
%mktex(3 ** 3, n=3**3, seed=238)

proc print; run;
```

The results are as follows:

Obs	x1	x2	x3
1	1	1	1
2	1	1	2
3	1	1	3
4	1	2	1
5	1	2	2
6	1	2	3
7	1	3	1
8	1	3	2
9	1	3	3
10	2	1	1
11	2	1	2
12	2	1	3
13	2	2	1
14	2	2	2
15	2	2	3
16	2	3	1
17	2	3	2
18	2	3	3
19	3	1	1
20	3	1	2
21	3	1	3
22	3	2	1
23	3	2	2
24	3	2	3
25	3	3	1
26	3	3	2
27	3	3	3

Next, you run the %ChoicEff macro to find an efficient design for the unbranded, purely generic choice

model assuming $\boldsymbol{\beta} = \mathbf{0}$ as follows:*

```
%choiceff(data=design,              /* candidate set of alternatives        */
          model=class(x1-x3 / sta), /* model with stdz orthogonal coding    */
          nsets=9,                  /* number of choice sets                */
          flags=3,                  /* 3 alternatives, generic candidates   */
          seed=289,                 /* random number seed                   */
          maxiter=60,               /* maximum number of designs to make    */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */

proc print; var x1-x3; id set; by set; run;

proc print data=bestcov label;
   title 'Variance-Covariance Matrix';
   id __label;
   label __label = '00'x;
   var x:;
   run;
title;

%mktdups(generic, data=best, factors=x1-x3, nalts=3)
```

The option `data=final` names the input data set, `model=class(x1-x3 / sta)` specifies the PROC TRANSREG `model` statement for coding the design (the `sta` option, short for `standorth`, uses a standardize orthogonal coding), `nsets=9` specifies nine choice sets, `options=relative` requests that relative *D*-efficiency be displayed, `flags=3` specifies that there are three alternatives in a purely generic design,† `beta=zero` specifies all zero parameters, and `seed=382` specifies the random number seed, and `maxiter=50` specifies the number of designs to create. The design and the covariance matrix is displayed, and the design is checked for duplicates. Some of the results are as follows:

n	Name	Beta	Label
1	x11	0	x1 1
2	x12	0	x1 2
3	x21	0	x2 1
4	x22	0	x2 2
5	x31	0	x3 1
6	x32	0	x3 2

.
.
.

*If you are not running version 9.2 or a later SAS release, remove the slash and the `sta` option from the `model=` specification. The standardized orthogonal contrast coding is first available with Version 9.2 of SAS. Without this option, you will not get a relative *D*-efficiency in the 0 to 100 range.

†The option name `flags=` comes from the fact that in the context of other types of designs (designs with brands or labeled alternatives), this option provides a set of "flag" variables that specify which candidates can be be used for which alternatives.

Design	Iteration	D-Efficiency	D-Error
52	0	4.74648	0.21068
	1	9.00000 *	0.11111
	2	9.00000	0.11111

.
.
.

Final Results

Design	52
Choice Sets	9
Alternatives	3
Parameters	6
Maximum Parameters	18
D-Efficiency	9.0000
Relative D-Eff	100.0000
D-Error	0.1111
1 / Choice Sets	0.1111

n	Variable Name	Label	Variance	DF	Standard Error
1	x11	x1 1	0.11111	1	0.33333
2	x12	x1 2	0.11111	1	0.33333
3	x21	x2 1	0.11111	1	0.33333
4	x22	x2 2	0.11111	1	0.33333
5	x31	x3 1	0.11111	1	0.33333
6	x32	x3 2	0.11111	1	0.33333
				==	
				6	

Set	x1	x2	x3
1	3	1	3
	1	3	1
	2	2	2
2	2	2	3
	1	3	2
	3	1	1
3	3	2	2
	2	1	1
	1	3	3

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

5       1       2       1
        2       1       2
        3       3       3

6       3       1       2
        1       2       3
        2       3       1

7       3       2       1
        2       3       3
        1       1       2

8       1       1       3
        3       3       2
        2       2       1

9       1       1       1
        3       2       3
        2       3       2
```

Variance-Covariance Matrix

	x1 1	x1 2	x2 1	x2 2	x3 1	x3 2
x1 1	0.11111	0.00000	-0.00000	0.00000	0.00000	0.00000
x1 2	0.00000	0.11111	0.00000	-0.00000	0.00000	-0.00000
x2 1	-0.00000	0.00000	0.11111	0.00000	0.00000	0.00000
x2 2	0.00000	-0.00000	0.00000	0.11111	0.00000	-0.00000
x3 1	0.00000	0.00000	0.00000	0.00000	0.11111	0.00000
x3 2	0.00000	-0.00000	0.00000	-0.00000	0.00000	0.11111

The output from the `%ChoicEff` macro consists of a list of the parameter names, values and labels, followed by a series of iteration histories (each based on a different random initial design), followed by a brief report on the most efficient design found, and finally a table with the parameter names, variances, *df*, and standard errors. The design and covariance matrix are displayed using PROC PRINT.

This design is optimal; it has 100% relative *D*-efficiency. Because a generic (main-effects only) design is requested with $\boldsymbol{\beta} = \mathbf{0}$, and the standardized orthogonal contrast coding is used, it is possible to get a relative *D*-efficiency on a 0 to 100 scale. For many other models, this will not be the case. Relative *D*-efficiency is *D*-efficiency divided by the number of choice sets, then multiplied by 100. In an optimal generic design such as this one, all of the following hold:

- *D*-efficiency equals the number of choice sets.
- *D*-error equals one over the number of choice sets.
- All of the variances equal one over the number of choice sets.

- All of the covariances are zero.
- Relative *D*-efficiency equals 100.

Note that *D*-error is by definition equal to the inverse of *D*-efficiency. Also note that in practice, the computed values of the covariances are often not precisely zero because inexact floating point arithmetic is involved. This is why some display as –0.00000 in the output.

The `%MktDups` macro reports the following:

```
Design:          Generic
Factors:         x1-x3
                 x1 x2 x3
Sets w Dup Alts: 0
Duplicate Sets:  0
```

There are no duplicate choice sets or duplicate alternatives within choice sets.

You can assign more descriptive names to the factors and values for the levels, for example, as follows:

```
proc format;
   value sf 1 = '12 oz' 2 = '16 oz' 3 = '24 oz';
   value cf 1 = 'Red  ' 2 = 'Green' 3 = 'Blue ';
   run;

data ChoiceDesign;
   set best;
   format x1 sf. x2 cf. x3 dollar6.2;
   label x1 = 'Size' x2 = 'Color' x3 = 'Price';
   x3 + 0.49;
   run;

proc print label; var x1-x3; id set; by set; run;
```

The final design is as follows:

Set	Size	Color	Price
1	24 oz	Red	$3.49
	12 oz	Blue	$1.49
	16 oz	Green	$2.49
2	16 oz	Green	$3.49
	12 oz	Blue	$2.49
	24 oz	Red	$1.49
3	24 oz	Green	$2.49
	16 oz	Red	$1.49
	12 oz	Blue	$3.49

```
4     16 oz    Red       $3.49
      12 oz    Green     $2.49
      24 oz    Blue      $1.49

5     12 oz    Green     $1.49
      16 oz    Red       $2.49
      24 oz    Blue      $3.49

6     24 oz    Red       $2.49
      12 oz    Green     $3.49
      16 oz    Blue      $1.49

7     24 oz    Green     $1.49
      16 oz    Blue      $3.49
      12 oz    Red       $2.49

8     12 oz    Red       $3.49
      24 oz    Blue      $2.49
      16 oz    Green     $1.49

9     12 oz    Red       $1.49
      24 oz    Green     $3.49
      16 oz    Blue      $2.49
```

Flag Variables

This example uses the `flags=3` option in the `%ChoicEff` macro as follows:

```
%choiceff(data=design,              /* candidate set of alternatives        */
          model=class(x1-x3 / sta), /* model with stdz orthogonal coding    */
          nsets=9,                  /* number of choice sets                */
          flags=3,                  /* 3 alternatives, generic candidates   */
          seed=289,                 /* random number seed                   */
          maxiter=60,               /* maximum number of designs to make    */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

This is a short-hand notation for specifying flag variables. Your specifications must contain information that show or "flag" which candidate can be used in which alternative. In a generic design, you can create this information by creating flag variables yourself, or you can use the short-hand specification and have the `%ChoicEff` macro create these variables for you. When you create the flag variables yourself, you create one variable for each alternative. (With three alternatives, you create three flag variables.) The flag variables flag which candidates can be used for which alternative(s). Since this is a generic choice model, each candidate can appear in any alternative, which means you need to add flags that are constant and all one: `f1=1 f2=1 f3=1`. You can use the `%MktLab` macro to add the flag variables, essentially by specifying that you have three intercepts, you can program it yourself with a DATA step, or you can let the `%ChoicEff` macro create the flag variables for you. The option `int=f1-f3` in the `%MktLab` macro creates three variables with values that are all one. A 1 in variable `f1` indicates that the candidate can be used for the first alternative, a 1 in `f2` indicates that the candidate

can be used for the second alternative, and so on. In this case, all candidates can be used for all alternatives, otherwise the flag variables contain zeros for candidates that cannot be used for certain alternatives. The default output data set from the %MktLab macro is called FINAL and is specified in the data= option in the %ChoiceEff macro. The following step illustrates:

```
%mktlab(data=design, int=f1-f3)

proc print data=final; run;

%choiceff(data=final,               /* candidate set of alternatives        */
          model=class(x1-x3 / sta), /* model with stdz orthogonal coding    */
          nsets=9,                  /* number of choice sets                */
          flags=f1-f3,              /* flag which alt can go where, 3 alts  */
          seed=289,                 /* random number seed                   */
          maxiter=60,               /* maximum number of designs to make    */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

The candidates along with the flag variables are as follows:

Obs	f1	f2	f3	x1	x2	x3
1	1	1	1	1	1	1
2	1	1	1	1	1	2
3	1	1	1	1	1	3
4	1	1	1	1	2	1
5	1	1	1	1	2	2
6	1	1	1	1	2	3
7	1	1	1	1	3	1
8	1	1	1	1	3	2
9	1	1	1	1	3	3
10	1	1	1	2	1	1
11	1	1	1	2	1	2
12	1	1	1	2	1	3
13	1	1	1	2	2	1
14	1	1	1	2	2	2
15	1	1	1	2	2	3
16	1	1	1	2	3	1
17	1	1	1	2	3	2
18	1	1	1	2	3	3

19	1	1	1	3	1	1
20	1	1	1	3	1	2
21	1	1	1	3	1	3
22	1	1	1	3	2	1
23	1	1	1	3	2	2
24	1	1	1	3	2	3
25	1	1	1	3	3	1
26	1	1	1	3	3	2
27	1	1	1	3	3	3

The results of the `%ChoicEff` macro match the results from the previous part of the example.

Direct Construction of an Optimal Generic Choice Design

These next steps directly create an optimal design for this generic choice model and evaluate its efficiency using the `%ChoicEff` macro and the initial design options. This generic design is created with only 3 choice sets this time, and it is constructed as follows:

```
%mktex(3 ** 4, n=9)

%mktlab(data=design, vars=Set Size Color Price)

proc print data=final; id set; by set; var size -- price; run;
```

The design is as follows:

Set	Size	Color	Price
1	1	1	1
	2	3	2
	3	2	3
2	1	3	3
	2	2	1
	3	1	2
3	1	2	2
	2	1	3
	3	3	1

Notice that each attribute contains all three levels in each choice set.

The following steps evaluate the design:

```
%choiceff(data=final,              /* candidate set of choice sets         */
          init=final(keep=set),    /* select these sets from candidates    */
          intiter=0,               /* evaluate without internal iterations */
          model=class(size color   /* main-effects model with stdz         */
                      price / sta), /* orthogonal coding                   */
          nsets=3,                 /* number of choice sets                */
          nalts=3,                 /* number of alternatives               */
          options=relative,        /* display relative D-efficiency        */
          beta=zero)               /* assumed beta vector, Ho: b=0         */

%mktdups(generic, data=best, factors=size color price, nalts=3)
```

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 initial design is constructed from the candidate set. The initial design is constructed by selecting the candidates specified in the `Set` variable in the `init=` data set. This is accomplished with the `init=final(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. Specify `intiter=0` (internal iterations equals zero) when you just want to evaluate the efficiency of a given design and not improve on it. The output from the `%ChoicEff` macro is as follows:

---

n	Name	Beta	Label
1	x11	0	x1 1
2	x12	0	x1 2
3	x21	0	x2 1
4	x22	0	x2 2
5	x31	0	x3 1
6	x32	0	x3 2

Design	Iteration	D-Efficiency	D-Error
1	0	3.00000 *	0.33333

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

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	

This design is optimal. Relative *D*-efficiency is 100%, and all of the variances of the parameter estimates are equal to one over the number of choice sets.

The `%MktDups` macro reports the following:

```
Design:          Generic
Factors:         size color price
                 Color Price Size
Sets w Dup Alts: 0
Duplicate Sets:  0
```

There are no duplicate choice sets or duplicate alternatives within choice sets.

The following steps create and evaluate an equivalent but slightly different version of the optimal generic choice design:

```
%mktex(3 ** 4, n=9, options=nosort)

proc sort; by x4 x1; run;

%mktlab(data=design, vars=Size Color Price Set)

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(size color   /* main-effects model with stdz        */
                      price / sta), /* orthogonal coding                  */
          nsets=3,                 /* number of choice sets               */
          nalts=3,                 /* number of alternatives              */
          options=relative,        /* display relative D-efficiency       */
          beta=zero)               /* assumed beta vector, Ho: b=0        */

%mktdups(generic, data=best, factors=size color price, nalts=3)
```

The full results are not shown, but the design is as follows:

Set	Size	Color	Price
1	1	1	1
	2	2	2
	3	3	3
2	1	2	3
	2	3	1
	3	1	2
3	1	3	2
	2	1	3
	3	2	1

It has a cyclic structure where the second and third alternatives are constructed from the previous alternative by adding 1 (mod 3).* The levels for just the first alternative for each set are as follows:

1	1	1
1	2	3
1	3	2

This matrix is known as a 3×3 difference scheme of order 3.† While we will not prove any of this here or anywhere else in this book, the following fact is used in a number of places in this book. The cyclic development of a difference scheme (that is, making subsequent p-level alternative from previous alternatives by adding 1 mod p), is an algorithm for making optimal generic choice designs. See the section beginning on 102 for more information about optimal generic choice designs.

*More precisely, since these numbers are based on one instead of zero, the operation is: (x mod 3) + 1.

†Although more typically, we think of this matrix minus one as the difference scheme.

Generic Choice Design Constructed from Candidate Choice Sets

This example is provided for complete coverage of the `%ChoicEff` macro. If you are just getting started, concentrate instead on examples of the `%ChoicEff` macro that use candidate sets of alternatives. The next example starts on page 820.

These next steps use the `%MktEx` and `%MktRoll` macros to create a candidate set of choice sets and the `%ChoicEff` macro to search for an efficient design using the candidate-set-swapping algorithm:

```
%mktex(3 ** 9, n=2187, seed=368)

%mktroll(design=design, key=3 3, out=rolled)

%choiceff(data=rolled,              /* candidate set of choice sets         */
          model=class(x1-x3 / sta), /* model with stdz orthogonal coding    */
          nsets=9,                  /* number of choice sets                */
          nalts=3,                  /* number of alternatives               */
          maxiter=20,               /* maximum number of designs to make    */
          seed=205,                 /* random number seed                   */
          options=relative nodups,  /* display relative D-eff, avoid dups   */
          beta=zero)                /* assumed beta vector, Ho: b=0         */

%mktdups(generic, data=best, factors=x1-x3, nalts=3)
```

The first steps create a candidate set of choice sets. The `%MktEx` macro creates a design with nine factors, three for each of the three alternatives. The `%MktRoll` macro with `key=3 3` makes the following `Key` data set:

x1	x2	x3
x1	x2	x3
x4	x5	x6
x7	x8	x9

It specifies that the first alternative is made from the linear arrangement factors `x1-x3`, the second alternative is made from `x4-x6`, and the third alternative is made from `x7-x9`. The `%MktRoll` macro turns a linear arrangement of a choice design into a true choice design using the rules specified in the `Key` data set.

In the `%ChoicEff` macro, the `nalts=3` option specifies that there are three alternatives. There must always be a constant number of alternatives in each choice set, even if all of the alternatives will not be used. When a nonconstant number of alternatives is desired, you must use a weight variable to flag those alternatives that the subject will not see (see for example page 912). When you swap choice sets, you need to specify `nalts=`. The output from these steps is not appreciably different from what we saw previously, so it is not shown. The `%ChoicEff` macro can on occasion find a 100% *D*-efficient generic choice design with this approach. However, the optimal design is much harder to find when using a large candidate set of choice sets than when using a small candidate set of alternatives. This is one reason why the candidate set of alternatives approach (potentially with restrictions) is usually preferred over creating a candidate set of choice sets.

Avoiding Dominated Alternatives

In this next example, there are 6 four-level attributes, which are all quantitative in nature. As the factor level value increases, the desirability of the feature increases. Of course, dominance could go in the other direction (the desirability increases as the level decreases as in price), and that can easily be handled as well. When one alternative contains levels that are all less than or equal to the levels for another alternative, the first alternative is dominated by the second. When one alternative is dominated by another, the choice task becomes easier for respondents. Eliminating dominated alternatives forces the respondents to consider all of the attributes and all of the alternatives in making a choice. The goal in this example is to write a restrictions macro that prevents dominated alternatives from occurring. The first step makes a candidate set of alternatives:

```
%mktex(4 ** 6, n=32, seed=104)
```

The next steps find a choice design where no alternatives dominate:

```
%macro res;
   do i = 1 to nalts;
      do k = i + 1 to nalts;
         if all(x[i,] >= x[k,])     /* alt i dominates alt k               */
            then bad = bad + 1;
         if all(x[k,] >= x[i,])     /* alt k dominates alt i               */
            then bad = bad + 1;
         end;
      end;
   %mend;

%choiceff(data=randomized,          /* candidate set of choice sets        */
          model=class(x1-x6 / sta), /* model with stdz orthogonal coding   */
          nsets=8,                  /* number of choice sets               */
          flags=4,                  /* 4 alternatives, generic candidates  */
          seed=104,                 /* random number seed                  */
          options=relative          /* display relative D-efficiency       */
                  resrep,           /* detailed report on restrictions     */
          restrictions=res,         /* name of the restrictions macro      */
          resvars=x1-x6,            /* vars used in defining restrictions  */
          maxiter=1,                /* maximum number of designs to make   */
          beta=zero)                /* assumed beta vector, Ho: b=0        */
```

First, a macro is created that counts dominated alternatives. The macro is written in IML. An IML scalar `bad` is increased by one every time a dominated alternative is found. Note that the scalar `bad` is automatically initialized to zero. In this example, the restrictions macro is evaluating each choice set at the time that the `%ChoicEff` macro is constructing it. The current choice set that is being considered is stored in the matrix `x`. When all elements in the *ith* row of `x` are greater than or equal to all elements in the *kth* row of `x`, the *ith* row dominates the *kth* row and `bad` is increased by one. Similarly, when all elements in the *kth* row of `x` are greater than or equal to all elements in the *ith* row of `x`, the *kth* row dominates the *ith* row and `bad` is increased by one. The `do` loops set `i` and `k` to (1,2), (1,3), (1,4) (2,3), (2,4), and (3,4), which are all pairs of attributes within a choice set.

The expression `all(x[i,] >= x[k,])` works as follows. The expression `(x[i,] >= x[k,])` compares two row vectors, the *ith* row in `x` and the *kth* row in `x`. The result of comparing two vectors with a Boolean operation (in this case, greater than or equal to) is another vector, of the same order as the two vectors being compared, that consists of ones when the element-wise comparison is true and zeros when the element-wise comparisons are false. The `all` function returns a 1 or true when all elements in the scalar, vector, or matrix argument are nonzero and nonmissing. Otherwise if there are any zeros or missings, it returns a zero or false. For example, `all({1 2 3} >= {3 2 1})` = `all({0 1 1})` = 0 (or false) and `all({3 2 4} >= {1 2 3})` = `all({1 1 1})` = 1 (or true).

The `%ChoicEff` macro uses the `res` macro when it considers swapping alternatives into the design. You must specify two options when there are restrictions. The `restrictions=`*macro-name* option provides the name of the macro that evaluates the badness of each choice set. The `resvars=`*variable-list* option provides the names of the variables that are in the design. The option `maxiter=1` is used to make only one design for now during the testing phase until you are sure that you are specifying restrictions correctly. The option `options=resrep` adds additional output to the iteration history to show how the macro is progressing in producing a design that conforms to the restrictions.

Some of the results are as follows:

```
                Design   Iteration  D-Efficiency       D-Error
                ------------------------------------------------
                     1       0          2.69601 *     0.37092
at    1   1 swapped in     1       2.91185 bad =       1
at    1   2 swapped in     2       3.04604 bad =       0
at    1   3 swapped in    11       3.14498 bad =       0
at    1   4 swapped in     9       3.22393 bad =       0
at    1   4 swapped in    20       3.28990 bad =       0
at    1   4 swapped in    27       3.34674 bad =       0
at    2   1 swapped in     1       3.54869 bad =       2
at    2   1 swapped in     2       3.54869 bad =       0
at    2   1 swapped in     3       3.57907 bad =       0
at    2   1 swapped in     5       3.69173 bad =       0
at    2   1 swapped in    31       3.72044 bad =       0
at    2   2 swapped in     2       3.92700 bad =       0
at    2   2 swapped in     4       3.93246 bad =       0
at    2   2 swapped in     5       4.05283 bad =       0
at    2   2 swapped in    23       4.06014 bad =       0
at    2   3 swapped in     3       4.08614 bad =       0
at    2   3 swapped in     5       4.17255 bad =       0
at    2   3 swapped in     8       4.18298 bad =       0
at    2   3 swapped in    20       4.26063 bad =       0
at    2   4 swapped in    29       4.26063 bad =       0
at    3   1 swapped in     1       4.42830 bad =       2
at    3   1 swapped in     2       4.42830 bad =       0
at    3   1 swapped in     3       4.43255 bad =       0
at    3   1 swapped in     4       4.45001 bad =       0
at    3   1 swapped in     5       4.63752 bad =       0
at    3   2 swapped in     3       4.67839 bad =       0
```

```
at    3   2 swapped in     4       4.69266 bad =       0
at    3   2 swapped in     6       4.77758 bad =       0
at    3   3 swapped in     1       4.82692 bad =       0
at    3   3 swapped in     4       4.91595 bad =       0
at    3   4 swapped in     7       4.96346 bad =       0
at    3   4 swapped in    12       5.06320 bad =       0
.
.
.
at    7   4 swapped in    17       5.76862 bad =       0
at    8   1 swapped in     1       5.83926 bad =       0
at    8   1 swapped in     3       5.91170 bad =       0
at    8   2 swapped in    30       5.91170 bad =       0
at    8   3 swapped in    24       5.94511 bad =       0
at    8   4 swapped in    22       5.94511 bad =       0
                       1           5.94511 *     0.16821
.
.
.
at    1   1 swapped in     1       6.39883 bad =       0
at    1   2 swapped in     2       6.39883 bad =       0
at    1   3 swapped in    20       6.39883 bad =       0
at    1   4 swapped in    16       6.39883 bad =       0
at    3   1 swapped in     5       6.39883 bad =       0
at    3   2 swapped in     9       6.39883 bad =       0
at    3   3 swapped in     4       6.39883 bad =       0
at    3   3 swapped in     7       6.46427 bad =       0
at    3   4 swapped in    12       6.46427 bad =       0
.
.
.
at    8   1 swapped in     3       6.54545 bad =       0
at    8   2 swapped in    30       6.54545 bad =       0
at    8   3 swapped in    24       6.54545 bad =       0
at    8   4 swapped in    22       6.54545 bad =       0
                       3           6.54545 *     0.15278
at    1   1 swapped in     1       6.54545 bad =       0
at    1   2 swapped in     2       6.54545 bad =       0
at    1   3 swapped in    20       6.54545 bad =       0
at    1   4 swapped in    16       6.54545 bad =       0
                       4           6.54545       0.15278
```

First, an ordinary line is printed in the iteration history table, which displays the efficiency of the initial random design. Next, there is a line that provides information about every swap that is performed. In choice set i, alternative j, candidate alternative k was swapped in resulting in a D-efficiency of a and a new value for `bad` of b. You can see that badness does not always start out at zero but it quickly goes to zero. An ordinary line is written to the iteration history table whenever a full pass through the design is completed.

Lines beginning with "`at`" are added by `options=resrep`. All other lines appear by default. The first line (`1 0 2.69601 0.37092`) provides the design number (1), iteration number (0), *D*-efficiency (2.69601), and *D*-error (0.37092) for the initial random design. The next line (`at 1 1 swapped in 1 2.91185 bad = 1`) specifies that in choice set 1 and alternative 1, candidate alternative 1 is swapped in resulting in a *D*-efficiency of 2.911854 and a badness value of 1. The initial badness is not stated, but it must have been greater than zero. No other candidate alternatives improve the badness or efficiency for alternative 1 of set 1, so no other swaps are performed. However, the next line (`at 1 2 swapped in 2 3.04604 bad = 0`) shows that in alternative 2 of set 1, candidate alternative 2 is swapped in and badness is reduced to zero for this choice set. Recall that with this restrictions macro, badness is only evaluated within the current choice set, so a badness of zero at this point does not mean that the design conforms to all restrictions. This can be seen in the first swap for choice set 2 (`at 2 1 swapped in 1 3.54869 bad = 2`) where the badness for the second choice set is reduced to 2 for the first swap. Subsequent swaps reduce the badness to zero.

Iterations progress through the 8 choice sets until the line (`1 5.945110 0.168205`) shows that at the end of the first iteration (the end of the first full pass through the choice design) the *D*-efficiency is 5.94511 and the *D*-error is 0.16821. For this problem, badness is never more than zero in the second and subsequent iterations, although in general there are no guarantees that all violations will be fixed in the first iteration. The final line of the iteration history (`4 6.54545 0.15278`) displays the final *D*-efficiency and *D*-error. Notice that the number of swaps decreases for each iteration. This is usually the case. If `maxiter=1` had not been specified, this process is repeated with a different initial design selected at random from the candidates.

The design summary and the final efficiency results are as follows:

Final Results

Design	1
Choice Sets	8
Alternatives	4
Parameters	18
Maximum Parameters	24
D-Efficiency	6.5455
Relative D-Eff	81.8181
D-Error	0.1528
1 / Choice Sets	0.1250

n	Variable Name	Label	Variance	DF	Standard Error
1	x11	x1 1	0.13396	1	0.36601
2	x12	x1 2	0.15708	1	0.39633
3	x13	x1 3	0.19401	1	0.44046
4	x21	x2 1	0.16605	1	0.40749
5	x22	x2 2	0.13352	1	0.36540
6	x23	x2 3	0.18423	1	0.42922
7	x31	x3 1	0.24205	1	0.49199
8	x32	x3 2	0.18793	1	0.43351
9	x33	x3 3	0.14790	1	0.38458
10	x41	x4 1	0.17957	1	0.42376
11	x42	x4 2	0.19117	1	0.43723
12	x43	x4 3	0.14291	1	0.37803
13	x51	x5 1	0.18529	1	0.43045
14	x52	x5 2	0.12508	1	0.35367
15	x53	x5 3	0.15012	1	0.38746
16	x61	x6 1	0.16184	1	0.40229
17	x62	x6 2	0.14248	1	0.37747
18	x63	x6 3	0.17398	1	0.41711
				==	
				18	

This construction method creates a design that is 82% efficient relative to the optimal design with no restrictions. With only one iteration, we have no way of knowing how large the value might be with this candidate set. However, our concern at the moment is in evaluating our restrictions macro, not in finding the absolute maximum for *D*-efficiency.

The following steps assign names and levels for the attributes and display the design:

```
proc format;
   value x1f 1='Bad'   2='Good'    3='Better'   4='Best';
   value x2f 1='Small' 2='Average' 3='Bigger'   4='Large';
   value x3f 1='Ugly'  2='OK'      3='Average'  4='Nice ';
   value x4f 1='Slow'  2='Fast'    3='Faster'   4='Fastest';
   value x5f 1='Rough' 2='Normal'  3='Smoother' 4='Smoothest';
   value x6f 1='$9.99' 2='$8.99'   3='$7.99'    4='$6.99';
   run;

proc print label;
   label x1 = 'Quality'   x2 = 'Size'       x3 = 'Appearance'
         x4 = 'Speed'     x5 = 'Smoothness' x6 = 'Price';
   format x1 x1f. x2 x2f. x3 x3f. x4 x4f. x5 x5f. x6 x6f.;
   by set; id set; var x:;
   run;
```

Notice that levels are assigned so that in terms of the original values (1, 2, 3, 4), larger values are always better than smaller values. In particular, notice that the largest price is assigned to the smallest level (1 becomes $9.99) and the smallest price is assigned to the largest level (4 becomes $6.99).

The design is as follows:

Set	Quality	Size	Appearance	Speed	Smoothness	Price
1	Good	Average	OK	Slow	Smoother	$9.99
	Best	Large	Ugly	Faster	Smoothest	$9.99
	Best	Small	OK	Slow	Rough	$8.99
	Bad	Large	Nice	Fastest	Normal	$7.99
2	Best	Average	Nice	Fast	Normal	$9.99
	Better	Large	Average	Slow	Smoothest	$8.99
	Bad	Large	Ugly	Fast	Rough	$8.99
	Bad	Bigger	OK	Faster	Rough	$6.99
3	Good	Small	Ugly	Fastest	Smoother	$7.99
	Best	Large	Nice	Slow	Smoother	$6.99
	Good	Bigger	Ugly	Faster	Normal	$8.99
	Better	Average	Average	Fast	Rough	$7.99
4	Best	Small	Average	Faster	Normal	$7.99
	Best	Average	Ugly	Fastest	Rough	$6.99
	Bad	Small	OK	Fastest	Smoothest	$9.99
	Better	Bigger	Ugly	Fast	Smoother	$9.99
5	Bad	Average	Ugly	Slow	Smoothest	$7.99
	Better	Large	OK	Faster	Smoother	$7.99
	Best	Bigger	Average	Fastest	Smoother	$8.99
	Good	Small	Nice	Fast	Smoothest	$8.99
6	Good	Large	Average	Fastest	Rough	$9.99
	Good	Average	Average	Faster	Smoothest	$6.99
	Best	Bigger	OK	Fast	Smoothest	$7.99
	Better	Small	Nice	Faster	Rough	$9.99
7	Good	Large	OK	Fast	Normal	$6.99
	Best	Large	Ugly	Faster	Smoothest	$9.99
	Better	Small	Ugly	Slow	Normal	$6.99
	Bad	Average	Nice	Faster	Smoother	$8.99
8	Bad	Bigger	Average	Slow	Normal	$9.99
	Good	Bigger	Nice	Slow	Rough	$7.99
	Bad	Small	Average	Fast	Smoother	$6.99
	Better	Average	OK	Fastest	Normal	$8.99

The following steps provide a report and check on dominance:

```
title;
proc iml;
   use best(keep=x1-x6); read all into x;
   sets = 8;
   alts = 4;
   if sets # alts ^= nrow(x) then print 'ERROR: Invalid sets and/or alts.';
   do a = 1 to sets;
      print a[label='Set'] '   '
            (x[((a - 1) * alts + 1) : a * alts,])[format=1.] '          ';
      ii = 0;
      do i = (a - 1) * alts + 1 to a * alts;
         ii = ii + 1;
         kk = ii;
         do k = i + 1 to a * alts;
            kk = kk + 1;
            print ii[label='Alt'] '   ' (x[i,])[format=1.]
                  (sum(x[i,] >= x[k,]))[label='Sum'],
                  kk[label=none]  '   ' (x[k,])[format=1.]
                  (sum(x[k,] >= x[i,]))[label=none];
            if all(x[i,] >= x[k,]) | all(x[k,] >= x[i,]) then
               print "ERROR: Sum=0.";
            end;
         end;
      end;
   quit;
```

You should always double check your design to make sure you wrote your restrictions macro correctly. Some of the results are as follows:

```
Set

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

Alt                       Sum

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

Alt                       Sum

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

```
Alt                       Sum

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

Alt                       Sum

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

Alt                       Sum

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

Alt                       Sum

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

The IML step displays each choice set, each pair of alternatives, and the number of attributes in which each alternative dominates the other. An error is printed if any alternative dominates another alternative in all attributes. For this design, no alternatives dominate.

Now consider for a moment what would happen if you made a mistake in your dominance evaluation macro and specified a set of restrictions that could not be satisfied:

```
%mktex(4 ** 6, n=32, seed=104)

%macro res;
   do i = 1 to nalts;
      do k = i + 1 to nalts;
         if any(x[i,] >= x[k,]) then bad = bad + 1; /* should be all not any */
         if any(x[k,] >= x[i,]) then bad = bad + 1; /* should be all not any */
         end;
      end;
   %mend;

%choiceff(data=randomized,          /* candidate set of choice sets         */
          model=class(x1-x6 / sta), /* model with stdz orthogonal coding    */
          nsets=8,                  /* number of choice sets                */
          flags=4,                  /* 4 alternatives, generic candidates   */
          seed=104,                 /* random number seed                   */
          options=relative          /* display relative D-efficiency        */
                  resrep,           /* detailed report on restrictions      */
          restrictions=res,         /* name of the restrictions macro       */
          resvars=x1-x6,            /* vars used in defining restrictions   */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

Some of the results are as follows:

```
                Design   Iteration  D-Efficiency       D-Error
                ------------------------------------------------
                     1       0           2.69601 *     0.37092
at     1   1 swapped in     1       2.91185 bad =      12
at     1   2 swapped in     2       3.04604 bad =      12
at     1   3 swapped in    11       3.14498 bad =      12
at     1   4 swapped in     3       3.14498 bad =      12
at     1   4 swapped in     9       3.22393 bad =      12
at     1   4 swapped in    20       3.28990 bad =      12
at     1   4 swapped in    27       3.34674 bad =      12
at     2   1 swapped in     1       3.54869 bad =      12
at     2   1 swapped in     2       3.54869 bad =      12
at     2   1 swapped in     3       3.57907 bad =      12
at     2   1 swapped in     5       3.69173 bad =      12
at     2   1 swapped in    21       3.72136 bad =      12
at     2   2 swapped in     1       3.90634 bad =      12
at     2   2 swapped in     5       4.04435 bad =      12
at     2   3 swapped in     1       4.04516 bad =      12
at     2   3 swapped in     3       4.08971 bad =      12
at     2   3 swapped in     4       4.09739 bad =      12
at     2   3 swapped in     8       4.14098 bad =      12
at     2   3 swapped in    12       4.26015 bad =      12
at     2   3 swapped in    20       4.27800 bad =      12
at     2   4 swapped in    29       4.27800 bad =      12
       .
       .
       .
       .
```

```
                    Final Results

Design                 2
Choice Sets            8
Alternatives           4
Parameters            18
Maximum Parameters    24
D-Efficiency       5.9444
Relative D-Eff    74.3044
D-Error            0.1682
1 / Choice Sets    0.1250
WARNING: Restriction violations.
```

n	Variable Name	Label	Variance	DF	Standard Error
1	x11	x1 1	0.21317	1	0.46170
2	x12	x1 2	0.15990	1	0.39988
3	x13	x1 3	0.25107	1	0.50107
4	x21	x2 1	0.20685	1	0.45481
5	x22	x2 2	0.20666	1	0.45460
6	x23	x2 3	0.14918	1	0.38624
7	x31	x3 1	0.21191	1	0.46033
8	x32	x3 2	0.20148	1	0.44886
9	x33	x3 3	0.18835	1	0.43399
10	x41	x4 1	0.16921	1	0.41135
11	x42	x4 2	0.20802	1	0.45610
12	x43	x4 3	0.17490	1	0.41821
13	x51	x5 1	0.16740	1	0.40914
14	x52	x5 2	0.21489	1	0.46356
15	x53	x5 3	0.15014	1	0.38748
16	x61	x6 1	0.18323	1	0.42805
17	x62	x6 2	0.19784	1	0.44479
18	x63	x6 3	0.16916	1	0.41129
				==	
				18	

It is clear from these results that this set of restrictions is impossible and is not met.

Forcing Attributes to be Constant

The following steps again find a restricted design, but with a different set of restrictions:

```
%mktex(4 ** 6, n=32, seed=104)

%macro res;
   c = 0;                          /* n of constant attrs in x            */
   do i = 1 to ncol(x);
      c = c +                      /* n of constant attrs in x            */
         all(x[,i] = round(x[:,i]));/* all values equal average value     */
      end;
   bad = bad + abs(c - 2);         /* want two attrs constant             */
   %mend;
```

```
%choiceff(data=randomized,          /* candidate set of alternatives        */
          model=class(x1-x6 / sta), /* model with stdz orthogonal coding    */
          nsets=8,                  /* number of choice sets                */
          flags=4,                  /* 4 alternatives, generic candidates   */
          seed=104,                 /* random number seed                   */
          options=relative          /* display relative D-efficiency        */
                  resrep,           /* detailed report on restrictions      */
          restrictions=res,         /* name of the restrictions macro       */
          resvars=x1-x6,            /* variable names used in restrictions  */
          maxiter=1,                /* maximum number of designs to make    */
          bestout=desres2,          /* final choice design                  */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

In this case, a `do` statement loops over every column in the choice set. The IML scalar `c` is incremented by one every time all values in column i are equal to the average level value.* In other words, it counts the number of constant columns. The value of `bad` is the absolute difference between `c` and 2. The goal is to create a choice design where exactly two attributes are constant in each choice set.

Part of the iteration history is as follows:

```
                Design   Iteration  D-Efficiency       D-Error
                ------------------------------------------------
                     1           0           2.69601 *     0.37092
          at    1   1 swapped in     1       2.91185 bad =       2
          .
          .
          .
          at    1   2 swapped in     2       3.04604 bad =       2
          at    2   1 swapped in     1       3.32110 bad =       2
          .
          .
          .
          at    2   4 swapped in    29       3.98253 bad =       2
          at    3   1 swapped in     1       4.19037 bad =       2
          .
          .
          .
          at    3   4 swapped in     6       4.75032 bad =       2
          at    4   1 swapped in     1       4.80793 bad =       2
          .
          .
          .
          at    4   4 swapped in    13       5.05268 bad =       2
```

*Note that the expression `x[:,i]` extracts column i from matrix `x` then computes the mean over the rows of the selected column. The colon is a subscript reduction operator that computes the mean.

```
at    5   1 swapped in     1       5.06824 bad =       2
.
.
.
at    5   4 swapped in    19       4.72563 bad =       0
at    6   1 swapped in    11       4.77369 bad =       2
.
.
.
at    6   4 swapped in     7       4.88972 bad =       2
at    7   1 swapped in    16       4.88972 bad =       2
.
.
.
at    7   4 swapped in    24       5.05078 bad =       2
at    8   1 swapped in    10       5.05078 bad =       2
.
.
.
at    8   4 swapped in    30       5.08288 bad =       2
                        1          5.08288 *     0.19674
at    1   3 swapped in    23       5.11786 bad =       1
at    1   4 swapped in    25       5.12802 bad =       1
at    2   1 swapped in    21       5.12802 bad =       2
.
.
.
at    2   4 swapped in    26       5.37339 bad =       2
at    3   1 swapped in     3       5.07267 bad =       2
.
.
.
at    3   4 swapped in     6       5.07267 bad =       1
at    4   2 swapped in     1       5.07730 bad =       2
.
.
.
at    4   3 swapped in    18       5.31648 bad =       2
at    6   1 swapped in    11       5.31648 bad =       2
.
.
.
at    6   4 swapped in     9       5.44566 bad =       2
at    7   1 swapped in    16       5.44839 bad =       2
.
.
.
at    7   4 swapped in    24       5.47225 bad =       2
                        2          5.47225 *     0.18274
```

```
.
.
.
                        8          5.51207       0.18142
at    7   1 swapped in     6       5.51207 bad =       2
at    7   2 swapped in     8       5.51207 bad =       2
at    7   3 swapped in    17       5.51207 bad =       2
at    7   4 swapped in    24       5.51207 bad =       2
                        9          5.51207       0.18142
at    7   1 swapped in     6       5.51207 bad =       2
at    7   2 swapped in     8       5.51207 bad =       2
at    7   3 swapped in    17       5.51207 bad =       2
at    7   4 swapped in    24       5.51207 bad =       2
                       10          5.51207       0.18142

WARNING: Design 1 has TYPES=, OPTIONS=NODUP, or restrictions problems.
```

The macro does not succeed in imposing the restrictions.

The design summary and the final efficiency results are as follows:

Final Results

Design	1
Choice Sets	8
Alternatives	4
Parameters	18
Maximum Parameters	24
D-Efficiency	5.5121
Relative D-Eff	68.9009
D-Error	0.1814
1 / Choice Sets	0.1250

WARNING: Restriction violations.

Again, the output states that there are restriction violations. The following step displays the design:

```
proc print data=desres2; id set; by set; var x:; run;
```

The first two choice sets are as follows:

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

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

Neither choice set has two constant attributes. The `%ChoicEff` macro is very much like the `%MktEx` macro in the way that you must quantify badness. It is often not sufficient to have a badness value that is zero when everything is fine and not zero when things are not fine. Consider `x1` in the second choice set. Imagine changing the 3 to a 2. That moves the attribute closer to constant but has no effect on the badness criterion. The badness criterion is only affected when an attribute with 3 values that are constant is changed to one with 4 values that are constant or an attribute with 4 values that are constant is changed to one with 3 values that are constant. The `%ChoicEff` macro needs to be provided with more guidance than this. Creating constant attributes decreases efficiency, so that is not a direction that the `%ChoicEff` macro tends to go unless you guide it in that direction.

The following steps illustrate one way to guide it:

```
%mktex(4 ** 6, n=128, seed=104, maxdesigns=1, options=nodups)

%macro res;
   c = 0;                           /* n of constant attrs in x            */
   do i = 1 to ncol(x);
      c = c +                       /* n of constant attrs in x            */
         all(x[,i] = round(x[:,i]));/* all values equal average value      */
      end;
   bad = bad + abs(c - 2);          /* want two attrs constant             */
   if c < 2 then do;                /* refine bad if we need more constants */
      do i = 1 to ncol(x);
         bad = bad +                /* count values not at the average     */
               sum(x[,i] ^= round(x[:,i]));
         end;
      end;
   %mend;

%choiceff(data=randomized,          /* candidate set of alternatives       */
          model=class(x1-x6 / sta), /* model with stdz orthogonal coding   */
          nsets=8,                  /* number of choice sets               */
          flags=4,                  /* 4 alternatives, generic candidates  */
          seed=104,                 /* random number seed                  */
          options=relative          /* display relative D-efficiency       */
                  resrep,           /* detailed report on restrictions     */
          restrictions=res,         /* name of the restrictions macro      */
          resvars=x1-x6,            /* variable names used in restrictions */
          maxiter=1,                /* maximum number of designs to make   */
          bestout=desres3,          /* final choice design                 */
          beta=zero)                /* assumed beta vector, Ho: b=0        */

proc print data=desres3; id set; by set; var x:; run;
```

In this example, the restrictions macro begins the same way as before, but when fewer than two attributes are constant, the badness value is increased by the number of values in each of the attributes that are not equal to the average. Now, when any value is changed to a 2 in `x1` in the second choice set, the badness criterion decreases. The `%ChoicEff` macro now knows when it is taking a step in the right direction even if it has not gotten to its ultimate goal yet. This is very important! If you do not let the `%ChoicEff` macro know when it is doing the right thing, it might not succeed in doing what you want. You want the `%ChoicEff` macro to move toward a restricted design with a lower efficiency. The `%ChoicEff` macro tries to move in another direction, towards a more efficient design. When you want the `%ChoicEff` macro to look somewhere than other where it would prefer to look, you need to tell it when it is moving in the right direction. One other change was made to make this example. The `%MktEx` macro was changed to make a larger candidate set, 128 candidates, none of which are duplicates of any other candidate.

Part of the output is as follows:

Final Results

Design	1
Choice Sets	8
Alternatives	4
Parameters	18
Maximum Parameters	24
D-Efficiency	3.5824
Relative D-Eff	44.7803
D-Error	0.2791
1 / Choice Sets	0.1250

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

Now, the restrictions are all correctly imposed. This approach creates a design that is 45% efficient relative to the optimal design with no restrictions. Again, our concern at the moment is in evaluating our restrictions macro, not in finding the absolute maximum for *D*-efficiency.

The following steps jointly impose both sets of restrictions considered in this example so far and evaluate the design:

```
%mktex(4 ** 6, n=128, seed=104, maxdesigns=1, options=nodups)

%macro res;
   do i = 1 to nalts;
      do k = i + 1 to nalts;
         if all(x[i,] >= x[k,])     /* alt i dominates alt k               */
            then bad = bad + 1;
         if all(x[k,] >= x[i,])     /* alt k dominates alt i               */
            then bad = bad + 1;
         end;
      end;
   c = 0;                           /* n of constant attrs in x            */
   do i = 1 to ncol(x);
      c = c +                       /* n of constant attrs in x            */
         all(x[,i] = round(x[:,i]));/* all values equal average value      */
      end;
   bad = bad + abs(c - 2);          /* want two attrs constant             */
   if c < 2 then do;                /* refine bad if we need more constants */
      do i = 1 to ncol(x);
         bad = bad +                /* count values not at the average     */
               sum(x[,i] ^= round(x[:,i]));
         end;
      end;
   %mend;

%choiceff(data=randomized,          /* candidate set of alternatives       */
          model=class(x1-x6 / sta), /* model with stdz orthogonal coding   */
          nsets=8,                  /* number of choice sets               */
          flags=4,                  /* 4 alternatives, generic candidates  */
          seed=104,                 /* random number seed                  */
          options=relative          /* display relative D-efficiency       */
                  resrep,           /* detailed report on restrictions     */
          restrictions=res,         /* name of the restrictions macro      */
          resvars=x1-x6,            /* variable names used in restrictions */
          maxiter=1,                /* maximum number of designs to make   */
          bestout=desres4,          /* final choice design                 */
          beta=zero)                /* assumed beta vector, Ho: b=0        */

proc print data=desres4; id set; by set; var x:; run;
```

```
proc iml;
   use desres4(keep=x1-x6); read all into x;
   sets = 8;
   alts = 4;
   if sets # alts ^= nrow(x) then print 'ERROR: Invalid sets and/or alts.';
   do a = 1 to sets;
      print a[label='Set'] '   '
            (x[((a - 1) * alts + 1):a * alts,])[format=1.] '          ';
      ii = 0;
      do i = (a - 1) * alts + 1 to a * alts;
         ii = ii + 1;
         kk = ii;
         do k = i + 1 to a * alts;
            kk = kk + 1;
            print ii[label='Alt'] '   ' (x[i,])[format=1.]
                  (sum(x[i,] >= x[k,]))[label='Sum'],
                  kk[label=none]  '   ' (x[k,])[format=1.]
                  (sum(x[k,] >= x[i,]))[label=none];
            if all(x[i,] >= x[k,]) | all(x[k,] >= x[i,]) then
               print "ERROR: Sum=0.";
            end;
         end;
      end;
   quit;
```

Part of the output is as follows:

Final Results

Design	1
Choice Sets	8
Alternatives	4
Parameters	18
Maximum Parameters	24
D-Efficiency	3.5175
Relative D-Eff	43.9692
D-Error	0.2843
1 / Choice Sets	0.1250

This approach creates a design that is 44% efficient relative to the optimal design with no restrictions. Our concern at the moment is still in evaluating our restrictions macro, not in finding the absolute maximum for *D*-efficiency. We do not know what the maximum *D*-efficiency is for this design, but with 2 of 8 attributes constrained to be constant, the optimum value must be less than $100 \times 6/8 = 75\%$.

The first two choice sets are as follows:

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

The dominance evaluation results for the first choice set are as follows:

Set		
1	1 1 3 1 3 1	
	3 3 2 1 3 4	
	4 2 2 1 3 4	
	2 4 2 1 3 2	
Alt		**Sum**
1	1 1 3 1 3 1	3
2	3 3 2 1 3 4	5
Alt		**Sum**
1	1 1 3 1 3 1	3
3	4 2 2 1 3 4	5
Alt		**Sum**
1	1 1 3 1 3 1	3
4	2 4 2 1 3 2	5
Alt		**Sum**
2	3 3 2 1 3 4	5
3	4 2 2 1 3 4	5
Alt		**Sum**
2	3 3 2 1 3 4	5
4	2 4 2 1 3 2	4

```
Alt                         Sum

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

The design conforms to all restrictions in every choice set.

Now that we are certain that the design conforms to the restrictions, we can try to find a more efficient design. The following step creates a full-factorial candidate set with $4^6 = 4096$ candidate alternatives and specifies `maxiter=10` in the `%ChoicEff` macro:

```
%mktex(4 ** 6, n=4096, seed=104)

%macro res;
   do i = 1 to nalts;
      do k = i + 1 to nalts;
         if all(x[i,] >= x[k,])     /* alt i dominates alt k               */
            then bad = bad + 1;
         if all(x[k,] >= x[i,])     /* alt k dominates alt i               */
            then bad = bad + 1;
         end;
      end;
   c = 0;                           /* n of constant attrs in x            */
   do i = 1 to ncol(x);
      c = c +                       /* n of constant attrs in x            */
         all(x[,i] = round(x[:,i]));/* all values equal average value      */
      end;
   bad = bad + abs(c - 2);          /* want two attrs constant             */
   if c < 2 then do;                /* refine bad if we need more constants */
      do i = 1 to ncol(x);
         bad = bad +                /* count values not at the average     */
               sum(x[,i] ^= round(x[:,i]));
         end;
      end;
   %mend;

%choiceff(data=randomized,          /* candidate set of alternatives       */
          model=class(x1-x6 / sta), /* model with stdz orthogonal coding   */
          nsets=8,                  /* number of choice sets               */
          flags=4,                  /* 4 alternatives, generic candidates  */
          seed=104,                 /* random number seed                  */
          rscale=8 * 6 / 8,         /* relative D-efficiency scale factor  */
                                    /* 6 of 8 attrs in 8 sets vary         */
          options=resrep,           /* detailed report on restrictions     */
          restrictions=res,         /* name of the restrictions macro      */
          resvars=x1-x6,            /* variable names used in restrictions */
          maxiter=10,               /* maximum number of designs to make   */
          bestout=desres5,          /* final choice design                 */
          beta=zero)                /* assumed beta vector, Ho: b=0        */
```

```
proc print data=desres5; id set; by set; var x:; run;
```

The `rscale=` option specifies that relative *D*-efficiency is not scaled relative to 8 choice sets. Rather, it is scaled relative to a value that is three-quarters as large since only six of eight attributes can vary in each choice set.

Part of the output is as follows:

Final Results

Design	1
Choice Sets	8
Alternatives	4
Parameters	18
Maximum Parameters	24
D-Efficiency	4.6354
Relative D-Eff	77.2566
D-Error	0.2157
1 / Choice Sets	0.1250

This approach takes much longer than the previous approach. The design is better than the one found previously (unscaled *D*-efficiency of 4.64 compared to 3.52 previously). This approach creates a design that is 77% efficient relative to the optimal design with 6 of 8 attributes varying. It is $100 \times 4.6354/8 = 57.94\%$ *D*-efficient relative to the optimal design with no restrictions (compared to 43.7% found previously with the smaller candidate set and fewer iterations).

The final design is as follows:

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

5	2	2	2	2	2	2
	4	2	1	1	3	2
	1	2	4	3	1	2
	3	2	3	1	4	2
6	4	2	2	4	1	2
	1	1	2	4	4	3
	3	3	2	4	2	1
	2	4	2	4	3	4
7	2	3	3	3	1	2
	4	1	4	3	1	4
	3	2	4	3	1	3
	1	4	1	3	1	1
8	2	1	4	2	2	2
	2	2	3	3	2	4
	2	4	2	1	2	1
	2	3	1	4	2	3

The results of the PROC IML step that evaluates the design (not shown) show that the design conforms to all of the restrictions.

Restrictions Within and Across Choice Sets

This example uses a fairly complicated restrictions macro. The goal is to avoid dominated alternatives like before. Another goal is to require certain patterns of constant attributes. Each choice set is required to have one or two constant attributes. Furthermore, each attribute is required to be constant within a specified number of choice sets. Specifically, attributes 1, 3, and 5 are required to be constant in two choice sets and attributes 2, 4, and 6 are required to be constant in one choice set. This last requirement requires more than just simple variations on the technique shown previously.

In this example, like the last example, restrictions are formulated based on `x`, the candidate choice set. However, this example also has restrictions that are defined across choice sets not just within a choice set. Therefore, restrictions are also defined based on `xmat`, the full design. The macro provides you with the value of an index variable, `setnum`, that contains the number of the choice set being worked on. The choice set `x` corresponds to the value of `setnum`. For example, when `setnum` = 3, then `x` is the third choice set. The `setnum` choice set in `xmat` is currently in the design, and the choice set in `x` is being considered as a replacement for it. The scalar `altnum` is available as well, and it contains the number of the alternative that is being changed. This scalar is only available when you are using the algorithm that swaps candidate alternatives. It is not available when you provide a candidate set of choice sets. Two additional scalars are available for you to use: `nalts`, the number of alternatives in the design and `nsets`, the number of choice sets in the design. Using these scalars rather than hard-coded constants makes it easier for you to modify a macro for use in a different situation.

The following step creates a candidate set of 256 alternatives with no duplicates:

```
%mktex(4 ** 6, n=256, seed=104, maxdesigns=1, options=nodups)
```

The following step creates the restrictions macro:

```
%macro res;
   do i = 1 to nalts;
      do k = i + 1 to nalts;
         if all(x[i,] >= x[k,])     /* alt i dominates alt k               */
            then bad = bad + 1;
         if all(x[k,] >= x[i,])     /* alt k dominates alt i               */
            then bad = bad + 1;
         end;
      end;

   nattrs = ncol(x);                /* number of columns in design         */
   v = j(1, nattrs, 0);             /* n of constant attrs across sets     */
   c = 0;                           /* n of constant attrs within set      */

   do i = 1 to nattrs;              /* loop over all attrs                 */
      a    = all(x[,i] = x[1,i]);   /* 1 - attr i constant, 0 - varying    */
      c    = c + a;                 /* n of constant attrs within set      */
      v[i] = v[i] + a;              /* n of constant attrs across sets     */
      end;

   if c > 2 | c = 0 then            /* want 1 or 2 constant attrs in a set */
      bad = bad + 10 # abs(c - 2);  /* weight of 10 prevents trade offs    */

   do s = 1 to nsets;               /* loop over rest of design            */
      if s ^= setnum then do;       /* skip xmat part that corresponds to x */
         z = xmat[((s-1)*nalts+1) : /* pull out choice set s              */
                  (s * nalts),];
         do i = 1 to nattrs;        /* loop over attrs                     */
            v[i] = v[i] +           /* n of constant attrs across sets     */
                   all(z[,i] = z[1,i]);
            end;
         end;
      end;

   d   = abs(v - {2 1 2 1 2 1})[+]; /* see if constant attrs match target  */
   bad = bad + d;                   /* increase badness                    */
   if d then do;                    /* if not at target, fine tune badness */
      do i = 1 to nattrs;           /* loop over attrs                     */
         bad = bad +                /* add to badness as attrs are farther */
               (x[,i] ^= x[1,i])[+];/* from constant                       */
         end;
      end;
   %mend;
```

Note that v is a vector with m elements, one for each attribute. The statements `v[i] = v[i] + a` and `v[i] = v[i] + all(z[,i] = z[1,i])` add one to the *ith* element of v whenever the *ith* attribute in x or xmat is constant. Now consider the statement: `d = abs(v - {2 1 2 1 2 1})[+]`. The expression `abs(v - {2 1 2 1 2 1})` creates a vector with m elements containing the absolute differences between the counts of the number of constant attributes and the target counts. The subscript reduction operator `[+]` adds up the absolute differences, then the result is stored in d. When d is zero, the right number of attributes are constant across choice sets. Otherwise, d is a measure of how far the design is from conforming to the restrictions. The measure of design badness must be more sensitive than this. When d is not zero, badness is increased for every value that is different from constant. This way, the %ChoicEff macro knows when it is moving in the right direction toward making more constant attributes. The statement `bad = bad + (x[,i] ^= x[1,i])[+]` creates a vector (`x[,i] ^= x[1,i]`) with ones for all values that are not equal to the first value and zeros for values that are equal. Then the values in this vector are summed by the subscript reduction operator to provide a count of the number of values not equal to the first value, and badness is increased by that amount.

The statement `bad = bad + 10 # abs(c - 2)` weights the number of constant attributes within a choice set with a weight of 10.* This weight is greater than the implicit weights of one for the other components of the badness function. This means that minimizing this source of badness takes precedence over minimizing other sources, and there won't be any trade offs between this source and other sources. Sometimes, when there are multiple sources of badness, it is important to differentially weight them. It might not be important how you weight them or which source gets the most weight. Simply providing some differential weighting helps the %ChoicEff macro figure out how to impose all of the restrictions. Otherwise the %ChoicEff macro might get stuck increasing one source of badness every time it decreases another. Weighting can also help you interpret options=resrep results.

The following step searches the candidate set of alternatives and creates the restricted choice design:

```
%choiceff(data=randomized,          /* candidate set of alternatives        */
          model=class(x1-x6 / sta), /* model with stdz orthogonal coding    */
          nsets=8,                  /* number of choice sets                */
          flags=4,                  /* 4 alternatives, generic candidates   */
          seed=104,                 /* random number seed                   */
          options=relative          /* display relative D-efficiency        */
                  resrep,           /* detailed report on restrictions      */
          restrictions=res,         /* name of the restrictions macro       */
          resvars=x1-x6,            /* variable names used in restrictions  */
          maxiter=1,                /* maximum number of designs to make    */
          bestout=desres6,          /* final choice design                  */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

*The IML operator # performs scalar multiplication.

Part of the output is as follows:

Final Results

Design	1
Choice Sets	8
Alternatives	4
Parameters	18
Maximum Parameters	24
D-Efficiency	5.0372
Relative D-Eff	62.9645
D-Error	0.1985
1 / Choice Sets	0.1250

This step creates a design that is 63% efficient relative to an optimal design with no restrictions.

The following step displays the design:

```
proc print data=desres6; id set; by set; var x:; run;
```

The results are as follows:

Set	x1	x2	x3	x4	x5	x6
1	4	3	1	4	1	3
	1	3	1	2	4	1
	3	3	1	4	3	4
	4	3	1	1	2	2
2	4	3	1	3	3	4
	2	1	2	3	3	3
	1	4	3	3	4	2
	3	1	4	3	1	3
3	4	2	4	3	3	3
	4	3	2	2	2	1
	4	4	1	1	2	4
	4	1	3	4	4	2
4	2	3	4	4	1	4
	3	1	2	3	4	4
	3	2	3	1	3	4
	1	2	4	4	2	4
5	4	2	4	2	4	3
	3	4	4	3	1	1
	2	3	4	4	2	3
	1	4	4	2	3	4

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

7       2       3       4       1       4       4
        2       1       1       3       2       3
        2       4       2       4       4       1
        2       2       1       2       1       2

8       3       4       1       2       4       3
        1       3       2       3       4       2
        4       1       3       4       4       1
        2       2       4       1       4       4
```

Choice set one has 2 constant attributes and choice sets 2 through 8 have 1 constant attribute. Choice sets 1, 3, and 5 have two constant attributes within choice set. Choice sets 2, 4, and 6 have one constant attributes within choice set. No alternatives are dominated. Now that we are certain that our restrictions macro is correct, we can make a full-factorial candidate set as follows:

```
%mktex(4 ** 6, n=4096)
```

Using the same restrictions macro and `%ChoicEff` macro call as before (one iteration), the results are as follows:

Final Results

Design	1
Choice Sets	8
Alternatives	4
Parameters	18
Maximum Parameters	24
D-Efficiency	5.6758
Relative D-Eff	70.9478
D-Error	0.1762
1 / Choice Sets	0.1250

The *D*-efficiency is 71% compared to 63% previously. With more iteration (increasing the value of `maxiter=`), you would expect that value to increase.

There are many things that can go wrong when you are creating restricted choice designs. You might write a restrictions macro that is internally contradictory or otherwise write a set of restrictions that cannot possibly be satisfied. The `%ChoicEff` macro cannot analyze these problems for you, but it can tell you when it fails to meet restrictions. You might write a restrictions macro that is correct but fails to provide the `%ChoicEff` macro the guidance that it needs. You might instead create a candidate set that is too small and limited. You might need to create a larger candidate set so that the macro has more freedom to find an efficient design. However, you do not want to start with a candidate set that is too large at first, because it takes a long time to search large candidate sets, particularly when there

are restrictions. Often there is some trial and error involved in creating the right restrictions macro and the right set of candidates.

Restrictions Within and Across Choice Sets with Candidate Set Swapping

This next example tackles the same problem as the previous example, but this time we use the approach of creating a candidate set of choice sets instead of a candidate set of alternatives. This next example is *not* the recommended approach for this example or most other examples now that the `restrictions=` option is implemented in the `%ChoicEff` macro. It is simply provided here for completeness and because it illustrates important differences between the two approaches. Most of the previous examples created a candidate set of alternatives. With 6 four-level factors, there are $4^6 = 4096$ possible candidate alternatives. From those 4096 possible alternatives, all of the $4096^4 = 281,474,976,710,656$ (281 trillion) possible choice sets can potentially be constructed. In practice, only a tiny fraction of them are considered (perhaps several thousand or a few million), but all are possible. In contrast, in the choice set swapping algorithm, you must create a candidate set of choice sets, so only a few thousand choice sets at the most can be considered for most problems. It is all but guaranteed that the alternative swapping algorithm will do better than the choice set swapping algorithm.

You begin using the candidate set swapping algorithm by creating a candidate set of choice sets. You need to impose the within-choice-set restrictions at this point. The following steps create a candidate set of choice sets:

```
%macro res2;

   nattrs = 6;                      /* 6 attributes                          */
   nalts  = 4;                      /* 4 alternatives                        */
   z = shape(x, nalts, nattrs);     /* rearrange x to look like a choice set*/

   do ii = 1 to nalts;
      do k = ii + 1 to nalts;
         if all(z[ii,] >= z[k,])    /* alt ii dominates alt k               */
            then bad = bad + 1;
         if all(z[k,] >= z[ii,])    /* alt k dominates alt ii               */
            then bad = bad + 1;
         end;
      end;

   c = 0;                           /* n of constant attrs within set       */
   do ii = 1 to nattrs;             /* loop over all attrs                  */
      c = c +                       /* n of constant attrs within set       */
          all(z[,ii] = z[1,ii]);    /* 1 - attr i constant, 0 - varying     */
      end;
   if c > 2 | c = 0 then            /* want 1 or 2 constant attrs in a set  */
      bad = bad + 10 # abs(c - 2);  /* weight of 10 prevents trade offs     */
   %mend;
```

```
%mktex(4 ** 24, n=200, restrictions=res2, seed=104,
       target=90, options=quickr resrep, order=random)

%mktkey(4 6)

%mktroll(design=randomized, key=key, out=rolled)
```

The restrictions macro is very similar to the within-choice-set part of the previous restrictions macro. The first difference is due to the fact that the `%MktEx` macro is creating a linear arrangement of the attributes—one in which all attributes of all alternatives are arranged in a single row. The statement `z = shape(x, nalts, nattrs)` rearranges `x` into a matrix with one row for each alternative and one column for each attribute. This is not necessary, but it enables us to impose the restrictions using similar code to that which was used previously (only now based on `z` instead of `x`). Another difference is that the index `i` had been replaced with `ii`. In a `%MktEx` macro restrictions macro, `i` is the row number being worked on and you cannot change it.

The `%MktEx` macro makes a design with 24 factors (four alternatives times six attributes). It specifies the name of the restrictions macro in the `restrictions=res2` option. The option `target=90` specifies that iteration can stop when all restrictions are met and the design is 90% efficient. Since this is a candidate set, we do not need to maximize *D*-efficiency at this stage. The option `options=quickr` makes one design quickly using the coordinate exchange algorithm and a random initialization. The option `options=resrep` provides a detailed report of how the design is conforming to the restrictions. The option `order=random` loops over the columns in a random order. A candidate set of choice sets with 80 candidates is created. The `%MktKey` macro creates the rules for turning a linear arrangement into a choice design, and the `%MktKey` macro creates the candidate set of choice sets.

The following step creates the restrictions macro for the `%ChoicEff` macro:

```
%macro res;
   nattrs = ncol(x);                /* number of columns in design          */
   v = j(1, nattrs, 0);             /* n of constant attrs across sets      */
   do s = 1 to nsets;               /* loop over each choice set            */
      if s ^= setnum then           /* pull choice set out of xmat          */
         z = xmat[((s - 1) * nalts + 1) : (s * nalts),];
      else z = x;                   /* if current set, get from x           */
      do i = 1 to nattrs;           /* loop over attrs                      */
         v[i] = v[i] +              /* n of constant attrs across sets      */
                all(z[,i] = z[1,i]);
         end;
      end;
   bad = abs(v - {2 1 2 1 2 1})[+]; /* see if constant attrs match target   */
   %mend;
```

Since the candidate set of choice sets already has the within-choice-set restrictions imposed, only the between-choice-set restrictions are imposed. Furthermore, the code at the end of the macro (`if d then do; do i = 1 to nattrs; bad = bad + (x[,i] ^= x[1,i])[+]; end; end;`) is removed. There is no opportunity to refine a candidate choice set. It either conforms to the restrictions or it does not. That code only serves to obfuscate the badness criterion in this example with the candidate choice set search algorithm.

The following step searches for the design:

```
%choiceff(data=rolled,              /* candidate set of choice sets         */
          model=class(x1-x6 / sta), /* model with stdz orthogonal coding    */
          nsets=8,                  /* number of choice sets                */
          nalts=4,                  /* number of alternatives               */
          seed=104,                 /* random number seed                   */
          options=relative          /* display relative D-efficiency        */
                  resrep,           /* detailed report on restrictions      */
          restrictions=res,         /* name of the restrictions macro       */
          resvars=x1-x6,            /* variable names used in restrictions  */
          maxiter=2,                /* maximum number of designs to make    */
          bestout=desres7,          /* final choice design                  */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

Since a candidate set of choice sets is searched and not a candidate set of alternatives, the `nalts=` option is used instead of the `flags=` option. Some of the results are as follows:

Final Results

Design	2
Choice Sets	8
Alternatives	4
Parameters	18
Maximum Parameters	24
D-Efficiency	4.3467
Relative D-Eff	54.3337
D-Error	0.2301
1 / Choice Sets	0.1250

The design is 54% efficient compared to 71% previously. With 400 candidate choice sets (not shown) the design is 56% efficient.

The following steps display and evaluate the design:

```
proc print data=desres7; id set; by notsorted set; var x:; run;
```

```
proc iml;
   use desres7(keep=x1-x6); read all into x;
   sets = 8;
   alts = 4;
   if sets # alts ^= nrow(x) then print 'ERROR: Invalid sets and/or alts.';
   do a = 1 to sets;
      print a[label='Set'] '   '
            (x[((a - 1) * alts + 1):a * alts,])[format=1.] '          ';
      ii = 0;
      do i = (a - 1) * alts + 1 to a * alts;
         ii = ii + 1;
         kk = ii;
         do k = i + 1 to a * alts;
            kk = kk + 1;
            print ii[label='Alt'] '   ' (x[i,])[format=1.]
                  (sum(x[i,] >= x[k,]))[label='Sum'],
                  kk[label=none]  '   ' (x[k,])[format=1.]
                  (sum(x[k,] >= x[i,]))[label=none];
            if all(x[i,] >= x[k,]) | all(x[k,] >= x[i,]) then
               print "ERROR: Sum=0.";
            end;
         end;
      end;
   quit;
```

The design is as follows:

Set	x1	x2	x3	x4	x5	x6
56	3	4	4	2	3	1
	2	2	3	2	1	2
	1	2	1	2	2	3
	3	4	2	2	4	2
106	2	4	2	1	2	4
	4	2	2	2	1	4
	2	3	1	3	3	4
	1	3	3	4	4	4
92	2	3	4	1	2	3
	2	2	4	2	3	1
	2	4	3	3	4	4
	2	1	1	4	3	2
96	1	4	2	4	1	3
	4	1	2	2	2	4
	2	2	2	1	1	1
	1	3	2	1	3	2

```
185     1     4     3     4     4     4
        4     3     1     1     4     2
        2     3     2     2     4     3
        4     1     2     3     4     1

131     4     4     2     4     4     1
        1     4     4     1     4     2
        3     4     4     2     4     1
        3     4     2     3     4     3

 34     4     2     3     1     1     1
        3     2     3     4     2     4
        2     1     3     1     3     3
        1     4     3     2     4     2

182     1     2     4     3     4     3
        1     3     3     4     2     1
        1     2     2     1     4     4
        1     3     4     3     1     1
```

The design conforms to all restrictions, but again, this is not the recommended approach for this problem.

Brand Effects

This next example creates a design with a brand factor. There are three brands, three additional attributes, and three alternatives. A choice set has nine values: three alternatives times three attributes. The goal is to restrict the design so that each level occurs between three and five times in each of the nine positions across all choice sets. In other words, the goal is to ensure a nearly balanced choice design.

The following steps make and display a candidate set of alternatives:

```
%mktex(3 ** 3, n=3**3)

data full;
   Set + 1;
   Brand = 0;
   set design;
   retain f1-f3 0;
   array f[3];
   do brand = 1 to 3;
      f[brand] = 1; output; f[brand] = 0;
      end;
   run;

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

A few of the candidate alternatives are as follows:

Set	Brand	x1	x2	x3	f1	f2	f3
1	1	1	1	1	1	0	0
	2	1	1	1	0	1	0
	3	1	1	1	0	0	1
2	1	1	1	2	1	0	0
	2	1	1	2	0	1	0
	3	1	1	2	0	0	1
3	1	1	1	3	1	0	0
	2	1	1	3	0	1	0
	3	1	1	3	0	0	1
.							
.							
.							
27	1	3	3	3	1	0	0
	2	3	3	3	0	1	0
	3	3	3	3	0	0	1

The `%MktEx` macro makes the full-factorial candidate set of alternatives for all of the attributes except brand. The DATA step adds a choice set number, which is not needed, but it is useful for displaying the candidate set. Three brands are also added. The initial `Brand = 0` statement positions the brand variable after the set variable and before the other variables and sets the case that SAS uses to display the name. Three flag variables are added to the design. Brand one alternatives are flagged by `f1 = 1 f2 = 0 f3 = 0`, brand two alternatives are flagged by `f1 = 0 f2 = 1 f3 = 0`, and brand three alternatives are flagged by `f1 = 0 f2 = 0 f3 = 1`. The flag variables are retained so that the all zero values from the previous candidate are available as initial values for the next candidate. Each candidate

alternative is written out three times, once for each brand.

The following step creates the restrictions macro:

```
%macro res;
   c = j(9, 3, 0);                  /* counts - nine positions x 3 levels   */
   do s = 1 to nsets;               /* loop over sets                       */
      if s = setnum then z = x;     /* get choice set from x or xmat        */
      else z = xmat[((s - 1) # nalts + 1) : s # nalts,];
      k = 0;                        /* index into count matrix c            */
      do j = 1 to ncol(z);          /* loop over attributes                 */
         do i = 1 to nalts;         /* loop over alternatives               */
            k = k + 1;              /* index into the next row of c         */
            a = z[i,j];             /* index into c for the z[i,j] level    */
            c[k,a] = c[k,a] + 1;    /* add one to count                     */
            end;
         end;
      end;
   bad = sum((c < 3) # abs(c - 3)) +/* penalty for counts being less than 3 */
         sum((c > 5) # abs(c - 5)); /* penalty for counts greater than 5    */
   %mend;
```

A matrix `c` of counts is initialized to zero. It has nine rows for each of the nine positions in a choice set and three columns for the three levels of each attribute. The `do` statement loops over all of the choice sets. When the index `s` matches the choice set being worked on, the current choice set `x` is stored in `z`. Otherwise, the relevant choice set is extracted from `xmat` and is stored in `z`. The next two `do` statements loop over the nine positions in the choice set. The positions are indexed by `k`. When the level is `a` ($a = 1, 2, 3$) then the *ath* column of `c` is incremented by one to count how often the *ath* level occurs in each of the nine positions. Badness is a function of the number of counts less than three and the number of counts greater than five. Specifically, the operation `(c < 3)` produces a matrix of zeros and ones. When the (i, j) element of c is less than three, the (i, j) element of `(c < 3)` is 1, and it is zero otherwise. When the (i, j) element of `c` is less than three, the (i, j) element of `(c < 3) # abs(c - 3)` is the absolute deviation between `c[i,j]` and three, and it is zero otherwise. The operation `#` performs element-wise multiplication. The sum of the elements in `(c < 3) # abs(c - 3)` and `(c > 5) # abs(c - 5)` provides a measure of how far the design is from having values in the right range.

The following step searches for the design:

```
%choiceff(data=full,                /* candidate set of alternatives        */
          model=class(brand)        /* brand effects                        */
                class(brand*x1      /* alternative-specific effects         */
                      brand*x2
                      brand*x3 /
                      zero=' '),    /* use all brands in these effects      */
          nsets=12,                 /* number of choice sets                */
          seed=104,                 /* random number seed                   */
          options=resrep,           /* detailed report on restrictions      */
          restrictions=res,         /* name of the restrictions macro       */
          resvars=x1-x3,            /* variable names used in restrictions  */
          flags=f1-f3,              /* flag which alt can go where, 3 alts  */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

Some of the results are as follows:

Final Results

Design	2
Choice Sets	12
Alternatives	3
Parameters	20
Maximum Parameters	24
D-Efficiency	0.5163
D-Error	1.9369

n	Variable Name	Label	Variance	DF	Standard Error
1	Brand1	Brand 1	7.57495	1	2.75226
2	Brand2	Brand 2	7.16595	1	2.67693
3	Brand1x11	Brand 1 * x1 1	3.24298	1	1.80083
4	Brand1x12	Brand 1 * x1 2	2.85037	1	1.68830
5	Brand2x11	Brand 2 * x1 1	3.05369	1	1.74748
6	Brand2x12	Brand 2 * x1 2	2.73221	1	1.65294
7	Brand3x11	Brand 3 * x1 1	3.13116	1	1.76951
8	Brand3x12	Brand 3 * x1 2	3.38122	1	1.83881
9	Brand1x21	Brand 1 * x2 1	3.07311	1	1.75303
10	Brand1x22	Brand 1 * x2 2	2.74792	1	1.65769
11	Brand2x21	Brand 2 * x2 1	3.19973	1	1.78878
12	Brand2x22	Brand 2 * x2 2	2.85195	1	1.68877
13	Brand3x21	Brand 3 * x2 1	2.79820	1	1.67278
14	Brand3x22	Brand 3 * x2 2	3.32705	1	1.82402
15	Brand1x31	Brand 1 * x3 1	3.31434	1	1.82053
16	Brand1x32	Brand 1 * x3 2	2.75006	1	1.65833
17	Brand2x31	Brand 2 * x3 1	2.90460	1	1.70429
18	Brand2x32	Brand 2 * x3 2	3.23177	1	1.79771
19	Brand3x31	Brand 3 * x3 1	3.33634	1	1.82656
20	Brand3x32	Brand 3 * x3 2	2.46393	1	1.56969
				==	
				20	

The following step displays the design:

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

The results are as follows:

Set	Brand	x1	x2	x3
1	1	1	3	2
	2	2	3	1
	3	3	3	3
2	1	3	3	2
	2	3	2	3
	3	1	3	2
3	1	3	2	1
	2	2	3	3
	3	1	1	3
4	1	2	2	3
	2	1	3	1
	3	1	3	1
5	1	3	3	3
	2	2	2	2
	3	2	1	1
6	1	2	3	1
	2	3	3	2
	3	2	2	2
7	1	2	1	2
	2	3	1	1
	3	2	1	3
8	1	3	1	3
	2	3	1	3
	3	3	2	1
9	1	2	3	3
	2	2	1	1
	3	1	1	2
10	1	1	2	3
	2	3	2	1
	3	1	2	3
11	1	1	2	2
	2	1	1	2
	3	3	1	2
12	1	1	1	1
	2	1	2	3
	3	2	3	2

The following step evaluates the design:

```
proc iml;
   nsets = 12;  nalts = 3;
   use best(keep=x:); read all into xmat;
   c = j(9, 3, 0);                  /* counts - nine positions x 3 levels   */
   do s = 1 to nsets;               /* loop over sets                       */
      z = xmat[((s - 1) # nalts + 1) : s # nalts,];
      k = 0;                        /* index into count matrix c            */
      do j = 1 to ncol(z);          /* loop over attributes                 */
         do i = 1 to nalts;         /* loop over alternatives               */
            k = k + 1;              /* index into next row of c             */
            a = z[i,j];             /* index into c for the z[i,j] level    */
            c[k,a] = c[k,a] + 1;    /* add one to count                     */
            end;
         end;
      end;
   print c[format=2.];
   quit;
```

The results are as follows:

```
                    c

          4  4  4
          3  4  5
          5  4  3
          3  4  5
          4  4  4
          5  3  4
          3  4  5
          5  3  4
          3  5  4
```

All of the counts are in the right range. In each position, the counts always sum to 12, the number of choice sets.

You can force all of the counts to be exactly four as follows:

```
%macro res;
   c = j(9, 3, 0);                  /* counts - nine positions x 3 levels   */
   do s = 1 to nsets;               /* loop over sets                       */
      if s = setnum then z = x;     /* get choice set from x or xmat        */
      else z = xmat[((s - 1) # nalts + 1) : s # nalts,];
      k = 0;                        /* index into count matrix c            */
      do j = 1 to ncol(z);          /* loop over attributes                 */
         do i = 1 to nalts;         /* loop over alternatives               */
            k = k + 1;              /* index into the next row of c         */
            a = z[i,j];             /* index into c for the z[i,j] level    */
            c[k,a] = c[k,a] + 1;    /* add one to count                     */
            end;
         end;
      end;
   bad = sum(abs(c - 4));           /* penalty for counts not at 4          */
   %mend;

%choiceff(data=full,                /* candidate set of alternatives        */
          model=class(brand)        /* brand effects                        */
                class(brand*x1      /* alternative-specific effects         */
                      brand*x2
                      brand*x3 /
                      zero=' '),    /* use all brands in these effects      */
          nsets=12,                 /* number of choice sets                */
          seed=104,                 /* random number seed                   */
          options=resrep,           /* detailed report on restrictions      */
          restrictions=res,         /* name of the restrictions macro       */
          resvars=x1-x3,            /* variable names used in restrictions  */
          flags=f1-f3,              /* flag which alt can go where, 3 alts  */
          beta=zero)                /* assumed beta vector, Ho: b=0         */

proc iml;
   nsets = 12;  nalts = 3;
   use best(keep=x:); read all into xmat;
   c = j(9, 3, 0);                  /* counts - nine positions x 3 levels   */
   do s = 1 to nsets;               /* loop over sets                       */
      z = xmat[((s - 1) # nalts + 1) : s # nalts,];
      k = 0;                        /* index into count matrix c            */
      do j = 1 to ncol(z);          /* loop over attributes                 */
         do i = 1 to nalts;         /* loop over alternatives               */
            k = k + 1;              /* index into next row of c             */
            a = z[i,j];             /* index into c for the z[i,j] level    */
            c[k,a] = c[k,a] + 1;    /* add one to count                     */
            end;
         end;
      end;
   print c[format=2.];
   quit;
```

Some of the results are as follows:

Final Results

Design	2
Choice Sets	12
Alternatives	3
Parameters	20
Maximum Parameters	24
D-Efficiency	0.3688
D-Error	2.7117

n	Variable Name	Label	Variance	DF	Standard Error
1	Brand1	Brand 1	7.9094	1	2.81236
2	Brand2	Brand 2	37.7097	1	6.14083
3	Brand1x11	Brand 1 * x1 1	22.2501	1	4.71700
4	Brand1x12	Brand 1 * x1 2	9.8908	1	3.14497
5	Brand2x11	Brand 2 * x1 1	4.1258	1	2.03121
6	Brand2x12	Brand 2 * x1 2	13.0844	1	3.61724
7	Brand3x11	Brand 3 * x1 1	5.6523	1	2.37746
8	Brand3x12	Brand 3 * x1 2	9.0392	1	3.00653
9	Brand1x21	Brand 1 * x2 1	7.2808	1	2.69830
10	Brand1x22	Brand 1 * x2 2	11.7998	1	3.43508
11	Brand2x21	Brand 2 * x2 1	13.1864	1	3.63130
12	Brand2x22	Brand 2 * x2 2	8.2109	1	2.86546
13	Brand3x21	Brand 3 * x2 1	3.0455	1	1.74514
14	Brand3x22	Brand 3 * x2 2	11.8138	1	3.43713
15	Brand1x31	Brand 1 * x3 1	7.0919	1	2.66305
16	Brand1x32	Brand 1 * x3 2	5.9528	1	2.43983
17	Brand2x31	Brand 2 * x3 1	13.6297	1	3.69185
18	Brand2x32	Brand 2 * x3 2	9.2488	1	3.04119
19	Brand3x31	Brand 3 * x3 1	5.0102	1	2.23835
20	Brand3x32	Brand 3 * x3 2	11.5376	1	3.39670
				==	
				20	

```
   c

4  4  4
4  4  4
4  4  4
4  4  4
4  4  4
4  4  4
4  4  4
4  4  4
4  4  4
```

It is instructive to compare the variances of the parameter estimates for the two designs. In the second design, when the levels are more constrained, the variances are much larger and more variable. This is usually the sign of a bad design. There is a risk with constraints that you are hurting the efficiency and hence inflating some or all of the variances. For comparison, you can find the efficiency for the unconstrained design as follows:

```
%choiceff(data=full,              /* candidate set of alternatives       */
          model=class(brand)      /* brand effects                       */
                class(brand*x1    /* alternative-specific effects        */
                      brand*x2
                      brand*x3 /
                      zero=' '),  /* use all brands in these effects     */
          nsets=12,               /* number of choice sets               */
          seed=104,               /* random number seed                  */
          flags=f1-f3,            /* flag which alt can go where, 3 alts */
          beta=zero)              /* assumed beta vector, Ho: b=0        */
```

Some of the results are as follows:

Final Results

Design	1
Choice Sets	12
Alternatives	3
Parameters	20
Maximum Parameters	24
D-Efficiency	0.5099
D-Error	1.9611

n	Variable Name	Label	Variance	DF	Standard Error
1	Brand1	Brand 1	8.51332	1	2.91776
2	Brand2	Brand 2	8.90879	1	2.98476
3	Brand1x11	Brand 1 * x1 1	3.34578	1	1.82915
4	Brand1x12	Brand 1 * x1 2	3.65006	1	1.91051

5	Brand2x11	Brand 2 * x1 1	3.35948	1	1.83289
6	Brand2x12	Brand 2 * x1 2	3.52737	1	1.87813
7	Brand3x11	Brand 3 * x1 1	3.64162	1	1.90830
8	Brand3x12	Brand 3 * x1 2	3.63366	1	1.90622
9	Brand1x21	Brand 1 * x2 1	2.83014	1	1.68230
10	Brand1x22	Brand 1 * x2 2	2.99478	1	1.73054
11	Brand2x21	Brand 2 * x2 1	3.63553	1	1.90671
12	Brand2x22	Brand 2 * x2 2	3.28224	1	1.81169
13	Brand3x21	Brand 3 * x2 1	2.94706	1	1.71670
14	Brand3x22	Brand 3 * x2 2	3.27390	1	1.80939
15	Brand1x31	Brand 1 * x3 1	3.34840	1	1.82986
16	Brand1x32	Brand 1 * x3 2	2.60489	1	1.61397
17	Brand2x31	Brand 2 * x3 1	2.67943	1	1.63690
18	Brand2x32	Brand 2 * x3 2	3.92194	1	1.98039
19	Brand3x31	Brand 3 * x3 1	2.72245	1	1.64998
20	Brand3x32	Brand 3 * x3 2	3.21863	1	1.79405
				==	
				20	

Relative to the unconstrained design, the design with constraints in the range of three to five is $100 \times 0.5163/0.5099 = 101\%$ efficient. Relative to the unconstrained design, the design with constraints of four is $100 \times 0.3688/0.5099 = 72\%$ efficient. With more iterations (not shown), these numbers become: $100 \times 0.5172/0.5191 = 99.6\%$ efficient and $100 \times 0.4454/0.5191 = 85.8\%$ efficient.

Brand Effects and the Alternative-Swapping Algorithm

This next example has brand effects and uses the alternative-swapping algorithm. It also illustrates properties of the standardized orthogonal contrast coding and relative *D*-efficiency when a 100% efficient design does not exist. The following steps make and display a candidate set of branded alternatives:

```
%mktex(3 ** 4, n=3**4)

%mktlab(data=design, vars=x1-x3 Brand)

data full(drop=i);
   set final;
   array f[3];
   do i = 1 to 3; f[i] = (brand eq i); end;
   run;

proc print data=full(obs=9); run;
```

The `%MktEx` macro makes the linear candidate design. The `%MktLab` macro changes the name of the variable `x4` to `Brand` while retaining the original names for `x1-x3` and original levels (1, 2, 3) for all factors. The DATA step creates the flags. The flag variable, `f1`, flags brand 1 candidates as available for the first alternative. Similarly, `f2` flags brand 2 candidates as available for the second alternative, and so on. The Boolean expression `(brand eq i)` evaluates to 1 if true and 0 if false.

The first part of the candidate set is as follows:

Obs	x1	x2	x3	Brand	f1	f2	f3
1	1	1	1	1	1	0	0
2	1	1	1	2	0	1	0
3	1	1	1	3	0	0	1
4	1	1	2	1	1	0	0
5	1	1	2	2	0	1	0
6	1	1	2	3	0	0	1
7	1	1	3	1	1	0	0
8	1	1	3	2	0	1	0
9	1	1	3	3	0	0	1

Notice that the candidate set consists of branded alternatives with flags such that only brand i is considered for the *ith* alternative of each choice set.

The following `%ChoicEff` macro step makes the choice design from the candidate set of alternatives:

```
%choiceff(data=full,                /* candidate set of alternatives        */
                                    /* alternative-specific effects model   */
                                    /* zero=' ' no reference level for brand*/
                                    /* brand*x1 ... interactions            */
          model=class(brand brand*x1 brand*x2 brand*x3 / zero=' ' sta),
          nsets=15,                 /* number of choice sets                */
          flags=f1-f3,              /* flag which alt can go where, 3 alts  */
          seed=151,                 /* random number seed                   */
          converge=1e-12,           /* convergence criterion                */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

The `model=` specification states that `Brand` and `x1-x3` are classification or categorical variables and brand effects and brand by attribute interactions (which are also known as alternative-specific effects, see page 386) are desired. The `zero=' '` specification is like `zero=none` except `zero=none` applies to all factors in the specification whereas `zero=' '` applies to just the first. See page 78 for more information about the `zero=` option. This `zero=' '` specification specifies that there is no reference level for the first factor (`Brand`), and the last level will by default be the reference category for the other factors (`x1-x3`). Hence, the interactions are derived from indicator variables created for all three brands, but only two coded variables for the 3 three-level attributes. We need to do this because we need the alternative-specific effects for all brands, including Brand 3. A standardized orthogonal contrast coding is used for `x1-x3` but not for `Brand` (which uses less-than-full-rank indicators).

Some of the results are as follows:

Design	Iteration	D-Efficiency	D-Error
1	0	0	.
	1	0	.
		1.23291 (Ridged)	
	2	0	.
		1.24083 (Ridged)	
	3	0	.
		1.24689 (Ridged)	
	4	0	.
		1.25318 (Ridged)	
	5	0	.
		1.25318 (Ridged)	

Design	Iteration	D-Efficiency	D-Error
2	0	0	.
	1	0	.
		1.21367 (Ridged)	
	2	0	.
		1.24462 (Ridged)	
	3	0	.
		1.24565 (Ridged)	
	4	0	.
		1.24708 (Ridged)	
	5	0	.
		1.24738 (Ridged)	
	6	0	.
		1.25210 (Ridged)	
	7	0	.
		1.25210 (Ridged)	

Final Results

Design	1
Choice Sets	15
Alternatives	3
Parameters	20
Maximum Parameters	30
D-Efficiency	0
D-Error	.

n	Variable Name	Label	Variance	DF	Standard Error
1	Brand1	Brand 1	0.42191	1	0.64955
2	Brand2	Brand 2	0.42147	1	0.64921
3	Brand3	Brand 3	.	0	.
4	Brand1x11	Brand 1 * x1 1	0.33232	1	0.57648
5	Brand1x12	Brand 1 * x1 2	0.38822	1	0.62307
6	Brand2x11	Brand 2 * x1 1	0.30106	1	0.54869
7	Brand2x12	Brand 2 * x1 2	0.39711	1	0.63017
8	Brand3x11	Brand 3 * x1 1	0.35380	1	0.59481
9	Brand3x12	Brand 3 * x1 2	0.37744	1	0.61436
10	Brand1x21	Brand 1 * x2 1	0.39729	1	0.63031
11	Brand1x22	Brand 1 * x2 2	0.32450	1	0.56965
12	Brand2x21	Brand 2 * x2 1	0.38070	1	0.61701
13	Brand2x22	Brand 2 * x2 2	0.35623	1	0.59685
14	Brand3x21	Brand 3 * x2 1	0.36905	1	0.60749
15	Brand3x22	Brand 3 * x2 2	0.34511	1	0.58746
16	Brand1x31	Brand 1 * x3 1	0.39903	1	0.63169
17	Brand1x32	Brand 1 * x3 2	0.32132	1	0.56685
18	Brand2x31	Brand 2 * x3 1	0.42616	1	0.65281
19	Brand2x32	Brand 2 * x3 2	0.32347	1	0.56874
20	Brand3x31	Brand 3 * x3 1	0.38295	1	0.61883
21	Brand3x32	Brand 3 * x3 2	0.34997	1	0.59158
				==	
				20	

The following list is displayed in the log:

```
Redundant Variables:

Brand3
```

Notice that at each step, the efficiency is zero, but a nonzero ridged value is displayed. This model contains a structural-zero coefficient in `Brand3`. While we need alternative-specific effects for Brand 3 (like `Brand3x11` and `Brand3x12`), we do not need the Brand 3 effect (`Brand3`). This can be seen from both the redundant variables list and from looking at the variance and *df* table. The inclusion of the `Brand3` term in the model makes the efficiency of the design zero. However, the `%ChoicEff` macro can still optimize the goodness of the design by optimizing a ridged efficiency criterion—a small constant is added to each diagonal entry of the information matrix to make it nonsingular. That is what is shown in the iteration history. Unlike the `%MktEx` macro, the `%ChoicEff` macro does not have an explicit `ridge=` option. It automatically ridges, but only when needed. We specify `converge=1e-12` because for this example, iteration stops prematurely with the default convergence criterion.

The following step switches to a full-rank coding, dropping the redundant variable Brand3, and using the output from the last step as the initial design:

```
%choiceff(data=full,                /* candidate set of alternatives        */
          init=best(keep=index),    /* select these alts from candidates    */
                                    /* alternative-specific effects model   */
                                    /* zero=' ' no reference level for brand*/
                                    /* brand*x1 ... interactions            */
          model=class(brand brand*x1 brand*x2 brand*x3 / zero=' ' sta),
          drop=brand3,              /* extra model terms to drop from model */
          seed=522,                 /* random number seed                   */
          nsets=15,                 /* number of choice sets                */
          flags=f1-f3,              /* flag which alt can go where, 3 alts  */
          converge=1e-12,           /* convergence criterion                */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

The option drop=brand3 is used to drop the parameter with the zero coefficient. We could have moved the brand specification into its own class specification (separate from the alternative-specific effects) and not specified zero=' ' with it (see, for example, page 878). However, sometimes it is easier to specify a model with more terms than you really need, and then list the terms to drop, so that is what we illustrate here. See page 78 for more information about the zero= option.

In this usage of init= with alternative swapping, the only part of the initial design that is required is the Index variable. It contains indices into the candidate set of the alternatives that are used to make the initial design. This method can be used in the situation where the initial design was output from the %ChoicEff macro. The results are as follows:

Design	Iteration	D-Efficiency	D-Error
1	0	3.01341 *	0.33185
	1	3.01341	0.33185

Final Results

Design	1
Choice Sets	15
Alternatives	3
Parameters	20
Maximum Parameters	30
D-Efficiency	3.0134
Relative D-Eff	20.0894
D-Error	0.3318
1 / Choice Sets	0.0667

n	Variable Name	Label	Variance	DF	Standard Error
1	Brand1	Brand 1	0.42191	1	0.64955
2	Brand2	Brand 2	0.42147	1	0.64921

3	Brand1x11	Brand 1 * x1 1	0.33232	1	0.57648
4	Brand1x12	Brand 1 * x1 2	0.38822	1	0.62307
5	Brand2x11	Brand 2 * x1 1	0.30106	1	0.54869
6	Brand2x12	Brand 2 * x1 2	0.39711	1	0.63017
7	Brand3x11	Brand 3 * x1 1	0.35380	1	0.59481
8	Brand3x12	Brand 3 * x1 2	0.37744	1	0.61436
9	Brand1x21	Brand 1 * x2 1	0.39729	1	0.63031
10	Brand1x22	Brand 1 * x2 2	0.32450	1	0.56965
11	Brand2x21	Brand 2 * x2 1	0.38070	1	0.61701
12	Brand2x22	Brand 2 * x2 2	0.35623	1	0.59685
13	Brand3x21	Brand 3 * x2 1	0.36905	1	0.60749
14	Brand3x22	Brand 3 * x2 2	0.34511	1	0.58746
15	Brand1x31	Brand 1 * x3 1	0.39903	1	0.63169
16	Brand1x32	Brand 1 * x3 2	0.32132	1	0.56685
17	Brand2x31	Brand 2 * x3 1	0.42616	1	0.65281
18	Brand2x32	Brand 2 * x3 2	0.32347	1	0.56874
19	Brand3x31	Brand 3 * x3 1	0.38295	1	0.61883
20	Brand3x32	Brand 3 * x3 2	0.34997	1	0.59158
				==	
				20	

Notice that now there are no zero parameters so *D*-efficiency can be directly computed. In the preceding step, relative *D*-efficiency was requested, and the value is nowhere near 100. This is because the standardized orthogonal contrast coding was not used for all of the attributes, so relative *D*-efficiency is not on a 0 to 100 scale. It is also interesting to perform the evaluation one more time—this time with the standardized orthogonal contrast coding for brand and the other attributes. This lets us drop the `drop=` option. The following step evaluates the design:

```
%choiceff(data=full,                /* candidate set of alternatives        */
          init=best(keep=index),    /* select these alts from candidates    */
                                    /* alternative-specific effects model   */
                                    /* zero=' ' no reference level for brand*/
                                    /* brand*x1 ... interactions            */
          model=class(brand x1 x2 x3 brand*x1 brand*x2 brand*x3 / sta),
          seed=522,                 /* random number seed                   */
          nsets=15,                 /* number of choice sets                */
          flags=f1-f3,              /* flag which alt can go where, 3 alts  */
          converge=1e-12,           /* convergence criterion                */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

It is instructive to compare the two `model=` options from the previous evaluation and the current one. They are as follows:

```
model=class(brand          brand*x1 brand*x2 brand*x3 / zero=' ' sta),
model=class(brand x1 x2 x3 brand*x1 brand*x2 brand*x3 /          sta),
```

Previously, there was a brand effect ($3 - 1 = 2$ parameters) and attribute effects within each of the brands (3 brands times 3 attributes times (3 levels minus 1) = 18 parameters) for a total of 20 parameters. Now there is a brand effect ($3 - 1 = 2$ parameters), attribute effects (3 attributes times (3 levels minus 1) = 6 parameters), and ((3 brands minus 1) times 3 attributes times (3 levels minus 1) = 12 parameters) for a total of 20 parameters. The number of parameters has not changed, but the names and interpretation has changed. The results are as follows:

Final Results

Design	1
Choice Sets	15
Alternatives	3
Parameters	20
Maximum Parameters	30
D-Efficiency	9.5507
Relative D-Eff	63.6715
D-Error	0.1047
1 / Choice Sets	0.0667

n	Variable Name	Label	Variance	DF	Standard Error
1	Brand1	Brand 1	0.07032	1	0.26518
2	Brand2	Brand 2	0.07232	1	0.26892
3	x11	x1 1	0.11182	1	0.33440
4	x12	x1 2	0.13844	1	0.37208
5	x21	x2 1	0.13302	1	0.36472
6	x22	x2 2	0.10224	1	0.31975
7	x31	x3 1	0.10243	1	0.32005
8	x32	x3 2	0.13957	1	0.37360
9	Brand1x11	Brand 1 * x1 1	0.11015	1	0.33188
10	Brand1x12	Brand 1 * x1 2	0.12050	1	0.34713
11	Brand2x11	Brand 2 * x1 1	0.10709	1	0.32725
12	Brand2x12	Brand 2 * x1 2	0.12865	1	0.35867
13	Brand1x21	Brand 1 * x2 1	0.13216	1	0.36354
14	Brand1x22	Brand 1 * x2 2	0.10969	1	0.33119
15	Brand2x21	Brand 2 * x2 1	0.11716	1	0.34229
16	Brand2x22	Brand 2 * x2 2	0.13002	1	0.36058
17	Brand1x31	Brand 1 * x3 1	0.15565	1	0.39452
18	Brand1x32	Brand 1 * x3 2	0.09699	1	0.31142
19	Brand2x31	Brand 2 * x3 1	0.14463	1	0.38031
20	Brand2x32	Brand 2 * x3 2	0.09503	1	0.30826
				==	
				20	

While these two different codings are equivalent, the former does not use the standardized orthogonal contrast coding for brand, while the latter does. Now, the variances are closer to the hypothetical minimum of one over the number of choice sets, and relative *D*-efficiency is larger, but it is still not close to 100. To understand why, imagine that we were going to construct this design directly from an orthogonal array. We would need a design in 45 runs with a fifteen-level factor, for the choice set number, and 4 three-level factors (`Brand x1-x3`). We would additionally need to estimate the interactions between `Brand` and each of `x1-x3`. Such a design does not exist. We can see this by using the `%MktRuns` macro as follows:

```
%mktruns(15 3 3 3 3, interact=2*3 2*4 2*5)
```

The output of this macro (not shown) shows us that the smallest design in which this could possibly work is 135 runs. It is important to note, however, that unless it reports that it will work in 45 runs, it will not work. That is, we are looking for a design with 15 sets and 3 alternatives, and hence 45 runs. With 135 runs, you would have to see if a design with 45 choice sets worked. Now, returning to the 15 sets and 3 alternatives, the relative *D*-efficiency *is* on a 0 to 100 scale, but it is relative to a *hypothetical* optimal design that cannot possibly exist. This is often the case in both linear and choice modeling. Incidentally, for a main effects only model, an optimal design can be constructed as follows:

```
%mktex(15 3 ** 4, n=45)

%mktlab(data=design, vars=Set Brand x1-x3)

%choiceff(data=final,              /* candidate set of choice sets         */
          init=final(keep=set),    /* select these sets from candidates    */
          model=class(brand x1 x2 x3 / sta), /* model w stdzd orthog coding */
          nsets=15,                /* 6 choice sets                        */
          nalts=3,                 /* 3 alternatives per set               */
          options=relative,        /* display relative D-efficiency        */
          beta=zero)               /* assumed beta vector, Ho: b=0         */
```

Some of the results are as follows:

Final Results

Design	1
Choice Sets	15
Alternatives	3
Parameters	8
Maximum Parameters	30
D-Efficiency	15.0000
Relative D-Eff	100.0000
D-Error	0.0667
1 / Choice Sets	0.0667

n	Variable Name	Label	Variance	DF	Standard Error
1	Brand1	Brand 1	0.066667	1	0.25820
2	Brand2	Brand 2	0.066667	1	0.25820
3	x11	x1 1	0.066667	1	0.25820
4	x12	x1 2	0.066667	1	0.25820
5	x21	x2 1	0.066667	1	0.25820
6	x22	x2 2	0.066667	1	0.25820
7	x31	x3 1	0.066667	1	0.25820
8	x32	x3 2	0.066667	1	0.25820
				==	
				8	

The %ChoicEff macro can also find this design by searching the full-factorial candidate set, but it takes a while. It comes very close (in the neighborhood of 99% relative *D*-efficiency) very easily, but it has a hard time finding the exact optimal main-effect design for this problem. The relative *D*-efficiency calculations for the alternative-specific effects model are again based on the assumption that a design with a variance structure like this main-effects model exists for the alternative-specific effects model. It has no way of knowing what the optimal variance structure is for models like this where a "perfect" design does not exist.

Alternative-Specific Effects and the Alternative-Swapping Algorithm

This example is provided to show how you can use the %ChoicEff macro search for an efficient design for a model with alternative-specific effects. This example is based on the vacation example starting on page 339 in the "Discrete Choice" chapter. That example uses the linear arrangement of a choice design approach to construct a choice design from a near orthogonal array. The approach illustrated in the vacation example starting on page 339 is probably the optimal approach for this problem. However, the approach that is used here works almost as well. When you become a sophisticated designer of choice experiments, you will want to be facile with all of the tools in the discrete choice chapter, the experimental design chapter, and this chapter. However, when you are just starting, you might find it easer to concentrate on simply using the %ChoicEff macro with a candidate set of alternatives that you create by using the %MktEx macro.

In this example, a researcher is interested in studying choice of vacation destinations. There are five destinations (alternatives) of interest: Hawaii, Alaska, Mexico, California, and Maine. Each alternative is composed of three factors: package cost ($999, $1,249, $1,499), scenery (mountains, lake, beach), and accommodations (cabin, bed & breakfast, and hotel). In addition, there is a stay at home alternative. See page 339 for more information.

The following step creates formats for each of the destination attributes:

```
proc format;
   value price 1 = ' 999'      2 = '1249'            3 = '1499';
   value scene 1 = 'Mountains' 2 = 'Lake'            3 = 'Beach';
   value lodge 1 = 'Cabin'     2 = 'Bed & Breakfast' 3 = 'Hotel';
   run;
```

Since there are three attributes for each alternative, the following step creates a full-factorial design that consists of all of the combinations of levels that can occur for each destination:

```
%mktex(3 ** 3, n=27)
```

The results of this step are not shown, but the full-factorial design is 100% *D*-efficient, it has $3\times3\times3 = 27$ runs, and it is stored in the SAS data set `Design`.

The following step creates a candidate set of alternatives:

```
data cand;
   retain f1-f6 0;
   length Place $ 10 Price Scene Lodge 8;
   if _n_ = 1 then do; f6 = 1; Place = 'Home'; output; f6 = 0; end;
   set design(rename=(x1=Price x2=Scene x3=Lodge));
   price = input(put(price, price.), 5.);
   f1 = 1; Place = 'Hawaii';     output; f1 = 0;
   f2 = 1; Place = 'Alaska';     output; f2 = 0;
   f3 = 1; Place = 'Mexico';     output; f3 = 0;
   f4 = 1; Place = 'California'; output; f4 = 0;
   f5 = 1; Place = 'Maine';      output; f5 = 0;
   format scene scene. lodge lodge.;
   run;
```

Each of the 27 runs in the full-factorial design is read in once and written out 5 times, once for each destination. In addition, a constant alternative is written once for a total of $5\times27+1 = 136$ candidates. Some of the candidates are as follows:

f1	f2	f3	f4	f5	f6	Place	Price	Scene	Lodge
0	0	0	0	0	1	Home	.	.	.
1	0	0	0	0	0	Hawaii	999	Mountains	Cabin
0	1	0	0	0	0	Alaska	999	Mountains	Cabin
0	0	1	0	0	0	Mexico	999	Mountains	Cabin
0	0	0	1	0	0	California	999	Mountains	Cabin
0	0	0	0	1	0	Maine	999	Mountains	Cabin
1	0	0	0	0	0	Hawaii	999	Mountains	Bed & Breakfast
0	1	0	0	0	0	Alaska	999	Mountains	Bed & Breakfast
0	0	1	0	0	0	Mexico	999	Mountains	Bed & Breakfast
0	0	0	1	0	0	California	999	Mountains	Bed & Breakfast
0	0	0	0	1	0	Maine	999	Mountains	Bed & Breakfast
1	0	0	0	0	0	Hawaii	999	Mountains	Hotel
0	1	0	0	0	0	Alaska	999	Mountains	Hotel
0	0	1	0	0	0	Mexico	999	Mountains	Hotel
0	0	0	1	0	0	California	999	Mountains	Hotel
0	0	0	0	1	0	Maine	999	Mountains	Hotel
.									
.									
.									

```
1   0   0   0   0   0  Hawaii       1499  Lake       Cabin
0   1   0   0   0   0  Alaska       1499  Lake       Cabin
0   0   1   0   0   0  Mexico       1499  Lake       Cabin
0   0   0   1   0   0  California   1499  Lake       Cabin
0   0   0   0   1   0  Maine        1499  Lake       Cabin
.
.
.
1   0   0   0   0   0  Hawaii       1499  Beach      Hotel
0   1   0   0   0   0  Alaska       1499  Beach      Hotel
0   0   1   0   0   0  Mexico       1499  Beach      Hotel
0   0   0   1   0   0  California   1499  Beach      Hotel
0   0   0   0   1   0  Maine        1499  Beach      Hotel
```

Consider again the data step that creates this candidate set:

```
data cand;
   retain f1-f6 0;
   length Place $ 10 Price Scene Lodge 8;
   if _n_ = 1 then do; f6 = 1; Place = 'Home'; output; f6 = 0; end;
   set design(rename=(x1=Price x2=Scene x3=Lodge));
   price = input(put(price, price.), 5.);
   f1 = 1; Place = 'Hawaii';     output; f1 = 0;
   f2 = 1; Place = 'Alaska';     output; f2 = 0;
   f3 = 1; Place = 'Mexico';     output; f3 = 0;
   f4 = 1; Place = 'California'; output; f4 = 0;
   f5 = 1; Place = 'Maine';      output; f5 = 0;
   format scene scene. lodge lodge.;
   run;
```

The `retain` statement creates the six flag variables, `f1-f6` (one for each alternative), initializes them to zero (this candidate cannot be used for any alternatives), and retains their values so that they are not set to missing at each new pass through the DATA step. The `length` statement specifies that the variable `Place` is a character variable of length 10 and that the remaining variables are numeric. The placement of the `retain` and `length` statements ensures that the flag variables appear first in the data set followed by the destination and attribute variables. This is purely for aesthetic reasons when displaying the candidates and does not affect the design search. The next statement adds a single candidate for the constant alternative (`f6 = 1` and `f1-f5 = 0`). The attributes `Price`, `Scene`, and `Lodge` have missing values since the design has not been read yet. The next statement reads each of the 27 candidates and renames the factor names into attribute names. The next statement maps the price attribute from numeric values of 1, 2, 3 to numeric values of 999, 1249, and 1499 by formatting the numeric value by using the `put` function and then converting the result to numeric by using the `input` function. The next five lines write out a candidate for each of the five nonconstant alternatives. Flag variables are set such that `f1` is 1 and `f2-f4` are 0 for the first alternative, `f2` is 1 and `f1 f3-f4` are 0 for the second alternative, and so on. Finally, formats are assigned. The flag variables control which alternative (or for some designs, which alternatives) each candidate can be used.

The next step searches the candidate set for a design with alternative-specific effects:

```
%choiceff(data=cand,                /* candidate set of alternatives       */
          model=class(place /       /* alternative effects                 */
                      zero=none     /* zero=none - use all levels          */
                      order=data)   /* use ordering of levels from data set */
                class(place * price /* alternative-specific effect of price */
                      place * scene /* alternative-specific effect of scene */
                      place * lodge /* alternative-specific effect of lodge */
                     / zero=none    /* zero=none - use all levels of place  */
                      order=formatted)/* order=formatted - sort levels     */
               / lprefix=0          /* lpr=0 labels created from just levels*/
                 cprefix=0          /* cpr=0 names created from just levels */
                 separators=' ' ', ',/* use comma sep to build interact terms*/
          nsets=36,                 /* number of choice sets               */
          flags=f1-f6,              /* six alternatives, alt-specific      */
          seed=104,                 /* random number seed                  */
          beta=zero)                /* assumed beta vector, Ho: b=0        */
```

As we often do, we ask for every conceivable parameter during our first pass, including those that are structural zeros, which we will eliminate in a subsequent pass. Some of the results are as follows:

Design	Iteration	D-Efficiency	D-Error
1	0	0	.
	1	0	.
		0.00072512 (Ridged)	
	2	0	.
		0.00072588 (Ridged)	

Design	Iteration	D-Efficiency	D-Error
2	0	0	.
	1	0	.
		0.00072537 (Ridged)	
	2	0	.
		0.00072647 (Ridged)	

```
Redundant Variables:

Maine Alaska_1499 California_1499 Hawaii_1499 Home_999 Home_1249 Home_1499
Maine_1499 Mexico_1499 AlaskaMountains CaliforniaMountains HawaiiMountains
HomeBeach HomeLake HomeMountains MaineMountains MexicoMountains AlaskaHotel
CaliforniaHotel HawaiiHotel HomeBed___Breakfast HomeCabin HomeHotel MaineHotel
MexicoHotel
```

Final Results

Design	1
Choice Sets	36
Alternatives	6
Parameters	35
Maximum Parameters	180
D-Efficiency	0
D-Error	.

n	Variable Name	Label	Variance	DF	Standard Error
1	Home	Home	1.56505	1	1.25102
2	Hawaii	Hawaii	2.73192	1	1.65285
3	Alaska	Alaska	2.92946	1	1.71157
4	Mexico	Mexico	2.63759	1	1.62407
5	California	California	2.70000	1	1.64317
6	Maine	Maine	.	0	.
7	Alaska_999	Alaska, 999	1.22388	1	1.10629
8	Alaska_1249	Alaska, 1249	1.23861	1	1.11293
9	Alaska_1499	Alaska, 1499	.	0	.
10	California_999	California, 999	1.23793	1	1.11262
11	California_1249	California, 1249	1.21703	1	1.10319
12	California_1499	California, 1499	.	0	.
13	Hawaii_999	Hawaii, 999	1.22456	1	1.10660
14	Hawaii_1249	Hawaii, 1249	1.22929	1	1.10873
15	Hawaii_1499	Hawaii, 1499	.	0	.
16	Home_999	Home, 999	.	0	.
17	Home_1249	Home, 1249	.	0	.
18	Home_1499	Home, 1499	.	0	.
19	Maine_999	Maine, 999	1.23864	1	1.11294
20	Maine_1249	Maine, 1249	1.22918	1	1.10868
21	Maine_1499	Maine, 1499	.	0	.
22	Mexico_999	Mexico, 999	1.23168	1	1.10981
23	Mexico_1249	Mexico, 1249	1.22689	1	1.10765
24	Mexico_1499	Mexico, 1499	.	0	.
25	AlaskaBeach	Alaska, Beach	1.22235	1	1.10560
26	AlaskaLake	Alaska, Lake	1.22247	1	1.10565
27	AlaskaMountains	Alaska, Mountains	.	0	.
28	CaliforniaBeach	California, Beach	1.24270	1	1.11477
29	CaliforniaLake	California, Lake	1.25330	1	1.11951
30	CaliforniaMountains	California, Mountains	.	0	.
31	HawaiiBeach	Hawaii, Beach	1.22975	1	1.10894
32	HawaiiLake	Hawaii, Lake	1.22507	1	1.10683
33	HawaiiMountains	Hawaii, Mountains	.	0	.
34	HomeBeach	Home, Beach	.	0	.
35	HomeLake	Home, Lake	.	0	.
36	HomeMountains	Home, Mountains	.	0	.

```
 37 MaineBeach                Maine, Beach                1.23600  1  1.11175
 38 MaineLake                 Maine, Lake                 1.23112  1  1.10956
 39 MaineMountains            Maine, Mountains             .       0   .
 40 MexicoBeach               Mexico, Beach               1.23252  1  1.11019
 41 MexicoLake                Mexico, Lake                1.22701  1  1.10771
 42 MexicoMountains           Mexico, Mountains            .       0   .
 43 AlaskaBed___Breakfast     Alaska, Bed & Breakfast     1.23723  1  1.11231
 44 AlaskaCabin               Alaska, Cabin               1.22093  1  1.10496
 45 AlaskaHotel               Alaska, Hotel                .       0   .
 46 CaliforniaBed___Breakfast California, Bed & Breakfast 1.23418  1  1.11094
 47 CaliforniaCabin           California, Cabin           1.22615  1  1.10732
 48 CaliforniaHotel           California, Hotel            .       0   .
 49 HawaiiBed___Breakfast     Hawaii, Bed & Breakfast     1.22431  1  1.10648
 50 HawaiiCabin               Hawaii, Cabin               1.23680  1  1.11211
 51 HawaiiHotel               Hawaii, Hotel                .       0   .
 52 HomeBed___Breakfast       Home, Bed & Breakfast        .       0   .
 53 HomeCabin                 Home, Cabin                  .       0   .
 54 HomeHotel                 Home, Hotel                  .       0   .
 55 MaineBed___Breakfast      Maine, Bed & Breakfast      1.22375  1  1.10623
 56 MaineCabin                Maine, Cabin                1.22955  1  1.10885
 57 MaineHotel                Maine, Hotel                 .       0   .
 58 MexicoBed___Breakfast     Mexico, Bed & Breakfast     1.23665  1  1.11205
 59 MexicoCabin               Mexico, Cabin               1.23940  1  1.11328
 60 MexicoHotel               Mexico, Hotel                .       0   .
                                                                   ==
                                                                   35
```

Since there are extra parameters, the *D*-efficiency is zero, but the `%ChoicEff` macro optimizes a ridged efficiency criterion. A list of reference parameters is provided. These can be used to drop the extra parameters, or we can use the `zero=` option. There is one reference level for destination and one for each of the nonconstant destination attributes. Furthermore, all of the parameters associated with the constant stay at home alternative are zero.[†] There are a total of 35 parameters that can be estimated (5 destinations plus 5 destinations times 3 attributes times $(3-1)$ levels. The model specification `model=class(place / zero=none order=data) class(place * price place * scene place * lodge / zero=none order=formatted)` creates one term for each destination (minus one) and two terms for each destination and attribute combination.

The standardized orthogonal contrast coding is not used since we have five nonconstant alternatives and three-level factors. It will be hard to know the optimum *D*-efficiency for a design like this.

We can run this again designating the stay at home level as a reference level and ask for 100 designs this time as follows:

[†]When we more explicitly control the reference level, Maine will have an estimable parameter for the destination effect instead of the stay at home alternative.

```
%choiceff(data=cand,                /* candidate set of alternatives       */
          model=class(place /       /* alternative effects                 */
                      zero='Home'   /* use 'Home' as reference level       */
                      order=data)   /* use ordering of levels from data set */
                                    /* place * price ... - interactions or  */
                class(place * price /* alternative-specific effect of price */
                      place * scene /* alternative-specific effect of scene */
                      place * lodge /* alternative-specific effect of lodge */
                     / zero='Home'  /* use 'Home' as reference level       */
                      order=formatted)/* order=formatted - sort levels     */
               / lprefix=0          /* lpr=0 labels created from just levels*/
                 cprefix=0          /* cpr=0 names created from just levels */
                 separators=' ' ', ',/* use comma sep to build interact terms*/
          nsets=36,                 /* number of choice sets               */
          flags=f1-f6,              /* six alternatives, alt-specific      */
          maxiter=100,              /* maximum number of designs to make   */
          seed=104,                 /* random number seed                  */
          beta=zero)                /* assumed beta vector, Ho: b=0        */
```

Some of the results are as follows:

Design	Iteration	D-Efficiency	D-Error
1	0	1.02898 *	0.97183
	1	1.17248 *	0.85290
	2	1.17458 *	0.85137

Design	Iteration	D-Efficiency	D-Error
2	0	1.05615	0.94684
	1	1.17317	0.85239
	2	1.17621 *	0.85019

Design	Iteration	D-Efficiency	D-Error
3	0	1.05667	0.94637
	1	1.17486	0.85117
	2	1.17642 *	0.85003

Design	Iteration	D-Efficiency	D-Error
4	0	1.07743	0.92814
	1	1.17467	0.85130
	2	1.17467	0.85130

Design	Iteration	D-Efficiency	D-Error
5	0	1.07993	0.92598
	1	1.17571	0.85055
	2	1.17663 *	0.84989

.
.
.

Design	Iteration	D-Efficiency	D-Error
53	0	1.04235	0.95937
	1	1.17464	0.85132
	2	1.17728 *	0.84942

.
.
.

Design	Iteration	D-Efficiency	D-Error
100	0	1.04486	0.95706
	1	1.17310	0.85244
	2	1.17662	0.84989

Final Results

Design	53
Choice Sets	36
Alternatives	6
Parameters	35
Maximum Parameters	180
D-Efficiency	1.1773
D-Error	0.8494

n	Variable Name	Label	Variance	DF	Error
1	Hawaii	Hawaii	1.55830	1	1.24832
2	Alaska	Alaska	1.56263	1	1.25005
3	Mexico	Mexico	1.55387	1	1.24654
4	California	California	1.55178	1	1.24570
5	Maine	Maine	1.54976	1	1.24489
6	Alaska_999	Alaska, 999	1.23707	1	1.11224
7	Alaska_1249	Alaska, 1249	1.22530	1	1.10693
8	California_999	California, 999	1.22238	1	1.10561
9	California_1249	California, 1249	1.22445	1	1.10655
10	Hawaii_999	Hawaii, 999	1.21990	1	1.10449
11	Hawaii_1249	Hawaii, 1249	1.21886	1	1.10402
12	Maine_999	Maine, 999	1.22909	1	1.10864
13	Maine_1249	Maine, 1249	1.21766	1	1.10348
14	Mexico_999	Mexico, 999	1.22986	1	1.10899
15	Mexico_1249	Mexico, 1249	1.22239	1	1.10562
16	AlaskaBeach	Alaska, Beach	1.21993	1	1.10450
17	AlaskaLake	Alaska, Lake	1.23474	1	1.11119
18	CaliforniaBeach	California, Beach	1.22751	1	1.10793
19	CaliforniaLake	California, Lake	1.22852	1	1.10839

```
 20 HawaiiBeach               Hawaii, Beach               1.21878  1  1.10399
 21 HawaiiLake                Hawaii, Lake                1.23243  1  1.11015
 22 MaineBeach                Maine, Beach                1.23564  1  1.11159
 23 MaineLake                 Maine, Lake                 1.22895  1  1.10858
 24 MexicoBeach               Mexico, Beach               1.22240  1  1.10562
 25 MexicoLake                Mexico, Lake                1.21919  1  1.10417
 26 AlaskaBed___Breakfast     Alaska, Bed & Breakfast     1.24157  1  1.11426
 27 AlaskaCabin               Alaska, Cabin               1.22303  1  1.10591
 28 CaliforniaBed___Breakfast California, Bed & Breakfast 1.22586  1  1.10718
 29 CaliforniaCabin           California, Cabin           1.22318  1  1.10597
 30 HawaiiBed___Breakfast     Hawaii, Bed & Breakfast     1.23355  1  1.11065
 31 HawaiiCabin               Hawaii, Cabin               1.22510  1  1.10684
 32 MaineBed___Breakfast      Maine, Bed & Breakfast      1.21813  1  1.10369
 33 MaineCabin                Maine, Cabin                1.21869  1  1.10394
 34 MexicoBed___Breakfast     Mexico, Bed & Breakfast     1.22010  1  1.10458
 35 MexicoCabin               Mexico, Cabin               1.22302  1  1.10590
                                                                   ==
                                                                   35
```

Now, *D*-efficiency is not zero and all 35 parameters can be estimated. Note that the stay at home alternative is now consistently the reference level. The *D*-efficiency and variances are similar to those found in the example starting on page 339. However, the efficiency is a bit lower, and the variances are a bit larger. The approach outlined on 339 is a better approach for this problem, but this approach works quite well for this problem and a wide variety of other problems.

The design is displayed in the following step:

```
options ps=200 missing=' ';
proc print;
   id set;
   by set;
   var place -- lodge;
   run;
options ps=60 missing='.';
```

Some of the results are as follows:

```
            Set    Place         Price    Scene        Lodge

              1    Hawaii         1499    Beach        Hotel
                   Alaska          999    Beach        Bed & Breakfast
                   Mexico         1499    Lake         Hotel
                   California     1499    Mountains    Cabin
                   Maine          1249    Mountains    Hotel
                   Home
```

```
 2    Hawaii         1499    Lake         Cabin
      Alaska         1499    Lake         Hotel
      Mexico          999    Mountains    Bed & Breakfast
      California     1249    Beach        Bed & Breakfast
      Maine          1249    Lake         Hotel
      Home

 3    Hawaii         1499    Lake         Bed & Breakfast
      Alaska         1499    Beach        Bed & Breakfast
      Mexico         1499    Lake         Cabin
      California     1249    Lake         Cabin
      Maine           999    Beach        Bed & Breakfast
      Home

 4    Hawaii         1499    Beach        Cabin
      Alaska         1499    Beach        Cabin
      Mexico         1499    Mountains    Bed & Breakfast
      California     1499    Mountains    Bed & Breakfast
      Maine          1249    Mountains    Bed & Breakfast
      Home

 5    Hawaii          999    Mountains    Hotel
      Alaska         1249    Lake         Hotel
      Mexico          999    Lake         Hotel
      California     1499    Lake         Cabin
      Maine          1249    Lake         Cabin
      Home

 6    Hawaii         1499    Lake         Hotel
      Alaska         1249    Lake         Cabin
      Mexico         1499    Lake         Cabin
      California      999    Beach        Hotel
      Maine          1499    Lake         Hotel
      Home

 .
 .
 .

34    Hawaii         1499    Mountains    Bed & Breakfast
      Alaska         1249    Beach        Hotel
      Mexico          999    Mountains    Bed & Breakfast
      California     1499    Beach        Cabin
      Maine           999    Beach        Hotel
      Home

35    Hawaii         1249    Mountains    Hotel
      Alaska         1499    Lake         Bed & Breakfast
      Mexico         1249    Beach        Bed & Breakfast
      California     1249    Mountains    Hotel
      Maine          1249    Beach        Hotel
      Home
```

```
 36    Hawaii         1249    Mountains    Cabin
       Alaska         1499    Beach        Cabin
       Mexico         1249    Lake         Bed & Breakfast
       California     1249    Lake         Bed & Breakfast
       Maine          1499    Mountains    Cabin
       Home
```

Note that each destination always appears in the same alternative, and this was controlled at candidate set creation time.

The following step codes the design:

```
proc transreg data=best design norestoremissing;
   model class(place / zero='Home' order=data)
         class(place * price place * scene place * lodge /
               zero='Home' order=formatted) /
         lprefix=0 cprefix=0 separators=' ' ', ';
   output out=coded;
   run;
```

The `design` option specifies that no model is fit; the procedure is just being used to code a design. When `design` is specified, dependent variables are not required. The `norestoremissing` option specifies that missing values should not be restored when the `out=` data set is created. By default, the coded `class` variable contains a row of missing values for observations in which the `class` variable is missing. With the `norestoremissing` option, these observations contain a row of zeros instead. This option is useful when there is a constant alternative indicated by missing values.

The following steps display the first coded choice set:

```
proc print data=coded(obs=6) noobs label;
   var place -- lodge Hawaii -- Maine;
   run;

proc print data=coded(obs=6) noobs label;
   var place Alaska_999 Alaska_1249 California_999 California_1249;
   run;

proc print data=coded(obs=6) noobs label;
   var place Hawaii_999 Hawaii_1249 Maine_999 Maine_1249 Mexico_999 Mexico_1249;
   run;

proc print data=coded(obs=6) noobs label;
   var place AlaskaBeach AlaskaLake CaliforniaBeach CaliforniaLake;
   run;

proc print data=coded(obs=6) noobs label;
   var place HawaiiBeach HawaiiLake MaineBeach MaineLake MexicoBeach MexicoLake;
   run;
```

```
proc print data=coded(obs=6) noobs label;
   var place AlaskaBed___Breakfast AlaskaCabin
       CaliforniaBed___Breakfast CaliforniaCabin;
   run;

proc print data=coded(obs=6) noobs label;
   var place HawaiiBed___Breakfast HawaiiCabin MaineBed___Breakfast MaineCabin
        MexicoBed___Breakfast MexicoCabin;
   run;
```

The results are as follows:

Place	Price	Scene	Lodge	Hawaii	Alaska	Mexico	California	Maine
Hawaii	1499	Beach	Hotel	1	0	0	0	0
Alaska	999	Beach	Bed & Breakfast	0	1	0	0	0
Mexico	1499	Lake	Hotel	0	0	1	0	0
California	1499	Mountains	Cabin	0	0	0	1	0
Maine	1249	Mountains	Hotel	0	0	0	0	1
Home	.	.	.	0	0	0	0	0

Place	Alaska, 999	Alaska, 1249	California, 999	California, 1249
Hawaii	0	0	0	0
Alaska	1	0	0	0
Mexico	0	0	0	0
California	0	0	0	0
Maine	0	0	0	0
Home	0	0	0	0

Place	Hawaii, 999	Hawaii, 1249	Maine, 999	Maine, 1249	Mexico, 999	Mexico, 1249
Hawaii	0	0	0	0	0	0
Alaska	0	0	0	0	0	0
Mexico	0	0	0	0	0	0
California	0	0	0	0	0	0
Maine	0	0	0	1	0	0
Home	0	0	0	0	0	0

Place	Alaska, Beach	Alaska, Lake	California, Beach	California, Lake
Hawaii	0	0	0	0
Alaska	1	0	0	0
Mexico	0	0	0	0
California	0	0	0	0
Maine	0	0	0	0
Home	0	0	0	0

Place	Hawaii, Beach	Hawaii, Lake	Maine, Beach	Maine, Lake	Mexico, Beach	Mexico, Lake
Hawaii	1	0	0	0	0	0
Alaska	0	0	0	0	0	0
Mexico	0	0	0	0	0	1
California	0	0	0	0	0	0
Maine	0	0	0	0	0	0
Home	0	0	0	0	0	0

Place	Alaska, Bed & Breakfast	Alaska, Cabin	California, Bed & Breakfast	California, Cabin
Hawaii	0	0	0	0
Alaska	1	0	0	0
Mexico	0	0	0	0
California	0	0	0	1
Maine	0	0	0	0
Home	0	0	0	0

Note that all coded variables for the stay at home alternative are zero. This is true in all other choice sets as well. Hence the utility for that alternative is zero, and the utility for all other alternatives is relative to zero.

Brand Effects and the Choice Set Swapping Algorithm

This example is provided for complete coverage of the `%ChoicEff` macro. If you are just getting started, concentrate instead on examples of the `%ChoicEff` macro that use candidate sets of alternatives. These next steps handle the same problem, only this time, we use the set-swapping algorithm, and we will specify a parameter vector that is not zero. At first, we omit the `beta=` option, just to see the coding. We specify the `effects` option in the PROC TRANSREG `class` specification to get –1, 0, 1 coding. The following steps create the design:

```
%mktex(3 ** 9, n=2187, seed=121)

data key;
   input (Brand x1-x3) ($);
   datalines;
1 x1 x2 x3
2 x4 x5 x6
3 x7 x8 x9
;
```

```
%mktroll(design=design, key=key, alt=brand, out=rolled)

%choiceff(data=rolled,              /* candidate set of choice sets         */
                                    /* alternative-specific model           */
                                    /* effects coding of interactions       */
                                    /* zero=' ' no reference level for brand*/
                                    /* brand*x1 ... interactions            */
          model=class(brand)
                class(brand*x1 brand*x2 brand*x3 / effects zero=' '),
          nsets=15,                 /* number of choice sets                */
          nalts=3)                  /* number of alternatives               */
```

The output tells us the parameter names and the order in which we need to specify parameters. The results are as follows:

n	Name	Beta	Label
1	Brand1	.	Brand 1
2	Brand2	.	Brand 2
3	Brand1x11	.	Brand 1 * x1 1
4	Brand1x12	.	Brand 1 * x1 2
5	Brand2x11	.	Brand 2 * x1 1
6	Brand2x12	.	Brand 2 * x1 2
7	Brand3x11	.	Brand 3 * x1 1
8	Brand3x12	.	Brand 3 * x1 2
9	Brand1x21	.	Brand 1 * x2 1
10	Brand1x22	.	Brand 1 * x2 2
11	Brand2x21	.	Brand 2 * x2 1
12	Brand2x22	.	Brand 2 * x2 2
13	Brand3x21	.	Brand 3 * x2 1
14	Brand3x22	.	Brand 3 * x2 2
15	Brand1x31	.	Brand 1 * x3 1
16	Brand1x32	.	Brand 1 * x3 2
17	Brand2x31	.	Brand 2 * x3 1
18	Brand2x32	.	Brand 2 * x3 2
19	Brand3x31	.	Brand 3 * x3 1
20	Brand3x32	.	Brand 3 * x3 2

Now that we are sure we know the order of the parameters, we can specify the assumed betas in the `beta=` option. These numbers are based on prior research or our expectations of approximately what we expect the parameter estimates will be. We also specify `n=100` in this run, which is a sample size we are considering.

The following step creates the design:

```
%choiceff(data=rolled,              /* candidate set of choice sets         */
                                    /* alternative-specific model           */
                                    /* effects coding of interactions       */
                                    /* zero=' ' no reference level for brand*/
                                    /* brand*x1 ... interactions            */
          model=class(brand)
                class(brand*x1 brand*x2 brand*x3 / effects zero=' '),

          nsets=15,                 /* number of choice sets                */
          nalts=3,                  /* number of alternatives               */
          n=100,                    /* n obs to use in variance formula     */
          seed=462,                 /* random number seed                   */
          beta=1 2 -0.5 0.5 -0.75 0.75 -1 1
               -0.5 0.5 -0.75 0.75 -1 1 -0.5 0.5 -0.75 0.75 -1 1)
```

Some of the results are as follows:

Final Results

Design	2
Choice Sets	15
Alternatives	3
Parameters	20
Maximum Parameters	30
D-Efficiency	144.1951
D-Error	0.006935

n	Variable Name	Label	Variance	Assumed Beta	DF	Standard Error	Wald	Prob > Squared Wald
1	Brand1	Brand 1	0.011889	1.00	1	0.10903	9.1714	0.0001
2	Brand2	Brand 2	0.020697	2.00	1	0.14386	13.9021	0.0001
3	Brand1x11	Brand 1 * x1 1	0.008617	-0.50	1	0.09283	-5.3865	0.0001
4	Brand1x12	Brand 1 * x1 2	0.008527	0.50	1	0.09234	5.4147	0.0001
5	Brand2x11	Brand 2 * x1 1	0.009283	-0.75	1	0.09635	-7.7842	0.0001
6	Brand2x12	Brand 2 * x1 2	0.012453	0.75	1	0.11159	6.7208	0.0001
7	Brand3x11	Brand 3 * x1 1	0.021764	-1.00	1	0.14753	-6.7784	0.0001
8	Brand3x12	Brand 3 * x1 2	0.015657	1.00	1	0.12513	7.9917	0.0001
9	Brand1x21	Brand 1 * x2 1	0.012520	-0.50	1	0.11189	-4.4685	0.0001
10	Brand1x22	Brand 1 * x2 2	0.010685	0.50	1	0.10337	4.8370	0.0001
11	Brand2x21	Brand 2 * x2 1	0.010545	-0.75	1	0.10269	-7.3035	0.0001
12	Brand2x22	Brand 2 * x2 2	0.012654	0.75	1	0.11249	6.6672	0.0001
13	Brand3x21	Brand 3 * x2 1	0.018279	-1.00	1	0.13520	-7.3964	0.0001
14	Brand3x22	Brand 3 * x2 2	0.012117	1.00	1	0.11008	9.0845	0.0001
15	Brand1x31	Brand 1 * x3 1	0.009697	-0.50	1	0.09848	-5.0774	0.0001
16	Brand1x32	Brand 1 * x3 2	0.010787	0.50	1	0.10386	4.8141	0.0001

```
17 Brand2x31 Brand 2 * x3 1 0.009203  -0.75   1  0.09593  -7.8181 0.0001
18 Brand2x32 Brand 2 * x3 2 0.013923   0.75   1  0.11800   6.3562 0.0001
19 Brand3x31 Brand 3 * x3 1 0.016546  -1.00   1  0.12863  -7.7742 0.0001
20 Brand3x32 Brand 3 * x3 2 0.014235   1.00   1  0.11931   8.3815 0.0001
                                             ==
                                             20
```

First, notice that D-efficiency is not on a 0 to 100 scale with this design specification. Also notice that parameters and test statistics are incorporated into the output. The n= value is incorporated into the variance matrix and hence the efficiency statistics, variances and tests.

Cross-Effects and the Choice Set Swapping Algorithm

These next steps create a design for a cross-effects model with five brands at three prices and a constant alternative:

```
%mktex(3 ** 5, n=3**5)

data key;
   input (Brand Price) ($);
   datalines;
1 x1
2 x2
3 x3
4 x4
5 x5
. .
;

%mktroll(design=design, key=key, alt=brand, out=rolled, keep=x1-x5)

proc print; by set; id set; where set in (1, 48, 101, 243); run;
```

See the examples beginning on pages 444 and 468 for more information about cross-effects. Note the choice-set-swapping algorithm can handle cross-effects but not the alternative-swapping algorithm.

The keep= option in the %MktRoll macro is used to keep the price variables that are needed to make the cross-effects. The following display shows some of the candidate choice sets:

Set	Brand	Price	x1	x2	x3	x4	x5
1	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
		.	1	1	1	1	1

48	1	1	1	2	3	1	3
	2	2	1	2	3	1	3
	3	3	1	2	3	1	3
	4	1	1	2	3	1	3
	5	3	1	2	3	1	3
		.	1	2	3	1	3
101	1	2	2	1	3	1	2
	2	1	2	1	3	1	2
	3	3	2	1	3	1	2
	4	1	2	1	3	1	2
	5	2	2	1	3	1	2
		.	2	1	3	1	2
243	1	3	3	3	3	3	3
	2	3	3	3	3	3	3
	3	3	3	3	3	3	3
	4	3	3	3	3	3	3
	5	3	3	3	3	3	3
		.	3	3	3	3	3

Notice that `x1` contains the price for Brand 1, `x2` contains the price for Brand 2, and so on, and the price of brand i in a choice set is the same, no matter which alternative it is stored with.

The following `%ChoicEff` macro step creates the choice design with cross-effects:

```
%choiceff(data=rolled,              /* candidate set of choice sets         */
                                    /* model with cross-effects             */
                                    /* zero=none - use all levels           */
                                    /* ide(...) * class(...) - cross-effects*/
          model=class(brand brand*price / zero=none)
                identity(x1-x5) * class(brand / zero=none),
          nsets=20,                 /* number of choice sets                */
          nalts=6,                  /* number of alternatives               */
          seed=17,                  /* random number seed                   */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

Cross-effects are created by interacting the price factors with brand. See pages 452 and 509 for more information about cross-effects.

The following redundant variable list is displayed in the log:

```
Redundant Variables:

Brand1Price3 Brand2Price3 Brand3Price3 Brand4Price3 Brand5Price3 x1Brand1
x2Brand2 x3Brand3 x4Brand4 x5Brand5
```

Next, we will run the macro again, this time requesting a full-rank model. The list of dropped names was created by copying from the redundant variable list. Also, `zero=none` was changed to `zero=' '` so no level would be zeroed for `Brand` but the last level of `Price` would be zeroed. See page 78 for more information about the `zero=` option.

The following step creates the design:

```
%choiceff(data=rolled,              /* candidate set of choice sets         */
                                    /* zero=' ' no reference level for brand*/
                                    /* model with cross-effects             */
                                    /* zero=none - use all levels           */
                                    /* ide(...) * class(...) - cross-effects*/
          model=class(brand brand*price / zero=' ')
                identity(x1-x5) * class(brand / zero=none),

                                    /* extra model terms to drop from model */
          drop=x1Brand1 x2Brand2 x3Brand3 x4Brand4 x5Brand5,
          nsets=20,                 /* number of choice sets                */
          nalts=6,                  /* number of alternatives               */
          seed=17,                  /* random number seed                   */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

In the following results, notice that we have five brand parameters, two price parameters for each of the five brands, and four cross-effect parameters for each of the five brands:

n	Variable Name	Label	Variance	DF	Standard Error
1	Brand1	Brand 1	13.8149	1	3.71683
2	Brand2	Brand 2	13.5263	1	3.67782
3	Brand3	Brand 3	13.2895	1	3.64547
4	Brand4	Brand 4	13.5224	1	3.67728
5	Brand5	Brand 5	16.3216	1	4.04000
6	Brand1Price1	Brand 1 * Price 1	2.8825	1	1.69779
7	Brand1Price2	Brand 1 * Price 2	3.5118	1	1.87399
8	Brand2Price1	Brand 2 * Price 1	2.8710	1	1.69441
9	Brand2Price2	Brand 2 * Price 2	3.5999	1	1.89733
10	Brand3Price1	Brand 3 * Price 1	2.8713	1	1.69448
11	Brand3Price2	Brand 3 * Price 2	3.5972	1	1.89662
12	Brand4Price1	Brand 4 * Price 1	2.8710	1	1.69441
13	Brand4Price2	Brand 4 * Price 2	3.5560	1	1.88574
14	Brand5Price1	Brand 5 * Price 1	2.8443	1	1.68649
15	Brand5Price2	Brand 5 * Price 2	3.8397	1	1.95953
16	x1Brand2	x1 * Brand 2	0.7204	1	0.84878
17	x1Brand3	x1 * Brand 3	0.7209	1	0.84908
18	x1Brand4	x1 * Brand 4	0.7204	1	0.84878
19	x1Brand5	x1 * Brand 5	0.7204	1	0.84877
20	x2Brand1	x2 * Brand 1	0.7178	1	0.84722
21	x2Brand3	x2 * Brand 3	0.7178	1	0.84724
22	x2Brand4	x2 * Brand 4	0.7178	1	0.84720
23	x2Brand5	x2 * Brand 5	0.7248	1	0.85133

24	x3Brand1	x3 * Brand 1	0.7178	1	0.84722
25	x3Brand2	x3 * Brand 2	0.7178	1	0.84721
26	x3Brand4	x3 * Brand 4	0.7178	1	0.84720
27	x3Brand5	x3 * Brand 5	0.7248	1	0.85133
28	x4Brand1	x4 * Brand 1	0.7178	1	0.84722
29	x4Brand2	x4 * Brand 2	0.7178	1	0.84721
30	x4Brand3	x4 * Brand 3	0.7178	1	0.84724
31	x4Brand5	x4 * Brand 5	0.7293	1	0.85402
32	x5Brand1	x5 * Brand 1	0.7111	1	0.84325
33	x5Brand2	x5 * Brand 2	0.7180	1	0.84737
34	x5Brand3	x5 * Brand 3	0.7248	1	0.85135
35	x5Brand4	x5 * Brand 4	0.7179	1	0.84731
				==	
				35	

Asymmetric Factors and the Alternative Swapping Algorithm

In this `%ChoicEff` macro example, the goal is to create a design for a pricing study with ten brands plus a constant alternative. Each brand has a single attribute, price. However, the prices are potentially different for each brand and they do not even have the same numbers of levels. A model is desired with brand and alternative-specific price effects. The design specifications are as follows:

Brand	Levels	Prices
Brand 1	8	0.89 0.94 0.99 1.04 1.09 1.14 1.19 1.24
Brand 2	8	0.94 0.99 1.04 1.09 1.14 1.19 1.24 1.29
Brand 3	6	0.99 1.04 1.09 1.14 1.19 1.24
Brand 4	6	0.89 0.94 0.99 1.04 1.09 1.14
Brand 5	6	1.04 1.09 1.14 1.19 1.24 1.29
Brand 6	4	0.89 0.99 1.09 1.19
Brand 7	4	0.99 1.09 1.19 1.29
Brand 8	4	0.94 0.99 1.14 1.19
Brand 9	4	1.09 1.14 1.19 1.24
Brand 10	4	1.14 1.19 1.24 1.29

The challenging aspect of this problem is creating the candidate set while coping with the price asymmetries. The candidate set must contain 8 rows for the eight Brand 1 prices, 8 rows for the eight Brand 2 prices, 6 rows for the six Brand 3 prices, ..., and 4 rows for the four Brand 10 prices. It also must contain a constant alternative. Furthermore, if we are to use the alternative-swapping algorithm, the candidate set must contain 11 flag variables, each of which will designate the appropriate group of candidates for each alternative. We could run the `%MktEx` macro ten times to make a candidate set for each of the brands, but since we have only one factor per brand, it would be much easier to generate the candidate set with a DATA step. Before we discuss the code, it is instructive to examine the candidate set.

The candidate set is as follows:

Obs	Brand	Price	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11
1	1	0.89	1	0	0	0	0	0	0	0	0	0	0
2	1	0.94	1	0	0	0	0	0	0	0	0	0	0
3	1	0.99	1	0	0	0	0	0	0	0	0	0	0
4	1	1.04	1	0	0	0	0	0	0	0	0	0	0
5	1	1.09	1	0	0	0	0	0	0	0	0	0	0
6	1	1.14	1	0	0	0	0	0	0	0	0	0	0
7	1	1.19	1	0	0	0	0	0	0	0	0	0	0
8	1	1.24	1	0	0	0	0	0	0	0	0	0	0
9	2	0.94	0	1	0	0	0	0	0	0	0	0	0
10	2	0.99	0	1	0	0	0	0	0	0	0	0	0
11	2	1.04	0	1	0	0	0	0	0	0	0	0	0
12	2	1.09	0	1	0	0	0	0	0	0	0	0	0
13	2	1.14	0	1	0	0	0	0	0	0	0	0	0
14	2	1.19	0	1	0	0	0	0	0	0	0	0	0
15	2	1.24	0	1	0	0	0	0	0	0	0	0	0
16	2	1.29	0	1	0	0	0	0	0	0	0	0	0
17	3	0.99	0	0	1	0	0	0	0	0	0	0	0
18	3	1.04	0	0	1	0	0	0	0	0	0	0	0
19	3	1.09	0	0	1	0	0	0	0	0	0	0	0
20	3	1.14	0	0	1	0	0	0	0	0	0	0	0
21	3	1.19	0	0	1	0	0	0	0	0	0	0	0
22	3	1.24	0	0	1	0	0	0	0	0	0	0	0
23	4	0.89	0	0	0	1	0	0	0	0	0	0	0
24	4	0.94	0	0	0	1	0	0	0	0	0	0	0
25	4	0.99	0	0	0	1	0	0	0	0	0	0	0
26	4	1.04	0	0	0	1	0	0	0	0	0	0	0
27	4	1.09	0	0	0	1	0	0	0	0	0	0	0
28	4	1.14	0	0	0	1	0	0	0	0	0	0	0
29	5	1.04	0	0	0	0	1	0	0	0	0	0	0
30	5	1.09	0	0	0	0	1	0	0	0	0	0	0
31	5	1.14	0	0	0	0	1	0	0	0	0	0	0
32	5	1.19	0	0	0	0	1	0	0	0	0	0	0
33	5	1.24	0	0	0	0	1	0	0	0	0	0	0
34	5	1.29	0	0	0	0	1	0	0	0	0	0	0
35	6	0.89	0	0	0	0	0	1	0	0	0	0	0
36	6	0.99	0	0	0	0	0	1	0	0	0	0	0
37	6	1.09	0	0	0	0	0	1	0	0	0	0	0
38	6	1.19	0	0	0	0	0	1	0	0	0	0	0
39	7	0.99	0	0	0	0	0	0	1	0	0	0	0
40	7	1.09	0	0	0	0	0	0	1	0	0	0	0
41	7	1.19	0	0	0	0	0	0	1	0	0	0	0
42	7	1.29	0	0	0	0	0	0	1	0	0	0	0
43	8	0.94	0	0	0	0	0	0	0	1	0	0	0
44	8	0.99	0	0	0	0	0	0	0	1	0	0	0
45	8	1.14	0	0	0	0	0	0	0	1	0	0	0
46	8	1.19	0	0	0	0	0	0	0	1	0	0	0

47	9	1.09	0	0	0	0	0	0	0	0	1	0	0
48	9	1.14	0	0	0	0	0	0	0	0	1	0	0
49	9	1.19	0	0	0	0	0	0	0	0	1	0	0
50	9	1.24	0	0	0	0	0	0	0	0	1	0	0
51	10	1.14	0	0	0	0	0	0	0	0	0	1	0
52	10	1.19	0	0	0	0	0	0	0	0	0	1	0
53	10	1.24	0	0	0	0	0	0	0	0	0	1	0
54	10	1.29	0	0	0	0	0	0	0	0	0	1	0
55	None	.	0	0	0	0	0	0	0	0	0	0	1

It begins with eight candidates for the eight prices for the first brand (`Brand` = 1 `f1` = 1, `f2-f11` = 0). It is followed by eight alternatives for the eight prices for the second brand (`Brand` = 2 `f2` = 1, `f1` = 0, `f3` through `f11` = 0). The constant alternative is at the end. The following steps create and display the candidate design:

```
proc format;
   value bf 11 = 'None';
   run;

data cand(keep=brand price f:);
   retain Brand Price f1-f11 0;
   array p[8];
   array f[11];
   infile cards missover;
   input Brand p1-p8;
   do i = 1 to 8;
      Price = p[i];
      if n(price) or (i = 1 and brand = 11) then do;
         f[brand] = 1; output; f[brand] = 0;
         end;
      end;
   format brand bf.;

   datalines;
 1 0.89 0.94 0.99 1.04 1.09 1.14 1.19 1.24
 2 0.94 0.99 1.04 1.09 1.14 1.19 1.24 1.29
 3 0.99 1.04 1.09 1.14 1.19 1.24
 4 0.89 0.94 0.99 1.04 1.09 1.14
 5 1.04 1.09 1.14 1.19 1.24 1.29
 6 0.89 0.99 1.09 1.19
 7 0.99 1.09 1.19 1.29
 8 0.94 0.99 1.14 1.19
 9 1.09 1.14 1.19 1.24
10 1.14 1.19 1.24 1.29
11
;

proc print; run;
```

The PROC FORMAT step creates a format so that the constant alternative, Brand 11, displays as "None". The DATA step creates the candidate alternatives. The `retain` statement names the variables `Brand` and `Price` first so that they appear in the data set first. The important computational reason for the `retain` statement is it retains the values of the variables `f1-f11` across the passes through the DATA step (rather than setting them to missing each time) and it initializes their values to zero. The first array statement creates an array for the eight price variables, `p1-p8` that are read with the `input` statement. The second array statement creates an array for the eleven flag variables, `f1-f11`, that flag which candidates can be used for each of the 11 alternatives. The `infile` statement with the option `missover` is used so that missing values are automatically provided for lines that do not have eight prices. The `do` statement is used to write each price out as a separate observation in the output data set. The *ith* price is stored in the variable `Price`, and it is written to the output data set if it is not missing. In addition, one missing price value is written to the output data set for the constant alternative. The *ith* flag variable is set to 1 before before the *ith* brand is written to the output data set and its value is restored to zero before going on to the next observation.

The following `%ChoicEff` macro step creates the design, naming `Brand` and `Price` as classification variables:

```
%choiceff(data=cand,                /* candidate set of alternatives        */
          model=class(brand         /* model with brand and                 */
                      brand*price / /* brand by price effects               */
                      zero=none) /  /* zero=none - use all levels           */
                      lprefix=0     /* use just levels in variable labels   */
                      cprefix=1,    /* use one var name char in new names   */
          nsets=24,                 /* number of choice sets                */
          flags=f1-f11,             /* flag which alt can go where, 11 alts */
          seed=462,                 /* random number seed                   */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

Indicator variables are created for all nonmissing levels of the factors. Some of the results are as follows:

Design	Iteration	D-Efficiency	D-Error
1	0	0	.
	1	0	.
		0.00139 (Ridged)	
	2	0	.
		0.00139 (Ridged)	

Design	Iteration	D-Efficiency	D-Error
2	0	0	.
	1	0	.
		0.00140 (Ridged)	
	2	0	.
		0.00140 (Ridged)	

Final Results

Design	1
Choice Sets	24
Alternatives	11
Parameters	54
Maximum Parameters	240
D-Efficiency	0
D-Error	.

n	Variable Name	Label	Variance	DF	Standard Error
1	B1	1	6.5759	1	2.56436
2	B2	2	4.5072	1	2.12303
3	B3	3	4.5405	1	2.13086
4	B4	4	2.8785	1	1.69660
5	B5	5	3.4751	1	1.86415
6	B6	6	3.4899	1	1.86812
7	B7	7	2.8850	1	1.69854
8	B8	8	2.8778	1	1.69640
9	B9	9	3.5155	1	1.87497
10	B10	10	2.8860	1	1.69883
11	BNone	None	.	0	.
12	B1P0D89	1 * 0.89	9.2255	1	3.03736
13	B1P0D94	1 * 0.94	12.2905	1	3.50578
14	B1P0D99	1 * 0.99	10.2609	1	3.20327
15	B1P1D04	1 * 1.04	18.6877	1	4.32293
16	B1P1D09	1 * 1.09	18.6007	1	4.31286
17	B1P1D14	1 * 1.14	8.2432	1	2.87109
18	B1P1D19	1 * 1.19	8.6498	1	2.94105
19	B1P1D24	1 * 1.24	.	0	.
20	B1P1D29	1 * 1.29	.	0	.
21	B2P0D89	2 * 0.89	.	0	.
22	B2P0D94	2 * 0.94	8.1669	1	2.85778
23	B2P0D99	2 * 0.99	7.1946	1	2.68227
24	B2P1D04	2 * 1.04	8.2091	1	2.86515
25	B2P1D09	2 * 1.09	8.2183	1	2.86676
26	B2P1D14	2 * 1.14	10.3850	1	3.22258
27	B2P1D19	2 * 1.19	7.1970	1	2.68272
28	B2P1D24	2 * 1.24	10.2743	1	3.20536
29	B2P1D29	2 * 1.29	.	0	.

30	B3P0D89	3 * 0.89	.	0	.
31	B3P0D94	3 * 0.94	.	0	.
32	B3P0D99	3 * 0.99	7.2137	1	2.68584
33	B3P1D04	3 * 1.04	7.2159	1	2.68624
34	B3P1D09	3 * 1.09	6.2137	1	2.49273
35	B3P1D14	3 * 1.14	7.2513	1	2.69283
36	B3P1D19	3 * 1.19	8.2380	1	2.87020
37	B3P1D24	3 * 1.24	.	0	.
38	B3P1D29	3 * 1.29	.	0	.
39	B4P0D89	4 * 0.89	4.5410	1	2.13097
40	B4P0D94	4 * 0.94	6.5975	1	2.56855
41	B4P0D99	4 * 0.99	8.6131	1	2.93480
42	B4P1D04	4 * 1.04	6.6153	1	2.57202
43	B4P1D09	4 * 1.09	4.9623	1	2.22762
44	B4P1D14	4 * 1.14	.	0	.
45	B4P1D19	4 * 1.19	.	0	.
46	B4P1D24	4 * 1.24	.	0	.
47	B4P1D29	4 * 1.29	.	0	.
48	B5P0D89	5 * 0.89	.	0	.
49	B5P0D94	5 * 0.94	.	0	.
50	B5P0D99	5 * 0.99	.	0	.
51	B5P1D04	5 * 1.04	6.1601	1	2.48195
52	B5P1D09	5 * 1.09	7.2013	1	2.68352
53	B5P1D14	5 * 1.14	5.4956	1	2.34427
54	B5P1D19	5 * 1.19	6.1440	1	2.47872
55	B5P1D24	5 * 1.24	6.1582	1	2.48157
56	B5P1D29	5 * 1.29	.	0	.
57	B6P0D89	6 * 0.89	4.8869	1	2.21063
58	B6P0D94	6 * 0.94	.	0	.
59	B6P0D99	6 * 0.99	4.6185	1	2.14906
60	B6P1D04	6 * 1.04	.	0	.
61	B6P1D09	6 * 1.09	5.5433	1	2.35442
62	B6P1D14	6 * 1.14	.	0	.
63	B6P1D19	6 * 1.19	.	0	.
64	B6P1D24	6 * 1.24	.	0	.
65	B6P1D29	6 * 1.29	.	0	.
66	B7P0D89	7 * 0.89	.	0	.
67	B7P0D94	7 * 0.94	.	0	.
68	B7P0D99	7 * 0.99	4.2574	1	2.06333
69	B7P1D04	7 * 1.04	.	0	.
70	B7P1D09	7 * 1.09	4.9384	1	2.22225
71	B7P1D14	7 * 1.14	.	0	.
72	B7P1D19	7 * 1.19	4.2492	1	2.06136
73	B7P1D24	7 * 1.24	.	0	.
74	B7P1D29	7 * 1.29	.	0	.

75	B8P0D89	8 * 0.89	.	0	.
76	B8P0D94	8 * 0.94	4.2680	1	2.06592
77	B8P0D99	8 * 0.99	4.2502	1	2.06161
78	B8P1D04	8 * 1.04	.	0	.
79	B8P1D09	8 * 1.09	.	0	.
80	B8P1D14	8 * 1.14	4.9127	1	2.21645
81	B8P1D19	8 * 1.19	.	0	.
82	B8P1D24	8 * 1.24	.	0	.
83	B8P1D29	8 * 1.29	.	0	.
84	B9P0D89	9 * 0.89	.	0	.
85	B9P0D94	9 * 0.94	.	0	.
86	B9P0D99	9 * 0.99	.	0	.
87	B9P1D04	9 * 1.04	.	0	.
88	B9P1D09	9 * 1.09	5.5866	1	2.36360
89	B9P1D14	9 * 1.14	4.6559	1	2.15775
90	B9P1D19	9 * 1.19	4.9117	1	2.21623
91	B9P1D24	9 * 1.24	.	0	.
92	B9P1D29	9 * 1.29	.	0	.
93	B10P0D89	10 * 0.89	.	0	.
94	B10P0D94	10 * 0.94	.	0	.
95	B10P0D99	10 * 0.99	.	0	.
96	B10P1D04	10 * 1.04	.	0	.
97	B10P1D09	10 * 1.09	.	0	.
98	B10P1D14	10 * 1.14	4.9756	1	2.23060
99	B10P1D19	10 * 1.19	4.0131	1	2.00327
100	B10P1D24	10 * 1.24	4.5461	1	2.13216
101	B10P1D29	10 * 1.29	.	0	.
102	BNoneP0D89	None * 0.89	.	0	.
103	BNoneP0D94	None * 0.94	.	0	.
104	BNoneP0D99	None * 0.99	.	0	.
105	BNoneP1D04	None * 1.04	.	0	.
106	BNoneP1D09	None * 1.09	.	0	.
107	BNoneP1D14	None * 1.14	.	0	.
108	BNoneP1D19	None * 1.19	.	0	.
109	BNoneP1D24	None * 1.24	.	0	.
110	BNoneP1D29	None * 1.29	.	0	.
				==	
				54	

There are unneeded parameters in our model, and for the moment, that is fine. We see 10 parameters for `Brand`. The constant alternative is the reference alternative. We see 7 parameters for Brand 1's price (8 prices minus 1 = 7), 7 parameters for Brand 2's price, 5 parameters for Brand 3's price (6 prices minus 1 = 5), ..., and 3 parameters for Brand 10's price (4 prices minus 1 = 3). This all looks correct.

The following redundant variable list is displayed in the log:

```
Redundant Variables:

zBNone B1P1D24 B1P1D29 B2P0D89 B2P1D29 B3P0D89 B3P0D94 B3P1D24 B3P1D29 B4P1D14
B4P1D19 B4P1D24 B4P1D29 B5P0D89 B5P0D94 B5P0D99 B5P1D29 B6P0D94 B6P1D04 B6P1D14
B6P1D19 B6P1D24 B6P1D29 B7P0D89 B7P0D94 B7P1D04 B7P1D14 B7P1D24 B7P1D29 B8P0D89
B8P1D04 B8P1D09 B8P1D19 B8P1D24 B8P1D29 B9P0D89 B9P0D94 B9P0D99 B9P1D04 B9P1D24
B9P1D29 B10P0D89 B10P0D94 B10P0D99 B10P1D04 B10P1D09 B10P1D29 BNoneP0D89
BNoneP0D94 BNoneP0D99 BNoneP1D04 BNoneP1D09 BNoneP1D14 BNoneP1D19 BNoneP1D24
BNoneP1D29
```

The following list is the same, except it has been manually reformatted to group the brands:

```
Redundant Variables:

B1P1D24 B1P1D29
B2P0D89 B2P1D29
B3P0D89 B3P0D94 B3P1D24 B3P1D29
B4P1D14 B4P1D19 B4P1D24 B4P1D29
B5P0D89 B5P0D94 B5P0D99 B5P1D29
B6P0D94 B6P1D04 B6P1D14 B6P1D19 B6P1D24 B6P1D29
B7P0D89 B7P0D94 B7P1D04 B7P1D14 B7P1D24 B7P1D29
B8P0D89 B8P1D04 B8P1D09 B8P1D19 B8P1D24 B8P1D29
B9P0D89 B9P0D94 B9P0D99 B9P1D04 B9P1D24 B9P1D29
B10P0D89 B10P0D94 B10P0D99 B10P1D04 B10P1D09 B10P1D29
BNoneP0D89 BNoneP0D94 BNoneP0D99 BNoneP1D04 BNoneP1D09 BNoneP1D14
BNoneP1D19 BNoneP1D24 BNoneP1D29
```

For Brands 1 and 2, we have 7 parameters and the last level for price (8) is the reference level and does not appear in the model. The specification `class(brand brand*price / zero=none)` suppresses having a reference level for brand so that all 10 brands appear along with the constant alternative.

For all brands, the reference level is the last price in the list: 1.24 for brands 1, 3 and 9; 1.29 for brands 2, 5, 7, and 10; and so on. In addition, missing variances and 0 *df* are displayed for each price that does not appear with a brand. Every parameter associated with the constant alternative has a missing variance and 0 *df*. The following step creates the design:

```
%let vars=
BNone B1P1D24 B1P1D29 B2P0D89 B2P1D29 B3P0D89 B3P0D94 B3P1D24 B3P1D29 B4P1D14
B4P1D19 B4P1D24 B4P1D29 B5P0D89 B5P0D94 B5P0D99 B5P1D29 B6P0D94 B6P1D04 B6P1D14
B6P1D19 B6P1D24 B6P1D29 B7P0D89 B7P0D94 B7P1D04 B7P1D14 B7P1D24 B7P1D29 B8P0D89
B8P1D04 B8P1D09 B8P1D19 B8P1D24 B8P1D29 B9P0D89 B9P0D94 B9P0D99 B9P1D04 B9P1D24
B9P1D29 B10P0D89 B10P0D94 B10P0D99 B10P1D04 B10P1D09 B10P1D29 BNoneP0D89
BNoneP0D94 BNoneP0D99 BNoneP1D04 BNoneP1D09 BNoneP1D14 BNoneP1D19 BNoneP1D24
BNoneP1D29;
```

```
%choiceff(data=cand,                /* candidate set of alternatives        */
          model=class(brand         /* model with brand and                 */
                      brand*price / /* brand by price effects               */
                      zero=none) /  /* zero=none - use all levels           */
                      lprefix=0     /* use just levels in variable labels   */
                      cprefix=1,    /* use one var name char in new names   */
          drop=&vars,               /* extra terms to drop                  */
          nsets=24,                 /* number of choice sets                */
          flags=f1-f11,             /* flag which alt can go where, 11 alts */
          seed=462,                 /* random number seed                   */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

Some of the results are as follows:

Design	Iteration	D-Efficiency	D-Error
1	0	0.31384 *	3.18632
	1	0.34074 *	2.93476
	2	0.34074	2.93476

Design	Iteration	D-Efficiency	D-Error
2	0	0	.
	1	0.34101 *	2.93246

Final Results

Design	2
Choice Sets	24
Alternatives	11
Parameters	54
Maximum Parameters	240
D-Efficiency	0.3410
D-Error	2.9325

n	Variable Name	Label	Variance	DF	Standard Error
1	Brand1	Brand 1	4.50417	1	2.12230
2	Brand2	Brand 2	4.52567	1	2.12736
3	Brand3	Brand 3	3.47776	1	1.86487
4	Brand4	Brand 4	3.48724	1	1.86742
5	Brand5	Brand 5	3.49982	1	1.87078
6	Brand6	Brand 6	2.47337	1	1.57270
7	Brand7	Brand 7	2.45738	1	1.56760
8	Brand8	Brand 8	2.45142	1	1.56570
9	Brand9	Brand 9	2.45282	1	1.56615
10	Brand10	Brand 10	2.44575	1	1.56389
11	Brand1Price1	Brand 1 * Price 1	8.19264	1	2.86228
12	Brand1Price2	Brand 1 * Price 2	8.19269	1	2.86229
13	Brand1Price3	Brand 1 * Price 3	8.18940	1	2.86171
14	Brand1Price4	Brand 1 * Price 4	8.23067	1	2.86891
15	Brand1Price5	Brand 1 * Price 5	8.21587	1	2.86633
16	Brand1Price6	Brand 1 * Price 6	8.19365	1	2.86246
17	Brand1Price7	Brand 1 * Price 7	8.23031	1	2.86885
18	Brand2Price1	Brand 2 * Price 1	8.16830	1	2.85802
19	Brand2Price2	Brand 2 * Price 2	8.23185	1	2.86912
20	Brand2Price3	Brand 2 * Price 3	8.21687	1	2.86651
21	Brand2Price4	Brand 2 * Price 4	8.23295	1	2.86931
22	Brand2Price5	Brand 2 * Price 5	8.27059	1	2.87586
23	Brand2Price6	Brand 2 * Price 6	8.22612	1	2.86812
24	Brand2Price7	Brand 2 * Price 7	8.28203	1	2.87785
25	Brand3Price1	Brand 3 * Price 1	6.15558	1	2.48104
26	Brand3Price2	Brand 3 * Price 2	6.17640	1	2.48524
27	Brand3Price3	Brand 3 * Price 3	6.13255	1	2.47640
28	Brand3Price4	Brand 3 * Price 4	6.14840	1	2.47960
29	Brand3Price5	Brand 3 * Price 5	6.11249	1	2.47234
30	Brand4Price1	Brand 4 * Price 1	6.17231	1	2.48441
31	Brand4Price2	Brand 4 * Price 2	6.22760	1	2.49552
32	Brand4Price3	Brand 4 * Price 3	6.12111	1	2.47409
33	Brand4Price4	Brand 4 * Price 4	6.19792	1	2.48956
34	Brand4Price5	Brand 4 * Price 5	6.12131	1	2.47413
35	Brand5Price1	Brand 5 * Price 1	6.21514	1	2.49302
36	Brand5Price2	Brand 5 * Price 2	6.15748	1	2.48143
37	Brand5Price3	Brand 5 * Price 3	6.17697	1	2.48535
38	Brand5Price4	Brand 5 * Price 4	6.16121	1	2.48218
39	Brand5Price5	Brand 5 * Price 5	6.20067	1	2.49011
40	Brand6Price1	Brand 6 * Price 1	4.16170	1	2.04002
41	Brand6Price2	Brand 6 * Price 2	4.11324	1	2.02811
42	Brand6Price3	Brand 6 * Price 3	4.13298	1	2.03297
43	Brand7Price1	Brand 7 * Price 1	4.10703	1	2.02658
44	Brand7Price2	Brand 7 * Price 2	4.11083	1	2.02752
45	Brand7Price3	Brand 7 * Price 3	4.10632	1	2.02641

46	Brand8Price1	Brand	8 * Price 1	4.12107	1	2.03004
47	Brand8Price2	Brand	8 * Price 2	4.10075	1	2.02503
48	Brand8Price3	Brand	8 * Price 3	4.08366	1	2.02081
49	Brand9Price1	Brand	9 * Price 1	4.11157	1	2.02770
50	Brand9Price2	Brand	9 * Price 2	4.10049	1	2.02497
51	Brand9Price3	Brand	9 * Price 3	4.10522	1	2.02614
52	Brand10Price1	Brand	10 * Price 1	4.07896	1	2.01964
53	Brand10Price2	Brand	10 * Price 2	4.09065	1	2.02253
54	Brand10Price3	Brand	10 * Price 3	4.11148	1	2.02768
					==	
					54	

We can see that we now have all the terms for the final model. The following step displays part of the design:

```
proc print data=best(obs=22); id set; by set; var brand price; run;
```

The first two choice sets are as follows:

Set	Brand	Price
1	1	0.94
	2	1.29
	3	1.24
	4	1.14
	5	1.19
	6	1.19
	7	1.09
	8	1.19
	9	1.24
	10	1.24
	None	.
2	1	1.04
	2	1.29
	3	1.14
	4	0.99
	5	1.09
	6	1.19
	7	1.29
	8	1.14
	9	1.24
	10	1.24
	None	.

Alternative Swapping with Price Constraints and Discounts

This example finds a choice design where there are constraints on the price factor. Specifically, for the first alternative, price can have one of three values, \$0.75, \$1.00, or \$1.25. In the second alternative, price will either be a 3%, 4%, or 5% discount of the alternative one price. In the third alternative, price will either be a 6%, 7%, or 8% discount of the alternative one price. This example also has 6 two-level factors, which are used for other attributes. These other attributes require no special handling. Since the goal is to produce a price attribute with price dependencies across alternatives, we will construct a candidate set of choice sets and build the design from that. First, we use the `%MktRuns` macro to get suggestions about candidate set sizes as follows:

```
%mktruns(3 2 ** 6  3 2 ** 6  3 2 ** 6)
```

The levels specification has a "3" for the price attribute for the first alternative, a "3" for the 3 discounts for the second alternative, and a "3" for the 3 discounts for the third alternative. The two-level factors make up the other 6 attributes for each of the three alternatives.

Some of the results are as follows:

Design Summary

Number of Levels	Frequency
2	18
3	3

Saturated = 25
Full Factorial = 7,077,888

Some Reasonable Design Sizes	Violations	Cannot Be Divided By
36	0	
72 *	0	
48	3	9
60	3	9
28	60	3 6 9
32	60	3 6 9
40	60	3 6 9
44	60	3 6 9
52	60	3 6 9
56	60	3 6 9
25 S	231	2 3 4 6 9

```
* - 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.
```

The results show that an orthogonal array exists in 72 runs, so we will try that first using the `%MktEx` macro as follows:

```
%mktex(3 2 ** 6  3 2 ** 6  3 2 ** 6, n=72, seed=289)
```

Some of the results are as follows:

Algorithm Search History

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

The `%MktEx` macro found a 100% efficient orthogonal array. The "`Tab`" note in the algorithm search history in 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, the price factors (`x1`, `x8`, and `x15`) with levels 1, 2, and 3 need to be converted to actual prices. In addition, the discounts are stored in three new factors (`d1`, `d2`, and `d3`. We do not really need these factors; they will just be created for our reference. The following step does the conversion and displays the first 10 choice sets:

```
data cand;                          /* new candidates with recoded price    */
   set randomized;                  /* use randomized design from MktEx     */
   x1  = 0.5 + 0.25 * x1;           /* map 1, 2, 3 to 0.75, 1.0, 1.25       */
   d1  = 0;                         /* discount for first alt is no discount*/
   d2  = 2 + x8;                    /* map 1, 2, 3 to 3%, 4%, 5% discount   */
   d3  = 5 + x15;                   /* map 1, 2, 3 to 6%, 7%, 8% discount   */
   x8  = round(x1 * (1 - d2 / 100), 0.01);/* apply discount to second alt   */
   x15 = round(x1 * (1 - d3 / 100), 0.01);/* apply discount to third alt    */
   run;

proc print data=cand(obs=10); run;
```

The first 10 candidates are as follows:

```
0                                x x x x x   x  x x x x x x
b   x  x x x x x x   x  x 1 1 1 1 1   1  1 1 1 1 2 2 d d d
s   1  2 3 4 5 6 7   8  9 0 1 2 3 4   5  6 7 8 9 0 1 1 2 3

1 0.75 1 2 2 1 1 1 0.72 2 2 2 2 1 2 0.70 2 2 1 2 1 1 0 4 7
2 1.00 1 1 2 1 2 1 0.96 2 2 2 1 2 1 0.94 2 1 1 2 2 2 0 4 6
3 1.00 1 1 1 1 1 1 0.97 2 2 2 1 1 2 0.93 1 1 2 1 1 1 0 3 7
4 1.00 2 2 2 1 2 1 0.96 2 1 1 2 2 1 0.94 2 1 2 1 1 1 0 4 6
5 0.75 2 1 2 2 2 1 0.72 2 1 1 1 1 2 0.70 2 2 2 2 2 1 0 4 7
6 1.25 1 2 2 1 2 2 1.21 1 1 1 2 1 2 1.18 1 2 2 1 2 1 0 3 6
7 1.00 1 1 2 2 2 2 0.97 2 2 1 1 2 2 0.93 2 1 2 2 1 2 0 3 7
8 1.25 2 2 2 1 2 1 1.20 2 2 1 2 1 2 1.15 1 1 1 2 2 2 0 4 8
9 1.25 2 1 1 1 1 2 1.20 2 1 1 2 2 1 1.15 1 1 2 2 2 2 0 4 8
10 1.25 2 2 2 2 1 2 1.19 1 1 2 1 2 2 1.16 1 2 1 2 1 2 0 5 7
```

Next, our goal is to convert our linear candidate design into a choice candidate design. We will need a `Key` data set that provides the rules for conversion, and the `%MktKey` macro can help us by constructing part of this data set. The following step creates the names `x1-x21` in a 3×7 array as follows:

```
%mktkey(3 7)
```

The results are as follows:

x1	x2	x3	x4	x5	x6	x7
x1	x2	x3	x4	x5	x6	x7
x8	x9	x10	x11	x12	x13	x14
x15	x16	x17	x18	x19	x20	x21

We will use these results to construct and display the `Key` data set as follows:

```
data key;
   input (Price x2-x7 Discount) ($);
   datalines;
x1     x2     x3     x4     x5     x6     x7   d1
x8     x9     x10    x11    x12    x13    x14  d2
x15    x16    x17    x18    x19    x20    x21  d3
;

proc print; run;
```

The results are as follows:

Obs	Price	x2	x3	x4	x5	x6	x7	Discount
1	x1	x2	x3	x4	x5	x6	x7	d1
2	x8	x9	x10	x11	x12	x13	x14	d2
3	x15	x16	x17	x18	x19	x20	x21	d3

The price attribute is made from `x1`, `x8`, and `x15`. The discount (informational only) attribute is made from `d1`, `d2`, and `d3`). The other attributes in the choice design are made from the other factors in the linear arrangement in the usual way. We use this information to create the choice design from our linear arrangement that has the actual prices and the discounts as follows:

```
%mktroll(design=cand, key=key, out=cand2)

proc print data=cand2(obs=30); id set; by set; run;
```

The first 10 candidate choice sets are as follows:

Set	_Alt_	Price	x2	x3	x4	x5	x6	x7	Discount
1	1	0.75	1	2	2	1	1	1	0
	2	0.72	2	2	2	2	1	2	4
	3	0.70	2	2	1	2	1	1	7
2	1	1.00	1	1	2	1	2	1	0
	2	0.96	2	2	2	1	2	1	4
	3	0.94	2	1	1	2	2	2	6
3	1	1.00	1	1	1	1	1	1	0
	2	0.97	2	2	2	1	1	2	3
	3	0.93	1	1	2	1	1	1	7
4	1	1.00	2	2	2	1	2	1	0
	2	0.96	2	1	1	2	2	1	4
	3	0.94	2	1	2	1	1	1	6
5	1	0.75	2	1	2	2	2	1	0
	2	0.72	2	1	1	1	1	2	4
	3	0.70	2	2	2	2	2	1	7
6	1	1.25	1	2	2	1	2	2	0
	2	1.21	1	1	1	2	1	2	3
	3	1.18	1	2	2	1	2	1	6
7	1	1.00	1	1	2	2	2	2	0
	2	0.97	2	2	1	1	2	2	3
	3	0.93	2	1	2	2	1	2	7

8	1	1.25	2	2	2	1	2	1	0
	2	1.20	2	2	1	2	1	2	4
	3	1.15	1	1	1	2	2	2	8
9	1	1.25	2	1	1	1	1	2	0
	2	1.20	2	1	1	2	2	1	4
	3	1.15	1	1	2	2	2	2	8
10	1	1.25	2	2	2	2	1	2	0
	2	1.19	1	1	2	1	2	2	5
	3	1.16	1	2	1	2	1	2	7

Next, we use the `%ChoicEff` macro to search for a design as follows:

```
%choiceff(data=cand2,              /* candidate set of choice sets         */
          model=ide(Price)         /* model with quantitative price effect */
                class(x2-x7 / effects), /* binary attributes, effects coded */
          nsets=20,                /* 20 choice sets                       */
          nalts=3,                 /* 3 alternatives per set               */
          morevars=discount,       /* add this var to the output data set  */
          drop=discount,           /* do not add this var to the model     */
          seed=292,                /* random number seed                   */
          options=nodups,          /* no duplicate choice sets             */
          maxiter=10,              /* maximum number of designs to create  */
          beta=zero)               /* assumed beta vector, Ho: b=0         */
```

A subset of the results are as follows:

Final Results

Design	1
Choice Sets	20
Alternatives	3
Parameters	7
Maximum Parameters	40
D-Efficiency	6.2005
D-Error	0.1613

n	Variable Name	Label	Variance	DF	Standard Error
1	Price	Price	41.7548	1	6.46179
2	x21	x2 1	0.0674	1	0.25953
3	x31	x3 1	0.0687	1	0.26204
4	x41	x4 1	0.0600	1	0.24489
5	x51	x5 1	0.0664	1	0.25762
6	x61	x6 1	0.0570	1	0.23866
7	x71	x7 1	0.0693	1	0.26329
				==	
				7	

You can display the covariances as follows:

```
proc print data=bestcov label;
   title 'Variance-Covariance Matrix';
   id __label;
   label __label = '00'x;
   var price x:;
   run;
title;
```

The results are as follows:

Variance-Covariance Matrix

	Price	x2 1	x3 1	x4 1	x5 1	x6 1	x7 1
Price	41.7548	0.055064	-0.045773	0.016390	-0.043134	0.001150	0.076607
x2 1	0.0551	0.067356	-0.002937	-0.003431	0.001635	0.003638	0.006400
x3 1	-0.0458	-0.002937	0.068664	-0.000748	0.000304	0.003032	-0.012407
x4 1	0.0164	-0.003431	-0.000748	0.059972	0.001552	-0.004627	0.003199
x5 1	-0.0431	0.001635	0.000304	0.001552	0.066369	-0.000065	-0.001835
x6 1	0.0012	0.003638	0.003032	-0.004627	-0.000065	0.056960	0.001193
x7 1	0.0766	0.006400	-0.012407	0.003199	-0.001835	0.001193	0.069321

Clearly, the variance for the price attribute is much greater than the variances for the other attributes. This is not surprising. Two points define a line. Having more points is inefficient. We can see how many price points we actually have as follows:

```
proc freq data=best; tables price; run;
```

The results are as follows:

The FREQ Procedure

Price	Frequency	Percent	Cumulative Frequency	Cumulative Percent
0.69	1	1.67	1	1.67
0.7	1	1.67	2	3.33
0.71	4	6.67	6	10.00
0.72	1	1.67	7	11.67
0.73	1	1.67	8	13.33
0.75	4	6.67	12	20.00
0.92	3	5.00	15	25.00
0.94	1	1.67	16	26.67
0.95	3	5.00	19	31.67
0.96	1	1.67	20	33.33
1	4	6.67	24	40.00
1.15	6	10.00	30	50.00
1.16	6	10.00	36	60.00
1.19	6	10.00	42	70.00
1.2	6	10.00	48	80.00
1.25	12	20.00	60	100.00

Note that the `%ChoicEff` macro selects many more \$1.25's than the other levels. Two extreme points define a line. It has a clear maximum it can grab and try to "load up" on. The minimum is a bit fuzzier. Alternatively, you can treat the price attribute as a classification variable as follows:

```
%choiceff(data=cand2,              /* candidate set of choice sets         */
          model=class(Price / zero=none)/* model with qualitative price     */
                class(x2-x7 / effects), /* binary attributes, effects coded */
          nsets=20,                /* 20 choice sets                       */
          nalts=3,                 /* 3 alternatives per set               */
          morevars=discount,       /* add this var to the output data set  */
          drop=discount,           /* do not add this var to the model     */
          seed=292,                /* random number seed                   */
          options=nodups,          /* no duplicate choice sets             */
          maxiter=10,              /* maximum number of designs to create  */
          beta=zero)               /* assumed beta vector, Ho: b=0         */
```

Some of the results are as follows:

Final Results

Design	1
Choice Sets	20
Alternatives	3
Parameters	23
Maximum Parameters	40
D-Efficiency	0
D-Error	.

n	Variable Name	Label	Variance	DF	Standard Error
1	Price0D69	Price 0.69	3.00000	1	1.73205
2	Price0D7	Price 0.7	2.18594	1	1.47849
3	Price0D71	Price 0.71	2.25000	1	1.50000
4	Price0D72	Price 0.72	2.00000	1	1.41421
5	Price0D73	Price 0.73	3.00000	1	1.73205
6	Price0D75	Price 0.75	.	0	.
7	Price0D92	Price 0.92	3.00000	1	1.73205
8	Price0D93	Price 0.93	2.06302	1	1.43632
9	Price0D94	Price 0.94	3.00000	1	1.73205
10	Price0D95	Price 0.95	3.00000	1	1.73205
11	Price0D96	Price 0.96	3.00000	1	1.73205
12	Price0D97	Price 0.97	2.17145	1	1.47358
13	Price1	Price 1	.	0	.
14	Price1D15	Price 1.15	3.00000	1	1.73205
15	Price1D16	Price 1.16	3.00000	1	1.73205
16	Price1D18	Price 1.18	3.00000	1	1.73205
17	Price1D19	Price 1.19	3.00000	1	1.73205
18	Price1D2	Price 1.2	3.00000	1	1.73205
19	Price1D21	Price 1.21	3.00000	1	1.73205
20	Price1D25	Price 1.25	.	0	.
21	x21	x2 1	0.06091	1	0.24681
22	x31	x3 1	0.08720	1	0.29530
23	x41	x4 1	0.08171	1	0.28585
24	x51	x5 1	0.07465	1	0.27322
25	x61	x6 1	0.07592	1	0.27554
26	x71	x7 1	0.08752	1	0.29584
				==	
				23	

It is interesting to note that the price parameters for our alternative 1 levels are all missing with zero *df*. Because the prices for alternatives two and three are discounts of the alternative one price, they come before the alternative one price in the list of sorted prices. Then because of the dependencies in the prices, only the discounted prices are estimable in the model. You can see the price frequencies as follows:

```
proc freq data=best; tables price; run;
```

The results are as follows:

The FREQ Procedure

Price	Frequency	Percent	Cumulative Frequency	Cumulative Percent
0.69	2	3.33	2	3.33
0.7	3	5.00	5	8.33
0.71	4	6.67	9	15.00
0.72	3	5.00	12	20.00
0.73	2	3.33	14	23.33
0.75	7	11.67	21	35.00
0.92	2	3.33	23	38.33
0.93	3	5.00	26	43.33
0.94	2	3.33	28	46.67
0.95	2	3.33	30	50.00
0.96	2	3.33	32	53.33
0.97	3	5.00	35	58.33
1	7	11.67	42	70.00
1.15	2	3.33	44	73.33
1.16	2	3.33	46	76.67
1.18	2	3.33	48	80.00
1.19	2	3.33	50	83.33
1.2	2	3.33	52	86.67
1.21	2	3.33	54	90.00
1.25	6	10.00	60	100.00

If your goal is a more even price distribution, designating price as a classification variable will work better than designating it as an identity variable. Note that you can design the experiment designating price as a classification variable and analyze it with price as an identity variable. The opposite approach is not guaranteed to work. If you generate a design using a model with one *df* for price, you should not attempt to then fit a model with multiple *df* for price.

You might be able to do better in this example by using a larger candidate set. Multiplying 72 by numbers like 2, 3, 4, or 6 might help, for example, as follows:

```
%mktex(3 2 ** 6  3 2 ** 6  3 2 ** 6, n=2 * 72, seed=289)

%mkteval;
```

If you do this, be sure to look at the n-way frequencies in the `%MktEval` output to ensure that you are not simply getting duplicate candidate choice sets. Since the point of this example is to illustrate how to construct the price attribute with discounts and dependencies, and that has already been accomplished, we will not explore varying candidate sets sizes here.

Multiple Alternative and Choice Set Types

This next example is an order of magnitude more complicated than the kinds of design problems that most analysts ever encounter. If you are just learning the `%ChoicEff` macro, you should skip ahead to page 916 and come back to this section later. If instead, you are a sophisticated analyst, this example shows you the power that you have to make complicated designs using the `%ChoicEff` macro and SAS programming statements.

The goal in this example is to create a design for a choice study. Choice sets consist of two different types of alternatives, and the choice study needs to consist of choice sets with an even mix of 6, 7, and 8 alternatives. Each choice set must contain a subset (of size 6, 7, or 8) of the 10 available brands, and a brand may not appear more than once in any given choice set. There are two types of alternatives, because there are two types of brands. Within each type of brand, the numbers of factor are the same, although they could be different. To make all of this clearer, three possible choice sets are as follows:

Set	Alt	Brand	Set Type	Alt Type	x1	x2	x3	x4	x5	x6
11	1	1	1	1	2	3	2	2	1	1
	2	2	1	1	1	2	2	1	1	3
	3	3	1	2	2	2	1	2	1	4
	4	5	1	2	5	3	1	3	2	2
	5	7	1	2	1	1	3	3	2	1
	6	9	1	2	4	4	4	1	2	2
23	1	1	2	1	2	1	2	3	1	4
	2	2	2	1	1	3	3	1	1	1
	3	3	2	2	2	2	2	1	2	1
	4	4	2	2	3	2	1	2	1	1
	5	6	2	2	1	2	4	2	2	3
	6	7	2	2	3	1	3	2	2	4
	7	10	2	2	4	1	2	3	1	2
56	1	1	3	1	2	2	3	1	2	4
	2	2	3	1	2	3	3	3	2	3
	3	3	3	2	5	4	2	1	1	1
	4	4	3	2	2	1	1	2	2	2
	5	5	3	2	4	1	2	3	1	2
	6	6	3	2	5	2	2	2	2	2
	7	7	3	2	3	3	4	1	1	2
	8	8	3	2	4	4	2	3	1	1

The first choice set (set 11) is of type 1 (6 alternatives), the second (set 23) is of type 2 (7 alternatives), and the third (set 56) is of type 3 (8 alternatives). In this design, `x1` will become the price factor, and `x2` will become the product size factor. For Brand 1 and Brand 2 alternatives, `x1` has 2 levels and `x2` has 3 levels. For Brand 3 through Brand 10 alternatives, `x1` has 5 levels and `x2` has 4 levels. For all brands, `x3` has 4 levels, `x4` has 3 levels, `x5` has 2 levels, and `x6` has 4 levels.

With a complicated problem such as this, there is always more than one way to proceed. In this example, we take the following approach:

- Create a candidate set of alternatives for the first type of alternative (Brand 1 and Brand 2).
- Create a candidate set of alternatives for the second type of alternative (Brand 3 through Brand 10).
- Combine the candidate sets and create the brand factor.
- Use the `%ChoicEff` macro to search the candidate set of alternatives and create a set of candidate choice sets with 6 alternatives.
- Use the `%ChoicEff` macro to search the candidate set of alternatives and create a set of candidate choice sets with 7 alternatives.
- Use the `%ChoicEff` macro to search the candidate set of alternatives and create a set of candidate choice sets with 8 alternatives.
- Combine the three individual candidate sets of choice sets into one big candidate set.
- Extract just the choice sets where no brand appears more than once.
- Array the candidate choice sets with a fixed number of alternatives (8). Add all missing alternatives to the choice sets with 6 or 7 alternatives, and provide a weight variable that identifies those alternatives that are actually used (weight = 1) and those alternatives that are not used (weight = 0).
- Search this candidate set of choice sets using the `%ChoicEff` macro.
- Display the final design.
- Check the final design for duplicate choice sets.

The rest of this section works through each of these steps and discusses the programming involved in creating this design.

Candidate Set of Alternatives

In this example, there are two different types of brands. One type is based on current brands and the other type is based on some new proposed brands. The number of factor levels is different for the two types. In this first set of steps, two candidate sets of candidate alternatives are created. All attributes but brand are created now; brand is added in later. The following statements create two sets of candidate alternatives, one with one type of alternative, and the other with the other type of alternative:

```
* Eliminate or replace options=quick when you know what you are doing.
*
* Increase n=.  Small values are good for testing.
* Larger values should give better designs.
;

%mktruns(2 3 4 3 2 4, interact=1*2)

%mktex(2 3 4 3 2 4,                /* attrs for proposed brands           */
       interact=1*2,               /* x1*x2 interaction                   */
       n=24,                       /* number of candidate alternatives    */
       seed=292,                   /* random number seed                  */
       out=d1,                     /* output experimental design          */
       options=quick)              /* provides a quick run initially      */

%mktruns(5 4 4 3 2 4, interact=1*2)

%mktex(5 4 4 3 2 4,                /* attrs for current brands            */
       interact=1*2,               /* x1*x2 interaction                   */
       n=40,                       /* number of candidate alternatives    */
       seed=292,                   /* random number seed                  */
       out=d2,                     /* output experimental design          */
       options=quick)              /* provides a quick run initially      */
```

For each of the two types of factors, the `%MktRuns` macro is used to suggest a size for the candidate design, and then the `%MktEx` macro is used to create the candidate alternatives. In both cases, one of the smaller suggestions from the `%MktRuns` macro is used. When you have your code thoroughly tested, then you should consider both specifying larger values for `n=` and removing `options=quick`. Both are specified for now to make these and subsequent steps run faster.

Some of the output from the first `%MktRuns` step is as follows:

Some Reasonable Design Sizes	Violations	Cannot Be Divided By
144	0	
72	1	16
48	2	9 18
96	2	9 18
192	2	9 18
24	3	9 16 18
120	3	9 16 18
168	3	9 16 18
36	7	8 16 24
108	7	8 16 24
15 S	23	2 4 6 8 9 12 16 18 24

Some of the output from the second `%MktRuns` step is as follows:

```
   Some Reasonable                  Cannot Be
      Design Sizes     Violations   Divided By

               120              5   16 80
                80              7    3  6 12 15 60
               160              7    3  6 12 15 60
                60              9    8 16 40 80
               180              9    8 16 40 80
                48             10    5 10 15 20 40 60 80
                96             10    5 10 15 20 40 60 80
               144             10    5 10 15 20 40 60 80
               192             10    5 10 15 20 40 60 80
                40             12    3  6 12 15 16 60 80
                29 S           25    2  3  4  5  6  8 10 12 15 16 20 40 60 80
```

Based on these results, candidate sets of alternatives of size 24 and 40 are selected. Later, once the code is tested and working, you should try larger sizes that will make the program run more slowly. They might also result in better designs. These results suggest sizes like 144 and 120.

The `%MktEx` macro is run to make the two candidate designs. In both cases, one two-way interaction, price by product size, is required to be estimable. The results are stored in two data sets, `d1` and `d2`. The designs and iteration histories for these steps are not shown here.

Combine the Candidate Sets

The following step combines the two sets of candidate alternatives into one set:

```
* Create full candidate design.;
data all;
   retain f1-f8 Brand 1;
   set d1(in=d1) d2(in=d2);

   * Make alternative-specific changes to the price
   * and other attribute levels in here as necessary.
   ;

   * For Brand 1 and Brand 2, write out the other attrs;
   if d1 then do;
      do brand = 1 to 2; output; end;
      end;

   * For Brand 3 through Brand 10, write out the other attrs;
   if d2 then do;
      do brand = 3 to 10; output; end;
      end;
   run;
```

Subsequent steps use the alternative-swapping algorithm to make designs, so flags must be added to the candidate set indicating which candidate can appear in which alternative position in the choice design. In this case, any candidate alternative can appear in any design position. Hence, all flags are constant, and all are set to 1 for every candidate. Eight flag variables, `f1-f8`, are set to 1 for the entire candidate set using the `retain` statement. The `Brand` factor is also named in the `retain` statement. This is just for aesthetic reasons, so that the brand variable gets positioned in the SAS data set after the flag variables and before the `x1-x8` attributes that come in with the `set` statement. The `set` statement reads and concatenates the two candidate sets. The data set options `in=d1` and `in=d2` create two binary variables that are 1 or true when the observation comes from the designated data set and 0 otherwise. With a `set` statement specification like this, when `d1` is 1 then `d2` must be zero and vice versa. Hence, you could just create one of these variables. SAS automatically drops these variables from the output data set.

The remaining steps make copies of the candidates, one copy for each brand that applies. There could be more statements here changing the levels in alternative-specific ways. For example, actual sizes and prices could be substituted for the raw (1, 2, 3, ...) design values, and prices could be assigned differently for the different brands or brand types.

Search the Candidate Set of Alternatives

The following three steps search the full candidate set of alternatives and create designs with 6, 7, and 8 alternatives:

```
* For the next three calls to the choiceff macro:
* Recode model with alternative-specific effects?
* Drop maxiter=1 or increase the value later.
* Consider increasing nsets= later.
;

                                  /* get a design with 6 alternatives     */
%choiceff(data=all,               /* candidate set of alternatives        */
          bestout=b1,             /* name of output design data set       */
                                  /* model with main effects, interaction */
          model=class(brand x1-x6 x1 * x2),
          flags=f1-f6,            /* flag which alt can go where, 6 alts  */
          nsets=20,               /* number of choice sets                */
          maxiter=1,              /* maximum number of designs to make    */
          seed=109,               /* random number seed                   */
          beta=zero)              /* assumed beta vector, Ho: b=0         */

                                  /* get a design with 7 alternatives     */
%choiceff(data=all,               /* candidate set of alternatives        */
          bestout=b2,             /* name of output design data set       */
                                  /* model with main effects, interaction */
          model=class(brand x1-x6 x1 * x2),
          flags=f1-f7,            /* flag which alt can go where, 7 alts  */
          nsets=20,               /* number of choice sets                */
          maxiter=1,              /* maximum number of designs to make    */
          seed=114,               /* random number seed                   */
          beta=zero)              /* assumed beta vector, Ho: b=0         */
```

```
                                    /* get a design with 8 alternatives     */
%choiceff(data=all,                 /* candidate set of alternatives        */
          bestout=b3,               /* name of output design data set       */
                                    /* model with main effects, interaction */
          model=class(brand x1-x6 x1 * x2),
          flags=f1-f8,              /* flag which alt can go where, 8 alts  */
          nsets=20,                 /* number of choice sets                */
          maxiter=1,                /* maximum number of designs to make    */
          seed=121,                 /* random number seed                   */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

These three designs consist of full choice sets consisting of differing numbers of alternatives. They are further processed in subsequent steps to remove sets with duplicate brands, then they are searched to make the final design. These three steps differ in only two important ways. The `flags=` variable names 6, 7, and 8 flag variables, which means that the steps produce 6, 7, and 8 alternative choice sets. Also, the output data sets with the best designs are all given different names. These steps could have alternative-specific factors coded. Later, when the code is all debugged, you can increase the `maxiter=` value to make more designs from which to choose the best one.

Create a Candidate Set of Choice Sets

The following step combines the three output data sets into one:

```
* Concatenate the three designs, create a new Set variable,
* and flag the three different sizes of choice sets with SetType=1, 2, 3.
* We will use this (with a bit more modification) as a candidate set for
* making the final design.
;
data best(keep=Set Brand SetType AltType x1-x6);
   set b1(in=b1) b2(in=b2) b3(in=b3);
   if set ne lag(set) then newset + 1;
   SetType = b1 + 2 * b2 + 3 * b3;
   AltType = 1 + (brand gt 2);
   set = newset;
   run;
```

Again, the `in=` option is used to flag which observations come from which data set. A new `SetType` variable is created with values of 1 (`b1`) when the observation is of the first type (from the first data set), 2 (2 * `b2`) when the observation is of the from the second data set, and 3 (3 * `b3`) when the observation is of the from the third data set. Only one of the `b1-b3` variables is true or 1 at a time. The type of alternative is also stored here in the variable `AltType`. One other thing is done in this step. Each input data set has its own choice set ID variable, `set`, so this variable starts over at one for each type of choice set. A new `set` variable is created that does not start over at one. Whenever the original `set` variable changes, a new set variable is incremented, and its value is stored in place of the original set variable.

Exclude Choice Sets with Duplicate Brands

There is nothing in the preceding steps that ensures that brands occur only once in each choice set. The following statements identity the choice sets where each brand occurs only once and excludes the choice sets where brands occur more than once:

```
* Extract just the choice sets where no brand appears more than once.
* In other words, if the maximum frequency by set is 1, keep it.
* Start by seeing how often each brand occurs within each set;
;
proc freq data=best noprint; tables set * brand / out=list; run;

* Find the maximum frequency.;
proc means noprint; var count; by set;
   output out=maxes(where=(_stat_ eq 'MAX'));
   run;

* Output the set number if the maximum frequency is one.;
data sets; set; if count = 1; keep set; run;

* Select the sets where the maximum frequency is one.;
data best; merge best sets(in=one); by set; if one; run;

* Report on the set and SetType variables as an error check.;
proc freq; tables set * SetType / list; run;

* Sort the design by brand within set.;
proc sort data=best; by set brand; run;

* Add alternative numbers within set.  Get consecutive set numbers again.;
data best(drop=oldset);
   set best(rename=(set=oldset)); by oldset;
   if first.oldset then do; Alt = 0; Set + 1; end;
   alt + 1;
   call symputx('maxset', set);
   run;

* Display the candidate design.;
proc print; var settype alttype brand x1-x6; by set; id set alt; run;
```

First, PROC FREQ is used to create a list of the number of times each brand occurs in each choice set. This list is stored in a SAS data set. PROC MEANS is used to process this data set and output the maximum brand frequency within each choice set. Next, a data set SETS is created that contains only the choice set numbers where the maximum brand frequency is 1. These are the sets we want. Next, the full candidate set is merged with the list of choice set numbers. Choice sets that are represented in both input data sets are kept, and the rest are deleted. PROC FREQ is run to create a list of observation types within choice set as a check on the results. Some of the results are as follows:

The FREQ Procedure

Set	SetType	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	1	6	1.46	6	1.46
.					
.					
.					
20	1	6	1.46	120	29.13
21	2	7	1.70	127	30.83
.					
.					
.					
40	2	7	1.70	260	63.11
41	3	8	1.94	268	65.05
.					
.					
.					
54	3	8	1.94	372	90.29
56	3	8	1.94	380	92.23
57	3	8	1.94	388	94.17
58	3	8	1.94	396	96.12
59	3	8	1.94	404	98.06
60	3	8	1.94	412	100.00

The full results show that all choice sets in the range 1–20 are of type 1, all choice sets in the range 21–40 are of type 2, all choice sets in the range 41–60 are of type 3. Furthermore, some choice sets (e.g. set 55) have been excluded.

This candidate set of choice sets is still not in the final form for the `%ChoicEff` macro to search. The problem is the `%ChoicEff` macro insists that candidate sets of choice sets must all contain the same number of alternatives. This is not a problem, because a mechanism is in place for dealing with varying numbers of alternatives. Extra (dummy) alternatives are added and given zero weight. This is accomplished in the following steps:

```
* Make the choice set alternative numbers you would have if all
* sets had the maximum 8 alternatives.  We will need this kind of layout
* because ChoicEff assumes all sets have the same alternatives, and uses
* weights when they don't.
;
data frame;
   do Set = 1 to &maxset;
      do Alt = 1 to 8;
         output;
         end;
      end;
   run;

* Add missing observations to the sets with 6 and 7 alternatives.
* Flag alternatives actually there with w = 1 and the rest with w = 0.
;
data all; merge frame best(in=b); by set alt; w = b; run;

proc print; var settype brand alttype x1-x6; by set; id set alt w; run;
```

The first step creates a SAS data set with the choice set and alternative numbers that we need. The number of choice sets comes from a macro variable set in a previous step. This is the number of choice sets that are left after the sets with duplicate brands are excluded. This data set is merged with the actual design and alternatives that come from the actual design are flagged with a weight of one (`w` = 1).

Three of the choice sets are as follows:

Set	Alt	w	Set Type	Brand	Alt Type	x1	x2	x3	x4	x5	x6
1	1	1	1	5	2	5	1	1	3	1	3
	2	1	1	6	2	4	4	4	1	2	2
	3	1	1	7	2	3	3	1	2	2	3
	4	1	1	8	2	5	4	2	1	1	1
	5	1	1	9	2	5	2	3	3	1	4
	6	1	1	10	2	5	3	1	3	2	2
	7	0	.	.	.	.	.	.	.	.	.
	8	0	.	.	.	.	.	.	.	.	.
21	1	1	2	1	1	1	2	2	1	1	3
	2	1	2	2	1	2	3	2	2	1	1
	3	1	2	3	2	5	4	3	3	2	3
	4	1	2	4	2	3	1	3	2	2	4
	5	1	2	6	2	4	2	3	1	2	2
	6	1	2	8	2	2	1	1	2	2	2
	7	1	2	9	2	2	4	4	3	2	1
	8	0	.	.	.	.	.	.	.	.	.

41	1	1	3	3	2	5	1	1	3	1	3
	2	1	3	4	2	4	1	1	1	2	1
	3	1	3	5	2	5	3	1	3	2	2
	4	1	3	6	2	1	4	4	3	2	4
	5	1	3	7	2	3	4	4	3	1	1
	6	1	3	8	2	2	4	2	1	1	3
	7	1	3	9	2	3	1	3	2	2	4
	8	1	3	10	2	1	2	3	1	1	4

You can see that each choice set has 8 data set observations even if it has only 6 or 7 alternatives. This is the form that is needed for making the final design.

Search the Candidate Set of Choice Sets

The following step creates the final design:

```
* Create a final design with 3 types of choice sets,
* 12 with 6 alternatives, 12 with 7, and 12 with 8, using weights
* to ignore the dummy alternatives.
* The candidate set has all three types of choice sets.
;

%choiceff(data=all,                 /* candidate set of choice sets         */
                                    /* model with main effects, interaction */
          model=class(brand x1-x6 x1 * x2),
          nalts=8,                  /* number of alternatives               */
          weight=w,                 /* weight to ignore dummy alternatives  */
          types=12 12 12,           /* number of each type of set           */
          typevar=settype,          /* choice set types variable            */
          nsets=36,                 /* number of choice sets                */
          maxiter=1,                /* maximum number of designs to make    */
          seed=396,                 /* random number seed                   */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

The combination of the `typevar=settype` option and the `types=12 12 12` option ensures that each type of choice set appears in the design exactly 12 times. The `weight=` option ensures that only the actual alternatives are used in the design.

The main results from the `%ChoicEff` macro are as follows:

Final Results

Design	1
Choice Sets	36
Alternatives	8
Parameters	37
Maximum Parameters	252
D-Efficiency	2.1018
D-Error	0.4758

n	Variable Name	Label	Variance	DF	Standard Error
1	Brand1	Brand 1	0.72908	1	0.85386
2	Brand2	Brand 2	0.78231	1	0.88448
3	Brand3	Brand 3	0.58411	1	0.76427
4	Brand4	Brand 4	0.56562	1	0.75208
5	Brand5	Brand 5	0.53984	1	0.73473
6	Brand6	Brand 6	0.59600	1	0.77201
7	Brand7	Brand 7	0.58271	1	0.76335
8	Brand8	Brand 8	0.58995	1	0.76808
9	Brand9	Brand 9	0.55157	1	0.74268
10	x11	x1 1	1.61917	1	1.27247
11	x12	x1 2	1.44374	1	1.20156
12	x13	x1 3	1.61199	1	1.26964
13	x14	x1 4	1.55168	1	1.24566
14	x21	x2 1	1.47910	1	1.21618
15	x22	x2 2	1.49892	1	1.22430
16	x23	x2 3	1.58831	1	1.26028
17	x31	x3 1	0.31441	1	0.56072
18	x32	x3 2	0.33885	1	0.58211
19	x33	x3 3	0.34725	1	0.58928
20	x41	x4 1	0.21259	1	0.46108
21	x42	x4 2	0.22298	1	0.47220
22	x51	x5 1	0.12020	1	0.34669
23	x61	x6 1	0.29928	1	0.54706
24	x62	x6 2	0.30377	1	0.55115
25	x63	x6 3	0.35651	1	0.59709
26	x11x21	x1 1 * x2 1	2.95839	1	1.72000
27	x11x22	x1 1 * x2 2	3.05417	1	1.74762
28	x11x23	x1 1 * x2 3	3.05268	1	1.74719
29	x12x21	x1 2 * x2 1	2.86368	1	1.69224
30	x12x22	x1 2 * x2 2	2.70056	1	1.64334
31	x12x23	x1 2 * x2 3	3.13789	1	1.77141
32	x13x21	x1 3 * x2 1	3.09384	1	1.75893
33	x13x22	x1 3 * x2 2	2.95469	1	1.71892
34	x13x23	x1 3 * x2 3	3.01531	1	1.73647
35	x14x21	x1 4 * x2 1	2.79842	1	1.67285
36	x14x22	x1 4 * x2 2	3.07207	1	1.75273
37	x14x23	x1 4 * x2 3	3.36417	1	1.83417
				==	
				37	

The following steps display the final design and check it for duplicate choice sets:

```
* Display final design.;
proc print;
   var brand settype alttype x1-x6;
   by notsorted set;
   id set alt;
   where w;
   run;

* Check for duplicate choice sets.;
proc freq data=best; tables set; run;
```

Three of the final choice sets are as follows:

Set	Alt	Brand	Set Type	Alt Type	x1	x2	x3	x4	x5	x6
11	1	1	1	1	2	3	2	2	1	1
	2	2	1	1	1	2	2	1	1	3
	3	3	1	2	2	2	1	2	1	4
	4	5	1	2	5	3	1	3	2	2
	5	7	1	2	1	1	3	3	2	1
	6	9	1	2	4	4	4	1	2	2
23	1	1	2	1	2	1	2	3	1	4
	2	2	2	1	1	3	3	1	1	1
	3	3	2	2	2	2	2	1	2	1
	4	4	2	2	3	2	1	2	1	1
	5	6	2	2	1	2	4	2	2	3
	6	7	2	2	3	1	3	2	2	4
	7	10	2	2	4	1	2	3	1	2
56	1	1	3	1	2	2	3	1	2	4
	2	2	3	1	2	3	3	3	2	3
	3	3	3	2	5	4	2	1	1	1
	4	4	3	2	2	1	1	2	2	2
	5	5	3	2	4	1	2	3	1	2
	6	6	3	2	5	2	2	2	2	2
	7	7	3	2	3	3	4	1	1	2
	8	8	3	2	4	4	2	3	1	1

The choice set numbers refer to the input candidate design, so numbers can go up and down. Only the alternatives that are actually used are displayed. The full design shows that different subsets of brands appear in different choice sets and no brand appears more than once in any particular choice set.

The PROC FREQ output (not shown) shows that each alternative occurs equally often (8 times) in the final design. Note that the full design including the dummy alternatives is input to PROC FREQ.

Most designs are not nearly as complicated as this one. However, it is good to know that when complex design problems come up, the tools are there to tackle them.

Making the Candidate Set

The %ChoicEff macro can be used in two ways, either with a candidate set of alternatives or with a candidate set of choice sets. Either way, typically you will use the %MktEx macro to make the candidate set. Before you make the candidate set, you have to decide how big to make it. You can use the %MktRuns macro to get some ideas. Next, based on the information provided by the %MktRuns macro, you make sets of different sizes, try each one, and then see which one works best. Sometimes, bigger is better; other times it is not. It is always the case that a very small candidate set that contains all of the right information for constructing the optimal design is better than a very large candidate set that contains many additional and nonoptimal candidates. However, you typically cannot know how good a candidate set is until you try using it. Typically, you should begin by trying a few iterations using the smallest candidate set that you can reasonably make. Then you should try increasingly bigger sizes, again with just a few iterations. The number of iterations might be on the order of less than ten up to several hundred depending on the problem. At this point, you should not let the %ChoicEff macro run for more than a few minutes. Based on the results, you should pick the size that seems to be working best, and then try more iterations with it. This process is illustrated in the rest of this section.

Candidate Set of Choice Sets

This example uses the %ChoicEff macro to create an efficient choice design from a set of candidate choice sets. The experiment has 3 three-level attributes and three alternatives. First, the %MktRuns macro is run to get suggestions for design sizes. The model specification is 3^9. Later, the design with 9 three-level factors is converted to a set of choice sets with three attributes and three alternatives. The following step produces the design suggestions:

```
%mktruns(3 ** 9)
```

Some of the results are as follows:

```
Saturated      = 19
Full Factorial = 19,683

Some Reasonable                      Cannot Be
   Design Sizes       Violations     Divided By

          27 *               0
          36 *               0
          45 *               0
          54 *               0
          21                36     9
          24                36     9
          30                36     9
          33                36     9
          39                36     9
          42                36     9
          19 S              45     3 9
```

```
          * - 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.
```

Explicitly, these results suggest using the sizes 27, 36, 45, and 54. Extrapolating, the results suggest using sizes that are multiples of powers of 3 beginning with 27. The following statements consider some relevant values:

```
%mktex(3 ** 9, n=27, seed=292)

%mktroll(design=randomized, key=3 3, out=cand)

%choiceff(data=cand,                /* candidate set of choice sets         */
          model=class(x1-x3 / sta), /* model with stdz orthogonal coding    */
          seed=513,                 /* random number seed                   */
          maxiter=5,                /* maximum number of designs to make    */
          options=relative nodups,  /* display relative D-efficiency        */
          nsets=18,                 /* number of choice sets                */
          nalts=3,                  /* number of alternatives               */
          beta=zero)                /* assumed beta vector, Ho: b=0         */

%mktex(3 ** 9, n=36, seed=292)
%mktroll(design=randomized, key=3 3, out=cand)
%choiceff(data=cand, model=class(x1-x3 / sta), seed=513, maxiter=5,
          options=relative nodups, nsets=18, nalts=3, beta=zero)

%mktex(3 ** 9, n=45, seed=292)
%mktroll(design=randomized, key=3 3, out=cand)
%choiceff(data=cand, model=class(x1-x3 / sta), seed=513, maxiter=5,
          options=relative nodups, nsets=18, nalts=3, beta=zero)

%mktex(3 ** 9, n=54, seed=292)
%mktroll(design=randomized, key=3 3, out=cand)
%choiceff(data=cand, model=class(x1-x3 / sta), seed=513, maxiter=5,
          options=relative nodups, nsets=18, nalts=3, beta=zero)

%mktex(3 ** 9, n=3 ** 4, seed=292)
%mktroll(design=randomized, key=3 3, out=cand)
%choiceff(data=cand, model=class(x1-x3 / sta), seed=513, maxiter=5,
          options=relative nodups, nsets=18, nalts=3, beta=zero)

%mktex(3 ** 9, n=3 ** 5, seed=292)
%mktroll(design=randomized, key=3 3, out=cand)
%choiceff(data=cand, model=class(x1-x3 / sta), seed=513, maxiter=5,
          options=relative nodups, nsets=18, nalts=3, beta=zero)
```

```
%mktex(3 ** 9, n=3 ** 6, seed=292)
%mktroll(design=randomized, key=3 3, out=cand)
%choiceff(data=cand, model=class(x1-x3 / sta), seed=513, maxiter=5,
          options=relative nodups, nsets=18, nalts=3, beta=zero)

%mktex(3 ** 9, n=3 ** 7, seed=292)
%mktroll(design=randomized, key=3 3, out=cand)
%choiceff(data=cand, model=class(x1-x3 / sta), seed=513, maxiter=5,
          options=relative nodups, nsets=18, nalts=3, beta=zero)

%mktex(3 ** 9, n=3 ** 8, seed=292)
%mktroll(design=randomized, key=3 3, out=cand)
%choiceff(data=cand, model=class(x1-x3 / sta), seed=513, maxiter=5,
          options=relative nodups, nsets=18, nalts=3, beta=zero)

%mktex(3 ** 9, n=3 ** 9, seed=292)
%mktroll(design=randomized, key=3 3, out=cand)
%choiceff(data=cand, model=class(x1-x3 / sta), seed=513, maxiter=5,
          options=relative nodups, nsets=18, nalts=3, beta=zero)
```

In each case, a candidate set is constructed with nine attributes (one for each of the three attributes for each of the three alternatives), it is rolled into a choice design, then the `%ChoicEff` macro is used to create a choice design from the candidate set of choice sets. More information about these macros can be found elsewhere in this chapter and throughout the examples in the design chapter beginning on page 53 and the "Discrete Choice" chapter beginning on page 285. The results of all of these steps are not shown, but a summary of the resulting *D*-Efficiencies is as follows:

n	*D*-Efficiency
27	13.574097
36	13.818249
45	13.830425
54	14.209785
81	15.218103
243	15.286949
729	16.635262
2,187	18.000000
6,561	17.962771
19,683	17.962771

In this example, with these seeds and this number of iterations, the *D*-efficiency increases with the size of the candidate set up to `n=2187`, then it slightly decreases. *D*-efficiency first increases as the richness of the candidate set increases, then it decreases as the candidate set contains more nonoptimal candidates from which to choose. The results would quite likely be different with different seeds, different numbers of iterations, and different problems. However, this pattern of results is not unusual.

In this problem, the model specification is simple enough that run time is not an issue even with large candidate sets, so you could easily try more than five iterations, particularly with the smaller candidate sets. With other problems, you need to be careful to not make candidate sets that are too big. For more complicated models, candidate sets of several thousand choice sets might be too big.

Examination of the results for `n=2187` (not shown) shows that this design is in fact optimal. Had it not been, you would have run the `%ChoicEff` macro again with candidate set, this time requesting more iterations.

Candidate Set of Alternatives

This section illustrates design creation with a candidate set of alternatives. This section illustrates the fact that a bigger candidate set is not always better. The goal is to make a design with 6 three-level factors in six choice sets each with three alternatives. The following `%MktRuns` macro step suggests candidate set sizes:

```
%mktruns(3 ** 6)
```

Some of the results are as follows:

```
Saturated      = 13
Full Factorial = 729

Some Reasonable                      Cannot Be
   Design Sizes       Violations     Divided By

          18 *              0
          27 *              0
          36 *              0
          15               15     9
          21               15     9
          24               15     9
          30               15     9
          33               15     9
          13 S             21     3 9
          14               21     3 9

* - 100% Efficient design can be made with the MktEx macro.
S - Saturated Design - The smallest design that can be made.
```

Explicitly, these results suggest using the sizes 18, 27, and 36. More generally, the results show that suitable candidate set sizes include multiples of powers of three that are greater than $6(3-1) = 12$ including 18, 27, 36, 54, 72, 81, 108, and so on. The following steps make and search candidate sets of varying sizes:

```
%mktex(3 ** 6, n=18, seed=306)

%choiceff(data=design,              /* candidate set of alternatives        */
          model=class(x1-x6 / sta), /* model with stdz orthogonal coding    */
          maxiter=400,              /* maximum number of designs to make    */
          seed=121,                 /* random number seed                   */
          flags=3,                  /* 3 alternatives, generic candidates   */
          nsets=6,                  /* number of choice sets                */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */


%mktex(3 ** 6, n=27, seed=306)
%choiceff(data=design, model=class(x1-x6 / sta), maxiter=400, seed=121,
          flags=3, nsets=6, options=relative, beta=zero)

%mktex(3 ** 6, n=36, seed=306)
%choiceff(data=design, model=class(x1-x6 / sta), maxiter=400, seed=121,
          flags=3, nsets=6, options=relative, beta=zero)

%mktex(3 ** 6, n=54, seed=306)
%choiceff(data=design, model=class(x1-x6 / sta), maxiter=400, seed=121,
          flags=3, nsets=6, options=relative, beta=zero)

%mktex(3 ** 6, n=72, seed=306)
%choiceff(data=design, model=class(x1-x6 / sta), maxiter=400, seed=121,
          flags=3, nsets=6, options=relative, beta=zero)

%mktex(3 ** 6, n=81, seed=306)
%choiceff(data=design, model=class(x1-x6 / sta), maxiter=400, seed=121,
          flags=3, nsets=6, options=relative, beta=zero)

%mktex(3 ** 6, n=108, seed=306)
%choiceff(data=design, model=class(x1-x6 / sta), maxiter=400, seed=121,
          flags=3, nsets=6, options=relative, beta=zero)
```

The results of these steps are not shown, but a summary is as follows:

n	*D*-Efficiency
18	6.000000
27	4.711253
36	4.996099
54	6.000000
72	4.996099
81	4.711253
108	6.000000

The following discussion provides some context for these results. In this problem, the optimal design can be directly constructed, displayed, and evaluated as follows:

```
%mktex(6 3 ** 6, n=18, seed=306)

%mktlab(data=design, vars=Set x1-x6, out=final)

proc print data=final;
   by set;
   id set;
   var x1-x6;
   run;

%choiceff(data=final,               /* candidate set of choice sets         */
          init=final(keep=set),     /* select these sets from candidates    */
          model=class(x1-x6 / sta), /* model with stdz orthogonal coding    */
          nalts=3,                  /* number of alternatives               */
          nsets=6,                  /* number of choice sets                */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

In other words, the 18-run candidate set is the optimal design (*D*-Efficiency = 6.0 and relative *D*-Efficiency = 100) when you add a six-level factor and use it as the choice set number. The optimal design is as 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
```

It has a cyclic structure where the second and third alternatives are constructed from the previous alternative by adding 1 (mod 3).* This is discussed in the section beginning on page 102. Both the optimal choice design and the 18-run orthogonal array are made using a combinatorial algorithm by developing a 6×6 difference scheme of order 3. The `%ChoicEff` macro does not know that, of course, but it does succeed in sorting the 18-run candidate set into the optimal choice design using its modified-Fedorov search algorithm. The macro also succeeds in constructing the optimal design from candidate sets of size 18, 54 and 108, but not from candidates of size 27, 36, 72, or 81. The right structure is either not in those candidate sets, or the candidate sets are large enough that the optimal design is not found. The former explanation is probably the correct one in this case. For most real-life applications, an optimal design cannot be constructed by combinatorial means, and you do not know which candidate set is best. Usually, the best you can do is try multiple candidate sets and see which one works best.

The rest of this section is optional and can be skipped by all but the most interested readers. The next section starts on page 946. It is interesting to explore the reasons why the optimal design can be found in a candidate set of 18 runs, but not in one of 36 or 72 runs. This is discussed next, but the full combinatorial details are not discussed; you will have to take some aspects of this discussion on faith. Orthogonal arrays have many different underlying constructions. As was mentioned previously, the design in 18 runs is made by developing a 6×6 difference scheme of order 3, which is also the optimal strategy for constructing this choice design. The designs in 27 and 81 runs are made from 9×9 and 27×27 difference scheme of order 3, respectively. This is not the optimal strategy for constructing this choice design. The arrays in 36, 72, and 108 runs can potentially be made in many different ways. You can see this by having the `%MktOrth` macro list all known orthogonal arrays in 36 runs that have at least 6 three-level factors. Note, however, that the words "all known" in the preceding section need some qualification. This list does not include duplicate or inferior designs that are generated with alternative lineages. This point is more fully explained throughout the rest of this section.

The following statements generate the list of designs:

```
%mktorth(range=n=36, options=lineage)

proc print data=mktdeslev; var lineage; where x3 ge 6; run;
```

*More precisely, since these numbers are based on one instead of zero, the operation is: (x mod 3) + 1.

The results are as follows:

Obs	Lineage
9	36 ** 1 : 36 ** 1 > 3 ** 12 12 ** 1 : 12 ** 1 > 2 ** 11
10	36 ** 1 : 36 ** 1 > 2 ** 10 3 ** 8 6 ** 1 (parent)
14	36 ** 1 : 36 ** 1 > 3 ** 12 12 ** 1 : 12 ** 1 > 2 ** 4 3 ** 1
16	36 ** 1 : 36 ** 1 > 2 ** 3 3 ** 9 6 ** 1 (parent)
18	36 ** 1 : 36 ** 1 > 3 ** 12 12 ** 1 : 12 ** 1 > 2 ** 2 6 ** 1
21	36 ** 1 : 36 ** 1 > 3 ** 7 6 ** 3 : 6 ** 1 > 2 ** 1 3 ** 1
23	36 ** 1 : 36 ** 1 > 3 ** 12 12 ** 1 : 12 ** 1 > 3 ** 1 4 ** 1
24	36 ** 1 : 36 ** 1 > 3 ** 12 12 ** 1 (parent)
25	36 ** 1 : 36 ** 1 > 3 ** 7 6 ** 3 (parent)

The design lineage is a set of instructions for making a design with smaller-level factors from a design with higher-level factors. For example, the first design is $3^{12}2^{11}$, which is made by replacing a single 36-level factor with $3^{12}12^1$ and then by replacing 12^1 with 2^{11}. There are four underlying parent designs capable of making at least 6 three-level factors in 36 runs and five more child designs. The `%MktEx` macro is capable of using any one of them. You can find out which one the `%MktEx` macro actually uses by default by specifying `options=lineage`, as follows:

```
%mktex(3 ** 6,                    /* 6 three-level factors               */
       n=36,                      /* 36 runs                             */
       seed=306,                  /* random number seed                  */
       options=lineage)           /* display OA construction instructions */
```

The preceding step displays the following lineage:

```
Design Lineage:
36 ** 1 : 36 ** 1 > 3 ** 12 12 ** 1 : 12 ** 1 > 2 ** 11
```

The macro uses the first lineage in the list. None of these involve a 6×6 difference scheme of order 3. A design in 36 runs is twice as big as a design in 18 runs, but it typically will not consist of replicates of the smaller design. Its construction is often quite different. You cannot expect a design of size cn (where c is an integer greater than 1) to contain all of the optimal information found in a design of size n. Sometimes it might, as in the case of the designs in 54 and 108 runs. Other times, the larger designs will not work as well.

The rest of this section explores one more bit of esoteric information that is related to this example. While this information is interesting to those interested in the finer points of choice design, it is not something you should ever need to do in practice. You can control the method that the `%MktEx` macro uses to make the design by explicitly controlling the design catalog that `%MktEx` otherwise creates automatically. You can use the `%MktOrth` macro to make the full catalog of n-run designs and then feed the lineage for just the desired design into the `%MktEx` macro.

The following steps create the design:

```
%mktorth(range=n=36, options=lineage dups)

data lev;
   set mktdeslev;
   where lineage ? '2 ** 2 18' and lineage ? '3 ** 6 6' and
         not (lineage ? ': 6');
   run;

%mktex(3 ** 6,                      /* 6 three-level factors               */
       n=36,                        /* 36 runs                             */
       seed=306,                    /* random number seed                  */
       options=lineage,             /* display OA construction instructions */
       cat=lev)                     /* OA catalog comes from lev data set  */

%choiceff(data=design,              /* candidate set of alternatives       */
          model=class(x1-x6 / sta), /* model with stdz orthogonal coding   */
          maxiter=400,              /* maximum number of designs to make   */
          seed=121,                 /* random number seed                  */
          flags=3,                  /* 3 alternatives, generic candidates  */
          nsets=6,                  /* number of choice sets               */
          options=relative,         /* display relative D-efficiency       */
          beta=zero)                /* assumed beta vector, Ho: b=0        */
```

The `dups` option in the `%MktOrth` macro is used to include duplicate and inferior designs in the catalog. These are designs that are normally removed from the catalog. An inferior design has only a subset of the factors that are available in a competing design. By default (without the `dups` option), the `%MktOrth` macro lists 26 designs with 36 runs. With the `dups` option, the number increases to 61. In this case, the design of interest is $2^2 3^6 6^1$ in 36 runs, where $3^6 6^1$ is constructed from an 18-level factor. Normally, it is removed from the catalog since it has fewer two-level and three-level factors than $2^{10} 3^8 6^1$. While it is inferior as an orthogonal array, it is a better candidate set than the default 36-run design, but less good than the 18-run design (since it is twice as big as it needs to be).

The `where` clause in the DATA step selects a design with 2 two-level factors and an 18-level factor in its lineage, 6 three-level factors and a six-level factor in its lineage, and no mention of expanding the six-level factor. This selects just the one design that we are interested in. The new catalog, with just the design of interest, is input to the `%MktEx` macro using the `cat=` (catalog input data set) option. The resulting design is then prepared as a candidate set and is input the `%ChoicEff` macro. With this candidate set, the `%ChoicEff` macro finds an optimal design.

Initial Designs and Evaluating a Design

The `%ChoicEff` macro can be used to either search for a design or to evaluate an existing design. This section discusses using the `%ChoicEff` macro to evaluate a design. In all cases, whether you are searching for a design or evaluating a design, you must provide the `%ChoicEff` macro with a candidate set. It might be the candidate set that was used to construct the design or it might simply be the design itself. When you evaluate a design, you must also provide information about how to construct the initial design from the candidate set. This can happen in one of several ways:

- If you have a design you want to evaluate in a SAS data set (say `Final`), and it has the choice set number (say `Set`), use the options `data=Final, init=Final(keep=set), intiter=0` to evaluate the design. This approach uses the choice set numbers in the `init=` data set to select the matching choice sets (the entire design) from the `data=` data set. It performs zero internal iterations. You might want to use this approach when you construct a design using a method other than the `%ChoicEff` macro.

- If you have a design you want to evaluate in a SAS data set (say `Design`), and it does not have the choice set number, but it has factors of say `x1-x6`, use the options `data=Design, init=Design, initvars=x1-x6, intiter=0` to evaluate the design. This approach uses the initial variables in the `init=` data set to select the matching observations (the entire design) in the `data=` data set. It performs zero internal iterations. You might want to use this approach when you are given a design rather than processing it further to add a choice set variable.

- If you have a design you want to evaluate in a SAS data set (say `Best`) that might have been constructed using the `%ChoicEff` macro using the alternative swapping algorithm, and it has a variable (say `Index`) that contains the alternative number from the candidate set of alternatives (say `Cand`) used to make the design, use the options `data=Cand, init=Best(keep=index), intiter=0` to evaluate the design. This approach uses the `Index` variable in the `init=` data set to select the matching alternatives in the `data=` data set. It performs zero internal iterations. You might want to use this approach when you found a design using the `%ChoicEff` macro, and now you want to evaluate it with other options or codings.

- If you have a design you want to evaluate in a SAS data set (say `Best`) that might have been constructed using the `%ChoicEff` macro using the choice set swapping algorithm, and it has a variable (say `Set`) that contains the alternative number from the candidate set of choice sets (say `Cand`) used to make the design, use the options `data=Cand, init=Best(keep=set), intiter=0` to evaluate the design. This approach uses the `Set` variable in the `init=` data set to select the matching choice sets in the `data=` data set. It performs zero internal iterations. You might want to use this approach when you found a design using the `%ChoicEff` macro, and now you want to evaluate it with other options or codings.

With the `intiter=0` option specified the design is evaluated. If you leave this option out, the design is used as an initial design, and the `%ChoicEff` macro will try to iterate to improve it.

The following example constructs a design with the choice set number and evaluates it:

```
%mktex(5 ** 6, n=25)

%mktlab(data=design, vars=Set x1-x5)

                                  /* evaluate design                      */
%choiceff(data=final,             /* candidate set of choice sets         */
          init=final(keep=set),   /* select these sets from candidates    */
          intiter=0,              /* evaluate without internal iterations */
          model=class(x1-x5 / sta), /* model with stdzd orthogonal coding   */
          nsets=5,                /* 5 choice sets                        */
          nalts=5,                /* 5 alternatives per set               */
          options=relative,       /* display relative D-efficiency        */
          beta=zero)              /* assumed beta vector, Ho: b=0         */
```

Some of the evaluation results are as follows:

Final Results

Design	1
Choice Sets	5
Alternatives	5
Parameters	20
Maximum Parameters	20
D-Efficiency	5.0000
Relative D-Eff	100.0000
D-Error	0.2000
1 / Choice Sets	0.2000

This design is 100% *D*-efficient.

The following example constructs a design without the choice set number and evaluates it:

```
%mktex(3 ** 6 6, n=18, options=nosort)

data design(keep=x1-x6);          /* There are easier ways to make this   */
   set design(obs=6);             /* design.  This example is just for    */
   array x[6];                    /* illustration.                        */
   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;
```

```
%choiceff(data=design,              /* candidate set of choice sets         */
          init=design,              /* initial design                       */
          initvars=x1-x6,           /* factors in the initial design        */
          intiter=0,                /* evaluate without internal iterations */
          model=class(x1-x6 / sta), /* model with stdzd orthogonal coding   */
          nsets=6,                  /* 6 choice sets                        */
          nalts=3,                  /* 3 alternatives per set               */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

Some of the evaluation results are 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 design is 100% *D*-efficient.

The following example constructs a design using the alternative-swapping algorithm and then evaluates it using the standardize orthogonal contrast coding:

```
%mktex(3 ** 3, n=27, seed=238)

                                  /* search for a design                  */
%choiceff(data=randomized,        /* candidate set of alternatives        */
          model=class(x1-x3),     /* main effects with ref cell coding    */
          nsets=3,                /* number of choice sets                */
          flags=3,                /* 3 alternatives, generic candidates   */
          seed=382,               /* random number seed                   */
          beta=zero)              /* assumed beta vector, Ho: b=0         */

                                  /* evaluate design                      */
%choiceff(data=randomized,        /* candidate set of alternatives        */
          init=best(keep=index),  /* select these alts from candidates    */
          intiter=0,              /* evaluate without internal iterations */
          model=class(x1-x3 / sta), /* model with stdz orthogonal coding  */
          nsets=3,                /* number of choice sets                */
          flags=3,                /* 3 alternatives, generic candidates   */
          options=relative,       /* display relative D-efficiency        */
          beta=zero)              /* assumed beta vector, Ho: b=0         */
```

Some of the design creation results are as follows:

```
                Final Results

Design                      2
Choice Sets                 3
Alternatives                3
Parameters                  6
Maximum Parameters          6
D-Efficiency           0.5774
D-Error                1.7321
```

Some of the evaluation 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
```

With the standardized orthogonal contrast coding and the relative *D*-efficiency displayed, it is clear that this design is 100% *D*-efficient. This was not as clear when the design was created without these options.

The following example constructs a design using the set-swapping algorithm and then evaluates it using the standardize orthogonal contrast coding:

```
%mktex(2 ** 6, n=64)

%mktroll(design=design, key=2 3, out=cand)

                                  /* search for a design                  */
%choiceff(data=cand,              /* candidate set of choice sets         */
          model=class(x1-x3),     /* main effects with ref cell coding    */
          nsets=8,                /* number of choice sets                */
          nalts=2,                /* number of alternatives               */
          seed=151,               /* random number seed                   */
          beta=zero)              /* assumed beta vector, Ho: b=0         */

                                  /* evaluate design                      */
%choiceff(data=cand,              /* candidate set of choice sets         */
          init=best(keep=set),    /* select these sets from candidates    */
          intiter=0,              /* evaluate without internal iterations */
          model=class(x1-x3 / sta), /* model with stdzd orthogonal coding */
          nsets=8,                /* number of choice sets                */
          nalts=2,                /* number of alternatives               */
          options=relative,       /* display relative D-efficiency        */
          beta=zero)              /* assumed beta vector, Ho: b=0         */
```

Some of the design creation results are as follows:

Final Results

Design	1
Choice Sets	8
Alternatives	2
Parameters	3
Maximum Parameters	8
D-Efficiency	2.0000
D-Error	0.5000

Some of the evaluation results are as follows:

```
                    Final Results

Design                      1
Choice Sets                 8
Alternatives                2
Parameters                  3
Maximum Parameters          8
D-Efficiency           8.0000
Relative D-Eff       100.0000
D-Error                0.1250
1 / Choice Sets        0.1250
```

With the standardized orthogonal contrast coding and the relative *D*-efficiency displayed, it is clear that this design is 100% *D*-efficient. This was not as clear when the design was created without these options.

Partial-Profile Designs

The following steps create and evaluate an optimal partial profile design where 4 of 16 attributes vary in each choice set:

```
%mktex(3 ** 4 6, n=18)

proc sort data=design out=design(drop=x5); by x1 x5; run;

%mktbibd(b=20, t=16, k=4, seed=104, out=b)

%mktppro(ibd=b, print=f p)

%choiceff(data=chdes,               /* candidate set of choice sets         */
          init=chdes(keep=set),     /* select these sets from candidates    */
          intiter=0,                /* no iterations, just evaluate         */
          model=class(x1-x6 / sta), /* model with stdz orthogonal coding    */
          nsets=120,                /* number of choice sets                */
          nalts=3,                  /* number of alternatives               */
          rscale=partial=4 of 16,   /* partial profiles, 4 of 16 vary       */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

The raw *D*-efficiency and the relative *D*-efficiency (scaled to range from 0 to 100) are as follows:

Final Results

Design	1
Choice Sets	120
Alternatives	3
Parameters	12
Maximum Parameters	240
D-Efficiency	30.0000
Relative D-Eff	100.0000
D-Error	0.0333
1 / Choice Sets	0.008333

In this case, since 4 of 16 attributes vary, the maximum *D*-efficiency is not the number of choice sets (120), it is 4/16 times the number of choice sets (30).

Other Uses of the RSCALE=PARTIAL= Option

The `rscale=partial=` option can also be used with constant alternatives. The following example creates and displays a generic design with a constant alternative:

```
%mktex(6 3 ** 6, n=18, seed=104);

%mktlab(data=randomized, vars=Set x1-x6)

proc sort data=final; by set; run;

data chdes;
   set final;
   by set;
   output;
   if last.set then do;
      x1 = .; x2 = .; x3 = .;
      x4 = .; x5 = .; x6 = .;
      output;
      end;
   run;

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

The first step makes an orthogonal array. The second step converts the six-level factor to the choice set number. The third step sorts the design into choice sets. The fourth step adds a constant alternative after the last alternative in each choice set. The fifth step displays the design.

The results are as follows:

Set	x1	x2	x3	x4	x5	x6
1	3	3	2	3	3	3
	1	2	1	2	2	2
	2	1	3	1	1	1
	.	.	.	.	.	.
2	1	3	1	1	1	3
	2	2	3	3	3	2
	3	1	2	2	2	1
	.	.	.	.	.	.
3	3	2	1	1	3	1
	1	1	3	3	2	3
	2	3	2	2	1	2
	.	.	.	.	.	.
4	1	1	2	1	3	2
	2	3	1	3	2	1
	3	2	3	2	1	3
	.	.	.	.	.	.
5	1	2	2	3	1	1
	2	1	1	2	3	3
	3	3	3	1	2	2
	.	.	.	.	.	.
6	3	1	1	3	1	2
	1	3	3	2	3	1
	2	2	2	1	2	3
	.	.	.	.	.	.

The following step evaluates the design:

```
%choiceff(data=chdes,               /* candidate set of choice sets         */
          init=chdes(keep=set),     /* select these sets from candidates    */
          intiter=0,                /* no iterations, just evaluate         */
          model=class(x1-x6 / sta), /* model with stdzd orthogonal coding   */
          nsets=6,                  /* 6 choice sets                        */
          nalts=4,                  /* number of alternatives               */
          rscale=partial=3 of 4,    /* relative D-eff, 3 of 4 attrs vary    */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

The option `rscale=partial=3 of 4` is specified since 3 of 4 attributes vary in each choice set. The raw *D*-efficiency and the relative *D*-efficiency (scaled to range from 0 to 100) are as follows:

Final Results

Design	1
Choice Sets	6
Alternatives	4
Parameters	12
Maximum Parameters	18
D-Efficiency	4.5000
Relative D-Eff	100.0000
D-Error	0.2222
1 / Choice Sets	0.1667

The variances and standard errors are as follows:

n	Variable Name	Label	Variance	DF	Standard Error
1	x11	x1 1	0.22222	1	0.47140
2	x12	x1 2	0.22222	1	0.47140
3	x21	x2 1	0.22222	1	0.47140
4	x22	x2 2	0.22222	1	0.47140
5	x31	x3 1	0.22222	1	0.47140
6	x32	x3 2	0.22222	1	0.47140
7	x41	x4 1	0.22222	1	0.47140
8	x42	x4 2	0.22222	1	0.47140
9	x51	x5 1	0.22222	1	0.47140
10	x52	x5 2	0.22222	1	0.47140
11	x61	x6 1	0.22222	1	0.47140
12	x62	x6 2	0.22222	1	0.47140
				==	
				12	

The following step displays the variance matrix:

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

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

The results are as follows:

Variance-Covariance Matrix

	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	0.22	0	0	0	0	0	0	0	0	0	0	0
x1 2	0	0.22	0	0	0	0	0	0	0	0	0	0
x2 1	0	0	0.22	0	0	0	0	0	0	0	0	0
x2 2	0	0	0	0.22	0	0	0	0	0	0	0	0
x3 1	0	0	0	0	0.22	0	0	0	0	0	0	0
x3 2	0	0	0	0	0	0.22	0	0	0	0	0	0
x4 1	0	0	0	0	0	0	0.22	0	0	0	0	0
x4 2	0	0	0	0	0	0	0	0.22	0	0	0	0
x5 1	0	0	0	0	0	0	0	0	0.22	0	0	0
x5 2	0	0	0	0	0	0	0	0	0	0.22	0	0
x6 1	0	0	0	0	0	0	0	0	0	0	0.22	0
x6 2	0	0	0	0	0	0	0	0	0	0	0	0.22

This is an optimal design with 100% relative *D*-efficiency and a diagonal variance matrix.

The following steps search for a design for this problem with a computerized search rather than a direct construction from an orthogonal array:

```
%mktex(3 ** 6, n=729, seed=104);

data cand;
   retain f1-f4 0;
   if _n_ = 1 then do;
      f4 = 1; output; f1 = 1; f2 = 1; f3 = 1; f4 = 0;
      end;
   set randomized;
   output;
   run;

proc print data=cand; run;

%choiceff(data=cand,                /* candidate set of alternatives        */
          model=class(x1-x6 / sta), /* model with stdzd orthogonal coding   */
          flags=f1-f4,              /* flag which alts go where             */
          nsets=6,                  /* 6 choice sets                        */
          maxiter=30,               /* maximum designs to make              */
          rscale=partial=3 of 4,    /* relative D-eff, 3 of 4 attrs vary    */
          seed=104,                 /* random number seed                   */
          beta=zero)                /* assumed beta vector, Ho: b=0         */

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

Part of the candidate set of alternatives is as follows:

Obs	f1	f2	f3	f4	x1	x2	x3	x4	x5	x6
1	0	0	0	1	.	.	.	.	.	.
2	1	1	1	0	3	2	3	2	1	1
3	1	1	1	0	3	3	1	3	3	3
4	1	1	1	0	1	2	3	1	1	1
5	1	1	1	0	2	1	1	2	1	2
6	1	1	1	0	1	2	1	1	3	2
7	1	1	1	0	2	3	1	3	3	3
8	1	1	1	0	2	3	3	2	3	1
9	1	1	1	0	3	1	3	1	2	1

```
  .
  .
  .
728    1    1    1    0    2    3    3    3    3    2
729    1    1    1    0    3    3    3    1    1    1
730    1    1    1    0    2    2    1    3    1    1
```

The first candidate provides the constant alternative for each choice set, and the remaining candidates provide the first through third alternatives.

The raw *D*-efficiency, the relative *D*-efficiency (scaled to range from 0 to 100), and the variances and standard errors are as follows:

Final Results

Design	21
Choice Sets	6
Alternatives	4
Parameters	12
Maximum Parameters	18
D-Efficiency	4.1425
Relative D-Eff	92.0562
D-Error	0.2414
1 / Choice Sets	0.1667

n	Variable Name	Label	Variance	DF	Standard Error
1	x11	x1 1	0.24444	1	0.49441
2	x12	x1 2	0.28889	1	0.53748
3	x21	x2 1	0.24444	1	0.49441
4	x22	x2 2	0.28889	1	0.53748
5	x31	x3 1	0.23611	1	0.48591
6	x32	x3 2	0.26389	1	0.51370
7	x41	x4 1	0.31111	1	0.55777
8	x42	x4 2	0.22222	1	0.47140
9	x51	x5 1	0.27778	1	0.52705
10	x52	x5 2	0.22222	1	0.47140
11	x61	x6 1	0.24444	1	0.49441
12	x62	x6 2	0.28889	1	0.53748
				==	
				12	

The variance matrix is as follows:

Variance-Covariance Matrix

	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	0.24	-0.04	0.02	-0.04	0	0	-0.07	0	0	0	-0.03	-0.06
x1 2	-0.04	0.29	-0.04	0.07	0	0	0.12	0	0	0	0.06	0.10
x2 1	0.02	-0.04	0.24	-0.04	0	0	-0.07	0	0	0	-0.03	-0.06
x2 2	-0.04	0.07	-0.04	0.29	0	0	0.12	0	0	0	0.06	0.10
x3 1	0	0	0	0	0.24	0.02	0	0	0.03	0	0	0
x3 2	0	0	0	0	0.02	0.26	0	0	0.05	0	0	0
x4 1	-0.07	0.12	-0.07	0.12	0	0	0.31	0	0	0	0.04	0.08
x4 2	0	0	0	0	0	0	0	0.22	0	0	0	0
x5 1	0	0	0	0	0.03	0.05	0	0	0.28	0	0	0
x5 2	0	0	0	0	0	0	0	0	0	0.22	0	0
x6 1	-0.03	0.06	-0.03	0.06	0	0	0.04	0	0	0	0.24	0.04
x6 2	-0.06	0.10	-0.06	0.10	0	0	0.08	0	0	0	0.04	0.29

This problem is large enough that it is hard to find the optimal design with a computerized search. Hence, the relative *D*-efficiency is 92.0562 (out of 100), most variances are larger than 0.22, and some covariances are larger than zero.

Optimal Alternative-Specific Design

The following steps create and evaluate an optimal design for a choice model with alternative-specific effects, three brands (and hence three alternatives), 27 choice sets, and 4 three-level attributes in addition to brand:

```
%mktex(3 ** 12, n=27, seed=104)

%mktkey(3 4)

data key; Brand = scan('A B C', _n_); set key; run;

%mktroll(design=randomized, key=key, out=rolled, alt=brand)
```

```
%choiceff(data=rolled,              /* candidate set of choice sets         */
          init=rolled(keep=set),    /* select these sets from candidates    */
          intiter=0,                /* no iterations, just evaluate         */
          model=class(brand / sta)  /* brand effects                        */
                class(brand * x1    /* alternative-specific effects x1      */
                      brand * x2    /* alternative-specific effects x2      */
                      brand * x3    /* alternative-specific effects x3      */
                      brand * x4 /  /* alternative-specific effects x4      */
                      sta zero=' '),/* std ortho coding, use all brands     */
          nalts=3,                  /* number of alternatives               */
          nsets=27,                 /* number of choice sets                */
          rscale=alt,               /* alt-specific design efficiency scale */
          beta=zero)                /* assumed beta vector, Ho: b=0         */

proc print data=bestcov label;
   title 'Variance-Covariance Matrix';
   id __label;
   label __label = '00'x;
   var B:;
   format _numeric_ zer5.2;
   run;
title;
```

The raw *D*-efficiency and the relative *D*-efficiency (scaled to range from 0 to 100) are as follows:

Final Results

Design	1
Choice Sets	27
Alternatives	3
Parameters	26
Maximum Parameters	54
D-Efficiency	6.7359
Relative D-Eff	100.0000
D-Error	0.1485
1 / Choice Sets	0.0370

The variances and standard errors are as follows:

n	Variable Name	Label	Variance	DF	Standard Error
1	BrandA	Brand A	0.03704	1	0.19245
2	BrandB	Brand B	0.03704	1	0.19245
3	BrandAx11	Brand A * x1 1	0.16667	1	0.40825
4	BrandAx12	Brand A * x1 2	0.16667	1	0.40825
5	BrandBx11	Brand B * x1 1	0.16667	1	0.40825
6	BrandBx12	Brand B * x1 2	0.16667	1	0.40825
7	BrandCx11	Brand C * x1 1	0.16667	1	0.40825
8	BrandCx12	Brand C * x1 2	0.16667	1	0.40825
9	BrandAx21	Brand A * x2 1	0.16667	1	0.40825
10	BrandAx22	Brand A * x2 2	0.16667	1	0.40825
11	BrandBx21	Brand B * x2 1	0.16667	1	0.40825
12	BrandBx22	Brand B * x2 2	0.16667	1	0.40825
13	BrandCx21	Brand C * x2 1	0.16667	1	0.40825
14	BrandCx22	Brand C * x2 2	0.16667	1	0.40825
15	BrandAx31	Brand A * x3 1	0.16667	1	0.40825
16	BrandAx32	Brand A * x3 2	0.16667	1	0.40825
17	BrandBx31	Brand B * x3 1	0.16667	1	0.40825
18	BrandBx32	Brand B * x3 2	0.16667	1	0.40825
19	BrandCx31	Brand C * x3 1	0.16667	1	0.40825
20	BrandCx32	Brand C * x3 2	0.16667	1	0.40825
21	BrandAx41	Brand A * x4 1	0.16667	1	0.40825
22	BrandAx42	Brand A * x4 2	0.16667	1	0.40825
23	BrandBx41	Brand B * x4 1	0.16667	1	0.40825
24	BrandBx42	Brand B * x4 2	0.16667	1	0.40825
25	BrandCx41	Brand C * x4 1	0.16667	1	0.40825
26	BrandCx42	Brand C * x4 2	0.16667	1	0.40825
				==	
				26	

The variance matrix (not shown) is diagonal. The brand effects have variances equal to one over the number of choice sets (just like in the optimal generic designs). The alternative-specific effects (with 3 alternatives and 27 choice sets) all have variances equal to $3^2/(3-1)/27 = 1/6$. The ratio $1/6$ is the inverse of $2/9$ of the number of choice sets. With an alternative-specific design with three alternatives, $1/3$ of the rows in the coded design have information. Furthermore, $(3-1)/3 = 2/3$ of the rows in any choice design contribute information to the variance matrix. The resulting product is $(1/3) \times (2/3) = 2/9$.

The determinant of the variance matrix can be decomposed into the product of the determinant of the variance submatrix for the brand effects and the determinant of the variance submatrix for the alternative-specific effects (since the off diagonal elements are zero). This determinant (with 2 and 24 parameters in each submatrix) is $27^2 \times (27 \times 2/9)^{24}$. The D-efficiency is the $26th$ root of this product since there are 26 parameters.

The following step computes and displays the maximum *D*-efficiency:

```
data _null_;
   sets  = 27;
   alts  = 3;
   m     = alts - 1;
   parms = m + alts * 4 * (3 - 1);
   det1  = sets ** m;
   det2  = (sets * (m / (alts ** 2))) ** (parms - m);
   scale = (det1 * det2) ** (1 / parms);
   put scale=;
   run;
```

The result is "`scale=6.7359424316`", which matches the raw *D*-efficiency in the final results table and is used as the scaling factor to get the relative *D*-efficiency.

The following steps illustrate this technique with an alternative-specific design created by an orthogonal array with 6-, 4-, 3-, and 2-level factors:

```
%mktex(6 3 2 2 4 4  6 3 2 2 4 4
       6 3 2 2 4 4  6 3 2 2 4 4  6 3 2 2 4 4, n=288)

%mktkey(5 6)

data key; Brand = scan('A B C D E F', _n_); set key; run;

%mktroll(design=randomized, key=key, out=rolled, alt=brand)

%choiceff(data=rolled,              /* candidate set of choice sets         */
          init=rolled(keep=set),    /* select these sets from candidates    */
          intiter=0,                /* no iterations, just evaluate         */
          model=class(brand / sta)  /* brand effects                        */
                class(brand * x1    /* alternative-specific effects x1      */
                      brand * x2    /* alternative-specific effects x2      */
                      brand * x3    /* alternative-specific effects x3      */
                      brand * x4    /* alternative-specific effects x4      */
                      brand * x5    /* alternative-specific effects x5      */
                      brand * x6 /  /* alternative-specific effects x6      */
                      sta zero=' '),/* std ortho coding, use all brands     */
          nalts=5,                  /* number of alternatives               */
          nsets=288,                /* number of choice sets                */
          rscale=alt,               /* alt-specific design efficiency scale */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

```
proc print data=bestcov label;
   title 'Variance-Covariance Matrix';
   id __label;
   label __label = '00'x;
   var B:;
   format _numeric_ zer5.2;
   run;
title;

proc iml;
   use bestcov(drop=__:); read all into x;
   x = shape(x, 1)`;
   create _cov from x[colname='Covariance']; append from x;
   quit;

proc freq; tables Covariance; format covariance zer5.; run;

data _null_;
   sets  = 288;
   alts  = 5;
   m     = alts - 1;
   parms = m + alts * (6 + 3 + 2 + 2 + 4 + 4 - 6);
   det1  = sets ** m;
   det2  = (sets * (m / (alts ** 2))) ** (parms - m);
   scale = (det1 * det2) ** (1 / parms);
   put scale=;
   run;
```

The raw *D*-efficiency and the relative *D*-efficiency (scaled to range from 0 to 100) are as follows:

Final Results

Design	1
Choice Sets	288
Alternatives	5
Parameters	79
Maximum Parameters	1152
D-Efficiency	50.5604
Relative D-Eff	100.0000
D-Error	0.0198
1 / Choice Sets	0.003472

The variances and covariances are not shown, but they are summarized in the following listing from PROC FREQ:

```
                        The FREQ Procedure

                                                  Cumulative    Cumulative
      Covariance    Frequency     Percent     Frequency      Percent
      ------------------------------------------------------------------
           0           6162       98.73          6162        98.73
       0.003              4        0.06          6166        98.80
       0.022             75        1.20          6241       100.00
```

The four variances for the brand effects are 0.003, and the 75 variances for the alternative-specific effects are 0.022. All covariances are zero. The results of the DATA _NULL_ step produce "`scale=50.560364105`", which matches the unscaled efficiency. Using this as a scale factor, the relative *D*-efficiency is 100%. This design is optimal, but is quite large with 288 choice sets. The following step creates a smaller design with a computerized search:

```
%mktex(6 3 2 2 4 4, n=6*3*2*2*4*4)

data des(drop=i);
   retain f1-f5 0;
   array f[5];
   set design;
   do i = 1 to 5;
      Brand = scan('A B C D E', i);
      f[i] = 1; output; f[i] = 0;
      end;
   run;

%choiceff(data=des,                 /* candidate set of alternatives        */
          model=class(brand / sta)  /* brand effects                        */
                class(brand * x1    /* alternative-specific effects x1      */
                      brand * x2    /* alternative-specific effects x2      */
                      brand * x3    /* alternative-specific effects x3      */
                      brand * x4    /* alternative-specific effects x4      */
                      brand * x5    /* alternative-specific effects x5      */
                      brand * x6 /  /* alternative-specific effects x6      */
                      sta zero=' '),/* std ortho coding, use all brands     */
          flags=f1-f5,              /* 5 alternatives, generic candidates   */
          nsets=32,                 /* number of choice sets                */
          maxiter=2,                /* maximum number of designs to make    */
          rscale=alt,               /* alt-specific design efficiency scale */
          seed=104,                 /* random number seed                   */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

The raw *D*-efficiency and the relative *D*-efficiency (scaled to range from 0 to 100) are as follows:

Final Results

Design	7
Choice Sets	32
Alternatives	5
Parameters	79
Maximum Parameters	128
D-Efficiency	5.0863
Relative D-Eff	90.5381
D-Error	0.1966
1 / Choice Sets	0.0313

The relative *D*-efficiency is based on comparing this design to a hypothetical alternative-specific choice design in 32 choice sets with a diagonal variance matrix like the one generated with 288 choice sets.

This next step creates a design with brand effects, alternative-specific price effects, and cross effects:

```
%let sets = %eval(3 ** 5);

%mktex(3 ** 5, n=&sets)

%mktlab(values=1.49 1.99 2.49)

data key;
  input (b p) ($);
  datalines;
1 x1
2 x2
3 x3
4 x4
5 x5
. .
;

%mktroll(design=final, key=key, out=crosscan, alt=b, keep=x1-x5)

data crosscan;
   set crosscan;
   label b  = 'Brand' p = 'Price' x1 = 'Brand 1 Price'
         x2 = 'Brand 2 Price' x3 = 'Brand 3 Price'
         x4 = 'Brand 4 Price' x5 = 'Brand 5 Price';
   run;
```

```
%choiceff(data=crosscan,            /* candidate set of choice sets         */
          init=crosscan(keep=set),  /* select these sets from candidates    */
          intiter=0,                /* no iterations, just evaluate         */
          model=class(b             /* brand effects                        */
                b*p / zero=' ')     /* alternative-specific effects         */
                class(b / zero=none)/* cross effects                        */
                * identity(x1-x5),
          drop=B1X1 B2X2 B3X3       /* drop cross effects of brand on self  */
               B4X4 B5X5,
          nsets=&sets,              /* number of choice sets                */
          nalts=6,                  /* number of alternatives               */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

This design is constructed from a full-factorial design of the price attributes. The final results table is as follows:

Final Results

Design	1
Choice Sets	243
Alternatives	6
Parameters	35
Maximum Parameters	1215
D-Efficiency	6.5104
D-Error	0.1536

This next step is similar to the previous step, but instead of evaluating the design constructed from the full-factorial design, we use it as a candidate set:

```
%choiceff(data=crosscan,            /* candidate set of choice sets         */
          model=class(b             /* brand effects                        */
                b*p / zero=' ')     /* alternative-specific effects         */
                class(b / zero=none)/* cross effects                        */
                * identity(x1-x5),
          maxiter=10,               /* maximum number of designs to make    */
          drop=B1X1 B2X2 B3X3       /* drop cross effects of brand on self  */
               B4X4 B5X5,
          nsets=&sets,              /* number of choice sets                */
          nalts=6,                  /* number of alternatives               */
          seed=104,                 /* random number seed                   */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

The final results table is as follows:

Final Results

Design	8
Choice Sets	243
Alternatives	6
Parameters	35
Maximum Parameters	1215
D-Efficiency	7.1536
D-Error	0.1398

Using the choice design constructed directly from the full-factorial design, we get a *D*-efficiency of 6.5. With the search, we get 7.15. We really have no way of knowing the maximum *D*-efficiency for this problem. We cannot even use the standardized orthogonal contrast coding. For this type of design, with all of the interactions involved in the coding, there is no reason to believe that a choice design constructed from an orthogonal array is going to good. At 243 choice sets, this design is too large to use without breaking it up into many blocks. However, we can use the 7.15 efficiency value to scale efficiency for smaller designs to a scale from 0 to approximately 100. The following step illustrates:

```
%let sets = 32;

%choiceff(data=crosscan,           /* candidate set of choice sets         */
          model=class(b            /* brand effects                        */
                b*p / zero=' ')    /* alternative-specific effects         */
                class(b / zero=none)/* cross effects                       */
                * identity(x1-x5),
          maxiter=10,              /* maximum number of designs to make    */
          drop=B1X1 B2X2 B3X3      /* drop cross effects of brand on self  */
               B4X4 B5X5,
          rscale=&sets * 7.1536/243,/* scaling factor for relative eff     */
          nsets=&sets,             /* number of choice sets                */
          nalts=6,                 /* number of alternatives               */
          seed=104,                /* random number seed                   */
          beta=zero)               /* assumed beta vector, Ho: b=0         */
```

The raw *D*-efficiency and the relative *D*-efficiency (scaled to range from 0 to approximately 100) are as follows:

```
                Final Results

Design                     3
Choice Sets               32
Alternatives               6
Parameters                35
Maximum Parameters       160
D-Efficiency          0.9390
Relative D-Eff       99.6825
D-Error               1.0649
1 / Choice Sets       0.0313
```

Relative to an unknown optimal design in 32 choice sets, this design is *approximately* 99.68% *D*-efficient. The *D*-efficiency is scaled relative to the number of choice sets times the proportion consisting of the approximate maximum *D*-efficiency divided by the number of choice sets in the comparison design. With 243 choice sets, *D*-efficiency is 7.1536. So 7.1536/243 provides the per set efficiency in the larger design. The number of sets is multiplied by this fraction to get the maximum expected *D*-efficiency for the smaller design. Since we do not know the maximum *D*-efficiency, you could be conservative and specify: `rscale=&sets * 7.2 / 243`, `rscale=&sets * 7.5 / 243`, or some other value.

Whenever you do not know the maximum *D*-efficiency, you can use this approach. Create a large design directly from a large candidate set or by searching a large candidate set. Then use its *D*-efficiency (or a slightly larger value to be conservative) and number of choice sets to scale the *D*-efficiency of a smaller and more realistic design.

%ChoicEff Macro Options

The following options can be used with the `%ChoicEff` macro:

	Option	Description
	`help`	(positional) "help" or "?" displays syntax summary
	`bestcov=`*SAS-data-set*	covariance matrix for the best design
	`bestout=`*SAS-data-set*	best design
	`beta=`*list*	true parameters
	`chunks=`*n*	number of observations to code at once
	`converge=`*n*	convergence criterion
	`cov=`*SAS-data-set*	all of the covariance matrices
	`data=`*SAS-data-set*	input choice candidate set
	`drop=`*variable-list*	variables to drop from the model
	`fixed=`*variable-list*	variable that flags fixed alternatives
*	`flags=`*variable-list*|*n*	variables that flag the alternatives
	`init=`*SAS-data-set*	input initial design data set
	`initvars=`*variable-list*	initial variables

* - a new option or an option with new features in this release.

	Option	Description
	`intiter=`*n*	maximum number of internal iterations
	`iter=`*n*	maximum iterations (designs to create)
	`maxiter=`*n*	maximum iterations (designs to create)
	`model=`*model-specification*	`model` statement list of effects
	`morevars=`*variable-list*	more variables to add to the model
	`n=`*n*	number of observations
	`nalts=`*n*	number of alternatives
	`nsets=`*n*	number of choice sets desired
	`options=coded`	displays the coded candidate set
	`options=detail`	displays the details of the swaps
∗	`options=nobeststar`	no asterisk when a better design is found
	`options=nocode`	skips the PROC TRANSREG coding stage
	`options=nodups`	prevents duplicate choice set creation
	`options=notests`	suppress the hypothesis tests
	`options=orthcan`	orthogonalizes the candidate set
∗	`options=outputall`	outputs all designs to OUT= and COV=
	`options=relative`	displays final relative *D*-efficiency
∗	`options=resrep`	same as `options=detail`
	`out=`*SAS-data-set*	all designs data set
∗	`restrictions=`*macro-name*	restrictions macro
∗	`resvars=`*variable-list*	variables for restrictions
∗	`rscale=`*r*	relative efficiency scaling factor
∗	`rscale=generic`	equivalent to `rscale=n` (*n* sets)
∗	`rscale=alt`	for simple alternative-specific designs
∗	`rscale=partial=`*p* `of` *q*	*p* of *q* alternatives or attributes vary
	`seed=`*n*	random number seed
	`submat=`*number-list*	submatrix for efficiency calculations
	`types=`*integer-list*	number of sets of each type
	`typevar=`*variable*	choice set types variable
	`weight=`*weight-variable*	optional weight variable

∗ - a new option or an option with new features in this release.

Help Option

You can specify either of the following to display the option names and simple examples of the macro syntax:

```
%choiceff(help)
%choiceff(?)
```

Required Options

You must specify both the `model=` and `nsets=` options and either the `flags=` or `nalts=` options. You can omit `beta=` if you just want a listing of effects, however you must specify `beta=` to create a design. The rest of the options are optional.

model= *model-specification*
specifies a PROC TRANSREG `model` statement, which lists the attributes and describes how they are coded. There are many potential forms for the model specification and a number of options. See the SAS/STAT PROC TRANSREG documentation. PROC TRANSREG has a new option with version 9.2 of SAS that is often useful in this macro, namely the standardized orthogonal contrast coding requested by the `sta` or `standorth` option. For some designs, with this option and a specification of `options=relative`, you can get a relative *D*-efficiency in the 0 to 100 range. If you are running an earlier version of SAS and cannot use this option, your functionality is in no way limited, but you will not have a 0 to 100 scale for relative *D*-efficiency.

The following option specifies generic effects:

```
model=class(x1-x3),
```

The following option specifies brand and alternative-specific effects:

```
model=class(b)
      class(b*x1 b*x2 b*x3 / effects zero=' '),
```

The following option specifies brand, alternative-specific, and cross effects:

```
model=class(b b*p / zero=' ')
      identity(x1-x5) * class(b / zero=none),
```

See pages 808 through 946 for other examples of `model` syntax. Furthermore, all of the PROC TRANSREG and `%ChoicEff` macro examples from pages 327 through 610 show examples of model syntax for choice models.

nsets= *n*
specifies the number of choice sets desired.

Other Required Options

You must specify exactly one of these next two options. When the candidate set consists of individual alternatives to be swapped, specify the alternative flags with `flags=`. When the candidate set consists of entire sets of alternatives to be swapped, specify the number of alternatives in each set with `nalts=`.

flags= *variable-list* | *number-of-alternatives*
specifies variables that flag the alternatives for which each candidate can be used. There must be one flag variable per alternative. If every candidate can be used in all alternatives, then the flags are constant. When the flags are all constant (in a purely generic design), you can have the macro create these flag variables for you by specifying the number of alternatives rather than a list of flag variables. Example: `flags=3`. Alternatively, you can make the flag variables yourself. For example, with three alternatives, create these constant flags: `f1=1 f2=1 f3=1`.

Otherwise, with designs with brands or alternative labels, with three alternatives, specify `flags=f1-f3` and create a candidate set where: alternative 1 candidates are indicated by `f1=1 f2=0 f3=0`, alternative 2 candidates are indicated by `f1=0 f2=1 f3=0`, and alternative 3 candidates are indicated by `f1=0 f2=0 f3=1`.

nalts= *n*
specifies the number of alternatives in each choice set for the set-swapping algorithm.

Other Options

The rest of the parameters are optional. You can specify zero or more of them.

bestcov= *SAS-data-set*
specifies a name for the data set containing the covariance matrix for the best design. By default, this data set is called BESTCOV.

bestout= *SAS-data-set*
specifies a name for the data set containing the best design. By default, this data set is called BEST. Often, you will want to specify a two-level name to create a permanent SAS data set so the design is available later for analysis.

beta= *list*
specifies the true parameters. By default, when `beta=` is not specified, the macro just reports on coding. You can specify `beta=zero` to assume all zeros. Otherwise specify a number list: `beta=1 -1 2 -2 1 -1`.

chunks= *n*
specifies the number of observations to process at one time with the coding step and PROC TRANSREG. By default, the entire data set is processed at once. You can specify a value, say 1/2 or 1/3 of the number of choice sets times the number of alternatives to break up the coding into smaller chunks if you run out of memory. Ideally, make the value a multiple of the number of choice sets. Be sure that you do not leave one or a few extra observations in the last chunk, particularly if you are using one of the orthogonal codings (for example, `sta`) or you will get an error. Usually, you will not need to specify this option.

converge= *n*
specifies the *D*-efficiency convergence criterion. By default, `converge=0.005`.

cov= *SAS-data-set*
specifies a name for the data set containing all of the covariance matrices for all of the designs. By default, this data set is called COV.

data= *SAS-data-set*
specifies the input choice candidate set. By default, the macro uses the last data set created.

drop= *variable-list*
specifies a list of variables to drop from the model. If you specified a less-than-full-rank model in the `model=` specification, you can use `drop=` to produce a full rank coding. When there are redundant variables, the macro displays a list that you can use in the `drop=` option in a subsequent run.

fixed= *variable-list*
specifies the variable that flags the fixed alternatives. When `fixed=variable` is specified, the `init=` data set must contain the named variable, which indicates which alternatives are fixed (cannot be swapped out) and which ones can be changed. Example: `fixed=fixed, init=init, initvars=x1-x3`. Values of the `fixed=` variable include:

1 – means this alternative can never be swapped out.

0 – means this alternative is used in the initial design, but it can be swapped out.

. – means this alternative should be randomly initialized, and it can be swapped out.

The `fixed=` option can be specified only when both `init=` and `initvars=` are specified.

init= *SAS-data-set*
specifies an input initial design data set. Null means a random start. One usage is to specify the `bestout=` data set for an initial start. When `flags=` is specified, `init=` must contain the index variable. Example: `init=best(keep=index)`. When `nalts=` is specified, `init=` must contain the choice set variable. Example: `init=best(keep=set)`.

Alternatively, the `init=` data set can contain an arbitrary design, potentially created outside this macro. In that case, you must also specify `initvars=factors`, where factors are the factors in the design, for example, `initvars=x1-x3`. When alternatives are swapped, this data set must also contain the `flags=` variables. When `init=` is specified with `initvars=`, the data set can also contain a variable specified in the `fixed=` option, which indicates which alternatives are fixed, and which ones can be swapped in and out.

intiter= *n*
specifies the maximum number of internal iterations. Specify `intiter=0` to just evaluate efficiency of an existing design. By default, `intiter=10`.

initvars= *variable-list*
specifies the factor variables in the `init=` data set that must match up with the variables in the `data=` data set. See `init=`. All of these variables must be of the same type.

maxiter= *n*
iter= *n*
specifies the maximum iterations (designs to create). By default, `maxiter=10`.

morevars= *variable-list*
specifies more variables to add to the model. This option gives you the ability to specify a list of variables to copy along as is, through the TRANSREG coding, then add them to the model.

n= *n*
specifies the number of observations to use in the variance matrix formula. By default, `n=1`.

options= *options-list*
specifies binary options. By default, none of these options are specified. Specify one or more of the following values after `options=`.

`coded`
displays the coded candidate set.

`detail`
displays the details of the swaps and any restriction violations. This option adds more information to the iteration history tables than is displayed by default. You can use `options=resrep` as an alias for `options=detail`. The former is the name of the option in the `%MktEx` macro that provides a report on restriction violations and conformance. It is a good idea to specify this option with restrictions until you are sure that your restrictions macro is correct.

`nobeststar`
do not print an asterisk when a better design is found. By default, an asterisk is printed in the iteration history table whenever a design is found with a *D*-efficiency that is greater than the previous best.

`nocode`
skips the PROC TRANSREG coding stage, assuming that WORK.TMP_CAND was created by a previous step. This is most useful with set swapping when the candidate set can be big. It is important with `options=nocode` to note that the effect of `morevars=` and `drop=` in previous runs has already been taken care of, so do not specify them (unless for instance you want to drop still more variables).

`nodups`
prevents the same choice set from coming out more than once. This option does not affect the initialization, so the random initial design might have duplicates. This option forces duplicates out during the iterations, so do not set `intiter=` to a small value. It might take several iterations to eliminate all duplicates. It is possible that efficiency will decrease as duplicates are forced out. With set swapping, this macro checks the candidate choice set numbers to avoid duplicates. With alternative swapping, this macro checks the candidate alternative index to avoid duplicates. The macro does not look at the actual factors. This makes the checks faster, but if the candidate set contains duplicate choice sets or alternatives, the macro might not succeed in eliminating all duplicates. Run the `%MktDups` macro (which looks at the actual factors) on the design to check and make sure all duplicates are eliminated. If you are using set swapping to make a generic design make sure you run the `%MktDups` macro on the candidate set to eliminate duplicate choice sets in advance.

`notests`
suppresses displaying the diagonal of the covariance matrix, and hypothesis tests for this n and $\boldsymbol{\beta}$. When $\boldsymbol{\beta}$ is not zero, the results include a Wald test statistic (β divided by the standard error), which is normally distributed, and the probability of a larger squared Wald statistic.

`orthcan`
orthogonalizes the candidate set.

`outputall`
outputs all designs to the `out=` and `cov=` data sets. When the `maxiter=` value is less than or equal to 100, this option is the default. However, when the `maxiter=` value is greater than 100, only designs that improve on the previous best design are output by default. This is a change from previous releases.

`relative`
displays the relative *D*-efficiency for the final design, which is 100 times the *D*-efficiency divided by the number of choice sets. In other words, this option scales the *D*-efficiency relative to a (perhaps hypothetical) design with *D*-efficiency equal to the number of choice sets and displays it. When `beta=zero` is specified along with the standardized orthogonal contrast coding in the model specification and a generic choice design is requested, this scales *D*-efficiency to a 0 to 100 scale. Certain optimal generic choice designs constructed through combinatorial methods will have a relative *D*-efficiency of 100. While you can display this value for any other type of design and specification, it will not generally be on a 0 to 100 scale except in certain special cases, and this is why it is not displayed by default. You can specify the `rscale=` option if you have used the standardized orthogonal contrast coding and would like D-efficiency scaled relative to a value other than the number of choice sets. The following steps show an example of where it would make sense to specify `options=relative`:

`resrep`
is the same as `options=detail`.

```
%mktex(4 ** 5, n=16)

%mktlab(data=design, vars=Set x1-x4)

%choiceff(data=final,               /* candidate set of choice sets  */
          init=final(keep=set),     /* select these sets from cands  */
          intiter=0,                /* eval without internal iters   */
          model=class(x1-x4 / sta), /* model with stdz orthog coding */
          options=relative,         /* display relative D-efficiency */
          nsets=4,                  /* number of choice sets         */
          nalts=4,                  /* number of alternatives        */
          beta=zero)                /* assumed beta vector, Ho: b=0  */
```

The standardized orthogonal contrast coding is requested with the `sta` option in the `class` specification. If you are not running version 9.2 or a later SAS release, remove the slash and the `sta` option from the *model* specification. The final results table contains the relative *D*-efficiency in addition to all of the other usual results. In this case, since an optimal generic design is being evaluated, relative *D*-efficiency is 100.

out= *SAS-data-set*

specifies a name for the output SAS data set with all of the final designs. The default is `out=results`.

restrictions= *macro-name*

specifies the name of a restrictions macro, written in IML, that quantifies the badness of the design in an IML scalar that must be called `bad`. By default, there are no restrictions. When a restrictions macro is specified, then the `resvars=` option must be specified as well.

revars= *variable-list*

specifies the variables for restrictions. These variables must all be numeric. The resvars variables are available for your restrictions macro in the following matrices:

`xmat` - the current design
`x` - the choice set that is being considered for the `setnum` position (the current choice set number) in `xmat`

When restrictions are posed in terms of the entire design, the code in the restrictions macro might have the following form (where `^=` means not equals):

```
do s = 1 to nsets;
   if s ^= setnum then z = xmat[((s - 1) * nalts + 1) : (s * nalts),];
   else z = x;
   ... evaluate choice set z and accumulate badness ...
   end;
```

The index vector `((s - 1) * nalts + 1) : (s * nalts)` (used as the row index in `xmat`) extracts one choice set. For example, with 3 alternatives, the index vector is: `1:3` when `s = 1` and extracts the first choice set from `xmat`, `4:6` when `s = 2` and extracts the second choice set from `xmat`, `7:9` when `s = 3`, and so on. The submatrix `xmat[((setnum - 1) * nalts + 1) : (setnum * nalts),]` contains the `setnum` choice set that is currently in the design, and `x` contains the candidate choice set that is being evaluated. The submatrix `xmat[((setnum - 1) * nalts + 1) : (setnum * nalts),]` is replaced by `x` when the replacement increases efficiency or decreases restriction violations.

The first `resvars=` variable is in the first column of `x` and `xmat` (`x[,1]` and `xmat[,1]`), ..., and the last `resvars=` variable is in the last column of `x` and `xmat` (`x[,m]` and `xmat[,m]`) where `m = ncol(xmat)`. The `resvars=` variables are typically the attributes, but they can contain additional information as well. All restrictions must be posed in terms of the values of `x` and `xmat` along with the following:

`nsets` - number of choice sets in the design
`nalts` - number of alternatives in the design
`setnum` - number of current choice set
`altnum` - number of current alternative (only available with alternative swapping)

rscale= r | `generic` | `alt` | `partial=`p `of` q
specifies the scaling factor to use for relative *D*-efficiency computations. When you specify `rscale=`, the option `options=relative` is implied. By default, when this option is not specified, the number of choice sets is used when `options=relative` is specified. If you specify `rscale=`r, where r is some number, then relative *D*-efficiency equals: 100 × *D*-efficiency / r. If you want relative *D*-efficiency, and you know that the number of choice sets is not the right scaling factor (perhaps because you have a constant alternative) and if you know the *D*-efficiency of an optimal design, you can specify it to get relative *D*-efficiency. Note that r must be a number and not an expression. However, you can use the `%sysevalf` function to evaluate an expression (for example, `rscale=%sysevalf(16 * 4 / 8)`).

The option `rscale=generic` (with n choice sets) is equivalent to `rscale=`n.

The option `rscale=alt` (with n choice sets, m alternatives, and p parameters, and $r = (n^{m-1}(n(m-1)/m^2)^{p-m+1})^{1/p}$) is equivalent to `rscale=`$r$. If a design has brand (or alternative label) effects such that brand i always occurs in alternative i, and all other effects are alternative-specific, then there are $m-1$ parameters with a maximum determinant of n, and the remaining $p-m+1$ have 1 of m alternatives contributing information to each set, and $m-1$ of m alternatives contribute information so the maximum *D*-efficiency is $n(m-1)/m^2$ for the $p-m+1$ parameters in that part of the design. This formula will not provide a true maximum for more complicated designs such as designs with constant alternatives.

The option `rscale=partial=`p `of` q, (where p and q are integers) is used with partial-profile designs (where p of q attributes vary) or with a generic choice design with a constant alternative (where p of q alternatives vary). It sets `rscale=` to np/q. For example, with 16 choice sets and 4 of 8 attributes varying, `rscale=partial=4 of 16` is equivalent to `rscale=%sysevalf(16 * 4 / 8)` and `rscale=8`.

seed= n
specifies the random number seed. By default, `seed=0`, and clock time is used as the random number seed. By specifying a random number seed, results should be reproducible within a SAS release for a particular operating system and for a particular version of the macro. However, due to machine and macro differences, some results might not be exactly reproducible everywhere, although you would expect the efficiency differences to be slight.

submat= *number-list*
specifies a submatrix for which efficiency calculations are desired. Specify an index vector. For example, with 3 three-level factors, `a`, `b`, and `c`, and the model `class(a b c a*b)`, specify `submat=1:6`, to see the efficiency of just the 6 × 6 matrix of main effects. Specify `submat=3:6`, to see the efficiency of just the 4 × 4 matrix of `b` and `c` main effects.

types= *integer-list*
specifies the number of sets of each type to put into the design. This option is used when you have multiple types of choice sets and you want the design to consist of only certain numbers of each type. This option can be specified with the set-swapping algorithm. The argument is an integer list. When you specify `types=`, you must also specify `typevar=`. Say you are creating a design with 30 choice sets, and you want the first 10 sets to consist of sets whose `typevar=` variable in the candidate set is type 1, and you want the rest to be type 2. You would specify `types=10 20`.

typevar= *variable*
specifies a variable in the candidate data set that contains choice set types. The types must be integers starting with 1. This option can only be specified with the set-swapping algorithm. When you specify `typevar=`, you must also specify `types=`.

weight= *weight-variable*
specifies an optional weight variable. Typical usage is with an availability design. Give unavailable alternatives a weight of zero and available alternatives a weight of one. The number of alternatives must always be constant, so varying numbers of alternatives are handled by giving unavailable or unseen alternatives a weight of zero.

%ChoicEff Macro Notes

This macro specifies `options nonotes` throughout most of its execution. If you want to see all of the notes, submit the statement `%let mktopts = notes;` before running the macro. To see the macro version, submit the statement `%let mktopts = version;` before running the macro.