The Macros

Warren F. Kuhfeld

Abstract

SAS provides a set of macros for designing experiments and analyzing choice data. The syntax and usage of these macros is discussed in this chapter. Two additional macros, one for scatter plots of labeled points and one for color interpolation, are documented here as well.[*]

Introduction

The following SAS autocall macros are available:

Macro	Page	Required Products	Purpose
`%ChoicEff`	806	STAT, IML	efficient choice design
`%MktAllo`	956		processes allocation data
`%MktBal`	959	QC	balanced main-effects designs
`%MktBIBD`	963	IML, QC	balanced incomplete block designs
`%MktBlock`	979	STAT, IML	block a linear or choice design
`%MktBSize`	989		sizes of balanced incomplete block designs
`%MktDes`	995	STAT, QC	efficient factorial design via candidate set search
`%MktDups`	1004		identify duplicate choice sets or runs
`%MktEval`	1012	STAT, IML	evaluate an experimental design
`%MktEx`	1017	STAT, IML, QC	efficient factorial design
`%MktKey`	1090		aid creation of the `key=` data set
`%MktLab`	1093		relabel, rename and assign levels to design factors
`%MktMDiff`	1105	STAT	MaxDiff (best-worst) analysis
`%MktMerge`	1125		merges a choice design with choice data
`%MktOrth`	1128		lists the orthogonal array catalog
`%MktPPro`	1145	IML	partial profiles through BIB designs
`%MktRoll`	1153		rolls a factorial design into a choice design
`%MktRuns`	1159		selecting number of runs in an experimental design
`%Paint`	1169		color interpolation
`%PHChoice`	1173		customizes the output from a choice model
`%PlotIt`	1178	STAT, GRAPH	graphical scatter plots of labeled points

[*]Copies of this chapter (MR-2010I), the other chapters, sample code, and all of the macros are available on the Web `http://support.sas.com/resources/papers/tnote/tnote_marketresearch.html`. Specifically, sample code is here `http://support.sas.com/techsup/technote/mr2010i.sas`. For help, please contact SAS Technical Support. See page 25 for more information.

These macros[*] are used for generating efficient factorial designs, efficient choice designs, processing and evaluating designs, processing data from choice experiments, and for certain graphical displays. The BASE product is required to run all macros along with any additional indicated products. To run the full suite of macros you need to have BASE, SAS/STAT, SAS/QC and SAS/IML installed, and you need SAS/GRAPH for the plotting macro. Once you have the right products installed, and you have installed the macros, you can call them and use them just as you would use a SAS procedure. However, the syntax for macros is a little different from SAS procedures. The following step makes an experimental design with 1 two-level and 7 three-level factors with 18 runs or profiles:

```
   %mktex(2 3 ** 7, n=18)
```

Changes and Enhancements

PROC TRANSREG has a new `sta` or `standorth` option for standardized orthogonal contrast coding that can be used with the `%ChoicEff` macro. This along with the new option, `options=relative`, can be used to get a relative *D*-efficiency in the 0 to 100 range for certain choice designs. The TRANSREG option is first available with SAS 9.2. It will not be recognized, and it will cause an error in earlier SAS releases.

The new `%MktBIBD` macro for balanced incomplete block designs (BIBDs) was added, and a number of the examples were revised to use it. This macro can also make BIBDs with directional and nondirectional row-neighbor balance. The new `%MktMDiff` macro analyzes data from MaxDiff (best-worst) choice models. Some new orthogonal arrays were added to the `%MktEx` macro, and there are a few other changes as well. There is a new discussion of Latin square and Graeco-Latin Square designs beginning on page 1026.

Installation

If your site has installed the autocall libraries supplied by SAS and uses the standard configuration of SAS supplied software, you need to ensure that the SAS system option `mautosource` is in effect to begin using the autocall macros. Note, however, that there might be differences between the macros used in this documentation and those that were shipped with your version of SAS. Be sure to get the latest macros from `http://support.sas.com/resources/papers/tnote/tnote_marketresearch.html`. These macros will work with Version 9 and later versions of SAS. You should install *all* of these macros, not just one or some. Some of the macros call other macros and will not work if the other macros are not there or if only older versions of the other macros are there. For example, the `%MktEx` macro calls: the `%MktRuns` macro to parse the factors list, the `%MktDes` macro for candidate set generation and search, and the `%MktOrth` macro to get the orthogonal array catalog.

[*]All of these macros come with the following disclaimer: This information is provided by SAS institute Inc. as a service to its users. It is provided “as is”. There are no warranties, expressed or implied, as to merchantability or fitness for a particular purpose regarding the accuracy of the materials or code contained herein.

The macros do not have to be included (for example, with a `%include` statement). They can be called directly once they are properly installed. For more information about autocall libraries, see *SAS Macro Language: Reference*. On a PC for example, the autocall library might be installed in the `stat\sasmacro` directory off of your SAS root directory. The name of your SAS root directory could be anything, but it is probably something like C:\Program Files\SAS\SASFoundation\9.2. One way to find the right directory is to use `Start` → `Find` to find one of the macros such as `mktex.sas`. The following are some examples of SAS/STAT autocall macro directories on a PC:

```
C:\Program Files\SAS\SASFoundation\9.2\stat\sasmacro
C:\Program Files\SAS\SAS 9.1\stat\sasmacro
```

Unix has a similar directory structure to the PC. The autocall library in Unix might be installed in the `stat/sasmacro` directory off of your SAS root directory. On MVS, each macro is a different member of a PDS. For details on installing autocall macros, consult your host documentation.

Usually, an autocall library is a directory containing individual files, each of which contains one macro definition. An autocall library can also be a SAS catalog. To use a directory as a SAS autocall library, store the source code for each macro in a separate file in the directory. The name of the file must be the same as the macro name, typically followed by `.sas`. For example, the macro `%MktEx` must typically be stored in a file named `mktex.sas`. On most hosts, the reserved `fileref sasautos` is assigned at invocation time to the autocall library supplied by SAS or another one designated by your site.

The libraries that SAS searches for autocall macros are controlled by the SASAUTOS option. The default value for this option is set in the configuration file. One way to have SAS find your autocall macros is to update this option in the SAS configuration file. For example, searching for SAS*.CFG might turn up SASV9.CFG on a PC that will contain lines like the following:

```
/* Setup the SAS System autocall library definition */
-SET SASAUTOS  (
                "!sasroot\core\sasmacro"
                "!sasext0\ets\sasmacro"
                "!sasext0\graph\sasmacro"
                "!sasext0\iml\sasmacro"
                "!sasext0\qc\sasmacro"
                "!sasext0\share\sasmacro"
                "!sasext0\stat\sasmacro"
               )
```

The directories listed in this option will depend on what SAS products you have licensed. If you replace the existing macros in `stat\sasmacro`, you should never have to change this file. However, if you install the new macros anywhere else, you need to ensure that that location appears in your configuration file *before* `stat\sasmacro` or you will not get all of the most recent macros. For details, see your host documentation and SAS macro language documentation.

%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

      Variable                                   Standard
n       Name       Label    Variance    DF        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 `%ChoicEff` 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 `%ChoiceEff` 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 `%ChoiceEff` 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
```

```
             Variable                                           Standard
        n      Name       Label     Variance     DF       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: $(x \bmod 3) + 1$.

[†]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
```

```
          Variable                                    Standard
     n      Name       Label     Variance     DF       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.
```

```
          Variable                                       Standard
     n      Name       Label     Variance     DF          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 `%ChoiceEff` 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 × 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 `%ChoiceEff` 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 `%ChoiceEff` macro figure out how to impose all of the restrictions. Otherwise the `%ChoiceEff` 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


            Variable                                                 Standard
      n     Name         Label               Variance    DF         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

             Variable                                                 Standard
        n    Name            Label                Variance    DF         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

            Variable                                                Standard
      n     Name            Label              Variance      DF        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                 .
```

```
         Variable                                               Standard
  n      Name          Label                 Variance    DF       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

             Variable                                               Standard
        n    Name          Label                Variance    DF        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

             Variable                                                 Standard
       n     Name          Label                Variance     DF         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
```

```
          Variable                                          Standard
 n          Name       Label       Variance     DF           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 $3 \times 3 \times 3 = 27$ 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 $5 \times 27 + 1 = 136$ 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                 .


                                                                        Standard
 n Variable Name             Label                           Variance DF   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

                      Alaska,    Alaska,    California,    California,
         Place          999        1249         999            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

                Hawaii,    Hawaii,    Maine,    Maine,    Mexico,    Mexico,
   Place          999        1249       999      1249       999        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

                      Alaska,    Alaska,    California,    California,
         Place         Beach       Lake        Beach           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
```

```
             Hawaii,    Hawaii,    Maine,    Maine,    Mexico,    Mexico,
Place         Beach       Lake      Beach     Lake      Beach       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

                   Alaska,                California,
                    Bed &      Alaska,       Bed &      California,
     Place        Breakfast     Cabin      Breakfast       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

                                                                     Prob >
       Variable                             Assumed    Standard              Squared
     n Name      Label          Variance      Beta  DF   Error       Wald     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:

```
          Variable                                                 Standard
     n    Name            Label                   Variance    DF      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                   .

         Variable                                             Standard
    n    Name           Label          Variance     DF          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
```

```
                                                                 Standard
  n    Variable Name    Label                  Variance    DF       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                       Cannot Be
                      Design Sizes       Violations     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

                                 Current          Best
          Design    Row,Col  D-Efficiency  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:

```
O                        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
```

```
             Variable                                        Standard
       n       Name       Label     Variance     DF       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

                                                   Cumulative    Cumulative
            Price    Frequency     Percent     Frequency      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                   .

             Variable                                               Standard
       n     Name          Label            Variance     DF          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

                                          Cumulative    Cumulative
      Price    Frequency     Percent     Frequency      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
        Set     Alt    Brand    Type    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                   Cannot Be
          Design Sizes       Violations    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

                                                   Cumulative    Cumulative
         Set    SetType    Frequency     Percent    Frequency      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
 Set    Alt     w    Type    Brand    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
```

```
           Variable                                           Standard
     n       Name       Label           Variance     DF         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
 Set    Alt    Brand    Type    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:

```
        Variable                                     Standard
   n      Name       Label    Variance    DF          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

                Variable                                       Standard
           n      Name       Label     Variance     DF          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:

```
      Variable                                                    Standard
 n    Name          Label                 Variance    DF           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 26*th* 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.

%MktAllo Macro

The `%MktAllo` autocall macro manipulates data for an allocation choice experiment. See the page 535 for an example. The `%MktAllo` macro takes as input a data set with one row for each alternative of each choice set. For example, in a study with 10 brands plus a constant alternative and 27 choice sets, there are $27 \times 11 = 297$ observations in the input data set. The following output displays an example of an input data set:

```
                       Obs    Set     Brand       Price    Count

                         1      1                              0
                         2      1    Brand  1      $50       103
                         3      1    Brand  2      $75        58
                         4      1    Brand  3      $50       318
                         5      1    Brand  4     $100        99
                         6      1    Brand  5     $100        54
                         7      1    Brand  6     $100        83
                         8      1    Brand  7      $75        71
                         9      1    Brand  8      $75        58
                        10      1    Brand  9      $75       100
                        11      1    Brand 10      $50        56
                         .
                         .
                         .
                       296     27    Brand  9     $100        94
                       297     27    Brand 10      $50        65
```

It contains a choice set variable, product attributes (`Brand` and `Price`) and a frequency variable (`Count`) that contains the total number of times that each alternative was chosen.

The end result is a data set with twice as many observations that contains the number of times each alternative was chosen and the number of times it was not chosen. This data set also contains a variable `c` with a value of 1 for first choice and 2 for second or subsequent choice. A portion of this data set is as follows:

```
                       Obs    Set     Brand       Price   Count     c

                         1      1                             0     1
                         2      1                          1000     2
                         3      1    Brand  1      $50      103     1
                         4      1    Brand  1      $50      897     2
                         5      1    Brand  2      $75       58     1
                         6      1    Brand  2      $75      942     2
                         7      1    Brand  3      $50      318     1
                         8      1    Brand  3      $50      682     2
```

```
                        .
                        .
                        .
                      593     27      Brand 10      $50        65     1
                      594     27      Brand 10      $50       935     2
```

The following step shows how you use the `%MktAllo` macro:

```
   %mktallo(data=allocs2, out=allocs3, nalts=11,
            vars=set brand price, freq=Count)
```

The option `data=` names the input data set, `out=` names the output data set, `nalts=` specifies the number of alternatives, `vars=` names the variables in the data set that are used in the analysis excluding the `freq=` variable, and `freq=` names the frequency variable.

%MktAllo Macro Options

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

Option	Description
`help`	(positional) "help" or "?" displays syntax summary
`data=`*SAS-data-set*	input SAS data set
`freq=`*variable*	frequency variable
`nalts=`*n*	number of alternatives
`out=`*SAS-data-set*	output SAS data set
`vars=`*variable-list*	input variables

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

```
   %mktallo(help)
   %mktallo(?)
```

You must specify the `nalts=`, `freq=`, and `vars=` options.

data= *SAS-data-set*

specifies the input SAS data set. By default, the macro uses the last data set created.

freq= *variable*

specifies the frequency variable, which contains the number of times this alternative was chosen. This option must be specified.

nalts= *n*

specifies the number of alternatives (including if appropriate the constant alternative). This option must be specified.

out= *SAS-data-set*
specifies the output SAS data set. The default is `out=allocs`.

vars= *variable-list*
specifies the variables in the data set that are used in the analysis but not the `freq=` variable. This option must be specified.

%MktAllo 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.

%MktBal Macro

The `%MktBal` autocall macro creates factorial designs using an algorithm that ensures that the design is perfectly balanced, or when the number of levels of a factor does not divide the number of runs, as close to perfectly balanced as possible. Before using the `%MktBal` macro, you should try the `%MktEx` macro to see if it makes a design that is balanced enough for your needs. The `%MktEx` macro can directly create thousands of orthogonal and balanced designs that the `%MktBal` algorithm will never find. Even when the `%MktEx` macro cannot create an orthogonal and balanced design, it will usually find a nearly balanced design. Designs created with the `%MktBal` macro, while perfectly balanced, might be less efficient than designs found with the `%MktEx` macro, and for large problems, the `%MktBal` macro can be slow.

The `%MktBal` macro is *not* a full-featured experimental design generator. For example, you cannot specify interactions that you want to estimate or specify restrictions such as which levels may or may not appear together. You must use the `%MktEx` macro for that. The `%MktBal` macro builds a design by creating a balanced first factor, optimally (or nearly optimally) blocking it to create the second factor, then blocking the first two factors to create the third, and so on. Once it creates all factors, it refines each factor. Each factor is in turn removed from the design, and the rest of the design is reblocked, replacing the initial factor if the new design is more *D*-efficient.

The following steps provide a simple example of creating and evaluating a design with 2 two-level factors and 3 three-level factors in 18 runs:

```
%mktbal(2 2 3 3 3, n=18, seed=151)

%mkteval(data=design)
```

The `%MktEval` macro evaluates the results. This design is in fact optimal.

In all cases, the factors are named `x1`, `x2`, `x3`, and so on. You can use the `%MktLab` macro to conveniently change them.

This next example, at 120 runs and with factor levels greater than 5, is starting to get big, and by default, it will run slowly. You can use the `maxstarts=`, `maxtries=`, and `maxiter=` options to make the macro run more quickly. The following steps create the design, with and without these options:

```
%mktbal(2 3 4 5 6 7 8 9 10, n=120, options=progress, seed=17)

%mktbal(2 3 4 5 6 7 8 9 10, n=120, options=progress, seed=17,
        maxstarts=1, maxiter=1, maxtries=1)
```

The second example, with the options, runs much faster than the first.

%MktBal Macro Options

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

Option	Description
`list`	(positional) list of the numbers of levels
	(positional) "help" or "?" displays syntax summary
`init=`*SAS-data-set*	initial input experimental design
`iter=`*n*	maximum iterations (designs to create)
`maxinititer=`*n*	maximum initialization iterations
`maxiter=`*n*	maximum iterations (designs to create)
`maxstarts=`*n*	maximum number of random starts
`maxtries=`*n*	times to try refining each factor
`n=`*n*	number of runs in the design
`options=noprint`	suppress the final *D*-efficiency
`options=nosort`	do not sort the final design
`options=oa`	an orthogonal array is sought
`options=progress`	reports on macro progress
`options=sequential`	factors are added but not refined
`out=`*SAS-data set*	output experimental design
`seed=`*n*	random number seed

Help Option

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

```
%mktbal(help)
%mktbal(?)
```

list

specifies a list of the numbers of levels of all the factors. For example, for 3 two-level factors specify either `2 2 2` or `2 ** 3`. Lists of numbers, like `2 2 3 3 4 4` or a *levels**number of factors* syntax like: `2**2 3**2 4**2` can be used, or both can be combined: `2 2 3**4 5 6`. The specification `3**4` means 4 three-level factors. You must specify a list. Note that the factor list is a positional parameter. This means it must come first, and unlike all other parameters, it is not specified after a name and an equal sign.

n= *n*

specifies the number of runs in the design. You must specify `n=`. You can use the `%MktRuns` macro to get suggestions for values of `n=`.

out= *sas-data set*
specifies the output experimental design. The default is `out=design`.

These next options control some of the details of the `%MktBal` macro.

init= *sas-data set*
specifies the initial design. You can specify just the first few columns of the design, in this data set with column names `x1`, `x2`, ..., and these columns will never change. This is different from the `init=` data set in the `%MktEx` macro. It is often the case that the first few columns can be constructed combinatorially, and it is known that the optimal design will just add new columns to the initial few orthogonal columns. This option makes that process more efficient. By default, when `init=` is not specified, all columns are iteratively found and improved.

maxinititer= *n*
specifies the maximum number of random starts for each factor during the initialization stage. This is the number of iterations that PROC OPTEX performs for each factor during the initialization. With larger values, the macro tends to find slightly better designs at a cost of slower run times. The default is the value of `maxstarts=`.

maxiter= *n*
iter= *n*
specifies the maximum iterations (designs to create). By default, `maxiter=5`. This is the outer most set of iterations.

maxstarts= *n*
specifies the maximum number of random starts for each factor. This is the number of iterations that PROC OPTEX performs for each factor. With larger values, the macro tends to find slightly better designs at a cost of slower run times. The default is `maxstarts=10`.

maxtries= *n*
specifies the maximum number of times to try refining each factor after the initialization stage. The default is `maxtries=10`. Increasing the value of this option usually has little effect since the macro stops refining each design when the efficiency stabilizes.

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

`noprint`
specifies that the final *D*-efficiency should not be displayed.

`nosort`
do not sort the final design.

`oa`
specifies that an orthogonal array is sought. Factors are added sequentially and they are acceptable only when D-efficiency reaches 100. With `options=oa`, you can use the `%MktBal` macro to search for small orthogonal arrays or in some cases augment existing orthogonal arrays.

`progress`
reports on the macro's progress. For large numbers of factors, a large number or runs, or when the number of levels is large, this macro is slow. The `options=progress` specification gives you information about which step is being executed.

`sequential`
sequentially adds factors to the design and does not go back and refine them.

seed= *n*
specifies the random number seed. By default, `seed=0`, and clock time is used to make 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.

%MktBal 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.

%MktBIBD Macro

The `%MktBIBD` autocall macro finds balanced incomplete block designs (BIBDs). A BIBD is a list of treatments that appear together in blocks. Each block contains a subset of the treatments. BIBDs can be used in marketing research to construct partial-profile designs (Chrzan and Elrod 1995). The entries in the BIBD say which attributes are to be shown in each set. For example, a BIBD could be used for the situation where there are t attributes or messages (treatments) and k are shown at a time. A total of b sets of attributes (blocks) are shown. For examples of using this macro in the partial-profile context, see page 1145. See the following pages for examples of using this macro in the discrete choice chapter: 642, 645, 650, 654, 656, 659 and 662. Additional examples appear throughout this chapter.

BIBDs are also used in marketing research to construct MaxDiff (best-worst) designs (Louviere 1991, Finn and Louviere 1992). In a MaxDiff study, subjects are shown sets (blocks) of messages or product attributes (treatments) and are asked to choose the best (or most important) from each set as well as the worst (or least important). For examples of using this macro in the MaxDiff context, see page 1105.

The documentation for the `%MktPPro` and `%MktMDiff` macros discuss the `%MktBIBD` macro in the context of sets and attributes, whereas most of the `%MktBIBD` macro documentation uses the more traditional statistical vocabulary of blocks and treatments. Furthermore, the output of the `%MktBIBD` macro can correspond to the marketing research vocabulary (when the **`nattrs=`** option is specified) or the statistical vocabulary (when the **`t=`** option is specified). The **`t=`** and **`nattrs=`** options are otherwise aliases for each other and specify the number of treatments (or attributes).

The following design is an example of a BIBD:

```
                Balanced Incomplete Block Design

                         x1    x2    x3

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

This BIBD has $b = 10$ blocks of treatments. Each of the 10 rows is a block. There are $t = 5$ treatments, since the entries in the design are the integers 1 through 5. Each block consists of $k = 3$ treatments. Note that the number of entries in the design, $b \times k = 10 \times 3 = 6 \times t = 30$. Each of the $t = 5$ treatments occurs exactly 6 times in the design, and each treatment occurs with every other treatment 3 times. This can be seen in the following treatment by treatment frequencies:

```
                    Treatment by Treatment Frequencies

                               1  2  3  4  5

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

When the `%MktBIBD` macro makes a BIBD, it tries to optimize the treatment by position frequencies. In other words, it tries to ensure that each treatment occurs in each of the k positions equally often, or at least close to equally often. The following are the treatment by position frequencies for this design, which in this case are perfect:

```
                    Treatment by Position Frequencies

                                  1  2  3

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

The following `%MktBIBD` macro step generates this BIBD:

```
   %mktbibd(b=10, t=5, k=3, seed=104)
```

In addition to the three tables above, the `%MktBIBD` macro also displays the following summary information:

```
            Block Design Efficiency Criterion       100.0000
            Number of Treatments, t                        5
            Block Size, k                                  3
            Number of Blocks, b                           10
            Treatment Frequency                            6
            Pairwise Frequency                             3
            Total Sample Size                             30
            Positional Frequencies Optimized?            Yes
```

The fact the that efficiency is 100 shows that `%MktBIBD` found a BIBD. In many cases, it will find a design that is close to a BIBD but each of the pairwise frequencies is not constant. For many marketing research problems, this is good enough.

The BIBD is available in the `out=BIBD` data set. The following step displays the design:

```
proc print data=bibd noobs; run;
```

The results are as follows:

```
                              x1    x2    x3

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

Every BIBD has a binary representation, or *incidence matrix*, with b rows and t columns and k ones that indicate which treatments appear in each block. The following step displays the `outi=incidence` matrix:

```
proc print data=incidence noobs; run;
```

The results are as follows:

```
                    b1    b2    b3    b4    b5

                     0     1     1     0     1
                     1     1     0     0     1
                     0     1     1     1     0
                     1     0     1     1     0
                     1     0     1     0     1
                     1     1     1     0     0
                     1     0     0     1     1
                     0     0     1     1     1
                     1     1     0     1     0
                     0     1     0     1     1
```

This shows that the first block consists of treatments 2, 3, and 5, just as the `out=BIBD` design does, but in a different format.

You can also view the BIBD arrayed as a block by treatment factorial design. The following step displays the first 9 (out of 30) observations:

```
proc print data=factorial(obs=9) noobs; run;
```

The results are as follows:

```
                    Block    Treatment

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

The following `%MktBIBD` macro step generates a BIBD with nondirectional row-neighbor balance:

```
%mktbibd(b=14, t=7, k=4, options=neighbor, seed=104)
```

The results are as follows:

```
Block Design Efficiency Criterion      100.0000
Number of Treatments, t                       7
Block Size, k                                 4
Number of Blocks, b                          14
Treatment Frequency                           8
Pairwise Frequency                            4
Total Sample Size                            56
Positional Frequencies Optimized?           Yes
Row-Neighbor Frequencies Optimized?         Yes

       Treatment by Treatment Frequencies

                1  2  3  4  5  6  7

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

```
                  Treatment by Position Frequencies

                              1  2  3  4

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

                      Row-Neighbor Frequencies

                         1  2  3  4  5  6  7

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

                   Balanced Incomplete Block Design

                         x1    x2    x3    x4

                          3     4     2     5
                          5     6     1     4
                          1     4     5     3
                          7     1     6     4
                          4     5     7     2
                          5     3     7     6
                          3     7     5     1
                          1     2     4     7
                          6     2     1     5
                          4     7     6     3
                          2     3     4     6
                          2     6     3     1
                          6     5     2     7
                          7     1     3     2
```

The resulting design is a BIBD with each of the 7 treatments appearing in each of the 4 positions within a block exactly twice. Furthermore, the row-neighbor frequencies show that each of the $7\times(7-1)/2 = 21$ pairs of treatments occur exactly twice. The pairs in this design are 3 with 4, 4 with 2, 2 with 5, 5 with 6, and so on. The order of the treatments in each pair is ignored. Hence, in this design, with the nondirectional row-neighbor balance, the 2 followed by 5 in the first row of the design is treated the same as the 5 followed by 2 in the second to last row of the design. This design is usually easily found in a few seconds. The last line of the first table ("Row-Neighbor Frequencies Optimized? Yes") along with the constant row-neighbor frequencies, shows that perfect row-neighbor balance was achieved.

The following finds a BIBD with row-neighbor balance where the order of the treatments *does* matter using `options=serial`:

```
%mktbibd(b=14, t=7, k=4, options=serial, seed=361699)
```

The results are as follows:

```
Block Design Efficiency Criterion      100.0000
Number of Treatments, t                       7
Block Size, k                                 4
Number of Blocks, b                          14
Treatment Frequency                           8
Pairwise Frequency                            4
Total Sample Size                            56
Positional Frequencies Optimized?           Yes
Row-Neighbor Frequencies Optimized?         Yes

       Treatment by Treatment Frequencies

                1  2  3  4  5  6  7

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

       Treatment by Position Frequencies

                    1  2  3  4

                 1  2  2  2  2
                 2  2  2  2  2
                 3  2  2  2  2
                 4  2  2  2  2
                 5  2  2  2  2
                 6  2  2  2  2
                 7  2  2  2  2
```

```
                    Row-Neighbor Frequencies

                       1  2  3  4  5  6  7

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

                Balanced Incomplete Block Design

                      x1    x2    x3    x4

                       1     4     2     5
                       3     1     5     7
                       4     5     1     7
                       2     3     7     1
                       6     7     5     3
                       7     6     4     1
                       7     2     6     5
                       1     3     4     6
                       3     5     2     4
                       4     7     3     2
                       5     4     3     6
                       5     6     1     2
                       6     2     7     4
                       2     1     6     3
```

In this set of output, with `options=serial`, the row-neighbor frequencies appear both above and below the diagonal. In contrast, in the previous set of output, with `options=neighbor`, the row-neighbor frequencies only appeared above the diagonal. You can see that each pair occurs exactly once in this design. In this design, with the serial (directional) row-neighbor balance, the 2 followed by 5 in the first row of the design is *not* treated the same as the 5 followed by 2 in the ninth row of the design.

An `options=serial` design is typically much harder to find than an `options=neighbor` design. In this case, a random number seed that is known to produce an optimal design reasonably quickly was chosen.[*] Usually, you will have to run the macro more than once or change some options such as the time value in the `positer=` option and iterate for up to a few hours to find an equivalent design.

The following requests this same design with a different seed:

```
   %mktbibd(b=14, t=7, k=4, options=serial, seed=104)
```

[*]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, results might not be exactly reproducible everywhere.

A portion of the output is as follows:

```
                    Positional Frequencies Optimized?            No
                    Row-Neighbor Frequencies Optimized?          No

                           Treatment by Position Frequencies

                                      1  2  3  4

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

                               Row-Neighbor Frequencies

                                  1  2  3  4  5  6  7

                               1     1  1  1  1  1  1
                               2  1     1  1  1  2  1
                               3  1  1     1  1  1  1
                               4  1  1  1     1  1  1
                               5  1  1  1  1     1  1
                               6  1  1  1  1  1     1
                               7  1  1  1  1  1  0
```

With most seeds and the default amount of iteration you will get results like these with nearly optimal position and row neighbor frequencies.

The following requests a design where it is not possible to have constant frequencies in the row-neighbor frequencies matrix:

```
   %mktbibd(b=7, t=7, k=4, options=serial, seed=104)
```

A portion of the output is as follows:

```
                    Positional Frequencies Optimized?           Yes
                    Row-Neighbor Frequencies Optimized?         Yes
                    Row-Neighbor Frequencies

                       1  2  3  4  5  6  7

                    1     0  1  1  1  0  0
                    2  0     1  0  0  1  1
                    3  1  0     0  0  1  1
                    4  0  1  0     1  1  0
                    5  1  1  0  0     0  1
                    6  0  0  1  1  1     0
                    7  1  1  0  1  0  0
```

The %MktBIBD macro reports that the row-neighbor frequencies are optimized since a mix of zeros and ones with no twos or larger values is optimal for this specification. However, the nonconstant frequencies show that row-neighbor balance is not possible for this BIBD.

BIBD Parameters

The parameters of a BIBD are:

- b specifies the number of blocks. In a partial-profile design, this is the number of profiles. In a MaxDiff design, this is the number of sets.
- t specifies the number of treatments. In a partial-profile or MaxDiff design, this is the total number of attributes or messages.
- k specifies the block size, which is the number of treatments in each block. In a partial-profile or MaxDiff design, this is the number of attributes or messages shown at one time.

When $r = b \times k/t$ and $l = r \times (k-1)/(t-1)$ are integers, and $k = t$ and $b \geq t$, then a complete block design might be possible. This is a necessary but not sufficient condition for the existence of a complete block design. When $r = b \times k/t$ and $l = r \times (k-1)/(t-1)$ are integers, and $k < t$ and $b \geq t$, then a balanced incomplete block design might be possible. This is a necessary but not sufficient condition for the existence of a BIBD. You can use the macro %MktBSize to find parameters in which BIBDs might exist. The %MktBIBD macro uses PROC OPTEX and a computerized search to find BIBDs. It does not have a library of BIBDs or use combinatorial constructions. Hence, it will not always find a BIBD even when one is known to exist. However, it usually works quite well in finding small BIBDs or something close for larger problems. The following step displays some of the smaller specifications for which a BIBD might exist:

```
%mktbsize(t=1 to 20, k=2 to 0.5 * t, b=t to 100)
```

The results are as follows:

```
                                                                    n
     t          k           b            r          Lambda         Total
Number of     Block    Number       Treatment    Pairwise       Sample
Treatments    Size     of Blocks    Frequency    Frequencies    Size

     4          2           6            3             1            12
     5          2          10            4             1            20
     6          2          15            5             1            30
     6          3          10            5             2            30
     7          2          21            6             1            42
     7          3           7            3             1            21
     8          2          28            7             1            56
     8          3          56           21             6           168
     8          4          14            7             3            56
     9          2          36            8             1            72
     9          3          12            4             1            36
     9          4          18            8             3            72
    10          2          45            9             1            90
    10          3          30            9             2            90
    10          4          15            6             2            60
    10          5          18            9             4            90
    11          2          55           10             1           110
    11          3          55           15             3           165
    11          4          55           20             6           220
    11          5          11            5             2            55
    12          2          66           11             1           132
    12          3          44           11             2           132
    12          4          33           11             3           132
    12          6          22           11             5           132
    13          2          78           12             1           156
    13          3          26            6             1            78
    13          4          13            4             1            52
    13          5          39           15             5           195
    13          6          26           12             5           156
    14          2          91           13             1           182
    14          4          91           26             6           364
    14          6          91           39            15           546
    14          7          26           13             6           182
    15          3          35            7             1           105
    15          5          21            7             2           105
    15          6          35           14             5           210
    15          7          15            7             3           105
```

16	3	80	15	2	240
16	4	20	5	1	80
16	5	48	15	4	240
16	6	16	6	2	96
16	7	80	35	14	560
16	8	30	15	7	240
17	4	68	16	3	272
17	5	68	20	5	340
17	8	34	16	7	272
18	6	51	17	5	306
18	9	34	17	8	306
19	3	57	9	1	171
19	4	57	12	2	228
19	6	57	18	5	342
19	7	57	21	7	399
19	9	19	9	4	171
20	4	95	19	3	380
20	5	76	19	4	380
20	8	95	38	14	760
20	10	38	19	9	380

%MktBIBD Macro Options

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

Option	Description
`help`	(positional) "help" or "?" displays syntax summary
`b=`*b*	number of blocks (alias for `nsets=`)
`group=`*n*	number of groups
`k=`*k*	block size (alias for `setsize=`)
`nattrs=`*t*	number of attributes (alias for `t=`)
`nsets=`*b*	number of sets (alias for `b=`)
`optiter=`*n1* < *n2* < *n3* >>	number of PROC OPTEX iterations
`options=position`	optimizes position frequencies
`options=neighbor`	nondirectional row-neighbor balance and position
`options=serial`	directional row-neighbor balance and position
`out=`*SAS-data-set*	output data set BIBD
`outf=`*SAS-data-set*	output data set factorial design
`outi=`*SAS-data-set*	output data set incidence matrix
`outs=`*SAS-data-set*	output data set sorted design
`positer=`*n1* < *n2* < *n3* >>	number of iterations position frequencies
`seed=`*n*	random number seed
`setsize=`*k*	set size (alias for `k=`)
`t=`*t*	number of treatments (alias for `nattrs=`)
`weights=`*n n*	position frequency badness weights

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

```
%mktbibd(help)
%mktbibd(?)
```

b= *b*
nsets= *b*

specifies the number of blocks. In a partial-profile design, this is the number of profiles. In a MaxDiff design, this is the number of sets. The `nsets=` and `b=` options are aliases. This option (in one of its two forms) must be specified.

group= *n*

specifies the number of groups into which the design is to be divided. This could be useful with MaxDiff and partial profiles. By default, the design is not divided into groups.

k= *k*
setsize= *k*

specifies the block size, which is the number of treatments in each block. In a partial-profile or MaxDiff design, this is the number of attributes or messages shown at one time in each set. The `setsize=` and `k=` options are aliases. This option (in one of its two forms) must be specified.

optiter= *n1 < n2 < n3 >>*

specifies the number of PROC OPTEX iterations. If one number is specified, it is just the number of iterations. With two numbers, *n1* and *n2*, *n1* iterations are performed and then efficiency is checked. If the block design efficiency criterion is 100, the iterations stop. Otherwise, this process is repeated up to *n2* times for a maximum of $n1 \times n2$ iterations. The default for *n2*, when it is missing, is 1000 when the parameters conform to the necessary conditions for a BIBD and 5 otherwise. The third value is the maximum amount of time in minutes to spend in PROC OPTEX. The default for the time value, when it is missing, is 5 minutes when the parameters conform to the necessary conditions for a BIBD and 0.5 minutes otherwise. Hence, when the parameters conform to the necessary conditions for a BIBD, the default is `optiter=500 1000 5`, otherwise, the default is `optiter=500 5 0.5`. For larger problems, you might want to specify values smaller than the defaults.

options= *options-list*

specifies binary options. By default, `options=position`.

`position`
optimizes position frequencies. The goal is for each treatment to appear equally often in each position.

`neighbor`
optimizes nondirectional row-neighbor balance and also position. The goal is for pairs of treatments, constructed from each of the first $k-1$ treatments in each block along with the treatment that follows it, to occur equally often in the design. The order of the treatments within each pair does not matter in evaluating row-neighbor balance. That is, treatment 1 appearing before treatment 2 is counted as the same as 2 appearing before 1.

`serial`
optimizes directional row-neighbor balance and also position. The goal is for pairs of treatments, constructed from each of the first $k-1$ treatments in each block along with the treatment that follows it, to occur equally often in the design. In contrast to `options=neighbor`, treatment 2 appearing before treatment 1 is treated as different from 1 appearing before 2.

out= *SAS-data-set*

specifies the output data set name for the $b \times k$ BIBD (or more generally, the incomplete block design). The default is `out=BIBD`.

outf= *SAS-data-set*

specifies the output data set name for the $bk \times 2$ factorial design matrix. The default is `outf=Factorial`.

outi= *SAS-data-set*

specifies the output data set name for the $b \times t$ incidence matrix. The default is `outi=Incidence`.

outs= *SAS-data-set*

specifies the output data set name for the $b \times k$ sorted design. The default is `outs=Sorted`.

positer= *n1 < n2 < n3 >>*

specifies the number of iterations for the algorithm that attempts to optimize the treatment by position frequencies. The default is `positer=200 5000 2`. The first value specifies the number of times to try to refine the design. The second value specifies the number of times to start from scratch with a different random start. Larger values increase the chances of finding better treatment by position frequencies at a cost of slower run times. The third value is the maximum amount of time in minutes to spend optimizing the positional frequencies. Specify `positer=` or `positer=0` to just request a design without optimizing position.

seed= *n*

specifies the random number seed. By default, `seed=0`, and clock time is used to make 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.

t= *t*
nattrs= *t*
specifies the number of treatments. In a partial-profile or MaxDiff design, this is the total number of attributes or messages. The `nattrs=` and `t=` options are aliases. This option (in one of its two forms) must be specified. When the `nattrs=` option is specified, the output will use the word "Attribute" rather than "Treatment" and "Set" rather than "Block".

weights= *n n*
specifies weights for position balance and row-neighbor balance. Specify two nonnegative numeric values. The total badness is a weighted sum of the position badness and the row-neighbor badness. The default is `weights=1 2`, so by default, row-neighbor balance is given the most weight. You can specify a weight of zero, for example, for the position balance (the first value) to just optimize row-neighbor balance.

Evaluating an Existing Block Design

The `%MktBIBD` macro does not have an option to evaluate existing block designs. However, you can run the following ad hoc macro to if you want to see the blocking design efficiency criterion, the treatment by position frequencies, and the treatment by treatment frequencies:

```
%macro mktbeval(design=_last_,      /* block design to evaluate         */
                attrortr=Attribute);/* string to print in the output    */
                                    /* specify Treatment or use default */

proc iml;
   use &design; read all into x;
   t = max(x);    k = ncol(x);    blocks = nrow(x);
   call symput('k', char(k));
   call symput('b', char(blocks));
   f = j(t, t, .);    p = j(t, k, 0);
   do i = 1 to blocks;
      do j = 1 to k;
         p[x[i,j],j] = p[x[i,j],j] + 1;
         do q = j to k;
            a = min(x[i,j],x[i,q]);
            b = max(x[i,j],x[i,q]);
            f[a,b] = sum(f[a,b], 1);
            end;
         end;
      end;
      options missing=' ';
      w = ceil(log10(t + 1)); t = right(char(1, w, 0) : char(t, w, 0));
      w = ceil(log10(k + 1)); q = right(char(1, w, 0) : char(k, w, 0));
      if max(f) < 100 then print "&attrortr by &attrortr Frequencies",,
         f[format=2. label='' rowname=t colname=t];
      else print "&attrortr by &attrortr Frequencies",,
         f[label='' rowname=t colname=t];
      if max(p) < 100 then print "&attrortr by Position Frequencies",,
         p[format=2. label='' rowname=t colname=q];
      else print "&attrortr by Position Frequencies",,
         p[label='' rowname=t colname=q];
      options missing='.';
      x = (1:blocks)` @ j(k, 1, 1) || shape(x, blocks * k);
      create __tmpbefac from x; append from x;
      quit;

   proc optex;
      class col2;
      model col2;
      generate initdesign=__tmpbefac method=sequential;
      blocks structure=(&b)&k init=chain iter=0;
      ods select BlockDesignEfficiencies;
      run;

   proc datasets nolist; delete __tmpbefac; run; quit;
   %mend;
```

The macro has two options. Specify the name of the block design in the `design=` option. By default, the treatments are labeled as "Attributes", however, you can specify `attrortr=Treatment` if you would like them labeled as treatments.

The following steps create a BIBD and evaluate it:

```
   %mktbibd(b=10, t=6, k=3, seed=104)

   %mktbeval;
```

The results of the evaluation step are as follows:

```
                        Attribute by Attribute Frequencies

                                  1  2  3  4  5  6

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

```
                         Attribute by Position Frequencies

                                      1  2  3

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

                              The OPTEX Procedure

          Design      Treatment        Treatment       Block Design
          Number     D-Efficiency     A-Efficiency     D-Efficiency
          ---------------------------------------------------------
               1        80.0000          80.0000         100.0000
```

These results match the results from the `%MktBIBD` macro (not shown). The efficiency result of interest is the block design *D*-efficiency, with is 100 for BIBDs.

%MktBIBD 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.

%MktBlock Macro

The `%MktBlock` autocall macro blocks a choice design or an ordinary linear experimental design. See the following pages for examples of using this macro in the discrete choice chapter: 426 and 497. Additional examples appear throughout this chapter. When a choice design is too large to show all choice sets to each subject, the design is blocked and a block of choice sets is shown to each subject. For example, if there are 36 choice sets, instead of showing each subject 36 sets, you could instead create 2 blocks and show 2 groups of subjects 18 sets each. You could also create 3 blocks of 12 choice sets or 4 blocks of 9 choice sets. You can also request just one block if you want to see the correlations and frequencies among all of the attributes of all of the alternatives of a choice design.

The design can be in one of two formats. Typically, a choice design has one row for each alternative of each choice set and one column for each of the attributes. Typically, this kind of design is produced by either the `%ChoicEff` or `%MktRoll` macro. Alternatively, a linear arrangement is an intermediate step in preparing some choice designs.[*] The linear arrangement has one row for each choice set and one column for each attribute of each alternative. Typically, the linear arrangement is produced by the `%MktEx` macro. The output from the `%MktBlock` macro is a data set containing the design, with the blocking variable added and hence not in the original order, with runs or choice sets nested within blocks.

The macro tries to create a blocking factor that is uncorrelated with every attribute of every alternative. In other words, the macro is trying to optimally add one additional factor, a blocking factor, to the linear arrangement. It is trying to make a factor that is orthogonal to all of the attributes of all of the alternatives. For linear arrangements, you can usually ask for a blocking factor directly as just another factor in the design, and then use the `%MktLab` macro to provide a name like `Block`, or you can use the `%MktBlock` macro.

The following steps create the blocking variable directly:

```
   %mktex(3 ** 7, n=27, seed=350)

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

The following steps create a design and then block it:

```
   %mktex(3 ** 6, n=27, seed=350)

   %mktblock(data=randomized, nblocks=3, seed=377, maxiter=50)
```

The results are as follows:

[*]See page 67 for an explanation of the linear arrangement of a choice design versus the arrangement of a choice design that is more suitable for analysis.

```
                    Canonical Correlations Between the Factors
                There are 0 Canonical Correlations Greater Than 0.316

               Block    x1      x2      x3      x4      x5      x6

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

                              Summary of Frequencies
                There are 0 Canonical Correlations Greater Than 0.316

                               Frequencies

               Block       9 9 9
               x1          9 9 9
               x2          9 9 9
               x3          9 9 9
               x4          9 9 9
               x5          9 9 9
               x6          9 9 9
               Block x1    3 3 3 3 3 3 3 3 3
               Block x2    3 3 3 3 3 3 3 3 3
               Block x3    3 3 3 3 3 3 3 3 3
               Block x4    3 3 3 3 3 3 3 3 3
               Block x5    3 3 3 3 3 3 3 3 3
               Block x6    3 3 3 3 3 3 3 3 3
               x1 x2       3 3 3 3 3 3 3 3 3
               x1 x3       3 3 3 3 3 3 3 3 3
               x1 x4       3 3 3 3 3 3 3 3 3
               x1 x5       3 3 3 3 3 3 3 3 3
               x1 x6       3 3 3 3 3 3 3 3 3
               x2 x3       3 3 3 3 3 3 3 3 3
               x2 x4       3 3 3 3 3 3 3 3 3
               x2 x5       3 3 3 3 3 3 3 3 3
               x2 x6       3 3 3 3 3 3 3 3 3
```

```
x3 x4       3 3 3 3 3 3 3 3 3
x3 x5       3 3 3 3 3 3 3 3 3
x3 x6       3 3 3 3 3 3 3 3 3
x4 x5       3 3 3 3 3 3 3 3 3
x4 x6       3 3 3 3 3 3 3 3 3
x5 x6       3 3 3 3 3 3 3 3 3
N-Way       1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
            1 1 1 1 1 1 1 1
```

The output shows that the blocking factor is uncorrelated with all of the factors in the design. This output comes from the `%MktEval` macro, which is called by the `%MktBlock` macro.

The following output displays the blocked linear arrangement (3 blocks of nine choice sets):

```
Block    Run    x1    x2    x3    x4    x5    x6

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

Block    Run    x1    x2    x3    x4    x5    x6

  2       1      1     3     3     1     3     2
          2      3     3     2     1     1     3
          3      2     2     3     2     2     2
          4      3     2     1     2     1     1
          5      2     1     2     3     2     3
          6      3     1     3     3     1     2
          7      2     3     1     1     2     1
          8      1     2     2     2     3     3
          9      1     1     1     3     3     1
```

```
Block     Run     x1     x2     x3     x4     x5     x6

  3        1      3      2      1      3      2      3
           2      3      1      3      1      2      1
           3      1      3      3      2      1      1
           4      2      2      3      3      3      1
           5      2      1      2      1      3      2
           6      1      2      2      3      1      2
           7      3      3      2      2      2      2
           8      2      3      1      2      3      3
           9      1      1      1      1      1      3
```

Note that in the linear version of the design, there is one row for each choice set and all of the attributes of all of the alternatives are in the same row.

Next, we will create and block a choice design with two blocks of nine sets instead of blocking the linear version of a choice design. The following steps create and then block a choice design:

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

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

%choiceff(data=out,                  /* candidate set of choice sets         */
          model=class(x1-x3 / sta),  /* model with stdz orthogonal coding    */
          nsets=18,                  /* number of choice sets                */
          nalts=2,                   /* number of alternatives               */
          seed=151,                  /* random number seed                   */
          options=nodups             /* do not create duplicate choice sets  */
                  relative,          /* display relative D-efficiency        */
          beta=zero)                 /* assumed beta vector, Ho: b=0         */

* Block the choice design.  Ask for 2 blocks;
%mktblock(data=best, nalts=2, nblocks=2, factors=x1-x3, seed=472)
```

(Note that if this had been a branded example, and if you are running SAS version 8.2 or an earlier release, specify `id=brand`; do not add your brand variable to the factor list. For SAS 9.0 and later SAS releases, it is fine to add your brand variable to the factor list.)

Both the design and the blocking are not as good this time. The variable names in the output are composed of `Alt`, the alternative number, and the factor name. Since there are two alternatives each composed of three factors plus one blocking variable ($2 \times 3 + 1 = 7$), a 7×7 correlation matrix is reported. Some of the results are as follows:

```
                 Canonical Correlations Between the Factors
           There are 7 Canonical Correlations Greater Than 0.316

          Block   Alt1_x1   Alt1_x2   Alt1_x3   Alt2_x1   Alt2_x2   Alt2_x3

Block     1        0.15      0         0.14      0.13      0.15      0.14
Alt1_x1   0.15     1         0.41      0.21      0.51      0.20      0.30
Alt1_x2   0        0.41      1         0.40      0.26      0.56      0.33
Alt1_x3   0.14     0.21      0.40      1         0.19      0.31      0.52
Alt2_x1   0.13     0.51      0.26      0.19      1         0.31      0.30
Alt2_x2   0.15     0.20      0.56      0.31      0.31      1         0.48
Alt2_x3   0.14     0.30      0.33      0.52      0.30      0.48      1

                          Summary of Frequencies
           There are 7 Canonical Correlations Greater Than 0.316
                    * - Indicates Unequal Frequencies

                               Frequencies

                Block            9 9
            *   Alt1_x1          8 5 5
            *   Alt1_x2          4 6 8
            *   Alt1_x3          6 7 5
            *   Alt2_x1          4 7 7
            *   Alt2_x2          8 5 5
            *   Alt2_x3          6 5 7

            *   Block Alt1_x1    4 2 3 4 3 2
            *   Block Alt1_x2    2 3 4 2 3 4
            *   Block Alt1_x3    3 3 3 3 4 2
            *   Block Alt2_x1    2 4 3 2 3 4
            *   Block Alt2_x2    4 3 2 4 2 3
            *   Block Alt2_x3    3 3 3 3 2 4

            *   Alt1_x1 Alt1_x2  3 3 2 1 1 3 0 2 3
            *   Alt1_x1 Alt1_x3  3 3 2 2 2 1 1 2 2
            *   Alt1_x1 Alt2_x1  0 4 4 2 0 3 2 3 0
            *   Alt1_x1 Alt2_x2  3 3 2 2 1 2 3 1 1
            *   Alt1_x1 Alt2_x3  3 3 2 2 1 2 1 1 3

            *   Alt1_x2 Alt1_x3  2 2 0 3 1 2 1 4 3
            *   Alt1_x2 Alt2_x1  1 2 1 1 3 2 2 2 4
            *   Alt1_x2 Alt2_x2  0 2 2 3 0 3 5 3 0
            *   Alt1_x2 Alt2_x3  1 1 2 3 2 1 2 2 4

            *   Alt1_x3 Alt2_x1  1 3 2 2 2 3 1 2 2
            *   Alt1_x3 Alt2_x2  3 2 1 3 1 3 2 2 1
            *   Alt1_x3 Alt2_x3  0 3 3 3 0 4 3 2 0
*    Alt2_x1 Alt2_x2    1 1 2 3 3 1 4 1 2
*    Alt2_x1 Alt2_x3    1 2 1 3 1 3 2 2 3
*    Alt2_x2 Alt2_x3    1 2 5 2 2 1 3 1 1
     N-Way              1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Note that in this example, the input is a choice design (as opposed to the linear version of a choice design) so the results are in choice design format. There is one row for each alternative of each choice set. Some of the results are as follows:

```
                         Block    Set    Alt    x1    x2    x3

                           1       1      1      3     2     3
                                          2      2     3     1

                           1       2      1      2     1     2
                                          2      1     3     1

                           .
                           .
                           .
                         Block    Set    Alt    x1    x2    x3

                           2       1      1      2     2     1
                                          2      1     3     2

                           2       2      1      2     3     1
                                          2      3     1     3

                           .
                           .
                           .
```

%MktBlock Macro Options

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

Option	Description
`help`	(positional) "help" or "?" displays syntax summary
`alt=`*variable*	alternative number variable
`block=`*variable*	block number variable
`data=`*SAS-data-set*	either the choice design or linear arrangement
`factors=`*variable-list*	factors in the design
`id=`*variable-list*	variables to copy to output data set
`initblock=`*variable*	initial blocking variable

Option	Description
`iter=`*n*	times to try to block the design
`list=`*n*	list larger canonical correlations
`maxiter=`*n*	times to try to block the design
`nalts=`*n*	number of alternatives in choice set
`nblocks=`*n*	number of blocks to create
`next=`*n*	where to look for the next exchange
`options=nosort`	do not sort the design into blocks
`out=`*SAS-data-set*	output data set with block numbers
`outr=`*SAS-data-set*	randomized output data set
`print=`*print-options*	output display options
`ridge=`*n*	ridging factor
`seed=`*n*	random number seed
`set=`*variable*	choice set number variable
`vars=`*variable-list*	factors in the design

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

```
%mktblock(help)
%mktblock(?)
```

alt= *variable*

specifies the alternative number variable. If this variable is in the input data set, it is excluded from the factor list. The default is `alt=Alt`.

block= *variable*

specifies the block number variable. If this variable is in the input data set, it is excluded from the factor list. The default is `block=Block`.

data= *SAS-data-set*

specifies either the choice design or the linear arrangement. The choice design has one row for each alternative of each choice set and one column for each of the attributes. Typically, this design is produced by either the `%ChoicEff` or `%MktRoll` macro. For choice designs, you must also specify the `nalts=` option. By default, the macro uses the last data set created. The linear arrangement has one row for each choice set and one column for each attribute of each alternative. Typically, this design is produced by the `%MktEx` macro. This is the design that is input into the `%MktRoll` macro.

factors= *variable-list*
vars= *variable-list*

specifies the factors in the design. By default, all numeric variables are used, except variables with names matching those specified in the `block=`, `set=`, `alt=`, and `id=` options. (By default, the variables `Block`, `Set`, `Run`, and `Alt` are excluded from the factor list.) If you are using version 8.2 or an earlier SAS release with a branded choice design (assuming the brand factor is called `Brand`), specify `id=Brand`. Do not add the brand factor to the factor list unless you are using SAS 9.0 or a later SAS release.

id= *variable-list*

specifies the `data=` data set variables to copy to the output data set. If you are using version 8.2 or an earlier SAS release with a branded choice design (assuming the brand factor is called `Brand`), specify `id=Brand`. Do not add the brand factor to the factor list unless you are using SAS 9.0 or a later SAS release.

initblock= *variable*

specifies the name of the variable in the data set that is to be used as the initial blocking variable for the first iteration.

list= *r*

lists canonical correlations larger than `list=`r. The default is $r = 0.316 \approx \sqrt{r^2 = 0.1}$.

maxiter= *n*
iter= *n*

specifies the number of times to try to block the design starting with a different random blocking. By default, the macro tries five random starts, and iteratively refines each until D-efficiency quits improving, then in the end selects the blocking with the best D-efficiency.

nalts= *n*

specifies the number of alternatives in each choice set. If you are inputting a choice design, you must specify `nalts=`, otherwise the macro assumes you are inputting a linear arrangement.

nblocks= *n*

specifies the number of blocks to create. The option `nblocks=1` just reports information about the design. The `nblocks=` option must be specified.

next= *n*

specifies how far into the design to go to look for the next exchange. The specification `next=1` specifies that the macro should try exchanging the level for each run with the level for the next run and all other runs. The specification `next=2` considers exchanges with half of the other runs, which makes the algorithm run more quickly. The macro considers exchanging the level for run i with run $i + 1$ then uses the `next=` value to find the next potential exchanges. Other values, including nonintegers can be specified as well. For example `next=1.5` considers exchanging observation 1 with observations 2, 4, 5, 7, 8, 10, 11, and so on. With smaller values, the macro tends to find a slightly better blocking variable at a cost of much slower run time.

options= *options-list*

specifies binary options. By default, no binary options are specified. Specify the following value after `options=`.

`nosort`
do not sort the design into blocks. This is useful anytime you want the order of the observations in the output data set to match the order of the observations in the input data set. You will typically not want to specify `options=nosort` when you are using the `%MktBlock` macro to block a design. However, `options=nosort` is handy when you are using the `%MktBlock` macro to add just another factor to the design.

out= *SAS-data-set*
specifies the output data set with the block numbers. The default is `out=blocked`. 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.

outr= *SAS-data-set*
specifies the randomized output data set if you would like the design randomly sorted within blocks. 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.

print= *print-options*
specifies both the `%MktBlock` and the `%MktEval` macro display options, which control the display of the results. The default is `print=normal`. Specify one or more values from the following list.

`all`	all output is displayed
`corr`	canonical correlations
`block`	canonical correlations within blocks
`design`	blocked design
`freqs`	long frequencies list
`list`	list of big canonical correlations
`nonzero`	like `ordered` but sets `list=1e-6`
`noprint`	no output is displayed
`normal`	`corr list summ design note`
`note`	blocking note
`ordered`	like `list` but ordered by variable names
`short`	`corr summ note`
`summ`	frequency summaries

ridge= *n*
specifies the value to add to the diagonal of $\mathbf{X}'\mathbf{X}$ to make it nonsingular. Usually, you will not need to change this value. If you do, you probably will not notice any effect. Specify `ridge=0` to use a generalized inverse instead of ridging. The default is `ridge=0.01`.

seed= *n*
specifies the random number seed. By default, `seed=0`, and clock time is used to make 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.

set= *variable*
specifies the choice set number variable. When `nalts=` is specified, the default is `Set`, otherwise the default is `Run`. If this variable is in the input data set, it is excluded from the factor list.

%MktBlock 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.

%MktBSize Macro

The `%MktBSize` autocall macro suggests sizes for balanced incomplete block designs (BIBDs). The sizes that it reports are sizes that meet necessary but not sufficient conditions for the existence of a BIBD, so a BIBD might not exist for every size reported. In the following example, a list of designs with 12 treatments, between 4 and 8 treatments per block, and between 12 and 30 blocks are requested:

```
%mktbsize(t=12, k=4 to 8, b=12 to 30)
```

The results are as follows:

```
                                                                          n
       t              k          b               r            Lambda      Total
   Number of      Block      Number        Treatment      Pairwise       Sample
   Treatments     Size       of Blocks     Frequency      Frequencies    Size

       12           6          22              11             5            132
```

You can use this information to create a BIBD with the `%MktBIBD` macro as follows:

```
%mktbibd(t=12, k=6, b=22, seed=104)
```

There is no guarantee that `%MktBIBD` will find a BIBD for any specification, but in this case it does, and it finds the following design:

```
                    Balanced Incomplete Block Design

                    x1    x2    x3    x4    x5    x6

                     7     9     3    10    12     2
                     5     2     4     6    10     7
                     5    10    12     9    11     8
                     7     9    11    12     1     4
                    12     3     4     5     9    10
                    10     6     9    11     2     1
                     3     8     7     1     9     6
                     2    10    12     6     8     1
                     9     7     6     4     8     5
                     1    12     5     7     6    11
                    12     1     2     3     5     7
                     8     4     1    10     7    12
                     6     2     9     8    12     4
                     4     7    11     3    10     6
                    11     5     1     2     4     9
                    11     3     2     8     7     9
                     6    11     8     5     3    12
```

```
                      3      6     10      9      1      5
                      2      4      6     12     11      3
                      1      8     10     11      4      3
                      4      1      3      2      5      8
                     10      5      8      7      2     11
```

The design has b=22 blocks (rows), k=6 treatments per block (columns), and t=12 treatments (the entries are the integers 1 to 12). Each of the t=12 treatments occurs r=11 times, and each treatment occurs in a block with every other treatment λ=5 times.

The following step creates a list of over 50 designs:

```
%mktbsize(t=5 to 20, k=3 to t - 1, b=t to 30)
```

Some of the results are as follows:

```
                                                                       n
         t           k           b            r            Lambda     Total
    Number of      Block     Number        Treatment     Pairwise     Sample
    Treatments     Size      of Blocks     Frequency     Frequencies  Size

         5           3          10             6              3          30
         5           4           5             4              3          20
         6           3          10             5              2          30
         6           4          15            10              6          60
         6           5           6             5              4          30
```

Note that by default, `maxreps=1` (the maximum number of replications is 1), so for example, the design t=5, k=3, b=20 is not listed since it consists of two replications of t=5, k=3, b=10, which is listed. Also note that b, the number of blocks, was specified so that it is never less than the number of treatments. Furthermore, k, the block size (number of treatments per block), is set to always be less than the number of treatments. Even more complicated expressions are permitted. For example, to limit the number of treatments per block to no more than half of the number of treatments, you could specify the following:

```
%mktbsize(t=2 to 10, k=2 to 0.5 * t, b=t to 10)
```

The results are as follows:

```
                                                                   n
      t            k          b            r           Lambda      Total
 Number of      Block      Number       Treatment     Pairwise     Sample
 Treatments     Size       of Blocks    Frequency     Frequencies  Size

      4            2           6            3             1          12
      5            2          10            4             1          20
      6            3          10            5             2          30
      7            3           7            3             1          21
```

To limit the number of blocks as a function of the number of treatments, you could specify the following:

```
%mktbsize(t=2 to 10, k=2 to t - 1, b=t to 2 * t)
```

However, if you want to limit the number of treatments as a function of the number of blocks, you need to use the `order=` option to ensure that the number of blocks loop comes first, for example, as follows:

```
%mktbsize(b=2 to 10, t=2 to 0.5 * b, k=2 to t - 1, order=btk)
```

The macro reports sizes in which $r = b \times k/t$ and $l = r \times (k-1)/(t-1)$ are integers, $2 \leq k < t$, and $b \geq t$. When $r = b \times k/t$ and $l = r \times (k-1)/(t-1)$ are integers, and $k = t$ and $b \geq t$, then a complete block design might be possible. This is a necessary but not sufficient condition for the existence of a complete block design. When $r = b \times k/t$ and $l = r \times (k-1)/(t-1)$ are integers, and $k < t$ and $b \geq t$, then a balanced incomplete block design might be possible. This is a necessary but not sufficient condition for the existence of a BIBD. When you specify `options=ubd` and $r = b \times k/t$ is an integer, then unbalanced block design sizes are reported as well. For example, if you want a design with `t=20` treatments and a block size of 6, you can run the following to find out how many blocks you need:

```
%mktbsize(t=20, k=6, options=ubd)
```

The results are as follows:

```
                                                                   n
      t            k          b            r          Lambda       Total
 Number of      Block      Number       Treatment     Pairwise     Sample
 Treatments     Size       of Blocks    Frequency     Frequencies  Size

     20            6          10            3           0.79         60
```

Then the `%MktBIBD` macro can be used to find a design where each treatment occurs 3 times, but the treatments do not appear together an equal number of times, for example, as follows:

```
%mktbibd(t=20, k=6, b=10, seed=104)
```

Some of the results are as follows:

```
                     Treatment by Treatment Frequencies

       1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20

 1     3   1   0   0   2   1   1   1   0   1   1   1   0   2   0   1   0   1   1   1
 2         3   0   1   1   1   1   1   1   1   0   1   1   0   1   0   1   0   2   1
 3             3   1   1   1   1   0   1   0   1   0   1   1   1   2   2   1   1   0
 4                 3   0   1   1   1   0   1   1   1   1   0   1   1   1   1   2   0
 5                     3   1   1   1   1   1   0   1   1   1   0   2   0   0   1   0
 6                         3   0   0   1   1   1   0   1   1   1   1   0   1   1   1
 7                             3   1   0   0   2   2   1   1   1   0   1   0   1   0
 8                                 3   1   1   1   1   2   1   0   1   1   0   0   1
 9                                     3   1   0   1   2   0   1   1   1   1   0   2
10                                         3   0   1   0   1   2   1   0   1   1   1
11                                             3   1   1   2   1   0   1   1   0   1
12                                                 3   1   0   1   0   0   1   1   1
13                                                     3   0   0   1   1   0   0   1
14                                                         3   1   1   1   1   0   1
15                                                             3   0   1   1   1   1
16                                                                 3   1   1   1   0
17                                                                     3   1   1   1
18                                                                         3   1   2
19                                                                             3   0
20                                                                                 3
```

%MktBSize Macro Options

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

Option	Description
`help`	(positional) “help” or “?” displays syntax summary
`b=`*do-list*	number of blocks (alias for `nsets=`)
`k=`*do-list*	block size (alias for `setsize=`)
`maxreps=`*n*	maximum number of replications
`nattrs=`*do-list*	number of attributes (alias for `t=`)
`nsets=`*do-list*	number of sets (alias for `b=`)
`order=`*order-list*	order of the loops
`options=nocheck`	suppress checking b, t, and k
`options=ubd`	lifts the balance restriction on the design
`out=`*SAS-data-set*	output data set design list
`setsize=`*do-list*	set size (alias for `k=`)
`t=`*do-list*	number of treatments (alias for `nattrs=`)

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

```
%mktbsize(help)
%mktbsize(?)
```

The `k=` or `setsize=`, and the `t=` or `nattrs=` options must be specified.

b= *do-list*
nsets= *do-list*

specifies the number of blocks. In a partial-profile design, this is the number of profiles. In a MaxDiff design, this is the number of sets. Specify either an integer or a list of integers in the SAS *do-list* syntax. The default is `b=2 to 500`. The `nsets=` and `b=` options are aliases.

k= *do-list*
setsize= *do-list*

specifies the block size, or the number of treatments in each block. In a partial-profile or MaxDiff design, this is the number of attributes or messages shown at one time in each set. Specify either an integer or a list of integers in the SAS *do-list* syntax. The `setsize=` and `k=` options are aliases. This option (in one of its two forms) must be specified.

maxreps= *n*

specifies the maximum number of replications. The default is `maxreps=1`. By default, this option prevents the `%MktBSize` macro from reporting designs of size $2b, 3b$, and so on after it has found a size with b blocks.

options= *options-list*

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

`nocheck`
by default, certain checks are performed on b, t, and k. Specify `options=nocheck` to turn them off. This lets you make some creative expressions that otherwise would not be permitted.

`ubd`
lifts the balance restriction on the design. Results are reported when $r = b \times k/t$ is in integer but $l = r \times (k-1)/(t-1)$ might or might not be an integer. Use this option when you want to see sizes where every treatment can occur equally often, but the pairwise frequencies can be unequal. The listing can contain both sizes where a BIBD might be possible (λ, the expected pairwise frequency, is an integer) and sizes where a BIBD is not possible (λ is not an integer). You might use this option, for example, when the block design is being used to make a partial-profile design.

order= `tkb` | `tbk` | `btk` | `bkt` | `kbt` | `ktb`
specifies the order of the loops, the default is `tkb`, *t* then *k* then *b*. If you specify expressions in `t=`, `b=`, or `k=`, you might need some other ordering. For example, if you specify something like `t = 2 to 0.5 * b`, then you must specify `order=bkt` or any other ordering that defines *b* before *t*. Alternatively, you can specify this option just to change the default ordering of the results.

out= *SAS-data-set*
specifies the output data set with the list of potential design sizes. The default is `out=bibd`.

t= *do-list*
nattrs= *do-list*
specifies the number of treatments. In a partial-profile or MaxDiff design, this is the total number of attributes or messages. Specify either an integer or a list of integers in the SAS *do-list* syntax. The `nattrs=` and `t=` options are aliases. This option (in one of its two forms) must be specified. When the `nattrs=` option is specified, the output will use the word "Attribute" rather than "Treatment" and "Set" rather than "Block".

%MktBSize 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.

%MktDes Macro

The `%MktDes` autocall macro creates efficient experimental designs. Usually, we will not need to call the `%MktDes` macro directly. Instead, we will usually use the `%MktEx` autocall macro, which calls the `%MktDes` macro as one of its many tools. At the heart of the `%MktDes` macro are PROC PLAN, PROC FACTEX, and PROC OPTEX. PROC PLAN creates full-factorial designs. PROC FACTEX creates fractional-factorial designs. Both procedures can be used to create a candidate set for PROC OPTEX to search. We use a macro instead of calling these procedures directly because the macro has a simpler syntax. You specify the names of the factors and the number of levels for each factor. You also specify the number of runs you want in your final design. For example, you can create a design in 18 runs with 2 two-level factors (`x1` and `x2`) and 3 three-level factors (`x3`, `x4`, and `x5`) as follows:

```
%mktdes(factors=x1-x2=2 x3-x5=3, n=18)
```

You can optionally specify interactions that you want to estimate. The macro creates a candidate design in which every effect you want to estimate is estimable, but the candidate design is bigger than you want. By default, the candidate set is stored in a SAS data set called CAND1. The macro then uses PROC OPTEX to search the candidate design for an efficient final design. By default, the final experimental design is stored in a SAS data set called DESIGN.

When the full-factorial design is small (by default less than 2189 runs, although sizes up to 5000 or 6000 runs are reasonably small), the experimental design problem is straightforward. First, the macro uses PROC PLAN to create a full-factorial candidate set. Next, PROC OPTEX searches the full-factorial candidate set. For very small problems (a few hundred candidates) PROC OPTEX will often find the optimal design, and for larger problems, it might not find *the* optimal design, but given sufficient iteration (for example, specify `maxiter=100` or more) it will find very good designs. Run time will typically be a few seconds or a few minutes, but it could be longer. The following shows a typical example of using the `%MktDes` macro to find an optimal nonorthogonal design when the full-factorial design is small (108 runs):

```
*---2 two-level factors and 3 three-level factors in 18 runs---;
%mktdes(factors=x1-x2=2 x3-x5=3, n=18, maxiter=500)
```

When the full-factorial design is larger, the macro uses PROC FACTEX to create a fractional-factorial candidate set. In those cases, the methods found in the `%MktEx` macro usually make better designs than those found with the `%MktDes` macro.

PROC FACTEX

The primary reason that the `%MktDes` macro exists is to provide a front end to PROC FACTEX, although it additionally provides a front end to PROC OPTEX and PROC PLAN. PROC FACTEX is powerful and general, and it does much more than we use it for here. However, that power leads to a somewhat long and detailed syntax. The `%MktDes` macro can help by writing that syntax for you. It should be pointed out that the syntax that the `%MktDes` macro writes for PROC FACTEX is often more verbose than it needs to be. It is simply easier for it to always write out a verbose style of syntax than try to be clever and take short cuts. Since PROC FACTEX is such an integral component of the `%MktDes` macro and hence the `%MktEx` macro, we will spend some time here reviewing what PROC FACTEX does and also the kinds of problems it was not designed to handle.

PROC FACTEX provides a powerful facility for constructing certain fractional-factorial designs. Specifically, PROC FACTEX is designed to make fractional-factorial designs based on prime numbers and powers of prime numbers. For example, since 2 is prime, you can use PROC FACTEX to make a design 2^{2^p-1} in 2^p runs. Examples include: 2^3 in $2^2 = 4$ runs, 2^7 in $2^3 = 8$ runs, 2^{15} in $2^4 = 16$ runs, 2^{31} in $2^5 = 32$ runs, 2^{63} in $2^6 = 64$ runs, and so on. Since 3 is prime, you can use PROC FACTEX to make a design $3^{(3^p-1)/2}$ in 3^p runs. Examples include: 3^4 in $3^2 = 9$ runs, 3^{13} in $3^3 = 27$ runs, 3^{40} in $3^4 = 81$ runs, and so on. Since 4 is a power of a prime, you can use PROC FACTEX to make a design $4^{(4^p-1)/3}$ in 4^p runs. Examples include: 4^5 in $4^2 = 16$ runs, 4^{21} in $4^3 = 64$ runs, and so on. Since 5 is prime, you can use PROC FACTEX to make a design $5^{(5^p-1)/4}$ in 5^p runs. Examples include: 5^6 in $5^2 = 25$ runs, 5^{31} in $5^3 = 125$ runs, and so on.

When the number of runs is a power of 2, you can use PROC FACTEX to make designs that contain mixes of 2, 4, 8, and other power-of-two-level factors. Examples include: $2^4 4^1$ in $2^3 = 8$ runs, $2^{3(5-q)} 4^q$ in $2^4 = 16$ runs for $q < 5$, $2^{13} 4^{12} 8^2$ in $2^6 = 64$ runs, and so on. When the number of runs is a power of 3, you can use PROC FACTEX to make designs that contain mixes of 3, 9, 27, and other power-of-three-level factors, and so on.

The `%MktEx` macro uses the `%MktDes` macro and PROC FACTEX for all of the cases described so far. PROC FACTEX is used both to make larger orthogonal arrays and to make all fractional-factorial candidate sets. There are many times, however, when PROC FACTEX cannot be used to make orthogonal arrays. You *cannot* use PROC FACTEX to make a design $6^{(6^p-1)/5}$ in 6^p runs since 6 is not a prime number. Even for designs that exist, like 6^3 in $6^2 = 36$ runs, you cannot use PROC FACTEX to make this design, because again, 6 is not a prime number. Similarly, you cannot use PROC FACTEX to make 2^{11} in 12 runs (12 is not a power of 2), 2^{19} in 20 runs (20 is not a power of 2), 3^7 in 18 runs (18 is not a power of 3), and so on. You have to use the `%MktEx` macro to get designs like these. You can use PROC FACTEX to create the design 6^3 in 64 runs by creating 3 eight-level factors in 64 runs and coding down. However, such a design, while orthogonal, would be nonoptimal both compared to 6^3 in 36 runs (which is 100% D-efficient) and compared to an optimal but nonorthogonal array 6^3 in 64 runs.

The `%MktDes` and `%MktEx` macros rely on PROC FACTEX to make many orthogonal arrays that `%MktEx` cannot otherwise make, and the `%MktEx` macro can make many designs that PROC FACTEX cannot make. However, for $2^p < 128$, and for many of the other orthogonal arrays not based on powers of 2, both the `%MktEx` macro and PROC FACTEX can make the same designs, but by using totally different approaches. PROC FACTEX does a computerized search, whereas the `%MktEx` macro develops a difference scheme. The `%MktEx` macro will usually use its own methods to make these orthogonal arrays when an orthogonal array was directly requested, but use PROC FACTEX to make the same orthogonal array when it is needed for a candidate set.

We discussed making mixed orthogonal arrays when the numbers of levels are all powers of the same prime (with mixes of 2's and 4's being the most common example). You can also use PROC FACTEX to create other mixes in several ways. You can create a mix of two-level and three-level factors by first creating a mix of two-level and four-level (or eight-level) factors and then coding down the four- or eight-level factors into three-level factors. This preserves orthogonality but leads to imbalance. You can instead create a design with just the two-level factors (in 2^p runs) and a separate design with just the three-level factors (in 3^q runs) and cross them (pair each of the 2^p runs with each of the 3^q runs) creating a design in $2^p \times 3^q$ runs. This preserves orthogonality and balance, but the designs tend to get big quite quickly, and other approaches to mixed orthogonal array creation provide more factors. The smallest example of this approach is $2^3 3^4$ in $2^2 \times 3^2 = 36$ runs. In contrast, the `%MktEx` macro can directly construct the mixed orthogonal array $2^{11} 3^{12}$ in 36 runs. The `%MktEx` macro uses the `%MktDes` macro to construct candidate sets using this first method, but does not employ the second method.

In summary, PROC FACTEX provides powerful software for constructing fractional-factorial designs. The `%MktDes` macro provides a front end to this procedure that makes it easier to call. The `%MktEx` macro uses the `%MktDes` macro for the things it is best at, but not for other things.

%MktDes Macro Options

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

Option	Description
`help`	(positional) "help" or "?" displays syntax summary
`big=`*n*	size of big candidate set
`cand=`*SAS-data-set*	candidate design
`classopts=`*options*	`class` statement options
`coding=`*name*	`coding=` option
`examine=`< I > < V >	matrices that you want to examine
`facopts=`*options*	PROC FACTEX statement options
`factors=`*factor-list*	factors and levels for each factor
`generate=`*options*	`generate` statement options
`interact=`*interaction-list*	interaction terms
`iter=`*n*	number of designs
`keep=`*n*	number of designs to keep
`maxiter=`*n*	number of designs
`method=`*name*	search method
`n=`*n* | SATURATED	number of runs
`nlev=`*n*	number of levels for pseudo-factors
`options=allcode`	shows all code
`options=check`	checks the `cand=` design
`options=nocode`	suppress the procedure code display
`otherfac=`*variable-list*	other factors
`otherint=`*terms*	multi-step interaction terms
`out=`*SAS-data-set*	output experimental design
`procopts=`*options*	PROC OPTEX statement options
`run=`*procedure-list*	list of procedures that can be run
`seed=`*n*	random number seed
`size=`*n* | MIN	candidate-set size
`step=`*n*	step number
`where=`*where-clause*	`where` clause

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

```
%mktdes(help)
%mktdes(?)
```

big= *n*

specifies the size at which the candidate set is considered to be big. By default, `big=2188`. If the size of the full-factorial design is less than or equal to this size, and if PROC PLAN is in the `run=` list, the macro uses PROC PLAN instead of PROC FACTEX to create the candidate set. The default of 2188 is $\max(2^{11}, 3^7) + 1$. Specifying values as large as `big=6000` or even slightly larger is often reasonable. However, run time is slower as the size of the candidate set increases. The `%MktEx` macro coordinate-exchange algorithm will usually work better than a candidate-set search when the full-factorial design has more than several thousand runs.

cand= *SAS-data-set*

specifies the output data set with the candidate design (from PROC FACTEX or PROC PLAN). The default name is `Cand` followed by the step number, for example: `Cand1` for step 1, `Cand2` for step 2, and so on. You should only use this option when you are reading an external candidate set. When you specify `step=` values greater than 1, the macro assumes the default candidate set names, CAND1, CAND2, and so on, were used in previous steps. Specify just a data set name, no data set options.

classopts= *options*

specifies PROC OPTEX `class` statement options. The default, is `classopts=param=orthref`. You probably never want to change this option.

coding= *name*

specifies the PROC OPTEX `coding=` option. This option is usually not needed.

examine= < I > < V >

specifies the matrices that you want to examine. The option `examine=I` displays the information matrix, $\mathbf{X}'\mathbf{X}$; `examine=V` displays the variance matrix, $(\mathbf{X}'\mathbf{X})^{-1}$; and `examine=I V` displays both. By default, these matrices are not displayed.

facopts= *options*

specifies PROC FACTEX statement options.

factors= *factor-list*

specifies the factors and the number of levels for each factor. The `factors=` option must be specified. All other options are not required. The following shows a simple example of creating a design with 10 two-level factors:

```
%mktdes(factors=x1-x10=2)
```

First, a factor list, which is a valid SAS variable list, is specified. The factor list must be followed by an equal sign and an integer, which gives the number of levels. Multiple lists can be specified. For example, to create 5 two-level factors, 5 three-level factors, and 5 five-level factors, specify the following:

```
%mktdes(factors=x1-x5=2 x6-x10=3 x11-x15=5)
```

By default, this macro creates each factor in a fractional-factorial candidate set from a minimum number of pseudo-factors. Pseudo-factors are not output; they are used to create the factors of interest and then discarded. For example, with `nlev=2`, a three-level factor `x1` is created from 2 two-level pseudo-factors (_1 and _2) and their interaction by coding down:

```
(_1=1, _2=1) -> x1=1
(_1=1, _2=2) -> x1=2
(_1=2, _2=1) -> x1=3
(_1=2, _2=2) -> x1=1
```

This creates imbalance—the 1 level appears twice as often as 2 and 3. Somewhat better balance can be obtained by instead using three pseudo-factors. The number of pseudo-factors can be specified in parentheses after the number of levels, for example, as follows:

```
%mktdes(factors=x1-x5=2 x6-x10=3(3))
```

The levels 1 to 8 are coded down to 1 2 3 1 2 3 1 3, which is better balanced. The cost is candidate-set size might increase and efficiency might actually decrease. Some researchers are willing to sacrifice a little bit of efficiency in order to achieve better balance.

generate= *options*

specifies the PROC OPTEX `generate` statement options. By default, additional options are not added to the `generate` statement.

interact= *interaction-list*

specifies interactions that must be estimable. By default, no interactions are guaranteed to be estimable. Examples:

```
interact=x1*x2
interact=x1*x2 x3*x4*x5
interact=x1|x2|x3|x4|x5@2
interact=@2
```

The interaction syntax is like PROC GLM's and many of the other modeling procedures. It uses "`*`" for simple interactions (`x1*x2` is the interaction between `x1` and `x2`), "`|`" for main effects and interactions (`x1|x2|x3` is the same as `x1 x2 x1*x2 x3 x1*x3 x2*x3 x1*x2*x3`) and "`@`" to eliminate higher-order interactions (`x1|x2|x3@2` eliminates `x1*x2*x3` and is the same as `x1 x2 x1*x2 x3 x1*x3 x2*x3`). The specification "`@2`" creates main effects and two-way interactions. Only "`@`" values of `2` or `3` are permitted. If you specify "@2" by itself, a resolution V design is requested when PROC FACTEX is run.

iter= *n*
maxiter= *n*

specifies the PROC OPTEX `iter=` option which creates *n* designs. By default, `iter=10`.

keep= *n*

specifies the PROC OPTEX `keep=` option which keeps the *n* best designs. By default, `keep=5`.

nlev= *n*

specifies the number of levels from which factors are constructed through pseudo-factors and coding down. The value must be a prime or a power of a prime: 2, 3, 4, 5, 7, 8, 9, 11 .... This option is used with PROC FACTEX as follows:

```
factors factors / nlev=&nlev;
```

By default, the macro uses the minimum prime or power of a prime from the `factors=` list or 2 if no suitable value is found.

method= *name*

specifies the PROC OPTEX `method=` search method option. The default is `method=m_Fedorov` (modified Fedorov).

n= *n* | `saturated`

specifies the PROC OPTEX `n=` option, which is the number of runs in the final design. The default is the PROC OPTEX default and depends on the problem. Typically, you will not want to use the default. Instead, you should pick a value using the information produced by the `%MktRuns` macro as guidance (see page 1159). The `n=saturated` option creates a design with the minimum number of runs.

options= *options-list*

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

`check`
checks the efficiency of a given design, specified in `cand=`.

`nocode`
suppresses the display of the PROC PLAN, PROC FACTEX, and PROC OPTEX code.

`allcode`
shows all code, even code that will not be run.

otherfac= *variable-list*

specifies other terms to mention in the `factors` statement of PROC FACTEX. These terms are not guaranteed to be estimable. By default, there are no other factors.

otherint= *terms*

specifies interaction terms that will only be specified with PROC OPTEX for multi-step macro invocations. By default, no interactions are guaranteed to be estimable. Normally, interactions that are specified via the `interact=` option affect both the PROC FACTEX and the PROC OPTEX `model` statements. In multi-step problems, part of an interaction might not be in a particular PROC FACTEX step. In that case, the interaction term must only appear in the PROC OPTEX step. For example, if `x1` is created in one step and `x4` is created in another, and if the `x1*x4` interaction must be estimable, specify `otherint=x1*x4` on the final step, the one that runs PROC OPTEX. The following steps create the design:

```
%mktdes(step=1, factors=x1-x3=2, n=30, run=factex)

%mktdes(step=2, factors=x4-x6=3, n=30, run=factex)

%mktdes(step=3, factors=x7-x9=5, n=30, run=factex optex,
        otherint=x1*x4)
```

out= *SAS-data-set*
specifies the output experimental design (from PROC OPTEX). By default, `out=design`. 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.

procopts= *options*
specifies PROC OPTEX statement options. By default, no options are added to the PROC OPTEX statement.

run= *procedure-list*
specifies the list of procedures that the macro can run. Normally, the macro runs either PROC FACTEX or PROC PLAN and then PROC OPTEX. By default, `run=plan factex optex`. You can skip steps by omitting procedure names from this list. When both PLAN and FACTEX are in the list, the macro chooses between them based on the size of the full-factorial design and the value of `big=`. When PLAN is not in the list, the macro generates code for PROC FACTEX.

seed= *n*
specifies the random number seed. By default, `seed=0`, and clock time is used to make 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.

size= *n* | `min`
specifies the candidate-set size. Start with the default `size=min` and see how big that design is. If you want, subsequently you can specify larger values that are `nlev=`*n* multiples of the minimum size. This option is used with PROC FACTEX as follows:

```
size design=&size;
```

When `nlev=`*n*, increase the `size=` value by a factor of *n* each time. For example, when `nlev=2`, increase the `size=` value by a factor of two each time. If `size=min` implies `size=128`, then 256, 512, 1024, and 2048 are reasonable sizes to try. Integer expressions like `size=128*4` are permitted.

step= *n*

specifies the step number. By default, there is only one step. However, sometimes, a better design can be found using a multi-step approach. Do not specify the `cand=` option in any step of a multi-step run. Consider the problem of making a design with 3 two-level factors, 3 three-level factors, and 3 five-level factors. The simplest approach is to do something like the following—create a design from two-level factors using pseudo-factors and coding down:

```
%mktdes(factors=x1-x3=2 x4-x6=3 x7-x9=5, n=30)
```

However, for small problems like this, the following three-step approach is usually better:

```
%mktdes(step=1, factors=x1-x3=2, n=30, run=factex)
%mktdes(step=2, factors=x4-x6=3, n=30, run=factex)
%mktdes(step=3, factors=x7-x9=5, n=30, run=factex optex)
```

Note, however, that the following `%MktEx` macro step will usually be better still:

```
%mktex(2 2 2 3 3 3 5 5 5, n=30)
```

The first `%MktDes` macro step uses PROC FACTEX to create a fractional-factorial design for the two-level factors. The second step uses PROC FACTEX to create a fractional-factorial design for the three-level factors and cross it with the two-level factors. The third step uses PROC FACTEX to create a fractional-factorial design for the five-level factors and cross it with the design for the two and three-level factors and then run PROC OPTEX.

Each step globally stores two macro variables (`&class1` and `&inter1` for the first step, `&class2` and `&inter2` for the second step, ...) that are used to construct the PROC OPTEX `class` and `model` statements. When `step` $>$ 1, variables from the previous steps are used in the `class` and `model` statements. In this example, the following PROC OPTEX code is created by step 3:

```
proc optex data=Cand3;
   class
      x1-x3
      x4-x6
      x7-x9
      / param=orthref;
   model
      x1-x3
      x4-x6
      x7-x9
      ;
   generate n=30 iter=10 keep=5 method=m_fedorov;
   output out=Design;
   run; quit;
```

This step uses the previously stored macro variables `&class1=x1-x3` and `&class2=x4-x6`.

where= *where-clause*
specifies a SAS `where` clause for the candidate design, which is used to restrict the candidates. By default, the candidate design is not restricted.

%MktDes Macro Notes

This macro specifies `options nonotes` throughout much 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.

%MktDups Macro

The `%MktDups` autocall macro detects duplicate choice sets and duplicate alternatives within generic choice sets. See the following pages for examples of using this macro in the design chapter: 147, 174, 198, and 206. Also see the following pages for examples of using this macro in the discrete choice chapter: 319, 368, 519, 564, 564, 567, 570, 576, 597, 607, 617, 628, 636, 645, 650, 654, 656, 659 and 662. Additional examples appear throughout this chapter. To illustrate, consider a simple experiment with these two choice sets. These choice sets are completely different and are not duplicates.

```
a b c       a b c
1 2 1       1 1 1
2 1 2       2 2 2
1 1 2       2 2 1
2 1 1       1 2 2
```

Now consider these two choice sets:

```
a b c       a b c
1 2 1       2 1 2
2 1 2       1 1 2
1 1 2       2 1 1
2 1 1       1 2 1
```

They are the same for a generic study because all of the same alternatives are there, they are just in a different order. However, for a branded study they are different. For a branded study, there would be a different brand for each alternative, so the choice sets would be the same only if all the same alternatives appeared in the same order. For both a branded and generic study, these choice sets are duplicates:

```
a b c       a b c
1 2 1       1 2 1
2 1 2       2 1 2
1 1 2       1 1 2
2 1 1       2 1 1
```

Now consider these choice sets for a generic study.

```
a b c       a b c
1 2 1       1 2 1
2 1 1       1 2 1
1 1 2       1 1 2
2 1 1       2 1 1
```

First, each of these choice sets has duplicate alternatives (2 1 1 in the first and 1 2 1 in the second). Second, these two choice sets are flagged as duplicates, even though they are not exactly the same. They are flagged as duplicates because every alternative in choice set one is also in choice set two, and every alternative in choice set two is also in choice set one. In generic studies, two choice sets are considered duplicates unless one has one or more alternatives that are not in the other choice set.

As an example, a design is created with the `%ChoicEff` macro choice-set-swapping algorithm for a branded study, then the `%MktDups` macro is run to check for and eliminate duplicate choice sets. The following steps create and evaluate the design:

```
%mktex(3 ** 9, n=27, seed=424)

data key;
   input (Brand x1-x3) ($);
   datalines;
Acme   x1 x2 x3
Ajax   x4 x5 x6
Widgit x7 x8 x9
;

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

%choiceff(data=cand,                /* candidate set of choice sets         */
          model=class(brand x1-x3 / sta),/* model with stdz orthog coding   */
          seed=420,                 /* random number seed                   */
          nsets=18,                 /* number of choice sets                */
          nalts=3,                  /* number of alternatives               */
          options=relative,         /* display relative D-efficiency        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */

proc freq; tables set; run;

%mktdups(branded, data=best, factors=brand x1-x3, nalts=3, out=out)

proc freq; tables set; run;
```

The following PROC FREQ results show that candidate choice sets occur more than once in the design:

```
                              The FREQ Procedure

                                                  Cumulative    Cumulative
                Set    Frequency     Percent     Frequency      Percent
                --------------------------------------------------------
                  1           3        5.56             3         5.56
                  2           6       11.11             9        16.67
                  3           3        5.56            12        22.22
                  5           6       11.11            18        33.33
                  8           3        5.56            21        38.89
                  9           3        5.56            24        44.44
                 12           3        5.56            27        50.00
                 13           3        5.56            30        55.56
                 14           6       11.11            36        66.67
```

```
              16             6       11.11             42        77.78
              19             3        5.56             45        83.33
              20             3        5.56             48        88.89
              24             3        5.56             51        94.44
              27             3        5.56             54       100.00
```

The %MktDups macro displays the following information in the log:

```
Design:           Branded
Factors:          brand x1-x3
                  Brand
                  x1 x2 x3
Duplicate Sets:   4
```

The output from the %MktDups macro contains the following table:

```
                                      Duplicate
                        Choice      Choice Sets
                           Set        To Delete

                             1               15

                             3               18

                             7               14

                            10               17
```

The first line of the first table tells us that this is a branded design as opposed to generic. The second line tells us the factors as specified in the `factors=` option. These are followed by the actual variable names for the factors. The last line reports the number of duplicates. The second table tells us that choice set 1 is the same as choice set 15. Similarly, 3 and 18 are the same, and so on. The `out=` data set will contain the design with the duplicate choice set eliminated.

Now consider an example with purely generic alternatives. The following steps create and evaluate the design:

```
%mktex(2 ** 5, n=2**5, seed=109)

%choiceff(data=randomized,          /* candidate set of alternatives       */
          model=class(x1-x5 / sta), /* model with stdz orthogonal coding   */
          seed=93,                  /* random number seed                  */
          nsets=42,                 /* number of choice sets               */
          flags=4,                  /* 4 alternatives, generic candidates  */
          options=relative,         /* display relative D-efficiency       */
          beta=zero)                /* assumed beta vector, Ho: b=0        */

%mktdups(generic, data=best, factors=x1-x5, nalts=4, out=out)
```

The macro produces the following tables:

```
Design:          Generic
Factors:         x1-x5
                 x1 x2 x3 x4 x5
Sets w Dup Alts: 1
Duplicate Sets:  1
```

```
                    Duplicate
         Choice    Choice Sets
            Set      To Delete

              2             25

             39                        Alternatives
```

For each choice set listed in the choice set column, either the other choice sets it duplicates are listed or the word `Alternatives` is displayed if the problem is with duplicate alternatives.

The following step displays just the choice sets with duplication problems:

```
proc print data=best;
   var x1-x5;
   id set; by set;
   where set in (2, 25, 39);
   run;
```

The results are as follows:

```
Set    x1    x2    x3    x4    x5

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

 25     1     1     2     2     2
        2     1     2     2     2
        2     2     1     1     1
        1     2     1     1     1

 39     1     1     2     1     1
        1     1     2     1     1
        2     2     1     2     2
        2     2     1     2     2
```

You can see that the macro detects duplicates even though the alternatives do not always appear in the same order in the different choice sets.

Now consider another example. The following steps create and evaluate a choice design:

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

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

%mktdups(generic, data=cand, factors=x1-x2, nalts=3, out=out)

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

Some of the results are as follows:

```
Design:          Generic
Factors:         x1-x2
                 x1 x2
Sets w Dup Alts: 40
Duplicate Sets:  50
```

```
                    Duplicate
         Choice    Choice Sets
            Set      To Delete

              1                     Alternatives

              2                     Alternatives
                             5
                             6
                            17
                            18
                            21
              .
              .
              .
```

The output lists, for each set of duplicates, the choice set that is kept (in the first column) and all the matching choice sets that are deleted (in the second column).

The unique choice sets are as follows:

```
         Set    _Alt_    x1    x2

           7      1       1     1
                  2       1     2
                  3       2     1

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

```
                                   12       1        1     1
                                            2        2     1
                                            3        2     2

                                   28       1        1     2
                                            2        2     1
                                            3        2     2
```

The following example creates a conjoint design[*] and tests it for duplicates:

```
   %mktex(3 ** 3 2 ** 2, n=19, seed=513)

   %mktdups(linear, data=design, factors=x1-x5)
```

The results are as follows:

```
Design:          Linear
Factors:         x1-x5
                 x1 x2 x3 x4 x5
Duplicate Runs:  2
```

```
                                             Duplicate
                                                  Runs
                                     Run     To Delete


                                      12            13

                                      16            17
```

%MktDups Macro Options

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

Option	Description
`options`	(positional) binary options
	(positional) “help” or “?” displays syntax summary
`data=`*SAS-data-set*	input choice design
`factors=`*variable-list*	factors in the design
`nalts=`*n*	number of alternatives
`out=`*SAS-data-set*	output data set
`outlist=`*SAS-data-set*	output data set with duplicates
`vars=`*variable-list*	factors in the design

[*]Normally, we would use 18 runs and not a prime number like 19 that is not divisible by any of the numbers of levels, 2 and 3. We picked a silly number like 19 to ensure duplicates for this example.

Help Option

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

```
%mktdups(help)
%mktdups(?)
```

Positional Parameter

The options list is a positional parameter. This means it must come first, and unlike all other parameters, it is not specified after a name and an equal sign.

options

specifies positional options. You can specify `noprint` and one of the following: `generic`, `branded`, or `linear`.

`branded`
specifies that since one of the factors is brand, the macro only needs to compare corresponding alternatives in each choice set.

`generic`
specifies a generic design and is the default. This means that there are no brands, so options are interchangeable, so the macro needs to compare each alternative with every other alternative in every choice set.

`linear`
specifies a linear not a choice design. Specify `linear` for a full-profile conjoint design, for an ANOVA design, or for the linear version of a branded choice design.

`noprint`
suppresses the output display. This option is used when you are only interested in the output data set or macro variable.

The following step shows an example of the `noprint` option:

```
%mktdups(branded noprint, data=design, nalts=3)
```

Required Options

This next option is mandatory with choice designs.

nalts= *n*

specifies the number of alternatives. This option must be specified with generic or branded choice designs. It is ignored with linear model designs. For generic or branded designs, the `data=` data set must contain `nalts=` observations for the first choice set, `nalts=` observations for the second choice set, and so on.

Other Options

The other options are as follows:

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

out= *SAS-data-set*
specifies an output data set that contains the design with duplicate choice sets excluded. By default, no data set is created, and the macro just reports on duplicates. 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.

outlist= *SAS-data-set*
specifies the output data set with the list of duplicates. By default, `outlist=outdups`.

vars= *variable-list*
factors= *variable-list*
specifies the factors in the design. By default, all numeric variables are used.

%MktDups 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.

%MktEval Macro

The %MktEval autocall macro evaluates an experimental design for a linear model. The %MktEval macro reports on balance and orthogonality. Typically, you will call it immediately after running the %MktEx macro. You do not call it after making a choice design by using the %ChoicEff macro. The descriptive statistics that the %MktEval macro produces are appropriate for linear models not choice models. However, you can reasonably call it with the linear arrangement that will later be transformed into a choice design, for example, with the %MktRoll macro. See page 130 for an example of using this macro in the design chapter. Also see the following pages for examples of using this macro in the discrete choice chapter: 306, 308, 349, 353, 413, 423, 480, 485, 489, 491, 493, 538, 588 and 591. Additional examples appear throughout this chapter.

The output from this macro contains two default tables. The first table shows the canonical correlations between pairs of coded factors. A canonical correlation is the maximum correlation between linear combinations of the coded factors. See page 101 for more information about canonical correlations. All zeros off the diagonal show that the design is orthogonal for main effects. Off-diagonal canonical correlations greater than 0.316 ($r^2 > 0.1$) are listed in a separate table.

For nonorthogonal designs and designs with interactions, the canonical-correlation matrix is not a substitute for looking at the variance matrix with the %MktEx macro. It just provides a quick and more-compact picture of the correlations between the factors. The variance matrix is sensitive to the actual model specified and the coding. The canonical-correlation matrix just tells you if there is some correlation between the main effects. When is a canonical correlation too big? You will have to decide that for yourself. In part, the answer depends on the factors and how the design will be used. A high correlation between the client's and the main competitor's price factor is a serious problem meaning you will need to use a different design. In contrast, a moderate correlation in a choice design between one brand's minor attribute and another brand's minor attribute might be perfectly fine.

The macro also displays one-way, two-way and n-way frequencies. Equal one-way frequencies occur when the design is balanced. Equal two-way frequencies occur when the design is orthogonal. Equal n-way frequencies, all equal to one, occur when there are no duplicate runs or choice sets.

The following steps create and evaluate a design:

```
%mktex(2 2 3 ** 6,                  /* 2 two-level and 6 three-level factors*/
       n=18,                        /* 18 runs                              */
       balance=0,                   /* require perfect balance in the end   */
       mintry=5*18,                 /* but imbalance OK for first 5 passes  */
       seed=289)                    /* random number seed                   */

%mkteval(data=randomized)
```

The results are as follows:

```
                 Canonical Correlations Between the Factors
            There is 1 Canonical Correlation Greater Than 0.316

          x1      x2      x3      x4      x5      x6      x7      x8

    x1    1       0.33    0       0       0       0       0       0
    x2    0.33    1       0       0       0       0       0       0
    x3    0       0       1       0       0       0       0       0
    x4    0       0       0       1       0       0       0       0
    x5    0       0       0       0       1       0       0       0
    x6    0       0       0       0       0       1       0       0
    x7    0       0       0       0       0       0       1       0
    x8    0       0       0       0       0       0       0       1

             Canonical Correlations > 0.316 Between the Factors
            There is 1 Canonical Correlation Greater Than 0.316


                                    r    r Square

                  x1    x2    0.33      0.11

                       Summary of Frequencies
            There is 1 Canonical Correlation Greater Than 0.316
                     * - Indicates Unequal Frequencies

                          Frequencies

                 x1       9 9
                 x2       9 9
                 x3       6 6 6
                 x4       6 6 6
                 x5       6 6 6
                 x6       6 6 6
                 x7       6 6 6
                 x8       6 6 6
            *    x1 x2    3 6 6 3
                 x1 x3    3 3 3 3 3 3
                 x1 x4    3 3 3 3 3 3
                 x1 x5    3 3 3 3 3 3
                 x1 x6    3 3 3 3 3 3
                 x1 x7    3 3 3 3 3 3
                 x1 x8    3 3 3 3 3 3
                 x2 x3    3 3 3 3 3 3
                 x2 x4    3 3 3 3 3 3
                 x2 x5    3 3 3 3 3 3
                 x2 x6    3 3 3 3 3 3
                 x2 x7    3 3 3 3 3 3
                 x2 x8    3 3 3 3 3 3
```

```
x3 x4    2 2 2 2 2 2 2 2 2
x3 x5    2 2 2 2 2 2 2 2 2
x3 x6    2 2 2 2 2 2 2 2 2
x3 x7    2 2 2 2 2 2 2 2 2
x3 x8    2 2 2 2 2 2 2 2 2
x4 x5    2 2 2 2 2 2 2 2 2
x4 x6    2 2 2 2 2 2 2 2 2
x4 x7    2 2 2 2 2 2 2 2 2
x4 x8    2 2 2 2 2 2 2 2 2
x5 x6    2 2 2 2 2 2 2 2 2
x5 x7    2 2 2 2 2 2 2 2 2
x5 x8    2 2 2 2 2 2 2 2 2
x6 x7    2 2 2 2 2 2 2 2 2
x6 x8    2 2 2 2 2 2 2 2 2
x7 x8    2 2 2 2 2 2 2 2 2
N-Way    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

All factors in this design are perfectly balanced, and almost all are orthogonal, but `x1` and `x2` are correlated with each other.

%MktEval Macro Options

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

Option	Description
`help`	(positional) "help" or "?" displays syntax summary
`blocks=`*variable*	blocking variable
`data=`*SAS-data-set*	input data set with design
`factors=`*variable-list*	factors in the design
`format=`*format*	format for canonical correlations
`freqs=`*frequency-list*	frequencies to display
`list=`*n*	minimum canonical correlation to list
`outcb=`*SAS-data-set*	within-block canonical correlations
`outcorr=`*SAS-data-set*	canonical correlation matrix
`outfreq=`*SAS-data-set*	frequencies
`outfsum=`*SAS-data-set*	frequency summaries
`outlist=`*SAS-data-set*	list of largest canonical correlations
`print=`*print-options*	controls the display of the results
`vars=`*variable-list*	factors in the design

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

```
%mkteval(help)
%mkteval(?)
```

blocks= *variable*
specifies a blocking variable. This option displays separate canonical correlations within each block. By default, there is one block.

data= *SAS-data-set*
specifies the input SAS data set with the experimental design. By default, the macro uses the last data set created.

factors= *variable-list*
vars= *variable-list*
specifies a list of the factors in the experimental design. The default is all of the numeric variables in the data set.

freqs= *frequency-list*
specifies the frequencies to display. By default, `freqs=1 2 n`, and 1-way, 2-way, and n-way frequencies are displayed. Do not specify the exact number of ways instead of `n`. For ways other than `n`, the macro checks for and displays zero cell frequencies. For n-ways, the macro does not output or display zero frequencies. Only the full-factorial design will have nonzero cells, so specifying something like `freqs=1 2 20` will make the macro take a *long* time, and it will try to create *huge* data sets and will probably run out of memory or disk space before it is done. However, `freqs=1 2 n` runs very reasonably.

format= *format*
specifies the format for displaying the canonical correlations. The default format is `4.2`.

list= *n*
specifies the minimum canonical correlation to list. The default is 0.316, the square root of $r^2 = 0.1$.

outcorr= *SAS-data-set*
specifies the output SAS data set for the canonical correlation matrix. The default data set name is CORR.

outcb= *SAS-data-set*
specifies the output SAS data set for the within-block canonical correlation matrices. The default data set name is CB.

outlist= *SAS-data-set*
specifies the output data set for the list of largest canonical correlations. The default data set name is LIST.

outfreq= *SAS-data-set*
specifies the output data set for the frequencies. The default data set name is FREQ.

outfsum= *SAS-data-set*

specifies the output data set for the frequency summaries. The default data set name is FSUM.

print= *print-options*

controls the display of the results. The default is `print=short`. Specify one or more values from the following list.

`all`	all output is displayed
`corr`	displays the canonical correlations matrix
`block`	displays the canonical correlations within block
`freqs`	displays the frequencies, specified by the `freqs=` option
`list`	displays the list of canonical correlations greater than the `list=` value
`nonzero`	like `ordered` but sets `list=1e-6`
`ordered`	like `list` but ordered by variable names
`short`	is the default and is equivalent to: `corr list summ block`
`summ`	displays the frequency summaries
`noprint`	no output is displayed

By default, the frequency list, which contains the factor names, levels, and frequencies is not displayed, but the more compact frequency summary list, which contains the factors and frequencies but not the levels is displayed.

%MktEval 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.

%MktEx Macro

The `%MktEx` autocall macro creates efficient factorial designs. The `%MktEx` macro is designed to be very simple to use and to run in seconds for trivial problems, minutes for small problems, and in less than an hour for larger and difficult problems. This macro is a full-featured factorial-experimental designer that can handle simple problems like main-effects designs and more complicated problems including designs with interactions and restrictions on which levels can appear together. The macro is designed to easily create the kinds of designs that marketing researchers need for conjoint and choice experiments and that other researchers need for other types of problems. For most factorial-design problems, you can simply run the macro once, specifying only the number of runs and the numbers of levels of all the factors. You no longer have to try different algorithms and different approaches to see which one works best. The macro does all of that for you. We state on page 244 "The best approach to design creation is to use the computer as a tool along with traditional design skills, not as a substitute for thinking about the problem." With the `%MktEx` macro, we try to automate some of the thought processes of the expert designer.

See the following pages for examples of using this macro in the design chapter: 81, 85, 98, 109, 112, 129, 166, 190 and 200. Also see the examples of using this macro in the discrete choice chapter from pages 285 through 663. Additional examples appear throughout this chapter.

The following example uses the `%MktEx` macro to create a design with 5 two-level factors, 4 three-level factors, 3 five-level factors, 2 six-level factors, all in 60 runs (row, experimental conditions, conjoint profiles, or choice sets):

```
%mktex(2 ** 5  3 ** 4  5 5 5  6 6, n=60)
```

The notation `m ** n` means m^n or n m-level factors. For example `2 ** 5` means $2 \times 2 \times 2 \times 2 \times 2$ or 5 two-level factors.

The `%MktEx` macro creates efficient factorial designs using several approaches. The macro will try to directly create an orthogonal design (strength-two orthogonal array), it will search a set of candidate runs (rows of the design), and it will use a coordinate-exchange algorithm using both random initial designs and also a partially orthogonal design initialization. The macro stops if at any time it finds a perfect, 100% efficient, orthogonal and balanced design. This first phase is the algorithm search phase. In it, the macro determines which approach is working best for this problem. At the end of this phase, the macro chooses the method that has produced the best design and performs another set of iterations using exclusively the chosen approach. Finally, the macro performs a third set of iterations where it takes the best design it found so far and tries to improve it.

In all phases, the macro attempts to optimize D-efficiency (sometimes known as D-optimality), which is a standard measure of the goodness of the experimental design. As D-efficiency increases, the standard errors of the parameter estimates in the linear model decrease. A perfect design is orthogonal and balanced and has 100% D-efficiency. A design is orthogonal when all of the parameter estimates are uncorrelated. A design is balanced when all of the levels within each of the factors occur equally often. A design is orthogonal and balanced when the variance matrix, which is proportional to $(\mathbf{X}'\mathbf{X})^{-1}$, is diagonal, where $\mathbf{X}$ is a suitable orthogonal coding (see page 73) of the design matrix. See pages 53 and 243, for more information about efficient experimental designs.

For most problems, you only need to specify the levels of all the factors and the number of runs. For more complicated problems, you might need to also specify the interactions that you want to estimate or restrictions on which levels may not appear together. Other than that, you should not need any other options for most problems. This macro is not like other design tools that you have to tell what to do. With this macro, you just tell it what you want, and it figures out a good way to do it. For some problems, the sophisticated user, with a lot of work, might be able to adjust the options to come up with a better design. However, this macro should always produce a very good design with minimal effort for even the most unsophisticated users.

Orthogonal Arrays

The `%MktEx` macro has the world's largest catalog of strength-two (main effects) orthogonal arrays. In an orthogonal array, all estimable effects are uncorrelated. The orthogonal arrays are constructed using methods and arrays from a variety of sources, including: Addelman (1962a,b); Bose (1947); Colbourn and de Launey (1996); Dawson (1985); De Cock and Stufken (2000); de Launey (1986, 1987a,b); Dey (1985); Ehrlich (1964); Elliott and Butson (1966); Hadamard (1893); Hedayat, Sloane, and Stufken (1999); Hedayat and Wallis (1978); Kharaghania and Tayfeh-Rezaiea (2004); Kuhfeld (2005); Kuhfeld and Suen (2005); Kuhfeld and Tobias (2005); Nguyen (2005); Nguyen (2006); Nguyen (2006); Paley (1933); Pang, Zhang, and Liu (2004a); Pang and Zhang (2004b); Rao (1947); Seberry and Yamada (1992); Sloane (2004); Spence (1975a); Spence (1975b); Spence (1977a); Spence (1977b); Suen (1989a,b, 2003a,b,c); Suen and Kuhfeld (2005); Taguchi (1987); Turyn (1972); Turyn (1974); Wang (1996a,b); Wang and Wu (1989, 1991); Williamson(1944); Xu (2002); Yamada (1986); Yamada (1989); Zhang (2004–2006); Zhang, Duan, Lu, and Zheng (2002); Zhang, Lu and Pang(1999); Zhang, Pang and Wang (2001); Zhang, Weiguo, Meixia and Zheng (2004); and the SAS FACTEX procedure.

For all n's up through 448 that are a multiple of 4, and many n's beyond that, the `%MktEx` macro can construct orthogonal designs with up to $n-1$ two-level factors. The two-level designs are constructed from Hadamard matrices (Hadamard 1893; Paley 1933; Williamson 1944; Hedayat, Sloane, and Stufken 1999). The `%MktEx` macro can construct these designs when n is a multiple of 4 and one or more of the following hold:

- $n \le 448$ or $n = 580$, 596, 604, 612, 724, 732, 756, or 1060
- $n-1$ is prime
- $n/2-1$ is a prime power and $\text{mod}(n/2, 4) = 2$
- n is a power of 2 (2, 4, 8, 16, ...) times the size of a smaller Hadamard matrix that is available.

When n is a multiple of 8, the macro can create orthogonal designs with a small number (say m) four-level factors in place of $3 \times m$ of the two-level factors (for example, $2^{70}\ 4^{3}$ in 80 runs).

You can see the Hadamard matrix sizes that `%MktEx` has in its catalog by running the following steps:

```
%mktorth(maxlev=2)

proc print; run;
```

Alternatively, you can see more details by instead running the following steps:

```
data x;
   length Method $ 30;
   do n = 4 to 1000 by 4;
      HadSize = n; method = ' ';
      do while(mod(hadsize, 8) eq 0); hadsize = hadsize / 2; end;
      link paley;
      if method eq ' ' and hadsize le 256 and not (hadsize in (188, 236))
         then method = 'Williamson';
      else if hadsize in (188, 236, 260, 268, 292, 356, 404, 436, 596)
         then method = 'Turyn, Hedayat, Wallis';
      else if hadsize = 324                    then method = 'Ehlich';
      else if hadsize in (372, 612, 732, 756)  then method = 'Turyn';
      else if hadsize in (340, 580, 724, 1060) then method = 'Paley 2';
      else if hadsize in (412, 604)            then method = 'Yamada';
      else if hadsize = 428 then method = 'Kharaghania and Tayfeh-Rezaiea';
      if method = ' ' then do;
         do while(hadsize lt n and method eq ' ');
            hadsize = hadsize * 2;
            link paley;
            end;
         end;
      if method ne ' ' then do; Change = n - lag(n); output; end;
      end;
   return;
   paley:;
      ispm1 = 1; ispm2 = mod(hadsize / 2, 4) eq 2;
      h = hadsize - 1;
      do i = 3 to sqrt(hadsize) by 2 while(ispm1);
         ispm1 = mod(h, i);
         end;
      h = hadsize / 2 - 1;
      do i = 3 to sqrt(hadsize / 2) by 2 while(ispm2);
         ispm2 = mod(h, i);
         end;
      if      ispm1 then method = 'Paley 1';
      else if ispm2 then method = 'Paley 2';
      return;
   run;

options ps=2100;
proc print label noobs;
   label hadsize = 'Reduced Hadamard Matrix Size';
   var n hadsize method change;
   run;
```

Note, however, that the fact that a number appears in this program's listing, does not guarantee that your computer will have enough memory and other resources to create it.

The following table provides a summary of the parent and design sizes up through 513 runs that are available with the `%MktEx` macro:

Number of Runs	Parents		All Designs	
4– 50	87	11.87%	181	0.15%
51–100	185	25.24%	611	0.52%
101–127 129–143 145–150	152	20.74%	287	0.24%
128	21	2.86%	740	0.63%
144	16	2.18%	1,241	1.06%
151–200	43	5.87%	1,073	0.91%
201–250	41	5.59%	1,086	0.92%
251–255 257–300	35	4.77%	3,451	2.94%
256	2	0.27%	6,101	5.19%
301–350	37	5.05%	2,295	1.95%
351–400	41	5.59%	4,734	4.03%
401–431 433–447 449–450	24	3.27%	425	0.36%
432	7	0.95%	10,839	9.22%
448	4	0.55%	8,598	7.31%
451–500	32	4.37%	1,789	1.52%
501–511 513	4	0.55%	113	0.10%
512	2	0.27%	73,992	62.96%
	733		117,556	

Designs with 128, 144, 256, 432, 448, and 512 runs are listed separately from the rest of their category since there are so many of them.

Not included in this list are 2296 additional designs that are explicitly in the catalog but have more than 513 runs. Most are constructed from the parent array 24^8 in 576 runs (which is useful for making Latin Square designs). The rest are constructed from Hadamard matrices.

The 733 parent designs are displayed following this paragraph. (Listing all of the 117,556 designs at 200 designs per page, would require 588 pages.) Many more orthogonal arrays that are not explicitly in this catalog can also be created. These include full-factorial designs with more than 144 runs, Hadamard designs with more than 1000 runs, and fractional-factorial designs in 256, 512, or more runs.

Parents	Designs	Runs	
1	1	**4**	2^{3}
1	1	**6**	$2^{1}3^{1}$
1	2	**8**	$2^{4}4^{1}$
1	1	**9**	3^{4}
1	1	**10**	$2^{1}5^{1}$
4	4	**12**	2^{11}
			$2^{4}3^{1}$
			$2^{2}6^{1}$
			$3^{1}4^{1}$
1	1	**14**	$2^{1}7^{1}$
1	1	**15**	$3^{1}5^{1}$
2	7	**16**	$2^{8}8^{1}$
			4^{5}
2	3	**18**	$2^{1}9^{1}$
			$3^{6}6^{1}$
4	4	**20**	2^{19}
			$2^{8}5^{1}$
			$2^{2}10^{1}$
			$4^{1}5^{1}$
1	1	**21**	$3^{1}7^{1}$
1	1	**22**	$2^{1}11^{1}$
5	8	**24**	$2^{20}4^{1}$
			$2^{13}3^{1}4^{1}$
			$2^{12}12^{1}$
			$2^{11}4^{1}6^{1}$
			$3^{1}8^{1}$
1	1	**25**	5^{6}
1	1	**26**	$2^{1}13^{1}$
1	2	**27**	$3^{9}9^{1}$
4	4	**28**	2^{27}
			$2^{12}7^{1}$
			$2^{2}14^{1}$
			$4^{1}7^{1}$
3	4	**30**	$2^{1}15^{1}$
			$3^{1}10^{1}$
			$5^{1}6^{1}$
2	20	**32**	$2^{16}16^{1}$
			$4^{8}8^{1}$
1	1	**33**	$3^{1}11^{1}$
1	1	**34**	$2^{1}17^{1}$
1	1	**35**	$5^{1}7^{1}$
20	26	**36**	2^{35}
			$2^{27}3^{1}$
			$2^{20}3^{2}$
			$2^{18}3^{1}6^{1}$
			$2^{16}9^{1}$
			$2^{13}3^{2}6^{1}$
			$2^{13}6^{2}$
			$2^{10}3^{8}6^{1}$
			$2^{10}3^{1}6^{2}$
			$2^{9}3^{4}6^{2}$
			$2^{8}6^{3}$
			$2^{4}3^{1}6^{3}$
			$2^{3}3^{9}6^{1}$
			$2^{3}3^{2}6^{3}$
			$2^{2}3^{5}6^{2}$
			$2^{2}18^{1}$
			$2^{1}3^{3}6^{3}$
			$3^{12}12^{1}$
			$3^{7}6^{3}$
			$4^{1}9^{1}$
1	1	**38**	$2^{1}19^{1}$
1	1	**39**	$3^{1}13^{1}$
5	8	**40**	$2^{36}4^{1}$
			$2^{25}4^{1}5^{1}$
			$2^{20}20^{1}$
			$2^{19}4^{1}10^{1}$
			$5^{1}8^{1}$
3	4	**42**	$2^{1}21^{1}$
			$3^{1}14^{1}$
			$6^{1}7^{1}$
4	4	**44**	2^{43}
			$2^{15}11^{1}$
			$2^{2}22^{1}$
			$4^{1}11^{1}$
2	3	**45**	$3^{9}15^{1}$
			$5^{1}9^{1}$
1	1	**46**	$2^{1}23^{1}$
6	58	**48**	$2^{40}8^{1}$
			$2^{33}3^{1}8^{1}$
			$2^{31}6^{1}8^{1}$
			$2^{24}24^{1}$
			$3^{1}16^{1}$
			$4^{12}12^{1}$
1	1	**49**	7^{8}
2	3	**50**	$2^{1}25^{1}$
			$5^{10}10^{1}$
1	1	**51**	$3^{1}17^{1}$
4	4	**52**	2^{51}
			$2^{16}13^{1}$
			$2^{2}26^{1}$
			$4^{1}13^{1}$
3	6	**54**	$2^{1}27^{1}$
			$3^{20}6^{1}9^{1}$
			$3^{18}18^{1}$
1	1	**55**	$5^{1}11^{1}$
5	8	**56**	$2^{52}4^{1}$
			$2^{37}4^{1}7^{1}$
			$2^{28}28^{1}$
			$2^{27}4^{1}14^{1}$
			$7^{1}8^{1}$
1	1	**57**	$3^{1}19^{1}$
1	1	**58**	$2^{1}29^{1}$
11	15	**60**	2^{59}
			$2^{30}3^{1}$
			$2^{24}6^{1}$
			$2^{23}5^{1}$
			$2^{21}10^{1}$
			$2^{17}15^{1}$
			$2^{15}6^{1}10^{1}$
			$2^{2}30^{1}$
			$3^{1}20^{1}$
			$4^{1}15^{1}$
			$5^{1}12^{1}$
1	1	**62**	$2^{1}31^{1}$
2	3	**63**	$3^{12}21^{1}$
			$7^{1}9^{1}$
7	123	**64**	$2^{32}32^{1}$
			$2^{5}4^{17}8^{1}$
			$2^{5}4^{10}8^{4}$
			$4^{16}16^{1}$
			$4^{14}8^{3}$
			$4^{7}8^{6}$
			8^{9}
1	1	**65**	$5^{1}13^{1}$
3	4	**66**	$2^{1}33^{1}$
			$3^{1}22^{1}$
			$6^{1}11^{1}$
4	4	**68**	2^{67}
			$2^{18}17^{1}$
			$2^{2}34^{1}$
			$4^{1}17^{1}$
1	1	**69**	$3^{1}23^{1}$
3	4	**70**	$2^{1}35^{1}$
			$5^{1}14^{1}$
			$7^{1}10^{1}$

Parents	Designs	Runs	
49	103	**72**	$2^{68}4^{1}$
			$2^{60}3^{1}4^{1}$
			$2^{53}3^{2}4^{1}$
			$2^{51}3^{1}4^{1}6^{1}$
			$2^{49}4^{1}9^{1}$
			$2^{46}3^{2}4^{1}6^{1}$
			$2^{46}4^{1}6^{2}$
			$2^{44}3^{12}4^{1}$
			$2^{43}3^{8}4^{1}6^{1}$
			$2^{43}3^{1}4^{1}6^{2}$
			$2^{42}3^{4}4^{1}6^{2}$
			$2^{41}4^{1}6^{3}$
			$2^{37}3^{13}4^{1}$
			$2^{37}3^{1}4^{1}6^{3}$
			$2^{36}3^{9}4^{1}6^{1}$
			$2^{36}3^{2}4^{1}6^{3}$
			$2^{36}36^{1}$
			$2^{35}3^{12}4^{1}6^{1}$
			$2^{35}3^{5}4^{1}6^{2}$
			$2^{35}4^{1}18^{1}$
			$2^{34}3^{8}4^{1}6^{2}$
			$2^{34}3^{3}4^{1}6^{3}$
			$2^{31}6^{4}$
			$2^{30}3^{1}6^{4}$
			$2^{28}3^{2}6^{4}$
			$2^{27}3^{11}6^{1}12^{1}$
			$2^{27}3^{6}6^{4}$
			$2^{19}3^{20}4^{1}6^{1}$
			$2^{18}3^{16}4^{1}6^{2}$
			$2^{17}3^{12}4^{1}6^{3}$
			$2^{15}3^{7}4^{1}6^{5}$
			$2^{14}3^{3}4^{1}6^{6}$
			$2^{12}3^{21}4^{1}6^{1}$
			$2^{11}3^{20}6^{1}12^{1}$
			$2^{11}3^{17}4^{1}6^{2}$
			$2^{10}3^{20}4^{1}6^{2}$
			$2^{10}3^{16}6^{2}12^{1}$
			$2^{10}3^{13}4^{1}6^{3}$
			$2^{9}3^{16}4^{1}6^{3}$
			$2^{9}3^{12}6^{3}12^{1}$
			$2^{8}3^{12}4^{1}6^{4}$
			$2^{8}3^{8}4^{1}6^{5}$
			$2^{7}3^{7}6^{5}12^{1}$
			$2^{7}3^{4}4^{1}6^{6}$
			$2^{6}3^{7}4^{1}6^{6}$
			$2^{6}3^{3}6^{6}12^{1}$
			$2^{5}3^{3}4^{1}6^{7}$
			$3^{24}24^{1}$
			$8^{1}9^{1}$
1	1	**74**	$2^{1}37^{1}$
2	3	**75**	$3^{1}25^{1}$
			$5^{8}15^{1}$
4	4	**76**	2^{75}
			$2^{19}19^{1}$
			$2^{2}38^{1}$
			$4^{1}19^{1}$
1	1	**77**	$7^{1}11^{1}$
3	4	**78**	$2^{1}39^{1}$
			$3^{1}26^{1}$
			$6^{1}13^{1}$
7	49	**80**	$2^{72}8^{1}$
			$2^{61}5^{1}8^{1}$
			$2^{55}8^{1}10^{1}$
			$2^{51}4^{3}20^{1}$
			$2^{40}40^{1}$
			$4^{10}20^{1}$
			$5^{1}16^{1}$
2	12	**81**	$3^{27}27^{1}$
			9^{10}
1	1	**82**	$2^{1}41^{1}$
12	14	**84**	2^{83}
			$2^{33}3^{1}$
			$2^{28}7^{1}$
			$2^{27}6^{1}$
			$2^{22}6^{1}7^{1}$
			$2^{20}3^{1}14^{1}$
			$2^{20}21^{1}$
			$2^{14}6^{1}14^{1}$
			$2^{2}42^{1}$
			$3^{1}28^{1}$
			$4^{1}21^{1}$
			$7^{1}12^{1}$
1	1	**85**	$5^{1}17^{1}$
1	1	**86**	$2^{1}43^{1}$
1	1	**87**	$3^{1}29^{1}$
5	8	**88**	$2^{84}4^{1}$
			$2^{56}4^{1}11^{1}$
			$2^{44}44^{1}$
			$2^{43}4^{1}22^{1}$
			$8^{1}11^{1}$
5	10	**90**	$2^{1}45^{1}$
			$3^{30}30^{1}$
			$3^{26}6^{1}15^{1}$
			$5^{1}18^{1}$
			$9^{1}10^{1}$
1	1	**91**	$7^{1}13^{1}$
4	4	**92**	2^{91}
			$2^{21}23^{1}$
			$2^{2}46^{1}$
			$4^{1}23^{1}$
1	1	**93**	$3^{1}31^{1}$
1	1	**94**	$2^{1}47^{1}$
1	1	**95**	$5^{1}19^{1}$
14	183	**96**	$2^{80}16^{1}$
			$2^{73}3^{1}16^{1}$
			$2^{71}6^{1}16^{1}$
			$2^{48}48^{1}$
			$2^{44}4^{11}8^{1}12^{1}$
			$2^{43}4^{15}8^{1}$
			$2^{43}4^{12}6^{1}8^{1}$
			$2^{39}3^{1}4^{14}8^{1}$
			$2^{26}4^{23}$
			$2^{19}3^{1}4^{23}$
			$2^{18}4^{22}12^{1}$
			$2^{17}4^{23}6^{1}$
			$2^{12}4^{20}24^{1}$
			$3^{1}32^{1}$
2	3	**98**	$2^{1}49^{1}$
			$7^{14}14^{1}$
2	3	**99**	$3^{13}33^{1}$
			$9^{1}11^{1}$
15	23	**100**	2^{99}
			$2^{51}5^{3}$
			$2^{40}5^{4}$
			$2^{34}5^{3}10^{1}$
			$2^{29}5^{5}$
			$2^{22}25^{1}$
			$2^{18}5^{9}10^{1}$
			$2^{16}5^{3}10^{3}$
			$2^{7}5^{10}10^{1}$
			$2^{5}5^{4}10^{3}$
			$2^{4}10^{4}$
			$2^{2}50^{1}$
			$4^{1}25^{1}$
			$5^{20}20^{1}$
			$5^{8}10^{3}$
3	4	**102**	$2^{1}51^{1}$
			$3^{1}34^{1}$
			$6^{1}17^{1}$
5	8	**104**	$2^{100}4^{1}$
			$2^{65}4^{1}13^{1}$
			$2^{52}52^{1}$
			$2^{51}4^{1}26^{1}$
			$8^{1}13^{1}$

Parents	Designs	Runs	
3	4	**105**	$3^{1}35^{1}$
			$5^{1}21^{1}$
			$7^{1}15^{1}$
1	1	**106**	$2^{1}53^{1}$
39	79	**108**	2^{107}
			$2^{40}6^{1}$
			$2^{34}3^{29}6^{1}$
			$2^{27}3^{33}9^{1}$
			$2^{22}27^{1}$
			$2^{21}3^{1}6^{2}$
			$2^{20}3^{34}9^{1}$
			$2^{18}3^{33}6^{1}9^{1}$
			$2^{18}3^{31}18^{1}$
			$2^{17}3^{29}6^{2}$
			$2^{15}6^{1}18^{1}$
			$2^{13}3^{30}6^{1}18^{1}$
			$2^{13}6^{3}$
			$2^{12}3^{29}6^{3}$
			$2^{10}3^{40}6^{1}9^{1}$
			$2^{10}3^{33}6^{2}9^{1}$
			$2^{10}3^{31}6^{1}18^{1}$
			$2^{9}3^{36}6^{2}9^{1}$
			$2^{9}3^{34}6^{1}18^{1}$
			$2^{8}3^{30}6^{2}18^{1}$
			$2^{4}3^{33}6^{3}9^{1}$
			$2^{4}3^{31}6^{2}18^{1}$
			$2^{3}3^{41}6^{1}9^{1}$
			$2^{3}3^{39}18^{1}$
			$2^{3}3^{34}6^{3}9^{1}$
			$2^{3}3^{32}6^{2}18^{1}$
			$2^{3}3^{16}6^{8}$
			$2^{2}3^{42}18^{1}$
			$2^{2}3^{37}6^{2}9^{1}$
			$2^{2}3^{35}6^{1}18^{1}$
			$2^{2}54^{1}$
			$2^{1}3^{35}6^{3}9^{1}$
			$2^{1}3^{33}6^{2}18^{1}$
			$3^{44}9^{1}12^{1}$
			$3^{39}6^{3}9^{1}$
			$3^{37}6^{2}18^{1}$
			$3^{36}36^{1}$
			$3^{4}6^{11}$
			$4^{1}27^{1}$
3	4	**110**	$2^{1}55^{1}$
			$5^{1}22^{1}$
			$10^{1}11^{1}$
1	1	**111**	$3^{1}37^{1}$
7	57	**112**	$2^{104}8^{1}$
			$2^{89}7^{1}8^{1}$
			$2^{79}8^{1}14^{1}$
			$2^{75}4^{3}28^{1}$
			$2^{56}56^{1}$
			$4^{12}28^{1}$
			$7^{1}16^{1}$
3	4	**114**	$2^{1}57^{1}$
			$3^{1}38^{1}$
			$6^{1}19^{1}$
1	1	**115**	$5^{1}23^{1}$
4	4	**116**	2^{115}
			$2^{23}29^{1}$
			$2^{2}58^{1}$
			$4^{1}29^{1}$
2	3	**117**	$3^{13}39^{1}$
			$9^{1}13^{1}$
1	1	**118**	$2^{1}59^{1}$
1	1	**119**	$7^{1}17^{1}$
16	31	**120**	$2^{116}4^{1}$
			$2^{87}3^{1}4^{1}$
			$2^{79}4^{1}5^{1}$
			$2^{75}4^{1}6^{1}$
			$2^{75}4^{1}10^{1}$
			$2^{74}4^{1}15^{1}$
			$2^{70}3^{1}4^{1}10^{1}$
			$2^{70}4^{1}5^{1}6^{1}$
			$2^{68}4^{1}6^{1}10^{1}$
			$2^{60}60^{1}$
			$2^{59}4^{1}30^{1}$
			$2^{30}6^{1}20^{1}$
			$2^{28}10^{1}12^{1}$
			$3^{1}40^{1}$
			$5^{1}24^{1}$
			$8^{1}15^{1}$
1	1	**121**	11^{12}
1	1	**122**	$2^{1}61^{1}$
1	1	**123**	$3^{1}41^{1}$
4	4	**124**	2^{123}
			$2^{22}31^{1}$
			$2^{2}62^{1}$
			$4^{1}31^{1}$
1	2	**125**	$5^{25}25^{1}$
7	10	**126**	$2^{1}63^{1}$
			$3^{24}14^{1}$
			$3^{23}6^{1}7^{1}$
			$3^{21}42^{1}$
			$3^{20}6^{1}21^{1}$
			$7^{1}18^{1}$
			$9^{1}14^{1}$
21	740	**128**	$2^{64}64^{1}$
			$2^{6}4^{33}8^{1}16^{1}$
			$2^{6}4^{26}8^{4}16^{1}$
			$2^{6}4^{19}8^{7}16^{1}$
			$2^{6}4^{12}8^{10}16^{1}$
			$2^{6}4^{5}8^{13}16^{1}$
			$2^{5}4^{31}8^{2}16^{1}$
			$2^{5}4^{24}8^{5}16^{1}$
			$2^{5}4^{17}8^{8}16^{1}$
			$2^{5}4^{10}8^{11}16^{1}$
			$2^{5}4^{8}8^{14}$
			$2^{4}4^{36}16^{1}$
			$2^{4}4^{29}8^{3}16^{1}$
			$2^{4}4^{22}8^{6}16^{1}$
			$2^{4}4^{15}8^{9}16^{1}$
			$2^{4}4^{8}8^{12}16^{1}$
			$2^{3}4^{25}8^{7}$
			$2^{3}4^{18}8^{10}$
			$2^{3}4^{11}8^{13}$
			$4^{32}32^{1}$
			$8^{16}16^{1}$
1	1	**129**	$3^{1}43^{1}$
3	4	**130**	$2^{1}65^{1}$
			$5^{1}26^{1}$
			$10^{1}13^{1}$
11	14	**132**	2^{131}
			$2^{42}6^{1}$
			$2^{27}11^{1}$
			$2^{22}33^{1}$
			$2^{18}3^{1}22^{1}$
			$2^{18}6^{1}11^{1}$
			$2^{15}6^{1}22^{1}$
			$2^{2}66^{1}$
			$3^{1}44^{1}$
			$4^{1}33^{1}$
			$11^{1}12^{1}$
1	1	**133**	$7^{1}19^{1}$
1	1	**134**	$2^{1}67^{1}$
3	6	**135**	$3^{32}9^{1}15^{1}$
			$3^{27}45^{1}$
			$5^{1}27^{1}$
5	8	**136**	$2^{132}4^{1}$
			$2^{83}4^{1}17^{1}$
			$2^{68}68^{1}$
			$2^{67}4^{1}34^{1}$
			$8^{1}17^{1}$
3	4	**138**	$2^{1}69^{1}$
			$3^{1}46^{1}$
			$6^{1}23^{1}$

Parents	Designs	Runs	
13	15	**140**	2^{139}
			$2^{38}7^{1}$
			$2^{36}10^{1}$
			$2^{34}14^{1}$
			$2^{27}5^{1}7^{1}$
			$2^{25}5^{1}14^{1}$
			$2^{22}35^{1}$
			$2^{21}7^{1}10^{1}$
			$2^{17}10^{1}14^{1}$
			$2^{2}70^{1}$
			$4^{1}35^{1}$
			$5^{1}28^{1}$
			$7^{1}20^{1}$
1	1	**141**	$3^{1}47^{1}$
1	1	**142**	$2^{1}71^{1}$
1	1	**143**	$11^{1}13^{1}$
16	1,241	**144**	$2^{136}8^{1}$
			$2^{117}8^{1}9^{1}$
			$2^{113}3^{1}24^{1}$
			$2^{111}6^{1}24^{1}$
			$2^{103}8^{1}18^{1}$
			$2^{76}3^{12}6^{4}8^{1}$
			$2^{76}3^{7}4^{1}6^{5}12^{1}$
			$2^{75}3^{3}4^{1}6^{6}12^{1}$
			$2^{74}3^{4}6^{6}8^{1}$
			$2^{72}72^{1}$
			$2^{44}3^{11}12^{2}$
			$2^{16}3^{3}6^{6}24^{1}$
			$3^{48}48^{1}$
			$4^{36}36^{1}$
			$4^{11}12^{2}$
			12^{7}
1	2	**147**	$7^{9}21^{1}$
1	1	**148**	2^{147}
1	5	**150**	$5^{11}30^{1}$
2	6	**152**	$2^{148}4^{1}$
			$2^{76}76^{1}$
1	1	**153**	$3^{25}17^{1}$
1	1	**156**	2^{155}
6	110	**160**	$2^{144}16^{1}$
			$2^{138}4^{7}$
			$2^{133}5^{1}16^{1}$
			$2^{127}10^{1}16^{1}$
			$2^{80}80^{1}$
			$4^{16}40^{1}$
3	61	**162**	$3^{65}6^{1}27^{1}$
			$3^{54}54^{1}$
			$9^{18}18^{1}$
1	1	**164**	2^{163}
2	16	**168**	$2^{164}4^{1}$
			$2^{84}84^{1}$
1	1	**169**	13^{14}
1	1	**171**	$3^{28}19^{1}$
1	1	**172**	2^{171}
1	2	**175**	$5^{10}35^{1}$
4	56	**176**	$2^{168}8^{1}$
			$2^{166}4^{3}$
			$2^{88}88^{1}$
			$4^{12}44^{1}$
3	24	**180**	2^{179}
			$3^{30}60^{1}$
			$6^{2}30^{1}$
2	6	**184**	$2^{180}4^{1}$
			$2^{92}92^{1}$
1	1	**188**	2^{187}
1	4	**189**	$3^{36}63^{1}$
4	726	**192**	$2^{160}32^{1}$
			$2^{96}96^{1}$
			$4^{48}48^{1}$
			$8^{8}24^{1}$
3	11	**196**	2^{195}
			$7^{28}28^{1}$
			14^{5}
1	5	**198**	$3^{30}66^{1}$
4	42	**200**	$2^{196}4^{1}$
			$2^{100}100^{1}$
			$5^{20}40^{1}$
			$10^{5}20^{1}$
1	1	**204**	2^{203}
1	1	**207**	$3^{25}23^{1}$
4	72	**208**	$2^{200}8^{1}$
			$2^{198}4^{3}$
			$2^{104}104^{1}$
			$4^{16}52^{1}$
1	1	**212**	2^{211}
6	258	**216**	$2^{212}4^{1}$
			$2^{108}108^{1}$
			$2^{11}3^{77}12^{1}18^{1}$
			$3^{72}72^{1}$
			$3^{66}6^{5}12^{1}18^{1}$
			$6^{7}36^{1}$
1	1	**220**	2^{219}
5	351	**224**	$2^{208}16^{1}$
			$2^{193}7^{1}16^{1}$
			$2^{183}14^{1}16^{1}$
			$2^{112}112^{1}$
			$4^{56}56^{1}$
3	13	**225**	$3^{27}75^{1}$
			$5^{20}45^{1}$
			15^{6}
1	1	**228**	2^{227}
2	6	**232**	$2^{228}4^{1}$
			$2^{116}116^{1}$
1	5	**234**	$3^{30}78^{1}$
1	1	**236**	2^{235}
6	302	**240**	$2^{232}8^{1}$
			$2^{230}4^{3}$
			$2^{205}5^{1}24^{1}$
			$2^{199}10^{1}24^{1}$
			$2^{120}120^{1}$
			$4^{20}60^{1}$
1	2	**242**	$11^{22}22^{1}$
2	58	**243**	$3^{81}81^{1}$
			$9^{27}27^{1}$
1	1	**244**	2^{243}
1	2	**245**	$7^{10}35^{1}$
2	6	**248**	$2^{244}4^{1}$
			$2^{124}124^{1}$
1	4	**250**	$5^{50}50^{1}$
3	23	**252**	2^{251}
			$3^{42}84^{1}$
			$6^{2}42^{1}$
2	6,101	**256**	$8^{32}32^{1}$
			16^{17}
1	1	**260**	2^{259}
1	2	**261**	$3^{27}87^{1}$
2	16	**264**	$2^{260}4^{1}$
			$2^{132}132^{1}$
1	1	**268**	2^{267}
1	11	**270**	$3^{90}90^{1}$
4	136	**272**	$2^{264}8^{1}$
			$2^{262}4^{3}$
			$2^{136}136^{1}$
			$4^{32}68^{1}$
1	2	**275**	$5^{11}55^{1}$
1	1	**276**	2^{275}
1	2	**279**	$3^{30}93^{1}$
2	17	**280**	$2^{276}4^{1}$
			$2^{140}140^{1}$
1	1	**284**	2^{283}

Parents	Designs	Runs	
8	3,201	**288**	$2^{272} 16^{1}$
			$2^{253} 9^{1} 16^{1}$
			$2^{239} 16^{1} 18^{1}$
			$2^{144} 144^{1}$
			$3^{96} 96^{1}$
			$4^{36} 72^{1}$
			$6^{10} 48^{1}$
			$12^{6} 24^{1}$
1	1	**289**	17^{18}
1	1	**292**	2^{291}
1	5	**294**	$7^{18} 42^{1}$
1	2	**296**	$2^{292} 4^{1}$
1	4	**297**	$3^{39} 99^{1}$
3	24	**300**	2^{299}
			$5^{20} 60^{1}$
			$10^{2} 30^{1}$
4	70	**304**	$2^{296} 8^{1}$
			$2^{294} 4^{3}$
			$2^{228} 76^{1}$
			$4^{16} 76^{1}$
1	5	**306**	$3^{48} 102^{1}$
1	1	**308**	2^{307}
1	2	**312**	$2^{308} 4^{1}$
1	5	**315**	$3^{29} 105^{1}$
1	1	**316**	2^{315}
5	725	**320**	$2^{288} 32^{1}$
			$2^{274} 4^{15}$
			$2^{240} 80^{1}$
			$4^{40} 80^{1}$
			$8^{10} 40^{1}$
6	974	**324**	2^{323}
			$3^{143} 12^{1} 27^{1}$
			$3^{108} 108^{1}$
			$6^{2} 54^{1}$
			$9^{36} 36^{1}$
			18^{3}
1	2	**325**	$5^{20} 65^{1}$
1	2	**328**	$2^{324} 4^{1}$
1	1	**332**	2^{331}
1	2	**333**	$3^{36} 111^{1}$
6	487	**336**	$2^{328} 8^{1}$
			$2^{326} 4^{3}$
			$2^{297} 7^{1} 24^{1}$
			$2^{287} 14^{1} 24^{1}$
			$2^{252} 84^{1}$
			$4^{36} 84^{1}$
1	2	**338**	$13^{26} 26^{1}$
1	1	**340**	2^{339}
1	5	**342**	$3^{30} 114^{1}$
1	2	**343**	$7^{49} 49^{1}$
1	2	**344**	$2^{340} 4^{1}$
1	1	**348**	2^{347}
1	5	**350**	$5^{20} 70^{1}$
1	4	**351**	$3^{39} 117^{1}$
4	203	**352**	$2^{336} 16^{1}$
			$2^{330} 4^{7}$
			$2^{264} 88^{1}$
			$4^{32} 88^{1}$
1	1	**356**	2^{355}
3	133	**360**	$2^{356} 4^{1}$
			$3^{48} 120^{1}$
			$6^{8} 60^{1}$
1	1	**361**	19^{20}
1	2	**363**	$11^{11} 33^{1}$
1	1	**364**	2^{363}
4	150	**368**	$2^{360} 8^{1}$
			$2^{358} 4^{3}$
			$2^{276} 92^{1}$
			$4^{36} 92^{1}$
1	2	**369**	$3^{30} 123^{1}$
1	1	**372**	2^{371}
1	4	**375**	$5^{40} 75^{1}$
1	2	**376**	$2^{372} 4^{1}$
1	11	**378**	$3^{72} 126^{1}$
1	1	**380**	2^{379}
3	3,252	**384**	$2^{320} 64^{1}$
			$4^{96} 96^{1}$
			$8^{16} 48^{1}$
1	2	**387**	$3^{48} 129^{1}$
1	1	**388**	2^{387}
4	28	**392**	$2^{388} 4^{1}$
			$2^{196} 7^{28} 28^{1}$
			$7^{28} 56^{1}$
			$14^{5} 28^{1}$
3	23	**396**	2^{395}
			$3^{132} 132^{1}$
			$6^{2} 66^{1}$
7	912	**400**	$2^{392} 8^{1}$
			$2^{390} 4^{3}$
			$2^{300} 100^{1}$
			$4^{36} 100^{1}$
			$5^{80} 80^{1}$
			$10^{6} 40^{1}$
			20^{5}
1	1	**404**	2^{403}
2	60	**405**	$3^{81} 135^{1}$
			$9^{18} 45^{1}$
1	2	**408**	$2^{404} 4^{1}$
1	1	**412**	2^{411}
1	5	**414**	$3^{48} 138^{1}$
4	299	**416**	$2^{400} 16^{1}$
			$2^{394} 4^{7}$
			$2^{312} 104^{1}$
			$4^{48} 104^{1}$
1	1	**420**	2^{419}
1	2	**423**	$3^{30} 141^{1}$
1	2	**424**	$2^{420} 4^{1}$
1	2	**425**	$5^{20} 85^{1}$
1	1	**428**	2^{427}
7	10,839	**432**	$2^{424} 8^{1}$
			$2^{389} 9^{1} 24^{1}$
			$2^{375} 18^{1} 24^{1}$
			$3^{144} 144^{1}$
			$4^{108} 108^{1}$
			$6^{12} 72^{1}$
			$12^{6} 36^{1}$
1	1	**436**	2^{435}
1	2	**440**	$2^{436} 4^{1}$
3	12	**441**	$3^{42} 7^{7} 21^{1}$
			$7^{14} 63^{1}$
			21^{7}
1	1	**444**	2^{443}
4	8,598	**448**	$2^{416} 32^{1}$
			$2^{336} 112^{1}$
			$4^{56} 112^{1}$
			$8^{56} 56^{1}$
3	35	**450**	$3^{150} 5^{11} 30^{1}$
			$5^{90} 90^{1}$
			$15^{5} 30^{1}$
1	2	**456**	$2^{452} 4^{1}$
1	2	**459**	$3^{72} 9^{1} 17^{1}$
1	1	**460**	2^{459}
4	150	**464**	$2^{456} 8^{1}$
			$2^{454} 4^{3}$
			$2^{348} 116^{1}$
			$4^{36} 116^{1}$
3	17	**468**	2^{467}
			$3^{49} 52^{1}$
			$6^{2} 78^{1}$
1	2	**472**	$2^{468} 4^{1}$
1	2	**475**	$5^{20} 95^{1}$
1	1	**477**	$3^{37} 53^{1}$
4	1,210	**480**	$2^{464} 16^{1}$
			$2^{458} 4^{7}$
			$2^{360} 120^{1}$
			$4^{56} 120^{1}$

Parents	Designs	Runs	
3	8	**484**	2^{483}
			$11^{44}44^{1}$
			22^{3}
1	277	**486**	$9^{54}54^{1}$
1	2	**488**	$2^{484}4^{1}$
1	5	**490**	$7^{18}70^{1}$
1	1	**492**	2^{491}
1	2	**495**	$3^{42}5^{1}33^{1}$
4	78	**496**	$2^{488}8^{1}$
			$2^{486}4^{3}$
			$2^{372}124^{1}$
			$4^{18}124^{1}$
3	29	**500**	2^{499}
			$5^{100}100^{1}$
			$10^{2}50^{1}$
3	111	**504**	$2^{500}4^{1}$
			$2^{84}3^{84}84^{1}$
			$6^{8}84^{1}$
2	73,992	**512**	$8^{64}64^{1}$
			$16^{32}32^{1}$
1	2	**513**	$3^{81}9^{1}19^{1}$
733	**117,561**		

The following step provides a simple example of using the `%MktEx` macro to request the L_{36} design, $2^{11}3^{12}$, which has 11 two-level factors and 12 three-level factors:

```
%mktex(n=36)
```

No iterations are needed, and the macro immediately creates the L_{36}, which is 100% efficient. This example runs in a few seconds. The factors are always named `x1`, `x2`, ... and the levels are always consecutive integers starting with 1. You can use the `%MktLab` macro to assign different names and levels (see page 1093).

Randomization

By default, the macro creates two output data sets with the design—one sorted and one randomized

- `out=Design` – the experimental design, sorted by the factor levels.
- `outr=Randomized` – the randomized experimental design.

The two designs are equivalent and have the same *D*-efficiency. The `out=Design` data set is sorted and hence is usually easier to look at, however the `outr=Randomized` design is usually the better one to use. The randomized design has the rows sorted into a random order, and all of the factor levels are randomly reassigned. For example with two-level factors, approximately half of the original (1, 2) mappings are reassigned (2, 1). Similarly, with three level factors, the mapping (1, 2, 3) are changed to one of the following: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), or (3, 2, 1). The reassignment of levels is usually not critical for the iteratively derived designs, but it can be very important the orthogonal designs, many of which have all ones in the first row.

Latin Squares and Graeco-Latin Square Designs

The `%MktEx` orthogonal array catalog can be used to make both Latin Square and Graeco-Latin Square (mutually orthogonal Latin Square) designs. A Latin square is an $p \times p$ table with p different values arranged so that each value occurs exactly once in each row and exactly once in each column. An orthogonal array p^3 in p^2 runs can be used to make a Latin square. The following matrices are Latin Squares of order $p = 3, 3, 4, 5, 6$:

$$\begin{bmatrix} 1 & 3 & 2 \\ 3 & 2 & 1 \\ 2 & 1 & 3 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 3 & 5 & 2 & 4 \\ 5 & 2 & 4 & 1 & 3 \\ 4 & 1 & 3 & 5 & 2 \\ 3 & 5 & 2 & 4 & 1 \\ 2 & 4 & 1 & 3 & 5 \end{bmatrix} \quad \begin{bmatrix} 3 & 6 & 4 & 1 & 5 & 2 \\ 1 & 4 & 5 & 2 & 6 & 3 \\ 2 & 5 & 3 & 6 & 1 & 4 \\ 4 & 1 & 2 & 5 & 3 & 6 \\ 5 & 2 & 6 & 3 & 4 & 1 \\ 6 & 3 & 1 & 4 & 2 & 5 \end{bmatrix}$$

The first two Latin Squares are obtained from `%MktEx` as follows:

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

proc print; run;
```

The design is as follows:

```
Obs    x1    x2    x3    x4

 1      1     1     1     1
 2      1     2     3     2
 3      1     3     2     3
 4      2     1     3     3
 5      2     2     2     1
 6      2     3     1     2
 7      3     1     2     2
 8      3     2     1     3
 9      3     3     3     1
```

To make a Latin square from an orthogonal array, treat `x1` as the row number and `x2` as the column number. The values in `x3` form one Latin square (the first in the list shown previously), and the values in `x4` form a different Latin square (the second in the list). You can use the following macro to display the design that `%MktEx` creates in the Latin square format:

```
%macro latin(x,y);
   proc iml;
      use design;
      read all into x;
      file print;
      s1 = '123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ+=-*';
      s2 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456789+=-*';
      k = 0;
      m = x[nrow(x),1];
      do i = 1 to m;
         put;
         do j = 1 to m;
            k = k + 1;
            put (substr(s1, x[k,&x], 1)) $1. @;
            %if &y ne %then %do; put +1 (substr(s2, x[k,&y], 1)) $1. +1 @; %end;
            put +1 @;
            end;
         end;

      put;
      quit;
   %mend;
```

You specify the column number as the first parameter of the macro. You can specify any p-level factor after the first two. The following statements create and display the five Latin squares that are displayed previously:

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

%mktex(4 ** 5, n=4 ** 2)
%latin(3)

%mktex(5 ** 3, n=25)
%latin(3)

%mktex(6 ** 3, n=36)
%latin(3)
```

Note that you can specify a number or an expression for `n=`, and this example does both. The results (from the `latin` macro only) are as follows:

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

1 2 3
3 1 2
2 3 1

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

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

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

Alternatively, you can construct a Latin square from the randomized design, for example, as follows:

```
%mktex(6 ** 3,                   /* 3 six-level factor                  */
       n=36,                     /* 36 runs                             */
       options=nohistory         /* do not display iteration history    */
               nofinal,          /* do not display final levels, D-eff  */
       seed=109)                 /* random number seed                  */

proc sort data=randomized out=design; by x1 x2; run;

%latin(3)
```

With different random number seeds, you will typically get different Latin squares, particularly for larger Latin squares. The results of this step are as follows:

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

When an orthogonal array has 3 or more p-level factors in p^2 runs, you can make one or more Latin squares. When an orthogonal array has 4 or more p-level factors in p^2 runs, you can make one or more Graeco-Latin squares, which are also known as mutually orthogonal Latin squares or Euler squares (named after the Swiss mathematician Leonhard Euler). The following example creates and displays a Graeco-Latin square of order $p = 3$:

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

The results are as follows:

```
1 A  3 B  2 C
3 C  2 A  1 B
2 B  1 C  3 A
```

Each entry consists of two values (the two columns of the design that are specified in the `latin` macro). The $p \times p = 9$ left-most values form a Latin square as do the $p \times p$ right-most values. In addition, the two factors are orthogonal to each other and to the row and column indexes (`x1` and `x2`). Graeco-Latin squares exist for all $p \geq 3$ except $p = 6$. The following steps create and display a Graeco-Latin square of order $p = 12$:

```
%mktex(12 ** 4, n=12 ** 2)
%latin(3,4)
```

The results are as follows:

```
1 A  7 G  5 H  b I  3 F  9 K  8 L  6 C  c E  4 J  a D  2 B
a K  2 A  8 G  6 H  c I  4 F  3 B  9 L  1 C  7 E  5 J  b D
5 F  b K  3 A  9 G  1 H  7 I  c D  4 B  a L  2 C  8 E  6 J
8 I  6 F  c K  4 A  a G  2 H  1 J  7 D  5 B  b L  3 C  9 E
3 H  9 I  1 F  7 K  5 A  b G  a E  2 J  8 D  6 B  c L  4 C
c G  4 H  a I  2 F  8 K  6 A  5 C  b E  3 J  9 D  1 B  7 L
2 L  c C  6 E  a J  4 D  8 B  7 A  1 G  b H  5 I  9 F  3 K
9 B  3 L  7 C  1 E  b J  5 D  4 K  8 A  2 G  c H  6 I  a F
6 D  a B  4 L  8 C  2 E  c J  b F  5 K  9 A  3 G  7 H  1 I
7 J  1 D  b B  5 L  9 C  3 E  2 I  c F  6 K  a A  4 G  8 H
4 E  8 J  2 D  c B  6 L  a C  9 H  3 I  7 F  1 K  b A  5 G
b C  5 E  9 J  3 D  7 B  1 L  6 G  a H  4 I  8 F  2 K  c A
```

The `latin` macro uses numerals, lower-case letters, upper case letters, and symbols to display the first Latin square. It uses upper case letters, lower-case letters, numerals, and symbols to display the second Latin square. Up to 64 different values can be displayed. Currently, the `%MktEx` macro is capable of making Graeco-Latin squares for all orders in the range 3–25 except 6 (does not exist), 18, and 22. It can make a few larger ones as well (when p is a power of a prime like 49, 64, and 81).

Split-Plot Designs

In a split-plot experiment, there are blocks or plots, and some factors can only be applied to an entire plot. The term "plot" comes from agricultural experiments, where it is only convenient to apply some treatments to entire plots of land. These factors are called "whole-plot factors." Within a plot, other factors can be applied to smaller sections, and these are called "subplot factors." The goal is to create an experimental design with n rows, and p plots of size s where $n = ps$. All whole plot factors must be constant within each of the p plots of size s.

Split-Plot Designs from Orthogonal Arrays

The following split-plot design, with 4 plots of size 6, is constructed from the orthogonal array 2^{23} in 24 runs:

```
Plot  Whole   ------------Subplot Factors------------

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

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

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

Notice that there is a maximum of three whole-plot factors available in this design, and that all of the 21 subplot-factors are perfectly balanced within each plot. You can construct this design as follows:

```
   %mktex(2 ** 23, n=24, options=nosort)

   data splitplot;
      Plot = floor((_n_ - 1) / 6) + 1; /* 6 1s, 6 2s, 6 3s, 6 4s */
      set design;
      run;

   proc print; id Plot; by Plot; var x22 x23 x21 x1-x20; run;
```

The `nosort` option is used to output the design just as it is created, without sorting first. In this particular case, it leads to a particularly nice arrangement of the design where the whole-plot factors can be constructed from `x22`, `x23`, and `x21`. To understand why this is the case, you need to see the lineage or construction method for this particular design. The design lineage is a set of instructions for making a design with smaller-level factors from a design with higher-level factors. You can begin by adding the `lineage` option as follows:

```
   %mktex(2 ** 23, n=24, options=nosort lineage)
```

In addition to making the design, the `%MktEx` macro reports the following:

```
   Design Lineage:
   24 ** 1 : 24 ** 1 > 2 ** 20 4 ** 1 : 4 ** 1 > 2 ** 3
```

The first part of the lineage states that the design begins as a single 24-level factor,[*] and it is replaced by the orthogonal array $2^{20}4^1$. The final array is constructed from $2^{20}4^1$ by replacing the four-level factor with 3 two-level factors (4^1 with 2^3), which creates the 3 two-level factors that we use for our whole-plot factors. Note that the factors come out in the order described by the lineage: the 20 two-level factors first, then the 3 two-level factors that come from the four-level factor. You cannot get the two-level factors that come from the four-level factor without also requesting the other 20 two-level factors. We want the two-level factors that come from the four-level factor for our whole-plot factors, because in an orthogonal array, the underlying four-level factor must be orthogonal to a six-level factor. This ensures that each of the four combinations of these particular two-level factors occurs exactly six times in the final design. This is not guaranteed for other triples of two-level factors in the design.

Now consider selecting whole-plot factors from other two-level factors, for example as follows:

[*]This allows the same code that replaces a 24-level factor (for example in 48, 72, 96, 120, or 144 runs) to be used to make designs in 24 runs.

```
proc sort data=design out=splitplot;
   by x1-x23;
   run;

data splitplot;
   Plot = floor((_n_ - 1) / 6) + 1;
   set splitplot;
   run;

proc print; id Plot; by Plot; run;
```

The results are as follows:

```
P
l                             x  x  x  x  x  x  x  x  x  x  x  x  x  x
o  x  x  x  x  x  x  x  x  x  1  1  1  1  1  1  1  1  1  1  2  2  2  2
t  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7  8  9  0  1  2  3

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

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

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

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

Now there are only two whole-plot factors available, and many of the subplot factors are not balanced within each plot. Many split-plot designs can be constructed by carefully selecting factors from an orthogonal array. To take full advantage of the orthogonal array capabilities in the %MktEx macro for use in split-plot designs, you need to understand something else about design lineages. Some designs, such as 2^{23}, can be constructed in multiple ways. Some of those ways work better for our purposes than others. Consider the following steps:

```
%mktorth(range=n=24, options=dups lineage)

data cat;
   set mktdescat;
   design  = left(transtrn(compbl(design), ' ** ', '^'));
   lineage = left(transtrn(substr(lineage, 21), ' ** ', '^'));
   run;

proc print noobs; run;
```

The %MktOrth macro manages the catalog of orthogonal arrays that the %MktEx macro can make, and provides the %MktEx macro with instructions (when the lineage option is specified) about how the designs are made. The option range=n=24 outputs instructions for only designs in 24 runs. The dups option suppresses normal filtering and removal of duplicate and inferior designs. A design is considered to be a duplicate of another design if it has exactly the same numbers of each type of factor as some other design. An inferior design has only a subset of the factors that are available in a competing design. This determination is solely based on the number of each type of factor, not on the actual levels of the factors. Depending on your purposes, a duplicate or inferior design might be a better than the design that %MktEx makes by default, because different construction methods lead to designs with different combinatorial properties. For this reason, %MktEx provides you with a way to create the designs that it filters out by default.

The %MktOrth macro creates two data sets. The MKTDESCAT data set is smaller and more suitable for display. The MKTDESLEV dat set is larger, and has more information for processing and subsetting. In particular, the MKTDESLEV data set has a variable x2 that contains the number of two-level factors, x3 that contains the number of three-level factors, and so on. The %MktEx macro uses the information in the MKTDESLEV dat set when it creates orthogonal arrays.

The DATA step simply subsets and reformats the results into a form that can be displayed in one panel without splitting. The transtrn function replaces the string ' ** ' with '^' indicating an exponent in much less space. Additionally, the parts of the lineages dealing with 24 level factors being replaced by a parent along with extra blanks are removed. The results are displayed next:

```
  n  Design        Reference         Lineage

 24  2^23          Hadamard          2^12 12^1 : 12^1 > 2^11
 24  2^23          Hadamard          2^20 4^1 : 4^1 > 2^3
 24  2^20 4^1      Orthogonal Array  2^20 4^1 (parent)
 24  2^16 3^1      Orthogonal Array  2^12 12^1 : 12^1 > 2^4 3^1
 24  2^16 3^1      Orthogonal Array  2^13 3^1 4^1 : 4^1 > 2^3
 24  2^15 3^1      Orthogonal Array  2^11 4^1 6^1 : 6^1 > 2^1 3^1 : 4^1 > 2^3
 24  2^15 3^1      Orthogonal Array  2^12 12^1 : 12^1 > 2^2 6^1 : 6^1 > 2^1 3^1
 24  2^15 3^1      Orthogonal Array  2^12 12^1 : 12^1 > 3^1 4^1 : 4^1 > 2^3
 24  2^14 6^1      Orthogonal Array  2^11 4^1 6^1 : 4^1 > 2^3
 24  2^14 6^1      Orthogonal Array  2^12 12^1 : 12^1 > 2^2 6^1
 24  2^13 3^1 4^1  Orthogonal Array  2^13 3^1 4^1 (parent)
 24  2^12 3^1 4^1  Orthogonal Array  2^11 4^1 6^1 : 6^1 > 2^1 3^1
 24  2^12 3^1 4^1  Orthogonal Array  2^12 12^1 : 12^1 > 3^1 4^1
 24  2^12 12^1     Orthogonal Array  2^12 12^1 (parent)
 24  2^11 4^1 6^1  Orthogonal Array  2^11 4^1 6^1 (parent)
 24  2^7 3^1       Orthogonal Array  3^1 8^1 : 8^1 > 2^4 4^1 : 4^1 > 2^3
 24  2^4 3^1 4^1   Orthogonal Array  3^1 8^1 : 8^1 > 2^4 4^1
 24  3^1 8^1       Full-Factorial    3^1 8^1 (parent)
```

The design 2^{23} can be made from the two parent arrays $2^{20}4^1$ and $2^{12}12^1$. By default, the parent $2^{20}4^1$ is used, and you can see this by examining the default catalog with the default filtering of duplicate and inferior designs, for example as follows:

```
%mktorth(range=n=24, options=lineage)

data cat;
   set mktdescat;
   design  = left(transtrn(compbl(design), ' ** ', '^'));
   lineage = left(transtrn(substr(lineage, 21), ' ** ', '^'));
   run;

proc print noobs; run;
```

The results are as follows:

```
  n    Design          Reference           Lineage

 24    2^23            Hadamard            2^20 4^1 : 4^1 > 2^3
 24    2^20 4^1        Orthogonal Array    2^20 4^1 (parent)
 24    2^16 3^1        Orthogonal Array    2^13 3^1 4^1 : 4^1 > 2^3
 24    2^14 6^1        Orthogonal Array    2^11 4^1 6^1 : 4^1 > 2^3
 24    2^13 3^1 4^1    Orthogonal Array    2^13 3^1 4^1 (parent)
 24    2^12 12^1       Orthogonal Array    2^12 12^1 (parent)
 24    2^11 4^1 6^1    Orthogonal Array    2^11 4^1 6^1 (parent)
 24    3^1 8^1         Full-Factorial      3^1 8^1 (parent)
```

You can force `%MktEx` to construct the array from the parent $2^{12}12^1$ by explicitly providing `%MktEx` with a design catalog that contains the lineage of just the design of interest, for example as follows:

```
data lev;
   set mktdeslev;
   if x2 eq 23 and index(lineage, '2 ** 11');
   run;

%mktex(2 ** 23, n=24, options=nosort, cat=lev)

data splitplot;
   Plot = floor((_n_ - 1) / 6) + 1;
   set design;
   run;

proc print; id Plot; by Plot; run;
```

By default, when you do not specify the `cat=` option, `%MktEx` uses `%MktOrth` to generate the filtered catalog automatically. When you specify the `cat=` option, `%MktEx` does not use `%MktOrth` to generate the catalog, and it uses only the catalog that you provide.

The results are as follows:

```
P
l                                   x  x  x  x  x  x  x  x  x  x  x  x  x  x
o  x  x  x  x  x  x  x  x  x  1  1  1  1  1  1  1  1  1  1  2  2  2  2
t  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7  8  9  0  1  2  3

1  1  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2
   1  1  2  1  2  2  2  1  1  1  2  1  1  2  1  2  2  2  1  1  1  2  1
   1  1  1  2  1  2  2  2  1  1  1  2  1  1  2  1  2  2  2  1  1  1  2
   1  2  1  1  2  1  2  2  2  1  1  1  2  1  1  2  1  2  2  2  1  1  1
   1  1  2  1  1  2  1  2  2  2  1  1  1  2  1  1  2  1  2  2  2  1  1
   1  1  1  2  1  1  2  1  2  2  2  1  1  1  2  1  1  2  1  2  2  2  1
```

```
2  1  1  1  1  2  1  1  2  1  2  2  2  1  1  1  2  1  1  2  1  2  2  2
   1  2  1  1  1  2  1  1  2  1  2  2  2  1  1  1  2  1  1  2  1  2  2
   1  2  2  1  1  1  2  1  1  2  1  2  2  2  1  1  1  2  1  1  2  1  2
   1  2  2  2  1  1  1  2  1  1  2  1  2  2  2  1  1  1  2  1  1  2  1
   1  1  2  2  2  1  1  1  2  1  1  2  1  2  2  2  1  1  1  2  1  1  2
   1  2  1  2  2  2  1  1  1  2  1  1  2  1  2  2  2  1  1  1  2  1  1

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

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

Now there is only one whole-plot factor that is easily identified. Other sorts might reveal more whole-plot factors. For this problem, we would stick with the design that `%MktEx` makes by default. For other problems, you might want to force `%MktEx` to use a lineage that would otherwise be filtered out.

Split-Plot Designs from Computerized Search

Many split-plot designs cannot be constructed directly from orthogonal arrays. However, you can create a split-plot design by using the `%MktEx` macro to create a design for the whole-plot factors and by using a second `%MktEx` step to append the subplot factors. The following steps show one way to create the whole-plot factors (applying techniques discussed previously) and add missing values or place holders for the additional subplot factors:

```
%mktorth(range=n=24, options=dups lineage)

data lev;
   set mktdeslev;
   where x2 = 14 and index(lineage, '2 ** 11 4 ** 1 6 ** 1');
   run;

%mktex(2 ** 14 6, n=24, out=whole(keep=x12-x14), options=nosort)
```

```
data in(keep=x1-x7);
   retain x1-x7 .;
   set whole;
   x1 = x13;
   x2 = x14;
   x3 = x12;
   run;

proc print; run;
```

These steps rely on creating two-level factors from a four-level factor that is orthogonal to a six-level factor. This is similar to logic used previously, but now a different parent design is used. The results are as follows:

```
Obs    x1    x2    x3    x4    x5    x6    x7

  1     1     1     1     .     .     .     .
  2     1     1     1     .     .     .     .
  3     1     1     1     .     .     .     .
  4     1     1     1     .     .     .     .
  5     1     1     1     .     .     .     .
  6     1     1     1     .     .     .     .

  7     1     2     2     .     .     .     .
  8     1     2     2     .     .     .     .
  9     1     2     2     .     .     .     .
 10     1     2     2     .     .     .     .
 11     1     2     2     .     .     .     .
 12     1     2     2     .     .     .     .

 13     2     2     1     .     .     .     .
 14     2     2     1     .     .     .     .
 15     2     2     1     .     .     .     .
 16     2     2     1     .     .     .     .
 17     2     2     1     .     .     .     .
 18     2     2     1     .     .     .     .

 19     2     1     2     .     .     .     .
 20     2     1     2     .     .     .     .
 21     2     1     2     .     .     .     .
 22     2     1     2     .     .     .     .
 23     2     1     2     .     .     .     .
 24     2     1     2     .     .     .     .
```

There are many other ways that you could make the initial data set with the whole plot factors, including more directly as follows:

```
data in(keep=x1-x7);
   retain x1-x7 .;
   input x1-x3;
   do i = 1 to 6; output; end;
   datalines;
1 1 1
1 2 2
2 2 1
2 1 2
;
```

This step reads an orthogonal array with no replication for the whole-plot factors and makes six copies of it. The following steps illustrate another way, this time by creating rather than reading an orthogonal array with no replication for the whole-plot factors:

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

data in(keep=x1-x7);
   retain x1-x7 .;
   set design(rename=(x2=x1 x3=x2 x1=x3));
   do i = 1 to 6; output; end;
   run;
```

The `rename=` option is not necessary. In this example, it simply ensures that the levels match those used previously. This particular ordering of the first two whole-plot factors ensures minimal change of the factor levels across plots, which is desirable for some industrial uses of split-plot designs.

Regardless of how it is created, this design is used as an input initial design in a subsequent `%MktEx` run. In the `%MktEx` step, missing values are replaced, while nonmissing values are fixed and are not changed by the iterations. The whole-plot factors are input and never change, while the subplot factors are optimally (or at least nearly optimally) appended in the next step. The following steps append the subplot factors and evaluate and display the results:

```
%mktex(2 2 2   6 6 2 3, n=24, init=in, seed=104, options=nosort, maxiter=100)

%mkteval;

data splitplot;
   Plot = floor((_n_ - 1) / 6) + 1;
   set design;
   run;

proc print; id plot; by Plot;  run;
```

The results are as follows:

```
                Canonical Correlations Between the Factors
             There is 1 Canonical Correlation Greater Than 0.316

               x1      x2      x3      x4      x5      x6      x7

         x1    1       0       0       0       0       0       0.10
         x2    0       1       0       0       0       0       0.10
         x3    0       0       1       0       0       0       0.19
         x4    0       0       0       1       0.50    0       0.29
         x5    0       0       0       0.50    1       0       0.29
         x6    0       0       0       0       0       1       0.10
         x7    0.10    0.10    0.19    0.29    0.29    0.10    1

              Canonical Correlations > 0.316 Between the Factors
             There is 1 Canonical Correlation Greater Than 0.316


                                         r    r Square

                            x4    x5    0.50      0.25

                         Summary of Frequencies
             There is 1 Canonical Correlation Greater Than 0.316
                      * - Indicates Unequal Frequencies

                        Frequencies

                x1       12 12
                x2       12 12
                x3       12 12
                x4       4 4 4 4 4 4
                x5       4 4 4 4 4 4
                x6       12 12
             *  x7       6 9 9
                x1 x2    6 6 6 6
                x1 x3    6 6 6 6
                x1 x4    2 2 2 2 2 2 2 2 2 2 2 2
                x1 x5    2 2 2 2 2 2 2 2 2 2 2 2
                x1 x6    6 6 6 6
             *  x1 x7    3 5 4 3 4 5
```

```
          x2 x3     6 6 6 6
          x2 x4     2 2 2 2 2 2 2 2 2 2 2 2
          x2 x5     2 2 2 2 2 2 2 2 2 2 2 2
          x2 x6     6 6 6 6
     *    x2 x7     3 5 4 3 4 5
          x3 x4     2 2 2 2 2 2 2 2 2 2 2 2
          x3 x5     2 2 2 2 2 2 2 2 2 2 2 2
          x3 x6     6 6 6 6
     *    x3 x7     2 5 5 4 4 4
     *    x4 x5     0 1 0 1 1 1 1 0 1 1 0 1 1 1 0 0 1 1 0
                    1 1 0 1 1 1 1 1 1 0 0 1 0 1 1 1 0
          x4 x6     2 2 2 2 2 2 2 2 2 2 2 2
     *    x4 x7     1 2 1 1 1 2 1 2 1 1 2 1 1 1 2 1 1 2
          x5 x6     2 2 2 2 2 2 2 2 2 2 2 2
     *    x5 x7     1 1 2 1 2 1 1 1 2 1 1 2 1 2 1 1 2 1
     *    x6 x7     3 5 4 3 4 5
          N-Way     1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                    1 1 1 1 1

       Plot    x1    x2    x3    x4    x5    x6    x7

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

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

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

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

The %MktEval macro automatically flags factors that are correlated with a canonical correlation greater than 0.316, which corresponds to $r^2 > 0.1$. Perfect orthogonality and balance are not possible since neither 6×6 nor 3×6 divides 24.

Split-Plot Designs with Better Within-Plot Balance

If balance in the subplot factors within the plots is a concern, then you can often force better balance by adding the plot number as a factor for the duration of the design creation, for example as follows:

```
data in(keep=x1-x8);
   retain x1-x8 .;
   set whole; /* same design with whole-plot factors as before */
   x1 = x13;
   x2 = x14;
   x3 = x12;
   x4 = floor((_n_ - 1) / 6) + 1;  /* Plot number: 1, 2, 3, 4 */
   run;

%mktex(2 2 2   4   6 6 2 3, n=24, init=in, seed=104, options=nosort,
       maxiter=100, ridge=1e-4)

%mkteval;

proc print label; by x4; id x4; label x4 = 'Plot'; run;
```

This results in a design with a linear dependency and hence zero efficiency[*], however, that does not pose a problem for the %MktEx macro. It iterates and finds a good design optimizing a ridged efficiency criterion (a constant value is added to the diagonal of the $\mathbf{X'X}$ matrix to make it nonsingular). The final design, when evaluated without the plot factor, has a reasonable *D*-efficiency. The ridge= option is not necessary in this example. Without it, the default is ridge=1e-7. However, if the addition of the plot factor results in more parameters than runs, then you must specify the ridge= option. The macro will not produce a design with more parameters than runs without an explicit ridge= specification. The results are as follows:

```
                Canonical Correlations Between the Factors
            There are 4 Canonical Correlations Greater Than 0.316

            x1      x2      x3      x4      x5      x6      x7      x8

      x1    1       0       0       1.00    0       0       0       0
      x2    0       1       0       1.00    0       0       0       0
      x3    0       0       1       1.00    0       0       0       0
      x4    1.00    1.00    1.00    1       0       0       0       0
      x5    0       0       0       0       1       0.43    0       0.25
      x6    0       0       0       0       0.43    1       0       0.25
      x7    0       0       0       0       0       0       1       0
      x8    0       0       0       0       0.25    0.25    0       1
```

[*]In this design, the plot variable equals $2\times$x1 + x3 $- 2$.

```
                 Canonical Correlations > 0.316 Between the Factors
               There are 4 Canonical Correlations Greater Than 0.316


                                         r    r Square

                          x1    x4    1.00      1.00
                          x2    x4    1.00      1.00
                          x3    x4    1.00      1.00
                          x5    x6    0.43      0.19

                             Summary of Frequencies
               There are 4 Canonical Correlations Greater Than 0.316
                        * - Indicates Unequal Frequencies

                            Frequencies

                 x1       12 12
                 x2       12 12
                 x3       12 12
                 x4       6 6 6 6
                 x5       4 4 4 4 4 4
                 x6       4 4 4 4 4 4
                 x7       12 12
                 x8       8 8 8
                 x1 x2    6 6 6 6
                 x1 x3    6 6 6 6
            *    x1 x4    6 6 0 0 0 0 6 6
                 x1 x5    2 2 2 2 2 2 2 2 2 2 2 2
                 x1 x6    2 2 2 2 2 2 2 2 2 2 2 2
                 x1 x7    6 6 6 6
                 x1 x8    4 4 4 4 4 4
                 x2 x3    6 6 6 6
            *    x2 x4    6 0 0 6 0 6 6 0
                 x2 x5    2 2 2 2 2 2 2 2 2 2 2 2
                 x2 x6    2 2 2 2 2 2 2 2 2 2 2 2
                 x2 x7    6 6 6 6
                 x2 x8    4 4 4 4 4 4
            *    x3 x4    6 0 6 0 0 6 0 6
                 x3 x5    2 2 2 2 2 2 2 2 2 2 2 2
                 x3 x6    2 2 2 2 2 2 2 2 2 2 2 2
                 x3 x7    6 6 6 6
                 x3 x8    4 4 4 4 4 4
                 x4 x5    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                          1 1 1 1 1
                 x4 x6    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
                          1 1 1 1 1
                 x4 x7    3 3 3 3 3 3 3 3
                 x4 x8    2 2 2 2 2 2 2 2 2 2 2 2
```

```
*    x5 x6    1 1 0 1 0 1 1 0 0 1 1 1 0 1 1 1 1 0 1
              1 1 0 1 0 1 1 1 0 0 1 0 0 1 1 1 1
     x5 x7    2 2 2 2 2 2 2 2 2 2 2 2
*    x5 x8    2 1 1 1 1 2 1 2 1 1 1 2 1 2 1 2 1 1
     x6 x7    2 2 2 2 2 2 2 2 2 2 2 2
*    x6 x8    1 1 2 1 2 1 2 1 1 2 1 1 1 1 2 1 2 1
     x7 x8    4 4 4 4 4 4
     N-Way    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
              1 1 1 1 1

    Plot   x1    x2    x3    x5    x6    x7    x8

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

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

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

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

Now, all of the factors are balanced within each plot. This can be seen by examining the two-way frequencies involving `x4` along with `x5` through `x8`. This design in fact looks better than the one found previously, although in general, there is no guarantee that this approach will make a better design. Ignoring `x4`, the plot number, all of the nonzero canonical correlations have gotten smaller in this design compared to what they were previously. You can evaluate the design ignoring the plot number by using this step: `%mkteval(data=design(drop=x4))`.

Candidate Set Search

The candidate-set search has two parts. First, either PROC PLAN is run to create a full-factorial design for small problems, or PROC FACTEX is run to create a fractional-factorial design for large problems. Either way, this design is a candidate set that in the second part is searched by PROC OPTEX using the modified Fedorov algorithm. A design is built from a selection of the rows of the candidate set (Fedorov 1972; Cook and Nachtsheim 1980). The modified Fedorov algorithm considers each run in the design and each candidate run. Candidate runs are swapped in and design runs are swapped out if the swap improves *D*-efficiency.

Coordinate Exchange

The `%MktEx` macro also uses the coordinate-exchange algorithm, based on Meyer and Nachtsheim (1995). The coordinate-exchange algorithm considers each level of each factor, and considers the effect on *D*-efficiency of changing a level ($1 \rightarrow 2$, or $1 \rightarrow 3$, or $2 \rightarrow 1$, or $2 \rightarrow 3$, or $3 \rightarrow 1$, or $3 \rightarrow 2$, and so on). Exchanges that increase efficiency are performed. Typically, the macro first tries to initialize the design with an orthogonal design (`Tab` refers to the orthogonal array table or catalog) and a random design (`Ran`) both. Levels that are not orthogonally initialized can be exchanged for other levels if the exchange increases efficiency.

The initialization might be more complicated. Say you asked for the design $4^1 5^1 3^5$ in 18 runs. The macro would use the orthogonal design $3^6 6^1$ in 18 runs to initialize the three-level factors orthogonally, and the five-level factor with the six-level factor coded down to five levels (which is imbalanced). The four-level factor would be randomly initialized. The macro would also try the same initialization but with a random rather than unbalanced initialization of the five-level factor, as a minor variation on the first initialization. In the next initialization variation, the macro would use a fully random initialization. If the number of runs requested were smaller than the number or runs in the initial orthogonal design, the macro would initialize the design with just the first n rows of the orthogonal design. Similarly, if the number of runs requested were larger than the number or runs in the initial orthogonal design, the macro would initialize part of the design with the orthogonal design and the remaining rows and columns randomly. The coordinate-exchange algorithm considers each level of each factor that is not orthogonally initialized, and it exchanges a level if the exchange improves *D*-efficiency. When the number or runs in the orthogonal design does not match the number of runs desired, none of the design is initialized orthogonally.

The coordinate-exchange algorithm is not restricted by having a candidate set and hence can *potentially* consider every possible design. That is, no design is precluded from consideration due to the limitations of a candidate set. In practice, however, both the candidate-set-based and coordinate-exchange algorithms consider only a *tiny* fraction of the possible designs. When the number of runs in the full-factorial design is very small (say 100 or 200 runs), the modified Fedorov algorithm and coordinate exchange algorithms usually work equally well. When the number of runs in the full-factorial design is small (up to several thousand), the modified Fedorov algorithm is usually superior to coordinate exchange, particularly in finding designs with interactions. When the full-factorial design is larger, coordinate exchange is usually the superior approach. However, heuristics like these are often wrong, which is why the macro tries both methods to see which one is really best for each problem.

Next, the `%MktEx` macro determines which algorithm (candidate set search, coordinate exchange with partial orthogonal initialization, or coordinate exchange with random initialization) is working best and tries more iterations using that approach. It starts by displaying the initial (`Ini`) best efficiency.

Next, the `%MktEx` macro tries to improve the best design it found previously. Using the previous best design as an initialization (`Pre`), and random mutations of the initialization (`Mut`) and simulated annealing (`Ann`), the macro uses the coordinate-exchange algorithm to try to find a better design. This step is important because the best design that the macro found might be an intermediate design and might not be the final design at the end of an iteration. Sometimes, the iterations deliberately make the designs less efficient, and sometimes, the macro never finds a design as efficient or more efficient again. Hence, it is worthwhile to see if the best design found so far can be improved. At the end, PROC OPTEX is called to display the levels of each factor and the final *D*-efficiency.

Random mutations involve adding random noise to the initial design before iterations start (levels are randomly changed). This might eliminate the perfect balance that will often be in the initial design. By default, random mutations are used with designs with fully random initializations and in the design refinement step; orthogonal initial designs are not mutated.

Coordinate exchange can be combined with the simulated annealing optimization technique (Kirkpatrick, Gellat, and Vecchi 1983). Annealing refers to the cooling of a liquid in a heat bath. The structure of the solid depends on the rate of cooling. Coordinate exchange without simulated annealing seeks to maximize *D*-efficiency at every step. Coordinate exchange with simulated annealing lets *D*-efficiency occasionally decrease with a probability that decreases with each iteration. This is analogous to slower cooling, and it helps overcome local optima.

For design 1, for the first level of the first factor, by default, the macro might execute an exchange (say change a 2 to a 1) that makes the design worse with probability 0.05. As more and more exchanges occur, this probability decreases so at the end of the processing of design 1, exchanges that decrease efficiency are hardly ever done. For design 2, this same process is repeated, again starting by default with an annealing probability of 0.05. This often helps the algorithm overcome local efficiency maxima. To envision this, imagine that you are standing on a molehill next to a mountain. The only way you can start going up the mountain is to first step down off the molehill. Once you are on the mountain, you might occasionally hit a dead end, where all you can do is step down and look for a better place to continue going up. Simulated annealing, by occasionally stepping down the efficiency function, often lets the macro go farther up it than it would otherwise. The simulated annealing is why you will sometimes see designs getting worse in the iteration history. The macro keeps track of the best design, not the final design in each step. By default, annealing is used with designs with fully random initializations and in the design refinement step. Simulated annealing is not used with orthogonally initialized designs.

Aliasing Structure

The following example illustrates the `examine=aliasing=2` option:

```
%mktex(3 ** 4, n=9, examine=aliasing=2)
```

The preceding step produces the following results:

```
                                  Aliasing Scheme
Estimable  Aliased
Effect     Effects

Intercept

x1         x2*x3 x2*x4 x3*x4

x2         x1*x3 x1*x4 x3*x4

x3         x1*x2 x1*x4 x2*x4

x4         x1*x2 x1*x3 x2*x3

NOTE: Some parameters in the estimable effects are aliased with some parameters
 in the aliased effects.  For effects with more than two levels, the aliasing
 scheme displayed here is potentially partial.  Specify EXAMINE=FULL ALIASING=n
 to see the full aliasing structure.
```

These results show that one or both of the two parameters in each of `x1-x4` are aliased with one or more of the $(3-1) \times (3-1) = 4$ parameters in the each of three two-way interactions that do not involve the estimable effect.

You can get the full results as follows:

```
%mktex(3 ** 4, n=9, examine=aliasing=2 full)
```

The full aliasing results involving the `x1` main effect are as follows:

```
x11 - x21x32 - x22x31 + 0.5*x21x41 - 0.5*x21x42 + 0.5*x22x41 + 1.5*x22x42 +
   0.5*x31x41 - 0.5*x31x42 - 0.5*x32x41 - 1.5*x32x42

x12 - 0.3333*x21x31 + x22x32 - 0.1667*x21x41 - 0.5*x21x42 + 0.5*x22x41 -
   0.5*x22x42 + 0.1667*x31x41 + 0.5*x31x42 + 0.5*x32x41 - 0.5*x32x42
```

These results show that the level 1 parameter of `x1` cannot be estimated independently of the interaction term involving `x2` level 1 and `x3` level 2, the interaction term involving `x2` level 2 and `x3` level 1, the interaction term involving `x2` level 1 and `x4` level 1, the interaction term involving `x2` level 1 and `x4` level 2, the interaction term involving `x2` level 2 and `x4` level 1, and so on.

Note, however, that even these results are not really the full results because only two-way interactions are included. To really see the full extent of the aliasing, you need to add all three-way and the four-way interaction as follows:

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

The first part of the full aliasing structure is as follows:

```
                              Aliasing Structure

Intercept - 0.6667*x11x21x32 - 0.6667*x11x22x31 - 0.6667*x12x21x31 +
   2*x12x22x32 + 0.3333*x11x21x41 - 0.3333*x11x21x42 + 0.3333*x11x22x41 +
   x11x22x42 - 0.3333*x12x21x41 - x12x21x42 + x12x22x41 - x12x22x42 +
   0.3333*x11x31x41 - 0.3333*x11x31x42 - 0.3333*x11x32x41 - x11x32x42 +
   0.3333*x12x31x41 + x12x31x42 + x12x32x41 - x12x32x42 + 0.3333*x21x31x41 -
   0.3333*x21x31x42 + 0.3333*x21x32x41 + x21x32x42 - 0.3333*x22x31x41 -
   x22x31x42 + x22x32x41 - x22x32x42

x11 - x21x32 - x22x31 + 0.3333*x11x21x31 - x11x22x32 + x12x21x32 + x12x22x31 +
   0.5*x21x41 - 0.5*x21x42 + 0.5*x22x41 + 1.5*x22x42 + 0.1667*x11x21x41 +
   0.5*x11x21x42 - 0.5*x11x22x41 + 0.5*x11x22x42 - 0.5*x12x21x41 +
   0.5*x12x21x42 - 0.5*x12x22x41 - 1.5*x12x22x42 + 0.5*x31x41 - 0.5*x31x42 -
   0.5*x32x41 - 1.5*x32x42 - 0.1667*x11x31x41 - 0.5*x11x31x42 - 0.5*x11x32x41 +
   0.5*x11x32x42 - 0.5*x12x31x41 + 0.5*x12x31x42 + 0.5*x12x32x41 +
   1.5*x12x32x42 + 0.3333*x11x21x31x41 - 0.3333*x11x21x31x42 +
   0.3333*x11x21x32x41 + x11x21x32x42 - 0.3333*x11x22x31x41 - x11x22x31x42 +
   x11x22x32x41 - x11x22x32x42

x12 - 0.3333*x21x31 + x22x32 + 0.3333*x11x21x32 + 0.3333*x11x22x31 -
   0.3333*x12x21x31 + x12x22x32 - 0.1667*x21x41 - 0.5*x21x42 + 0.5*x22x41 -
   0.5*x22x42 - 0.1667*x11x21x41 + 0.1667*x11x21x42 - 0.1667*x11x22x41 -
   0.5*x11x22x42 - 0.1667*x12x21x41 - 0.5*x12x21x42 + 0.5*x12x22x41 -
   0.5*x12x22x42 + 0.1667*x31x41 + 0.5*x31x42 + 0.5*x32x41 - 0.5*x32x42 -
   0.1667*x11x31x41 + 0.1667*x11x31x42 + 0.1667*x11x32x41 + 0.5*x11x32x42 +
   0.1667*x12x31x41 + 0.5*x12x31x42 + 0.5*x12x32x41 - 0.5*x12x32x42 +
   0.3333*x12x21x31x41 - 0.3333*x12x21x31x42 + 0.3333*x12x21x32x41 +
   x12x21x32x42 - 0.3333*x12x22x31x41 - x12x22x31x42 + x12x22x32x41 -
   x12x22x32x42
```

The aliasing scheme, which is the new summary, is as follows:

```
                                 Aliasing Scheme
Estimable  Aliased
Effect     Effects

Intercept  x1*x2*x3 x1*x2*x4 x1*x3*x4 x2*x3*x4

x1         x2*x3 x2*x4 x3*x4 x1*x2*x3 x1*x2*x4 x1*x3*x4 x1*x2*x3*x4

x2         x1*x3 x1*x4 x3*x4 x1*x2*x3 x1*x2*x4 x2*x3*x4 x1*x2*x3*x4

x3         x1*x2 x1*x4 x2*x4 x1*x2*x3 x1*x3*x4 x2*x3*x4 x1*x2*x3*x4

x4         x1*x2 x1*x3 x2*x3 x1*x2*x4 x1*x3*x4 x2*x3*x4 x1*x2*x3*x4

NOTE: Some parameters in the estimable effects are aliased with some parameters
 in the aliased effects.  For effects with more than two levels, the aliasing
 scheme displayed here is potentially partial.  Specify EXAMINE=FULL ALIASING=n
 to see the full aliasing structure.
```

For the complicated designs that we use in practice, particularly for choice models, the aliasing structure and aliasing scheme are often very long and complicated. In many cases, they cannot be displayed because the size of the model gets unwieldy with m main effect parameters, $m(m-1)/2$ two-way interaction parameters, and so on.

We can use the %MktEx macro to create a resolution IV design (all main effects are estimable free of each other and free of all two-factor interactions, but some two-factor interactions are confounded with other two-factor interactions), for m two-level factors (where m is a multiple of 4), and evaluate it as follows:

```
%let m = 12;
%mktorth(range=n=2 * &m, options=lineage dups, maxlev=&m)

%mktex(2 ** &m, n=2 * &m, examine=aliasing=2,
       cat=mktdeslev(where=(index(compbl(design), "2 ** &m &m ** 1"))))
```

This example uses the %MktOrth macro to create the instructions for creating all designs in $2m$ runs, and then sends the %MktEx macro instructions for just the design with m two-level factors and an m-level factor, which produces the desired design. Often, the %MktEx macro has many ways to make a specified design and will not choose the one you have in mind unless you give it specific instructions such as we did here. This step creates a design from the first m columns of a design of the form:

$$\begin{array}{rc} \mathbf{H}_m & \ell_m \\ -\mathbf{H}_m & \ell_m \end{array}$$

$\mathbf{H}_m$ is a Hadamard matrix of order m, and ℓ_m is a column vector where $\ell'_m = [0\ 1\ 2\ ...\ m-1]$. The resulting design has m two-level factors in $2m$ runs and has the following aliasing scheme:

```
                                  Aliasing Scheme
Estimable  Aliased
Effect     Effects

Intercept

x1

x2

x3

x4

x5

x6

x7

x8

x9

x10

x11

x12

x1*x2      x1*x7 x1*x8 x1*x11 x1*x12 x2*x7 x2*x8 x2*x11 x2*x12 x3*x7 x3*x8
           x3*x9 x3*x11 x3*x12 x4*x7 x4*x8 x4*x10 x4*x11 x4*x12 x5*x6 x5*x7
           x5*x8 x5*x11 x5*x12 x6*x7 x6*x8 x6*x11 x6*x12 x7*x9 x7*x10 x8*x9
           x8*x10 x9*x11 x9*x12 x10*x11 x10*x12

x1*x3      x1*x7 x1*x8 x1*x10 x1*x11 x2*x6 x2*x7 x2*x8 x2*x10 x2*x11 x3*x7
           x3*x8 x3*x10 x3*x11 x4*x7 x4*x8 x4*x10 x4*x11 x4*x12 x5*x7 x5*x8
           x5*x9 x5*x10 x5*x11 x6*x7 x6*x8 x6*x10 x6*x11 x7*x9 x7*x12 x8*x9
           x8*x12 x9*x10 x9*x11 x10*x12 x11*x12

x1*x4      x1*x7 x1*x9 x1*x10 x1*x11 x2*x7 x2*x9 x2*x10 x2*x11 x2*x12 x3*x6
           x3*x7 x3*x9 x3*x10 x3*x11 x4*x7 x4*x9 x4*x10 x4*x11 x5*x7 x5*x8
           x5*x9 x5*x10 x5*x11 x6*x7 x6*x9 x6*x10 x6*x11 x7*x8 x7*x12 x8*x9
           x8*x10 x8*x11 x9*x12 x10*x12 x11*x12

x1*x5      x1*x7 x1*x9 x1*x11 x1*x12 x2*x7 x2*x9 x2*x10 x2*x11 x2*x12 x3*x7
           x3*x8 x3*x9 x3*x11 x3*x12 x4*x6 x4*x7 x4*x9 x4*x11 x4*x12 x5*x7
           x5*x9 x5*x11 x5*x12 x6*x7 x6*x9 x6*x11 x6*x12 x7*x8 x7*x10 x8*x9
           x8*x11 x8*x12 x9*x10 x10*x11 x10*x12
```

```
    x1*x6      x1*x8 x1*x9 x1*x10 x1*x11 x1*x12 x2*x6 x2*x8 x2*x9 x2*x10 x2*x11
               x2*x12 x3*x6 x3*x8 x3*x9 x3*x10 x3*x11 x3*x12 x4*x6 x4*x8 x4*x9
               x4*x10 x4*x11 x4*x12 x5*x6 x5*x8 x5*x9 x5*x10 x5*x11 x5*x12 x6*x7
               x7*x8 x7*x9 x7*x10 x7*x11 x7*x12

    x1*x7      x1*x8 x1*x9 x1*x10 x1*x11 x1*x12 x2*x3 x2*x4 x2*x5 x2*x6 x2*x7 x2*x8
               x2*x9 x2*x10 x2*x11 x2*x12 x3*x4 x3*x5 x3*x6 x3*x7 x3*x8 x3*x9
               x3*x10 x3*x11 x3*x12 x4*x5 x4*x6 x4*x7 x4*x8 x4*x9 x4*x10 x4*x11
               x4*x12 x5*x6 x5*x7 x5*x8 x5*x9 x5*x10 x5*x11 x5*x12 x6*x7 x6*x8
               x6*x9 x6*x10 x6*x11 x6*x12 x7*x8 x7*x9 x7*x10 x7*x11 x7*x12 x8*x9
               x8*x10 x8*x11 x8*x12 x9*x10 x9*x11 x9*x12 x10*x11 x10*x12 x11*x12

    NOTE: Some parameters in the estimable effects are aliased with some parameters
     in the aliased effects.  For effects with more than two levels, the aliasing
     scheme displayed here is potentially partial.  Specify EXAMINE=FULL ALIASING=n
     to see the full aliasing structure.
```

The aliasing scheme shows that we did in fact find a resolution IV design with all main effects estimable free of each other and free of all two-factor interactions. You can see by looking at the interaction terms in the estimable effects and those that are part of the x1*x7 aliasing scheme that in this case at least part of every two-way interaction is aliased with at least part of some other two-way interactions.

%MktEx Macro Notes

The %MktEx macro displays notes in the SAS log to show you what it is doing while it is running. Most of the notes that would normally come out of the macro's procedure and DATA steps are suppressed by default by an `options nonotes` statement. 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. This section describes the notes that are normally not suppressed.

The macro will usually start by displaying one of the following notes (filling in a value after n=):

```
NOTE: Generating the Hadamard design, n=.
NOTE: Generating the full-factorial design, n=.
NOTE: Generating the fractional-factorial design, n=.
NOTE: Generating the orthogonal array design, n=.
```

These messages tell you which type of orthogonal design the macro is constructing. The design might be the final design, or it might provide an initialization for the coordinate exchange algorithm. In some cases, it might not have the same number of runs, n, as the final design. Usually this step is fast, but constructing some fractional-factorial designs might be time consuming.

If the macro is going to use PROC OPTEX to search a candidate set, it will display the following note:

```
NOTE: Generating the candidate set.
```

This step will usually be fast. Next, when a candidate set is searched, the macro will display the following note, substituting in values for the ellipses:

```
NOTE: Performing ... searches of ... candidates.
```

This step might take a while depending on the size of the candidate set and the size of the design. When there are a lot of restrictions and a fractional-factorial candidate set is being used, the candidate set might be so restricted that it does not contain enough information to make the design. In that case, you will get the following message:

```
NOTE: The candidate-set initialization failed,
      but the MKTEX macro is continuing.
```

Even though part of the macro's algorithm failed, it is *not* a problem. The macro just goes on to the coordinate-exchange algorithm, which will almost certainly work better than searching any severely-restricted candidate set.

For large designs, you usually will want to skip the PROC OPTEX iterations. The macro might display the following note:

```
NOTE: With a design this large, you may get faster results with OPTITER=0.
```

Sometimes you will get the following note:

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

When the macro keeps finding the same maximum *D*-efficiency over and over again in different designs, it might stop early. This might mean that the macro has found the optimal design, or it might mean that the macro keeps finding a very attractive local optimum. Either way, it is unlikely that the macro will do any better. You can control this using the `stopearly=` option.

The macro has options that control the amount of time it spends trying different techniques. When time expires, the macro might switch to other techniques before it completes the usual maximum number of iterations. When this happens, the macro tells you with the following notes:

```
NOTE: Switching to a random initialization after ... minutes and
      ... designs.
NOTE: Quitting the algorithm search after ... minutes and ... designs.
NOTE: Quitting the design search after ... minutes and ... designs.
NOTE: Quitting the refinement step after ... minutes and ... designs.
```

When there are restrictions, or when you specify that you do not want duplicate runs, you can also specify `options=accept`. This means that you are willing to accept designs that violate the restrictions. With `options=accept`, the macro will tell you if the restrictions are not met with the following notes:

```
NOTE: The restrictions were not met.
NOTE: The design has duplicate runs.
```

`%MktEx` optimizes a ridged efficiency criterion, that is, a small number is added to the diagonal of $(\mathbf{X}'\mathbf{X})^{-1}$. Usually, the ridged criterion is virtually the same as the unridged criterion. When `%MktEx` detects that this is not true, it displays the following notes:

```
NOTE: The final   ridged D-efficiency criterion is ....
NOTE: The final unridged D-efficiency criterion is ....
```

The macro ends with one of the following two messages:

```
NOTE: The MKTEX macro used ... seconds.
NOTE: The MKTEX macro used ... minutes.
```

%MktEx Macro Iteration History

This section provides information about interpreting the iteration history table produced by the `%MktEx` macro. Some of the results are as follows:

```
                        Algorithm Search History

                                 Current          Best
Design    Row,Col  D-Efficiency  D-Efficiency  Notes
----------------------------------------------------------
     1      Start       82.2172       82.2172  Can
     1        End       82.2172

     2      Start       78.5039                Tab,Ran
     2      5  14       83.2098       83.2098
     2      6  14       83.3917       83.3917
     2      6  15       83.5655       83.5655
     2      7  14       83.7278       83.7278
     2      7  15       84.0318       84.0318
     2      7  15       84.3370       84.3370
     2      8  14       85.1449       85.1449
     .
     .
     .
     2        End       98.0624
     .
     .
     .

    12      Start       51.8915                Ran,Mut,Ann
    12        End       93.0214
     .
     .
     .


                         Design Search History

                                 Current          Best
Design    Row,Col  D-Efficiency  D-Efficiency  Notes
----------------------------------------------------------
     0    Initial       98.8933       98.8933  Ini

     1      Start       80.4296                Tab,Ran
     1        End       98.8567
     .
     .
     .
```

```
                        Design Refinement History

                                Current          Best
          Design    Row,Col  D-Efficiency  D-Efficiency  Notes
          ----------------------------------------------------------
               0    Initial       98.9438       98.9438  Ini

               1      Start       94.7490                Pre,Mut,Ann
               1        End       92.1336
               .
               .
               .
```

The first column, `Design`, is a design number. Each design corresponds to a complete iteration using a different initialization. Initial designs are numbered zero. The second column is `Row,Col`, which shows the design row and column that is changing in the coordinate-exchange algorithm. This column also contains `Start` for displaying the initial efficiency, `End` for displaying the final efficiency, and `Initial` for displaying the efficiency of a previously created initial design (perhaps created externally or perhaps created in a previous step). The `Current D-Efficiency` column contains the *D*-efficiency for the design including starting, intermediate and final values. The next column is `Best D-Efficiency`. Values are put in this column for initial designs and when a design is found that is as good as or better than the previous best design. The last column, `Notes`, contains assorted algorithm and explanatory details. Values are added to the table at the beginning of an iteration, at the end of an iteration, when a better design is found, and when a design first conforms to restrictions. The note `Conforms` is displayed when a design first conforms to the restrictions. From then on, the design continues to conform even though `Conforms` is not displayed in every line. Details of the candidate search iterations are not shown. Only the *D*-efficiency for the best design found through candidate search is shown.

The notes are as follows:

`Can`	– the results of a candidate-set search
`Tab`	– design table or catalog (orthogonal array, full, or fractional-factorial) initialization (full or in part)
`Ran`	– random initialization (full or in part)
`Unb`	– unbalanced initial design (usually in part)
`Ini`	– initial design
`Mut`	– random mutations of the initial design were performed
`Ann`	– simulated annealing was used in this iteration
`Pre`	– using previous best design as a starting point
`Conforms`	– design conforms to restrictions
`Violations`	– number of restriction violations

Often, more than one note appears. For example, the triples `Ran,Mut,Ann` and `Pre,Mut,Ann` frequently appear together.

The iteration history consists of three tables.

`Algorithm Search History`	– searches for a design and the best algorithm for this problem
`Design Search History`	– read the order from a data set
`Design Refinement History`	– tries to refine the best design

%MktEx Macro Options

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

Option	Description
`list`	(positional) list of the numbers of factor levels
	(positional) "help" or "?" displays syntax summary
`anneal=`*n1* < *n2* < *n3* >>	starting probability for annealing
`annealfun=`*function*	annealing probability function
`anniter=`*n1* < *n2* < *n3* >>	first annealing iteration
`balance=`*n*	maximum level-frequency range
`big=`*n* < *choose* >	size of big full-factorial design
`canditer=`*n1* < *n2* >	iterations for OPTEX designs
`cat=`*SAS-data-set*	input design catalog
`detfuzz=`*n*	determinants change increment
`examine=`< I > < V > <*aliasing* >	matrices that you want to examine
`exchange=`*n*	number of factors to exchange
`fixed=`*variable*	indicates runs that are fixed
`holdouts=`*n*	adds holdout observations
`imlopts=`*options*	IML PROC statement options
`init=`*SAS-data-set*	initial input experimental design
`interact=`*interaction-list*	interaction terms
`iter=`*n1* < *n2* < *n3* >>	maximum number of iterations
`levels=`*value*	assigning final factor levels
`maxdesigns=`*n*	maximum number of designs to make
`maxiter=`*n1* < *n2* < *n3* >>	maximum number of iterations
`maxstages=`*n*	maximum number of algorithm stages
`maxtime=`*n1* < *n2* < *n3* >>	approximate maximum run time
`mintry=`*n*	minimum number of rows to process
`mutate=`*n1* < *n2* < *n3* >>	mutation probability
`mutiter=`*n1* < *n2* < *n3* >>	first iteration to consider mutating
`n=`*n*	number of runs in the design
`options=accept`	accept designs that violate restrictions
`options=check`	checks the efficiency of the `init=` design
`options=file`	renders the design to a file
`options=int`	add an intercept variable `x0` to the design
`options=justinit`	stop processing after making the initial design
`options=largedesign`	stop after `maxtime=` minutes have elapsed
`options=lineage`	displays the lineage of the orthogonal array
`options=nodups`	eliminates duplicate runs
`options=nofinal`	skips displaying the final efficiency
`options=nohistory`	does not display the iteration history
`options=nooadups`	check for orthogonal array with duplicate runs
`options=noqc`	do not use the SAS/QC product
`options=nosort`	does not sort the design
`options=nox`	suppress the the `x`*n* scalars with restrictions
`options=quick`	`optiter=0`, `maxdesigns=2`, `unbalanced=0`, `tabiter=1`
`options=quickr`	`optiter=0`, `maxdesigns=1`, `unbalanced=0`, `tabiter=0`
`options=quickt`	`optiter=0`, `maxdesigns=1`, `unbalanced=0`, `tabiter=1`

Option	Description
`options=render`	displays the design compactly in the SAS listing
`options=refine`	with `init=`, never reinitializes
`options=resrep`	reports on the progress of the restrictions
`options=+-`	renders –1 as – and 1 as + in two-level factors
`options=3`	applies `options=+-` to 3-level factors
`options=512`	adds some designs in 512 runs
`optiter=`*n1 < n2 >*	OPTEX iterations
`order=`*value*	coordinate exchange column order
`out=`*SAS-data-set*	output experimental design
`outall=`*SAS-data-set*	output data set with all designs
`outeff=`*SAS-data-set*	output data set with final efficiency
`outr=`*SAS-data-set*	randomized output experimental design
`partial=`*n*	partial-profile design
`repeat=`*n1 n2 n3*	times to iterate on a row
`reslist=`*list*	constant matrix list
`resmac=`*macro-name*	constant matrix creation macro
`restrictions=`*macro-name*	restrictions macro
`ridge=`*n*	ridging factor
`seed=`*n*	random number seed
`stopearly=`*n*	the macro can stop early
`tabiter=`*n1 < n2 >*	design table initialization iterations
`tabsize=`*n*	orthogonal array size
`target=`*n*	target efficiency criterion
`unbalanced=`*n1 < n2 >*	unbalance initial design iterations

Help Option

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

```
%mktex(help)
%mktex(?)
```

Required Options

The `n=` options is required, and the `list` option is almost always required.

list

specifies a list of the numbers of levels of all the factors. For example, for 3 two-level factors specify either `2 2 2` or `2 ** 3`. Lists of numbers, like `2 2 3 3 4 4` or a *levels**number of factors* syntax like: `2**2 3**2 4**2` can be used, or both can be combined: `2 2 3**4 5 6`. The specification `3**4` means 4 three-level factors. Note that the factor list is a positional parameter. This means that if it is specified, it must come first, and unlike all other parameters, it is not specified after a name and an equal sign. Usually, you have to specify a list. However, in some cases, you can just specify `n=` and omit the list and a default list is implied. For example, `n=18` implies a list of `2 3 ** 7`. When the list is omitted, and if there are no interactions, restrictions, or duplicate exclusions, then by default there are no OPTEX iterations (`optiter=0`).

n= *n*

specifies the number of runs in the design. You must specify `n=`. The following example uses the `%MktRuns` macro to get suggestions for values of `n=`:

```
   %mktruns(4 2 ** 5 3 ** 5)
```

In this case, this macro suggests several sizes including orthogonal designs with `n=72` and `n=144` runs and some smaller nonorthogonal designs including `n=36, 24, 48, 60`.

Basic Options

This next group of options contains some of the more commonly used options.

balance= *n*

specifies the maximum allowable level-frequency range. You use this option to tell the macro that it should make an extra effort to ensure that the design is balanced or at least nearly balanced. Specify a positive integer, usually 1 or 2, that specifies the degree of imbalance that is acceptable. The `balance=n` option specifies that for each factor, a difference between the frequencies of the most and least frequently occurring levels should be no larger than n.

When you specify `balance=`, particularly if you specify `balance=0`, you should also specify `mintry=` (perhaps something like `mintry=5 * n`, or `mintry=10 * n`). When `balance=` and `mintry=mt` are both specified, then the balance restrictions are ignored for the first `mt - 3 * n / 2` passes through the design. During this period, the badness function for the balance restrictions is set to 1 so that `%MktEx` knows that the design does not conform. After that, all restrictions are considered. The `balance=` option works best when its restrictions are imposed on a reasonably efficient design not an inefficient initial design. You can specify `balance=0`, without specify `mintry=`, however, this might not be a good idea because the macro needs the flexibility to have imbalance as it refines the design. Often, the design actually found will be better balanced than your `balance=n` specification would require. For this reason, it is good to start by specifying a value larger than the minimum acceptable value. The larger the value, the more freedom the algorithm has to optimize both balance and efficiency.

The `balance=` option works by adding restrictions to the design. The badness of each column (how far each column is from conforming to the balance restrictions) is evaluated and the results stored in a scalar `__bbad`. When you specify other restrictions, this is added to the `bad` value created by your restrictions macro. You can use your restrictions macro to change or differentially weight `__bbad` before the final addition of the components of design badness takes place (see page 1065).

The `%MktEx` macro usually does a good job of producing nearly balanced designs, but if balance is critically important, and your designs are not balanced enough, you can sometimes achieve better balance by specifying `balance=`, but usually at the price of worse efficiency, sometimes much worse. By default, no additional restrictions are added. Another approach is to instead use the `%MktBal` macro, which for main effects plans with no restrictions, produces designs that are guaranteed to have optimal balance.

examine= < `I` > < `V` > < `aliasing=`*n* > < `full` > < `main` >
specifies the matrices that you want to examine. The option `examine=I` displays the information matrix, $\mathbf{X}'\mathbf{X}$; `examine=V` displays the variance matrix, $(\mathbf{X}'\mathbf{X})^{-1}$; and `examine=I V` displays both. By default, these matrices are not displayed.

Specify `examine=aliasing=`*n* to examine the aliasing structure of the design. If you specify `examine= aliasing=2`, `%MktEx` will display the terms in the model and how they are aliased with up to two-factor interactions. More generally, with `examine=aliasing=`*n*, up to *n*-factor interactions are displayed. You can also specify `full` (e.g. `examine=aliasing=2 full`) to see the full (and often much more complicated) aliasing structure that PROC GLM produces directly. You can also specify `main` to see only the estimable functions that begin with main effects, and not the ones that begin with interactions. Interactions are still used with `examine=aliasing=2 main` and larger values of `aliasing=`. This option just removes some of the output.

Note that the `aliasing=`*n* option is resource intensive for larger problems. For some large problems, one of the underlying procedures might detect that the problem is too big, immediately issue an error, and quit. For other large problems, it might simply take a very long time before completing or printing an error due to insufficient resources. The number of two-way interaction terms is a quadratic function of the number of main effects, so it is not possible to print the aliasing structure even for some very reasonably sized main-effects designs.

interact= *interaction-list*
specifies interactions that must be estimable. By default, no interactions are guaranteed to be estimable. Examples:
`interact=x1*x2`
`interact=x1*x2 x3*x4*x5`
`interact=x1|x2|x3|x4|x5@2`
`interact=@2`

The interaction syntax is in most ways like PROC GLM's and many of the other modeling procedures. It uses "`*`" for simple interactions (`x1*x2` is the interaction between `x1` and `x2`), "`|`" for main effects and interactions (`x1|x2|x3` is the same as `x1 x2 x1*x2 x3 x1*x3 x2*x3 x1*x2*x3`) and "`@`" to eliminate higher-order interactions (`x1|x2|x3@2` eliminates `x1*x2*x3` and is the same as `x1 x2 x1*x2 x3 x1*x3 x2*x3`). The specification "`@2`" creates main effects and two-way interactions. Unlike PROC GLM's syntax, some short cuts are permitted. For the factor names, you can specify either the actual variable names (for example, `x1*x2` ...) or you can just specify the factor number without the "`x`" (for example, `1*2`). You can also specify `interact=@2` for all main effects and two-way interactions omitting the `1|2|`.... The following three specifications are equivalent:

```
   %mktex(2 ** 5, interact=@2, n=16)
   %mktex(2 ** 5, interact=1|2|3|4|5@2, n=16)
   %mktex(2 ** 5, interact=x1|x2|x3|x4|x5@2, n=16)
```

If you specify `interact=@2`, and if your specification matches a regular fractional-factorial design, then a resolution V design is requested. If instead you specify the full interaction list, (e.g. `interact=x1 | x2 | x3 | x4 | x5@2`) then the less-direct approach of requesting a design with the full list of interaction terms is taken, which in some cases might not work as well as directly requesting a resolution V design.

mintry= *n*

specifies the minimum number of rows to process before giving up for each design. For example, to ensure that the macro passes through each row of the design at least five times, you can specify `mintry=5 * n`. You can specify a number or a DATA step expression involving n (rows) and m (columns). By default, the macro will always consider at least n rows. This option can be useful with certain restrictions, particularly with `balance=`. When `balance=` and `mintry=mt` are both specified, then the balance restrictions are ignored for the first `mt - 3 * n / 2` passes through the design. During this period, the badness function for the balance restrictions is set to 1 so that `%MktEx` knows that the design does not conform. After that, all restrictions are considered. The `balance=` option works best when its restrictions are imposed on a reasonably efficient design not an inefficient initial design.

The `%MktEx` macro sometimes displays the following message:

```
WARNING: It may be impossible to meet all restrictions.
```

This message is displayed after `mintry=`*n* rows are passed without any success. Sometimes, it is premature to expect any success during the first pass. When you know this, you can specify this option to prevent that warning from coming out.

options= *options-list*

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

`accept`
lets the macro output designs that violate restrictions imposed by `restrictions=`, `balance=`, or `partial=`, or have duplicates with `options=nodups`. Normally, the macro will not output such designs. With `options=accept`, a design becomes eligible for output when the macro can no longer improve on the restrictions or eliminate duplicates. Without `options=accept`, a design is only eligible when all restrictions are met and all duplicates are eliminated.

`check`
checks the efficiency of a given design, specified in `init=`, and disables the `out=`, `outr=`, and `outall=` options. If `init=` is not specified, `options=check` is ignored.

`file`
renders the design to a file with a generated file name. For example, if the design $2^{11}3^{12}$ in 36 runs is requested, the generated file name is: `OA(36,2^11,3^12)`. This option is ignored unless `options=render` is specified.

`int`
add an intercept to the design, variable, `x0`.

`justinit`
specifies that the macro should stop processing as soon as it is done making the initial design, even if that design would not normally be the final design. Usually, this design is an orthogonal array or some function of an orthogonal array (e.g. some three-level factors could be recoded into two-level factors), but there are no guarantees. Use this option when you want to output the initial design, for example, if you want to see the orthogonal but unbalanced design that `%MktEx` sometimes uses as an initial design. The `options=justinit` specification implies `optiter=0` and `outr=`. Also, `options=justinit nofinal` both stops the processing and prevents the final design from being evaluated. Particularly when you specify `options=nofinal`, you must ensure that this design has a suitable efficiency.

`largedesign`
lets the macro stop after `maxtime=` minutes have elapsed in the coordinate exchange algorithm. Typically, you would use this with `maxstages=1` and other options that make the algorithm run faster. By default, the macro checks time after it finishes with a design. With this option, the macro checks the time at the end of each row, after it has completed the first full pass through the design, and after any restrictions have been met, so the macro might stop before *D*-efficiency has converged. For really large problems and problems with restrictions, this option might make the macro run much faster but at a price of lower *D*-efficiency. For example, for large problems with restrictions, you might just want to try one run through the coordinate exchange algorithm with no candidate set search, orthogonal arrays, or mutations.

`lineage`
displays the lineage or "family tree" of the orthogonal array. For example, the lineage of the design $2^1 3^{25}$ in 54 runs is `54 ** 1 :  54 ** 1 > 3 ** 20 6 ** 1 9 ** 1 :  9 ** 1 > 3 ** 4 :  6 ** 1 > 2 ** 1 3 ** 1`. This states that the design starts as a single 54-level factor, then 54^1 is replaced by $3^{20}6^1 9^1$, 9^1 is replaced by 3^4, and finally 6^1 is replaced by $2^1 3^1$ to make the final design.

`nodups`
eliminates duplicate runs.

`nofinal`
skips calling PROC OPTEX to display the efficiency of the final experimental design.

`nohistory`
does not display the iteration history.

`nooadups`
for orthogonal array construction, checks to see if a design with duplicate runs is created. If so, it tries using other factors from the larger orthogonal array to see if that helps avoid duplicates. There is no guarantee that this option will work. If you select only a small subset of the columns of an orthogonal array, duplicates might be unavoidable. If you really wish to ensure no duplicates, even at the expense of nonorthogonality, you must also specify `options=nodups`.

`noqc`
specifies that you do not have the SAS/QC product, so the `%MktEx` macro should try to get by without it. This means it tries to get by using the coordinate exchange and orthogonal array code without using PROC FACTEX and PROC OPTEX. The `%MktEx` macro will skip generating a candidate set, searching the candidate set, and displaying the final efficiency values. This is equivalent to specifying `optiter=0` and `options=nofinal`. This option also eliminates the check to see if the SAS/QC product is available. For some problems, the SAS/QC product is not necessary. For others, for example, for some orthogonal arrays in 128 runs, it is necessary. For other problems still, SAS/QC is not necessary, but the macro might find better designs if SAS/QC is available (for example, models with interactions and candidate sets on the order of a few thousand observations).

`nosort`
does not sort the design. One use of this option is with orthogonal arrays and Hadamard matrices. Some Hadamard matrices are generated with a banded structure that is lost when the design is sorted. If you want to see the original structure, and not just a design, specify `options=nosort`.

`nox`
suppresses the creation of `x1`, `x2`, `x3`, and so on, for use with the restrictions macro. If you are not using these names in your restrictions macro, specifying `options=nox` can make the macro run somewhat more efficiently. By default, `x1`, `x2`, `x3`, and so on are available for use.

`quick`
sets `optiter=0`, `maxdesigns=2`, `unbalanced=0`, and `tabiter=1`. This option provides a quick run that makes at most two design using coordinate exchange iterations—one using an initial design based on the orthogonal array table (catalog), and if necessary, one with a random initialization.

`quickr`
sets `optiter=0`, `maxdesigns=1`, `unbalanced=0`, and `tabiter=0`. This option provides an even quicker run than `options=quick` creating only one design using coordinate exchange and a random initialization. The "`r`" in `quickr` stands for random. You can use this option when you think using an initial design from the orthogonal array catalog will not help.

`quickt`
sets `optiter=0`, `maxdesigns=1`, `unbalanced=0`, and `tabiter=1`. This option provides an even quicker run than `options=quickr` with one design found by coordinate exchange using a design from the orthogonal array table (catalog) in the initialization. The "`t`" in `quickt` stands for table.

`render`
displays the design compactly in the SAS listing. If you specify `options=render file`, then the design is instead rendered to a file whose name represents the design specification. For example, if the design $2^{11}3^{12}$ in 36 runs is requested, the generated file name is: `OA(36,2^11,3^12)`.

`refine`
specifies that with an `init=` design data set with at least one nonpositive entry, each successive design iteration tries to refine the best design from before. By default, the part of the design that is not fixed is randomly reinitialized each time. The default strategy is usually superior.

`resrep`
reports on the progress of the restrictions. You should specify this option with problems with lots of restrictions. Always specify this option if you find that `%MktEx` is unable to make a design that conforms to the restrictions. By default, the iteration history is not displayed for the stage where `%MktEx` is trying to make the design conform to the restrictions. Specify `options=resrep` when you want to see the progress in making the design conform.

`+-`
with render, displays –1 as '–' and 1 as '+' in two-level factors. This option is typically used with `levels=i` for displaying Hadamard matrices.

`3`
modifies `options=+-` to apply to three-level factors as well: –1 as '–', 0 as '0', and 1 as '+'.

`512`
adds some larger designs in 512 runs with mixes of 16, 8, 4, and 2-level factors to the catalog, which gives added flexibility in 512 runs at a cost of potentially *much* slower run time. This option replaces the default $4^{160}32^1$ parent with $16^{32}32^1$ and adds over 60,000 new designs to the catalog. Many of these designs are automatically available with PROC FACTEX, so do not use this option unless you have first tried and failed to find the design without it.

partial= *n*

specifies a partial-profile design (Chrzan and Elrod 1995). The default is an ordinary linear model design. Specify, for example, `partial=4` if you only want 4 attributes to vary in each row of the design (except the first run, in which none vary). This option works by adding restrictions to the design (see `restrictions=`) and specifying `order=random` and `exchange=2`. The badness of each row (how far each row is from conforming to the partial-profile restrictions) is evaluated and the results stored in a scalar `__pbad`. When you specify other restrictions, this is added to the `bad` value created by your restrictions macro. You can use your restrictions macro to change or differentially weight `__pbad` before the final addition of the components of design badness takes place (see page 1065). Because of the default `exchange=2` with partial-profile designs, the construction is slow, so you might want to specify `maxdesigns=1` or other options to make `%MktEx` run faster. For large problems, you might get faster but less good results by specifying `order=seqran`. Specifying `options=accept` or `balance=` with `partial=` is *not* a good idea. The following steps create and display the first part of a partial-profile design with twelve factors, each of which has three levels that vary and one level that means the attribute is not shown:

```
%mktex(4 ** 12,                     /* 12 four-level factors             */
       n=48,                        /* 48 profiles                       */
       partial=4,                   /* four attrs vary                   */
       seed=205,                    /* random number seed                */
       maxdesigns=1)                /* just make one design              */

%mktlab(data=randomized, values=. 1 2 3, nfill=99)

options missing=' ';
proc print data=final(obs=10); run;
options missing='.';
```

The first part of the design is as follows:

```
 Obs   x1    x2    x3    x4    x5    x6    x7    x8    x9    x10    x11    x12

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

reslist= *list*

specifies a list of constant matrices. Begin all names with an underscore to ensure that they do not conflict with any of names that `%MktEx` uses. If you specify more than one name, then names must be separated by commas. Example: `reslist=%str(_a, _b)`.

resmac= *macro-name*

specifies the name of a macro that creates the matrices named in the `reslist=` option. Begin all names including all intermediate matrix names with an underscore to ensure that they do not conflict with any of the names that `%MktEx` uses.

The `reslist=` and `resmac=` options can be used jointly for certain complicated restrictions to set up some constant matrices that the restrictions macro can use. Since the restrictions macro is called a lot, anything you can do only once helps speed up the algorithm.

Another way you can use these options is when you want to access a `%MktEx` matrix in your restrictions macro that you normally could not access. This would require knowledge of the internal workings of the `%MktEx` macro, so it is not a capability that you would usually need.

restrictions= *macro-name*
specifies the name of a macro that places restrictions on the design. By default, there are no restrictions. If you have restrictions on the design, what combinations can appear with what other combinations, then you must create a macro that creates a variable called `bad` that contains a numerical summary of how bad the row of the design is. When everything is fine, set `bad` to zero. Otherwise set `bad` to a larger value that is a function of the number of restriction violations. The `bad` variable must not be binary (0 – ok, 1 – bad) unless there is only one simple restriction. You must set `bad` so that the `%MktEx` macro knows if the changes it is considering are moving the design in the right direction. See page 1079 for examples of restrictions. The macro must consist of PROC IML statements and possibly some macro statements.

When you have restrictions, you should usually specify `options=resrep` so that you can get a report on the restriction violations in the iteration history. This can be a great help in debugging your restrictions macro. Also, be sure to check the log when you specify `restrictions=`. The macro cannot always ensure that your statements are syntax-error free and stop if they are not. There are many options that can impose restrictions, including `restrictions=`, `options=nodups`, `balance=`, `partial=`, and `init=`. If you specify more than one of these options, be sure that the combination makes sense, and be sure that it is possible to simultaneously satisfy all of the restrictions.

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

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

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

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

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

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

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

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

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

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

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

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

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

Other than that, you can create intermediate variables without worrying about conflicts with the names in the macro. The levels of the factors for one row of the experimental design are stored in a vector `x`, and the first level is always 1, the second always 2, and so on. All restrictions must be defined in terms of `x[j]` (or alternatively, `x1`, `x2`, ..., and perhaps the other matrices). For example, if there are 5 three-level factors and if it is bad if the level of a factor equals the level for the following factor, you can create a macro `restrict` like the following and specify `restrictions=restrict` when you invoke the `%MktEx` macro:

```
%macro restrict;
   bad = (x1 = x2) +
         (x2 = x3) +
         (x3 = x4) +
         (x4 = x5);
   %mend;
```

Note that you specify just the macro name and no percents on the `restrictions=` option. Also note that IML does not have the full set of Boolean operators that the DATA step and other parts of SAS have. For example, these are *not* available: `OR AND NOT GT LT GE LE EQ NE`. Note that the expression `a <= b <= c` is perfectly valid in IML, but its meaning in IML is different than and less reasonable than its meaning in the DATA step. The DATA step expression checks to see if `b` is in the range of `a` to `c`. In contrast, the IML expression `a <= b <= c` is exactly the same as `(a <= b) <= c`, which evaluates `(a <= b)`, and sets the result to 0 (false) or 1 (true). Then IML compares the resulting 0 or 1 to see if it is less than or equal to `c`.

The operators you can use, along with their meanings, are as follows:

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

Restrictions can substantially slow down the algorithm.

With restrictions, the `Current D-Efficiency` column of the iteration history table might contain values larger than the `Best D-Efficiency` column. This is because the design corresponding to the current *D*-efficiency might have restriction violations. Values are only reported in the best *D*-efficiency column after all of the restriction violations have been removed. You can specify `options=accept` with `restrictions=` when it is okay if the restrictions are not met.

See page 1079 for more information about restrictions. See pages 471 and 604 for examples of restrictions. There are many examples of restrictions in the partial-profile examples starting on page 595.

seed= *n*
specifies the random number seed. By default, `seed=0`, and clock time is used to make 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. For most orthogonal and balanced designs, the results should be reproducible. When computerized searches are done, it is likely that you will not get the same design across different computers, operating systems and different SAS and macro releases, although you would expect the efficiency differences to be slight.

Data Set Options

These next options specify the names of the input and output data sets.

cat= *SAS-data-set*
specifies the input design catalog. By default, the `%MktEx` macro automatically runs the `%MktOrth` macro to get this catalog. However, many designs can be made in multiple ways, so you can instead run `%MktOrth` yourself, select the exact design that you want, and specify the resulting data set in the `cat=` option. The catalog data set for input to `%MktEx` is the `outlev=` data set from the `%MktOrth` macro, which by default is called MKTDESLEV. Be sure to specify `options=dups lineage` when you run the `%MktOrth` macro. For example, the design 2^{71} in 72 runs can be made from either $2^{36}36^1$ or $2^{68}4^1$.

The following example shows how to select the $2^{36}36^{1}$ parent:

```
%mktorth(range=n=72, options=dups lineage)

proc print data=mktdeslev; var lineage; run;

data lev;
   set mktdeslev(where=(x2 = 71 and index(lineage, '2 ** 36 36 ** 1')));
   run;

%mktex(2 ** 71,                    /* 71 two-level factors              */
       n=72,                       /* 72 runs                           */
       cat=lev,                    /* OA catalog comes from lev data set  */
       out=b)                      /* output design                     */
```

The results of the following steps (not shown) show that you are in fact getting a design that is different from the default:

```
%mktex(2 ** 71, n=72, out=a)

proc compare data=a compare=b noprint note;
   run;
```

init= *SAS-data-set*

specifies the initial input experimental design. If all values in the initial design are positive, then a first step evaluates the design, the next step tries to improve it, and subsequent steps try to improve the best design found. However, if any values in the initial design are nonpositive (or missing) then a different approach is used. The initial design can have three types of values:

- positive integers are fixed and constant and will not change throughout the course of the iterations.
- zero and missing values are replaced by random values at the start of each new design search and can change throughout the course of the iterations.
- negative values are replaced by their absolute value at the start of each new design attempt and can change throughout the course of the iterations.

When absolute orthogonality and balance are required in a few factors, you can fix them in advance. The following steps illustrate how:

```
* Get first four factors;
%mktex(8 6 2 2, n=48)

* Flag the first four as fixed and set up to solve for the next six;
data init;
   set design;
   retain x5-x10 .;
   run;

* Get the last factors holding the first 4 fixed;
%mktex(8 6 2 2 4 ** 6,              /* append 4 ** 6 to 8 6 2 2            */
       n=48,                        /* 48 runs                             */
       init=init,                   /* initial design                      */
       maxiter=100)                 /* 100 iterations                      */

%mkteval(data=design)
```

Alternatively, you can use `holdouts=` or `fixed=` to fix just certain rows.

out= *SAS-data-set*

specifies the output experimental design. The default is `out=Design`. By default, this design is sorted unless you specify `options=nosort`. This is the output data set to look at in evaluating the design. See the `outr=` option for a randomized version of the same design, which is usually more suitable for actual use. Specify a null value for `out=` if you do not want this data set created. 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.

outall= *SAS-data-set*

specifies the output data set containing all designs found. By default, this data set is not created. This data set contains the design number, efficiency, and the design. If you use this option you can break while the macro is running and still recover the best design found so far. Note, however, that designs are only stored in this data set when they are done being processed. For example, design 1 is stored once at the end, not every time an improvement is found along the way. Make sure you store the data set in a permanent SAS data set. If, for example, your macro is currently working on the tenth design, with this option, you could break and get access to designs 1 through 9.

The following steps illustrate how you can use this option:

```
%mktex(2 2 2 3 3 3, n=18, outall=sasuser.a)

proc means data=sasuser.a noprint;
   output out=m max(efficiency)=m;
   run;

data best;
   retain id .;
   set sasuser.a;
   if _n_ eq 1 then set m;
   if nmiss(id) and abs(m - efficiency) < 1e-12 then do;
      id = design;
      put 'NOTE: Keeping design ' design +(-1) '.';
      end;
   if id eq design;
   keep design efficiency x:;
   run;

proc print; run;
```

outeff= *SAS-data-set*
specifies the output data set with the final efficiencies, the method used to find the design, and the initial random number seed. By default, this data set is not created.

outr= *SAS-data-set*
specifies the randomized output experimental design. The default is `outr=Randomized`. Levels are randomly reassigned within factors, and the runs are sorted into a random order. Neither of these operations affects efficiency. When `restrictions=` or `partial=` is specified, only the random sort is performed. Specify a null value for `outr=` if you do not want a randomized design created. 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.

Iteration Options

These next options control some of the details of the iterations. The macro can perform three sets of iterations. The `Algorithm Search` set of iterations looks for efficient designs using three different approaches. It then determines which approach appears to be working best and uses that approach exclusively in the second set of `Design Search` iterations. The third set or `Design Refinement` iterations tries to refine the best design found so far by using level exchanges combined with random mutations and simulated annealing. Some of these options can take three arguments, one for each set of iterations.

The first set of iterations can have up to three parts. The first part uses either PROC PLAN or PROC FACTEX followed by PROC OPTEX, all called through the `%MktDes` macro, to create and search a candidate set for an optimal initial design. The second part might use an orthogonal array or fractional-factorial design as an initial design. The next part consists of level exchanges starting with random initial designs.

In the first part, if the full-factorial design is small and manageable (arbitrarily defined as < 5185 runs), it is used as a candidate set, otherwise a fractional-factorial candidate set is used. The macro tries `optiter=` iterations to make an optimal design using the `%MktDes` macro and PROC OPTEX.

In the second part, the macro tries to generate and improve a standard orthogonal array or fractional-factorial design. Sometimes, this can lead immediately to an optimal design, for example, with $2^{11}3^{12}$ and $n = 36$. In other cases, when only part of the desired design matches some standard design, only part of the design is initialized with the standard design and multiple iterations are run using the standard design as a partial initialization with the rest of the design randomly initialized.

In the third part, the macro uses the coordinate-exchange algorithm with random initial designs.

The following iteration options are available:

anneal= *n1 < n2 < n3 >>*

specifies the starting probability for simulated annealing in the coordinate-exchange algorithm. The default is `anneal=.05 .05 .01`. Specify a zero or null value for no annealing. You can specify more than one value if you would like to use a different value for the algorithm search, design search, and design refinement iterations. When you specify a value greater than zero and less than one, for example, 0.1 the design is permitted to get worse with decreasing probability as the number of iterations increases. This often helps the algorithm overcome local efficiency maxima. Permitting efficiency to decrease can help get past the bumps in the efficiency function.

Examples: `anneal=` or `anneal=0` specifies no annealing, `anneal=0.1` specifies an annealing probability of 0.1 during all three sets of iterations, `anneal=0 0.1 0.05` specifies no annealing during the initial iterations, an annealing probability of 0.1 during the search iterations, and an annealing probability of 0.05 during the refinement iterations.

anniter= *n1 < n2 < n3 >>*

specifies the first iteration to consider using annealing on the design. The default is `anniter=. . .`, which means that the macro chooses the values to use. The default is the first iteration that uses a fully random initial design in each of the three sets of iterations. Hence, by default, there is no random annealing in any part of the initial design when part of the initial design comes from an orthogonal design.

canditer= *n1 < n2 >*

specifies the number of coordinate-exchange iterations that are used to try to improve a candidate-set based, OPTEX-generated initial design. The default is `canditer=1 1`. Note that `optiter=` controls the number of OPTEX iterations. Unless you are using annealing or mutation in the `canditer=` iterations (by default you are not) or unless you are using `options=nodups`, do not change theses values. The default value of `canditer=1 1`, along with the default `mutiter=` and `anniter=` values of missing, mean that the results of the OPTEX iterations are presented once in the algorithm iteration history, and if appropriate, once in the design search iteration history. Furthermore, by default, OPTEX generated designs are not improved with level exchanges except in the design refinement phase.

maxdesigns= *n*

specifies that the macro should stop after `maxdesigns=` designs have been created. This option is useful with big, slow problems with restrictions. It is also useful as you are developing your design code. At first, just make one or a few designs, then when all of your code is finalized, let the macro run longer. You could specify, for example, `maxdesigns=3` and `maxtime=0` and the macro would perform one candidate-set-based iteration, one orthogonal design initialization iteration, and one random initialization iteration and then stop. By default, this option is ignored and stopping is based on the other iteration options. For large designs with restrictions, a typical specification is `options=largedesign quickr`, which is equivalent to `optiter=0, tabiter=0, unbalanced=0, maxdesigns=1, options=largedesign`.

maxiter= *n1 < n2 < n3 >>*
iter= *n1 < n2 < n3 >>*

specifies the maximum number of iterations or designs to generate. The default is `maxiter=21 25 10`. With larger values, the macro tends to find better designs at a cost of slower run times. You can specify more than one value if you would like to use a different value for the algorithm search, design search, and design refinement iterations. The second value is only used if the second set of iterations consists of coordinate-exchange iterations. Otherwise, the number of iterations for the second set is specified with the `tabiter=`, or `canditer=` and `optiter=` options. If you want more iterations, be sure to set the `maxtime=` option as well, because iteration stops when the maximum number of iterations is reached or the maximum amount of time, whichever comes first. Examples: `maxiter=10` specifies 10 iterations for the initial, search, and refinement iterations, and `maxiter=10 10 5` specifies 10 initial iterations, followed by 10 search iterations, followed by 5 refinement iterations.

maxstages= *n*

specifies that the macro should stop after `maxstages=` algorithm stages have been completed. This option is useful for big designs, and for times when the macro runs slowly, for example, with restrictions. You could specify `maxstages=1` and the macro will stop after the algorithm search stage, or `maxstages=2` and the macro will stop after the design search stage. The default is `maxstages=3`, which means the macro will stop after the design refinement stage.

maxtime= *n1 < n2 < n3 >>*

specifies the approximate maximum amount of time in minutes to run each phase. The default is `maxtime=10 20 5`. When an iteration completes (a design is completed), if more than the specified amount of time has elapsed, the macro quits iterating in that phase. Usually, run time is no more than 10% or 20% larger than the specified values. However, for large problems, with restrictions, and with `exchange=` values other than 1, run time might be quite a bit larger than the specified value, since the macro only checks time after a design finishes.

You can specify more than one value if you would like to use a different value for the algorithm search, design search, and design refinement iterations. By default, the macro spends up to 10 minutes on the algorithm search iterations, 20 minutes on the design search iterations, and 5 minutes in the refinement stage. Most problems run in much less time than this. Note that the second value is ignored for OPTEX iterations since OPTEX does not have any timing options. This option also affects, in the algorithm search iterations, when the macro switches between using an orthogonal initial design to using a random initial design. If the macro is not done using orthogonal initializations, and one half of the first time value has passed, it switches. Examples: `maxtime=60` specifies up to one hour for each phase. `maxtime=20 30 10` specifies 20 minutes for the first phase and 30 minutes for the second, and 10 for the third.

The option `maxtime=0` provides a way to get a quick run, with no more than one iteration in each phase. However, even with `maxtime=0`, run time can be several minutes or more for large problems. See the `maxdesigns=` and `maxstages=` options for other ways to drastically cut run time for large problems. If you specify really large time values (anything more than hours), you probably need to also specify `optiter=` since the default values depend on `maxtime=`.

mutate= *n1 < n2 < n3 >>*

specifies the probability at which each value in an initial design can mutate or be assigned a different random value before the coordinate-exchange iterations begin. The default is `mutate=.05 .05 .01`. Specify a zero or null value for no mutation. You can specify more than one value if you would like to use a different value for the algorithm search, design search, and design refinement iterations. Examples: `mutate=` or `mutate=0` specifies no random mutations. The `mutate=0.1` option specifies a mutation probability of 0.1 during all three sets of iterations. The `mutate=0 0.1 0.05` option specifies no mutations during the first iterations, a mutation probability of 0.1 during the search iterations, and a mutation probability of 0.05 during the refinement iterations.

mutiter= *n1 < n2 < n3 >>*

specifies the first iteration to consider mutating the design. The default is `mutiter=. . .`, which means that the macro chooses values to use. The default is the first iteration that uses a fully random initial design in each of the three sets of iterations. Hence, by default, there are no random mutations of any part of the initial design when part of the initial design comes from an orthogonal design.

optiter= *n1 < n2 >*

specifies the number of iterations to use in the OPTEX candidate-set based searches in the algorithm and design search iterations. The default is `optiter=. .`, which means that the macro chooses values to use. When the first value is "`.`" (missing), the macro will choose a value usually no smaller than 20 for larger problems and usually no larger than 200 for smaller problems. However, `maxtime=` values other than the defaults can make the macro choose values outside this range. When the second value is missing, the macro will choose a value based on how long the first OPTEX run took and the value of `maxtime=`, but no larger than 5000. When a missing value is specified for the first `optiter=` value, the default, the macro might choose to not perform any OPTEX iterations to save time if it thinks it can find a perfect design without them.

repeat= *n1 n2 n3*

specifies the maximum number of times to repeatedly work on a row to eliminate restriction violations. The default value of `repeat=25 . .` specifies that a row should be worked on up to 25 times to eliminate violations. The second value is the place in the design refinement where this processing starts. This is based on a zero-based total number of rows processed so far. This is like a zero-based row index, but it never resets within a design. The third value is the place where this extra repeated processing stops. Let m be the `mintry=`m value, which by default is n, the number of rows. By default, when the second value is missing, the process starts after m rows have been processed (the second complete pass through the design). By default, the process stops after `m + 10 * n` rows have been processed where m is the second (specified or derived) `repeat=` value.

tabiter= *n1 < n2 >*
specifies the number of times to try to improve an orthogonal or fractional-factorial initial design. The default is `tabiter=10 200`, which means 10 iterations in the algorithm search and 200 iterations in the design search.

unbalanced= *n1 < n2 >*
specifies the proportion of the `tabiter=` iterations to consider using unbalanced factors in the initial design. The default is `unbalanced=.2 .1`. One way that unbalanced factors occur is through coding down. Coding down, for example, creates a three-level factor from a four-level factor: (1 2 3 4) ⇒ (1 2 3 3) or a two-level factor from a three-level factor: (1 2 3) ⇒ (1 2 2). For any particular problem, this strategy is probably either going to work really well or not well at all, without much variability in the results, so it is not tried very often by default. This option will try to create two-level through five-level factors from three-level through six-level factors. It will not attempt, for example, to code down a twenty-level factor into a nineteen-level factor (although the macro is often capable of in effect doing just that through level exchanges).

In this problem, for example, the optimal design is constructed by coding down a six-level factor into a five-level factor:

```
   %mktex(3 3 3 3 5, n=18)

   %mkteval;
```

The frequencies for the levels of the five-level factor are 6, 3, 3, 3, and 3. The optimal design is orthogonal but unbalanced.

Miscellaneous Options

This section contains some miscellaneous options that some users might occasionally find useful.

big= *n* < `choose` >
specifies the full-factorial-design size that is considered to be big. The default is `big=5185 choose`. The default value was chosen because 5185 is approximately 5000 and greater than $2^6 3^4 = 5184$, $2^{12} = 4096$, and $2 \times 3^7 = 4374$. When the full-factorial design is smaller than the `big=` value, the `%MktEx` macro searches a full-factorial candidate set. Otherwise, it searches a fractional-factorial candidate set. When `choose` is specified as well (the default), the macro can choose to use a fractional-factorial even if the full-factorial design is not too big if it appears that the final design can be created from the fractional-factorial design. This might be useful, for example, when you are requesting a fractional-factorial design with interactions. Using FACTEX to create the fractional-factorial design might be a better strategy than searching a full-factorial design with PROC OPTEX.

exchange= *n*
specifies the number of factors to consider at a time when exchanging levels. You can specify `exchange=2` to do pairwise exchanges. Pairwise exchanges are *much* slower, but they might produce better designs. For this reason, you might want to specify `maxtime=0` or `maxdesigns=1` or other iteration options to make fewer designs and make the macro run faster. The `exchange=` option interacts with the `order=` option. The `order=seqran` option is faster with `exchange=2` than `order=sequential` or `order=random`. The default is `exchange=2` when `partial=` is specified. With `order=matrix`, the `exchange=` value is

the number of matrix columns. Otherwise, the default is `exchange=1`.

With partial-profile designs and certain other highly restricted designs, it is important to do pairwise exchanges. Consider, for example, the following design row with `partial=4`:

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

The `%MktEx` macro cannot consider changing a 1 to a 2 or 3 unless it can also consider changing one of the current 2's or 3's to 1 to maintain the partial-profile restriction of exactly four values not equal to 1. Specifying the `exchange=2` option gives `%MktEx` that flexibility.

fixed= *variable*

specifies an `init=` data set variable that indicates which runs are fixed (cannot be changed) and which ones can be changed. By default, no runs are fixed.

1 – (or any nonmissing) means this run must never change.
0 – means this run is used in the initial design, but it can be swapped out.
. – means this run should be randomly initialized, and it can be swapped out.

This option can be used to add holdout runs to a conjoint design, but see `holdouts=` for an easier way. To fix parts of the design in a much more general way, see the `init=` option.

holdouts= *n*

adds holdout observations to the `init=` data set. This option augments an initial design. Specifying `holdouts=`*n* optimally adds *n* runs to the `init=` design. The option `holdouts=`*n* works by adding a `fixed=` variable and extra runs to the `init=` data set. Do not specify both `fixed=` and `holdouts=`. The number of rows in the `init=` design, plus the value specified in `holdouts=` must equal the `n=` value.

levels= *value*

specifies the method of assigning the final factor levels. This recoding occurs after the design is created, so all restrictions must be expressed in terms of one-based factors, regardless of what is specified in the `levels=` option.

Values:

`1` – default, one based, the levels are 1, 2, ...

`0` – zero based, the levels are 0, 1, ...

`c` – centered, possibly resulting in nonintegers 1 2 → –0.5 0.5, 1 2 3 → –1 0 1.

`i` – centered and scaled to integers. 1 2 → –1 1, 1 2 3 → –1 0 1.

You can also specify separate values for two- and three-level factors by preceding a value by "2" or "3". For example, `levels=2 i 3 0 c` means two-level factors are coded –1, 1 and three-level factors are coded 0, 1, 2. The remaining factors are centered. Note that the centering is based on centering the level values not on centering the (potentially unbalanced) factor. So, for example, the centered levels for a two-level factor in five runs (1 2 1 2 1) are (–0.5 0.5 –0.5 0.5 –0.5) not (–0.4 0.6 –0.4 0.6 –0.4). If you want the latter form of centering, use `proc standard m=0`. See the `%MktLab` macro for more general level setting.

You can also specify three other values:

`first` – means the first row of the design should consist entirely of the first level.

`last` – means the first row of the design should consist entirely of the last level, which is useful for Hadamard matrices.

`int` – adds an intercept column to the design.

order= `col=`*n* | `matrix=SAS-data-set`| `random` | `random=`*n* | `ranseq` | `sequential`
specifies the order in which the columns are worked on in the coordinate exchange algorithm. Valid values include:

`col=`*n* – process *n* random columns in each row
`matrix=`*SAS-data-set* – read the order from a data set
`random` – random order
`random=`*n* – random order with partial-profile exchanges
`ranseq` – sequential from a random first column
`seqran` – alias for `ranseq`
`sequential` – 1, 2, 3, ...
null – (the default) `random` when there are partial-profile restrictions, `ranseq` when there are other restrictions, and `sequential` otherwise.

For `order=col=`*n*, specify an integer for *n*, for example, `order=col=2`. This option should only be used for huge problems where you do not care if you hit every column. Typically, this option is used in conjunction with `options=largedesign quickr`. You would use it when you have a large problem and you do not have enough time for one complete pass through the design. You just want to iterate for approximately the `maxtime=` amount of time then stop. You should not use `order=col=` with restrictions.

The options `order=random=`*n* is like `order=random`, but with an adaptation that is particularly useful for partial-profile choice designs. Use this option with `exchange=2`. Say you are making a partial-profile design with ten attributes and three alternatives. Then attribute 1 is made from `x1`, `x11`, and `x21`; attribute 2 is made from `x2`, `x12`, and `x22`; and so on. Specifying `order=random=10` means that the columns, as shown by column index `j1`, are traversed in a random order. A second loop (with variable `j2`) traverses all of the factors in the current attribute. So, for example, when `j1` is 13, then `j2` = 3, 13, 23. This performs pairwise exchanges within choice attributes.

The `order=` option interacts with the `exchange=` option. With a random order and `exchange=2`, the variable `j1` loops over the columns of the design in a random order and for each `j1`, `j2` loops over the columns greater than `j1` in a random order. With a sequential order and `exchange=2`, the variable `j1` loops over the columns in 1, 2, 3 order and for each `j1`, `j2` loops over the columns greater than `j1` in a `j1`+1, `j1`+2, `j1`+3 order. The `order=ranseq` option is different. With `exchange=2`, the variable `j1` loops over the columns in an order r, $r+1$, $r+2$, ..., m, 1, 2, ..., $r-1$ (for random r), and for each `j1` there is a single random `j2`. Hence, `order=ranseq` is the fastest option since it does not consider all pairs, just one pair. The `order=ranseq` option provides the only situation where you might try `exchange=3`.

The `order=matrix=SAS-data-set` option lets you specify exactly what columns are worked on, in what order, and in what groups. The SAS data set provides one row for every column grouping. Say you want to use this option to work on columns in pairs. (Note that you could just use `exchange=2` to do this.) Then the data set would have two columns. The first variable contains the number of a design column, and the second variable contains the number of a second column that is to be exchanged with the first. The names of the variables are arbitrary. The following steps create and display an example data set for five factors:

```
%let m = 5;
data ex;
   do i = 1 to &m;
      do j = i + 1 to &m;
         output;
         end;
      end;
   run;

proc print noobs; run;
```

The results are as follows:

```
i    j

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

The specified `exchange=` value is ignored, and the actual `exchange=` value is set to two because the data set has two columns. The values must be integers between 1 and m, where m is the number of factors. The values can also be missing except in the first column. Missing values are replaced by a random column (potentially a different random column each time).

In a model with interactions, you can use this option to ensure that the terms that enter into interactions together get processed together. This is illustrated in the following steps:

```
data mat;
   input x1-x3;
   datalines;
1 1 1
2 3 .
2 4 .
3 4 .
2 3 4
5 5 .
6 7 .
8 . .
;

%mktex(4 4 2 2 3 3 2 3,            /* levels of all the factors           */
       n=36,                       /* 36 runs                             */
       order=matrix=mat,           /* matrix of columns to work on        */
       interact=x2*x3 x2*x4 x3*x4 x6*x7, /* interactions                 */
       seed=472)                   /* random number seed                  */
```

The data set MAT contains eight rows, so there are eight column groupings processed. The data set contains three columns, so up to three-way exchanges are considered. The first row mentions column 1 three times. Any repeats of a column number are ignored, so the first group of columns simply consists of column 1. The second column consists of 2, 3, and ., so the second group consists of columns 2, 3, and some random column. The random column could be any of the columns including 2 and 3, so this will sometimes be a two-way and sometimes be a three-way exchange. This group was specified since `x2*x3` is one of the interaction terms. Similarly, other groups consist of the other two-way interaction terms and a random factor: 2 and 4, 3 and 4, and 6 and 7. In addition, to help with the 3 two-way interactions involving `x2`, `x3`, and `x4`, there is one three-way term. Each time, this will consider $4 \times 2 \times 2$ exchanges, the product of the three numbers of levels. In principle, there is no limit on the number of columns, but in practice, this number could easily get too big to be useful with more than a few exchanges at a time. The row `5 5 .` requests an exchange between column 5 and a random factor. The row `8 . .` requests an exchange between column 8 and two random factors.

stopearly= *n*

specifies that the macro can stop early when it keeps finding the same maximum *D*-efficiency over and over again in different designs. The default is `stopearly=5`. By default, during the design search iterations and refinement iterations, the macro will stop early if 5 times, the macro finds a *D*-efficiency essentially equal to the maximum but not greater than the maximum. This might mean that the macro has found the optimal design, or it might mean that the macro keeps finding a very attractive local optimum. Either way, it is unlikely it will do any better. When the macro stops for this reason, the macro will display the following message:

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

Specify either 0 or a very large value to turn off the stop-early checking.

tabsize= *n*
provides you with some control on which design (orthogonal array, FACTEX or Hadamard) from the orthogonal array table (catalog) is used for the partial initialization when an exact match is not found. Specify the number of runs in the orthogonal array. By default, the macro chooses an orthogonal design that best matches the specified design. See the `cat=` option for more detailed control.

target= *n*
specifies the target efficiency criterion. The default is `target=100`. The macro stops when it finds an efficiency value greater than or equal to this number. If you know what the maximum efficiency criterion is, or you know how big is big enough, you can sometimes make the macro run faster by letting it stop when it reaches the specified efficiency. You can also use this option if you just want to see the initial design that `%MktEx` is using: `target=1, optiter=0`. By specifying `target=1`, the macro will stop after the initialization as long as the initial efficiency is ≥ 1.

Esoteric Options

This last set of options contains all of the other miscellaneous options. Most of the time, most users should not specify options from this list.

annealfun= *function*
specifies the function that controls how the simulated annealing probability changes with each pass through the design. The default is `annealfun=anneal * 0.85`. Note that the IML operator `#` performs ordinary (scalar) multiplication. Most users will never need this option.

detfuzz= *n*
specifies the value used to determine if determinants are changing. The default is `detfuzz=1e-8`. If `newdeter > olddeter * (1 + detfuzz)` then the new determinant is larger. Otherwise if `newdeter > olddeter * (1 - detfuzz)` then the new determinant is the same. Otherwise the new determinant is smaller. Most users will never need this option.

imlopts= *options*
specifies IML PROC statement options. For example, for very large problems, you can use this option to specify the IML `symsize=` or `worksize=` options: `imlopts=symsize=`*n* `worksize=`*m*, substituting numeric values for n and m. The defaults for these options are host dependent. Most users will never need this option.

ridge= *n*
specifies the value to add to the diagonal of $\mathbf{X}'\mathbf{X}$ to make it nonsingular. The default is `ridge=1e-7`. Usually, for normal problems, you will not need to change this value. If you want the macro to create designs with more parameters than runs, you must specify some other value, usually something like 0.01. By default, the macro will quit when there are more parameters than runs. Specifying a `ridge=` value other than the default (even if you just change the "e" in 1e–7 to "E") lets the macro create a design with more parameters than runs. This option is sometimes needed for advanced design problems.

Advanced Restrictions

It is extremely important with restrictions to appropriately quantify the badness of the run. The `%MktEx` macro has to know when it considers an exchange if it is considering:

- eliminating restriction violations making the design better,
- causing more restriction violations making the design worse,
- a change that neither increases nor decreases the number of violations.

Your restrictions macro must tell `%MktEx` when it is making progress in the right direction. If it does not, `%MktEx` will probably not find an acceptable design.

Complicated Restrictions

Consider designing a choice experiment with two alternatives each composed of 25 attributes, and the first 22 of which have restrictions on them. Attribute one in the choice design is made from `x1` and `x23`, attribute two in the choice design is made from `x2` and `x24`, ..., and attribute 22 in the choice design is made from `x22` and `x44`. The remaining attributes are made from `x45 -- x50`. The restrictions are as follows: each choice attribute must contain two 1's between 5 and 9 times, each choice attribute must contain exactly one 1 between 5 and 9 times, and each choice attribute must contain two 2's between 5 and 9 times. The following steps show an example of how *NOT* to accomplish this:

```
%macro sumres;
   allone = 0; oneone = 0; alltwo = 0;
   do k = 1 to 22;
      if      (x[k] = 1 & x[k+22] = 1) then allone = allone + 1;
      else if (x[k] = 1 | x[k+22] = 1) then oneone = oneone + 1;
      else if (x[k] = 2 & x[k+22] = 2) then alltwo = alltwo + 1;
      end;

   * Bad example.  Need to quantify badness.;
   bad = (^((5 <= allone & allone <= 9) &
            (5 <= oneone & oneone <= 9) &
            (5 <= alltwo & alltwo <= 9)));
   %mend;

%mktex(3 ** 50,                    /* 50 three-level factors             */
       n=135,                      /* 135 runs                           */
       restrictions=sumres,        /* name of restrictions macro         */
       seed=289,                   /* random number seed                 */
       options=resrep              /* restrictions report                */
               quickr              /* very quick run with random init    */
               nox)                /* suppresses x1, x2, x3 ... creation */
```

The problem with the preceding approach is there are complicated restrictions but badness is binary. If all the counts are in the right range, badness is 0, otherwise it is 1. You need to write a macro that lets `%MktEx` know when it is going in the right direction or it will probably never find a suitable design. One thing that is correct about the preceding code is the compound Boolean range expressions like `(5`

`<= allone & allone <= 9)`. Abbreviated expressions like `(5 <= allone <= 9)` that work correctly in the DATA step work incorrectly and without warning in IML. Another thing that is correct is the way the `sumres` macro creates new variables, `k`, `allone`, `oneone`, and `alltwo`. Care was taken to avoid using names like `i` and `x` that conflict with the matrices that you can examine in quantifying badness. The full list of names that you must avoid are `i`, `try`, `x`, `x1`, `x2`, ..., through `x`*n* for *n* factors, `j1`, `j2`, `j3`, and `xmat`. The following steps show a slightly better but still bad example of the macro:

```
%macro sumres;
   allone = 0; oneone = 0; alltwo = 0;
   do k = 1 to 22;
      if      (x[k] = 1 & x[k+22] = 1) then allone = allone + 1;
      else if (x[k] = 1 | x[k+22] = 1) then oneone = oneone + 1;
      else if (x[k] = 2 & x[k+22] = 2) then alltwo = alltwo + 1;
      end;
   * Better, badness is quantified, and almost correctly too!;
   bad = (^((5 <= allone & allone <= 9) &
            (5 <= oneone & oneone <= 9) &
            (5 <= alltwo & alltwo <= 9))) #
         (abs(allone - 7) + abs(oneone - 7) + abs(alltwo - 7));
   %mend;
```

```
%mktex(3 ** 50,                   /* 50 three-level factors            */
       n=135,                     /* 135 runs                          */
       restrictions=sumres,       /* name of restrictions macro        */
       seed=289,                  /* random number seed                */
       options=resrep             /* restrictions report               */
               quickr             /* very quick run with random init   */
               nox)               /* suppresses x1, x2, x3 ... creation   */
```

This restrictions macro seems at first glance to do everything right—it quantifies badness. We need to examine this macro more closely. It counts in `allone`, `oneone`, and `alltwo` the number of times choice attributes are all one, have exactly one 1, or are all two. Everything is fine when the all one count is in the range 5 to 9 `(5 <= allone & allone <= 9)`, and the exactly one 1 count is in the range 5 to 9 `(5 <= oneone & oneone <= 9)`, and the all two count is in the range 5 to 9 `(5 <= alltwo & alltwo <= 9)`. It is bad when this is not true `(^((5 <= allone & allone <= 9) & (5 <= oneone & oneone <= 9) & (5 <= alltwo & alltwo <= 9)))`, the Boolean not operator "^" performs the logical negation. This Boolean expression is 1 for bad and 0 for OK. It is multiplied times a quantitative sum of how far these counts are outside the right range `(abs(allone - 7) + abs(oneone - 7) + abs(alltwo - 7))`. When the run meets all restrictions, this sum of absolute differences is multiplied by zero. Otherwise badness gets larger as the counts get farther away from the middle of the 5 to 9 interval.

In the `%MktEx` macro, we specify `options=resrep` to produce a report in the iteration history on the process of meeting the restrictions. When you run `%MktEx` and it is having trouble making a design that conforms to restrictions, this report can be extremely helpful. Next, we will examine some of the output from running the preceding macros.

Some of the results are as follows:

```
                           Algorithm Search History

                                  Current         Best
       Design    Row,Col  D-Efficiency  D-Efficiency  Notes
       -----------------------------------------------------------
            1      Start       59.7632                Ran,Mut,Ann
            1      1           60.0363                0 Violations
            1      2           60.3715                0 Violations
            1      3           60.9507                0 Violations
            1      4           61.2319                5 Violations
            1      5           61.6829                0 Violations
            1      6           62.1529                0 Violations
            1      7           62.4004                0 Violations
            1      8           62.9747                3 Violations
            .
            .
            .

            1    132           70.4482                6 Violations
            1    133           70.3394                4 Violations
            1    134           70.4054                0 Violations
            1    135           70.4598                0 Violations
```

So far we have seen the results from the first pass through the design. With `options=resrep` the macro displays one line per row with the number of violations when it is done with the row. Notice that the macro is succeeding in eliminating violations in some but not all rows. This is the first thing you should look for. If it is not succeeding in any rows, you might have written a set of restrictions that is impossible to satisfy. Some of the output from the second pass through the design is as follows:

```
            1      1           70.5586                0 Violations
            1      2           70.7439                0 Violations
            1      3           70.7383                0 Violations
            1      4           70.7429                5 Violations
            1      4           70.6392                4 Violations
            1      4           70.7081                4 Violations
            1      4           70.7717                4 Violations
            1      4           70.7717                4 Violations
            1      4           70.7717                4 Violations
            1      4           70.7717                4 Violations
            1      4           70.7717                4 Violations
            1      4           70.7202                4 Violations
            1      4           70.7717                4 Violations
            1      4           70.7717                4 Violations
            1      4           70.7717                4 Violations
```

```
                   1      4          70.7264                4 Violations
                   1      4          70.7717                4 Violations
                   1      4          70.7717                4 Violations
                   1      4          70.7717                4 Violations
                   1      4          70.7717                4 Violations
                   1      4          70.7717                4 Violations
                   1      4          70.7274                4 Violations
                   1      4          70.7717                4 Violations
                   1      4          70.7515                4 Violations
                   1      4          70.7636                4 Violations
                   1      4          70.7717                4 Violations
                   1      4          70.7591                4 Violations
                   1      4          70.7717                4 Violations
                   1      5          70.7913                0 Violations
                   1      6          70.9467                0 Violations
                   1      7          71.0102                0 Violations
                   1      8          71.0660                0 Violations
```

In the second pass, in situations where the macro had some reasonable success in the first pass, `%MktEx` tries extra hard to impose restrictions. We see it trying over and over again without success to impose the restrictions in the fourth row. All it manages to do is lower the number of violations from 5 to 4. We also see it has no trouble removing all violations in the eighth row that were still there after the first pass. The macro produces volumes of output like this. For several iterations, it will devote extra attention to rows with some violations but in this case without complete success. When you see this pattern, some success but also some stubborn rows that the macro cannot fix, there might be something wrong with your restrictions macro. Are you *really* telling `%MktEx` when it is doing a better job? These preceding steps illustrate some of the things that can go wrong with restrictions macros. It is important to carefully evaluate the results—look at the design, look at the iteration history, specify `options=resrep`, and so on to ensure your restrictions are doing what you want. The problem in this case is in the quantification of badness, in the following statement:

```
      bad = (^((5 <= allone & allone <= 9) &
               (5 <= oneone & oneone <= 9) &
               (5 <= alltwo & alltwo <= 9))) #
            (abs(allone - 7) + abs(oneone - 7) + abs(alltwo - 7));
```

Notice that we have three nonindependent contributors to the badness function, the three counts. As a level gets changed, it could increase one count and decrease another. There is a larger problem too. Say that `allone` and `oneone` are in the right range but `alltwo` is not. Then the function fragments `abs(allone - 7)` and `abs(oneone - 7)` incorrectly contribute to the badness function. The fix is to clearly differentiate the three sources of badness *and* weight the pieces so that one part never trades off against the other, for example, as follows:

```
%macro sumres;
   allone = 0; oneone = 0; alltwo = 0;
   do k = 1 to 22;
      if      (x[k] = 1 & x[k+22] = 1) then allone = allone + 1;
      else if (x[k] = 1 | x[k+22] = 1) then oneone = oneone + 1;
      else if (x[k] = 2 & x[k+22] = 2) then alltwo = alltwo + 1;
      end;
   bad = 100 # (^(5 <= allone & allone <= 9)) # abs(allone - 7) +
          10 # (^(5 <= oneone & oneone <= 9)) # abs(oneone - 7) +
               (^(5 <= alltwo & alltwo <= 9)) # abs(alltwo - 7);
   %mend;

%mktex(3 ** 50,                    /* 50 three-level factors            */
       n=135,                      /* 135 runs                          */
       restrictions=sumres,        /* name of restrictions macro        */
       seed=289,                   /* random number seed                */
       options=resrep              /* restrictions report               */
               quickr              /* very quick run with random init   */
               nox)                /* suppresses x1, x2, x3 ... creation  */
```

Now a component of the badness only contributes to the function when it is really part of the problem. We gave the first part weight 100 and the second part weight 10. Now the macro will never change `oneone` or `alltwo` if that causes a problem for `allone`, and it will never change `alltwo` if that causes a problem for `oneone`. Previously, the macro was getting stuck in some rows because it could never figure out how to fix one component of badness without making another component worse. *For some problems, figuring out how to differentially weight the components of badness so that they never trade off against each other is the key to writing a successful restrictions macro.* Often, it does not matter which component gets the most weight. What is important is that each component gets a *different* weight so that `%MktEx` does not get caught cycling back and forth making A better and B worse then making B better and A worse. Some of the output from the first pass through the design is as follows:

```
                        Algorithm Search History

                              Current          Best
       Design    Row,Col  D-Efficiency  D-Efficiency  Notes
       ----------------------------------------------------------
            1      Start       59.7632                Ran,Mut,Ann
            1      1           60.1415                0 Violations
            1      2           60.5303                0 Violations
            1      3           61.0148                0 Violations
            1      4           61.4507                0 Violations
            1      5           61.7717                0 Violations
            1      6           62.2353                0 Violations
            1      7           62.5967                0 Violations
            1      8           63.1628                3 Violations
            .
            .
            .
```

```
               1    126          72.3566               4 Violations
               1    127          72.2597               0 Violations
               1    128          72.3067               0 Violations
               1    129          72.3092               0 Violations
               1    130          72.0980               0 Violations
               1    131          71.8163               0 Violations
               1    132          71.3795               0 Violations
               1    133          71.4446               0 Violations
               1    134          71.2805               0 Violations
               1    135          71.3253               0 Violations
```

We can see that in the first pass, the macro is imposing all restrictions for most but not all of the rows. Some of the output from the second pass through the design is as follows:

```
               1      1          71.3968               0 Violations
               1      2          71.5017               0 Violations
               1      3          71.7295               0 Violations
               1      4          71.7839               0 Violations
               1      5          71.8671               0 Violations
               1      6          71.9544               0 Violations
               1      7          72.0444               0 Violations
               1      8          72.0472               0 Violations
               .
               .
               .
               1    126          77.1597               0 Violations
               1    127          77.1604               0 Violations
               1    128          77.1323               0 Violations
               1    129          77.1584               0 Violations
               1    130          77.0708               0 Violations
               1    131          77.1013               0 Violations
               1    132          77.1721               0 Violations
               1    133          77.1651               0 Violations
               1    134          77.1651               0 Violations
               1    135          77.2061               0 Violations
```

In the second pass, `%MktEx` has imposed all the restrictions in rows 8 and 126, the rows that still had violations after the first pass (and all of the other not shown rows too). The third pass ends with the following output:

```
        1    126          78.7813                   0 Violations
        1    127   1      78.7813       78.7813     Conforms
        1    127  18      78.7899       78.7899
        1    127  19      78.7923       78.7923
        1    127  32      78.7933       78.7933
        1    127  40      78.7971       78.7971
        1    127  44      78.8042       78.8042
        1    127  47      78.8250       78.8250
        1    127  50      78.8259       78.8259
        1    127   1      78.8296       78.8296
        1    127   5      78.8296       78.8296
        1    127   8      78.8449       78.8449
        1    127  10      78.8456       78.8456
        1    128  48      78.8585       78.8585
        1    128  49      78.8591       78.8591
        1    128   7      78.8591       78.8591
```

The `%MktEx` macro completes a full pass through row 126, the place of the last violation, without finding any new violations so the macro states in row 127 that the design conforms to the restrictions and the iteration history proceeds in the normal fashion from then on (not shown). The note `Conforms` is displayed at the place where `%MktEx` decides that the design conforms. The design will continue to conform throughout more iterations, even though the note `Conforms` is not displayed on every line. The final efficiency is as follows:

```
                              The OPTEX Procedure

                                                                        Average
                                                                      Prediction
     Design                                                            Standard
     Number     D-Efficiency     A-Efficiency     G-Efficiency          Error
     ------------------------------------------------------------------------
          1          85.0645          72.2858          95.6858          0.8650
```

These next steps create the choice design and display a subset of the design:

```
%mktkey(x1-x50)

data key;
   input (x1-x25) ($);
   datalines;
x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13
x14 x15 x16 x17 x18 x19 x20 x21 x22                  x45 x46 x47
x23 x24 x25 x26 x27 x28 x29 x30 x31 x32 x33 x34
x35 x36 x37 x38 x39 x40 x41 x42 x43 x44              x48 x49 x50
;
```

```
%mktroll(design=design, key=key, out=chdes)

proc print; by set; id set; where set le 2 or set ge 134; run;
```

Notice the slightly unusual arrangement of the Key data set due to the fact that the first 22 attributes get made from the first 44 factors of the linear arrangement.

The first four choice sets are as follows:

```
      -
      A
    S l                   x x x x x x x x x x x x x x x x
    e t x x x x x x x x x 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2
    t _ 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

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

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

  134 1 3 3 2 3 3 2 1 1 1 1 1 1 2 2 1 3 2 2 1 3 3 1 2 1 2
      2 1 1 2 2 1 2 1 1 1 1 1 3 2 2 3 1 1 2 1 1 3 3 1 3 3

  135 1 3 3 3 1 3 1 1 1 1 2 2 3 1 2 3 3 1 3 2 1 2 1 2 3 1
      2 2 1 1 1 1 1 1 3 1 2 2 1 3 2 1 3 3 1 2 1 2 1 2 2 2
```

Where the Restrictions Macro Gets Called

There is one more aspect to restrictions that must be understood for the most sophisticated usages of restrictions. The macro that imposes the restrictions is defined and called in four distinct places in the %MktEx macro. First, the restrictions macro is called in a separate, preliminary IML step, just to catch some syntax errors that you might have made. Next, it is called in between calling PROC PLAN or PROC FACTEX and calling PROC OPTEX. Here, the restrictions macro is used to impose restrictions on the candidate set. Next, it is used in the obvious way during design creation and the coordinate-exchange algorithm. Finally, when options=accept is specified, which means that restriction violations are acceptable, the macro is called after all of the iterations have completed to report on restriction violations in the final design. For some advanced restrictions, we might not want exactly the same code running in all four places. When the restrictions are purely written in terms of restrictions on x, which is the *ith* row of the design matrix, there is no problem. The same macro will work fine for all uses. However, when xmat (the full x matrix) or i or j1 (the row or column number) are used, the same code typically cannot be used for all applications, although sometimes it does not matter. Next are some notes on each of the four phases.

Syntax Check. In this phase, the macro is defined and called just to check for syntax errors. This step lets the macro end more gracefully when there are errors and provides better information about the nature of the error than it would otherwise. Your restrictions macro can recognize when it is in this phase because the macro variable `&main` is set to 0 and the macro variable `&pass` is set to null. The pass variable is null before the iterations begin, 1 for the algorithm search phase, 2 for the design search phase, 3 for the design refinement stage, and 4 after the iterations end. You can conditionally execute code in this step or not using the following macro statements:

```
%if      &main eq 0 and &pass eq  %then %do;  /* execute in syntax check     */
%if not (&main eq 0 and &pass eq) %then %do;  /* not execute in syntax check */
```

You will usually not need to worry about this step. It just calls the macro once and ignores the results to check for syntax errors. For this step, `xmat` is a matrix of ones, and `x` is a vector of ones (since the design does not exist yet) and `j1 = j2 = j3 = i = 1`. If you have complicated restrictions involving the row or column exchange indices (`i`, `j1`, `j2`, `j3`) you might need to worry about this step. You might need to either not execute your restrictions in this step or *conditionally* execute some assignment statements (just for this step) that set up `j1`, `j2`, and `j3` more appropriately. Sometimes, you can set things up appropriately by using the `resmac=` option. Be aware however, that this step checks (`i`, `try`, `j1`, `j2`, `j3`, `x`, and `xmat` after your macro is called to ensure that you are not changing them because this is usually a sign of an error. If you get the following warning, make sure you are not incorrectly changing one of the matrices that you should not be changing:

```
WARNING: Restrictions macro is changing i, try, j1, j2, j3, x, or xmat.
         This might be a serious problem.  Check your macro.
```

If this step detects a syntax error, it will try to tell you where it is and what the problem is. If you have syntax errors in your restrictions macro and you cannot figure out what they are, sometimes the best thing to do is directly submit the statements in your restrictions macro to IML to so you can see the syntax errors. First, you need to submit the following statements:

```
%let n = 27; /* substitute number of runs    */
%let m = 10; /* substitute number of factors */
proc iml;
   xmat = j(&n, &m, 1);
   i = 1; j1 = 1; j2 = 1; j3 = 1; bad = 0; x = xmat[i,];
```

Candidate Check. In this phase, the macro is used to impose restrictions on the candidate set created by PROC PLAN or PROC FACTEX before it is searched by PROC OPTEX. The macro is called once for each row with the column index, `j1` set to 1. For some problems, such as most partial-profile problems, the restrictions are so severe that virtually none of the candidates will conform. Also, restrictions that are based on row number and column number do not make sense in the context of a candidate design. Your restrictions macro can recognize when it is in this phase because the macro variable `&main` is set to 0 and the macro variable `&pass` is set to 1 or 2. You can conditionally execute code in this step or not by using the following macro statements:

```
%if      &main eq 0 and &pass ge 1 and &pass le 2
         %then %do;                             /* execute on candidates     */
%if not (&main eq 0 and &pass ge 1 and &pass le 2)
         %then %do;                             /* not execute on candidates */
```

For simple restrictions not involving the column exchange indices (`j1`, `j2`, `j3`), you probably do not need to worry about this step. If you use `j1`, `j2`, or `j3`, you will need to either not execute your restrictions in this step or conditionally execute some assignment statements that set up `j1`, `j2`, and `j3` appropriately. Ordinarily for this step, `xmat` contains the candidate design, `x` contains the *ith* row, `j1 = 0; j2 = 0; j3 = 0; try = 1;` `i` is set to the candidate row number.

Main Coordinate-Exchange Algorithm. In this phase, the macro is used to impose restrictions on the design as it is being built in the coordinate-exchange algorithm. Your restrictions macro can recognize when it is in this phase because the macro variable `&main` is set to 1 and the macro variable `&pass` is set to 1, 2, or 3. You can conditionally execute code in this step or not by using the following macro statements:

```
%if      &main eq 1 and &pass ge 1 and &pass le 3
         %then %do;                    /* execute on coordinate exchange     */
%if not (&main eq 1 and &pass ge 1 and &pass le 3)
         %then %do;                    /* not execute on coordinate exchange */
```

For this step, `xmat` contains the candidate design, `x` contains the *ith* row, `j1`, `j2`, and `j3` typically contain the column indices, `i` is the row number, and `try` is the zero-based cumulative row number. With `exchange=1`, `j1` exists, with `exchange=2`, `j1` and `j2` exist, and so on. Sometimes in this phase, the restrictions macro is called once per row with the `j*` indices all set to 1. If you use the `j*` indices in your restrictions, you might need to allow for this. For example, if you are checking the current `j1` column for balance, and you used an `init=` data set with column one fixed and unbalanced, you will not want to perform the check when `j1 = 1`. Note that for some designs that are partially initialized with an orthogonal array and for some uses of `init=`, not all columns or cells in the design are evaluated.

Restrictions Violations Check. In this phase, the macro is used to check the design when there are restrictions and `options=accept`. The macro is called once for each row of the design. Your restrictions macro can recognize when it is in this phase because the macro variable `&main` is set to 1 and the macro variable `&pass` is greater than 3. You can conditionally execute code in this step or not by using the following macro statements:

```
%if      &main eq 1 and &pass gt 3  %then %do; /* execute on final check     */
%if not (&main eq 1 and &pass gt 3) %then %do; /* not execute on final check */
```

For this step, `xmat` contains the candidate design, `x` contains the *ith* row, `j1 = 1; j2 = 1; j3 = 1; try = 1;` and `i` is the row number.

Using the following macro is equivalent to specifying `partial=4`:

```
%macro partprof;
   nvary = sum(x ^= 1);
   %if &main %then %do;
      if i = 1 then bad = nvary;
      else          bad = abs(nvary - 4);
      %end;
   %else %do;
      bad = ^ (nvary = 0 | nvary = 4);
      %end;
   %mend;
```

In the main algorithm, when imposing restrictions on the design, we restrict the first run to be constant and all other runs to have four attributes varying. For the candidate-set restrictions, when MAIN is zero, any observation with zero or four varying factors is acceptable. For the candidate-set restrictions, there is no reason to count the number of violations. A candidate run is either acceptable or not. We do not worry about the syntax error or final check steps; both versions will work fine in either.

%MktKey Macro

The `%MktKey` autocall macro creates expanded lists of variable names. See the following pages for examples of using this macro in the design chapter: 133 and 192. Also see the following pages for examples of using this macro in the discrete choice chapter: 356, 546, 556, 575, 607, 617, 628, and 636. Additional examples appear throughout this chapter. You can specify the number of rows followed by a number of columns. The output is a data set called KEY. This is illustrated in the following step:

```
%mktkey(5 10)
```

The KEY output data set with 5 rows and 10 columns and $5 \times 10 = 50$ variable names, `x1-x50` is as follows:

```
x1     x2     x3     x4     x5     x6     x7     x8     x9     x10

x1     x2     x3     x4     x5     x6     x7     x8     x9     x10
x11    x12    x13    x14    x15    x16    x17    x18    x19    x20
x21    x22    x23    x24    x25    x26    x27    x28    x29    x30
x31    x32    x33    x34    x35    x36    x37    x38    x39    x40
x41    x42    x43    x44    x45    x46    x47    x48    x49    x50
```

Alternatively, you can specify the number of rows and number of columns followed by a `t` or `T` and get the transpose of this data set. The output data set is again called KEY. The following step illustrates this option:

```
%mktkey(5 10 t)
```

The KEY output data set with 5 rows and 10 columns and $5 \times 10 = 50$ variable names, `x1-x50` is as follows:

```
x1    x2     x3     x4     x5     x6     x7     x8     x9     x10

x1    x6     x11    x16    x21    x26    x31    x36    x41    x46
x2    x7     x12    x17    x22    x27    x32    x37    x42    x47
x3    x8     x13    x18    x23    x28    x33    x38    x43    x48
x4    x9     x14    x19    x24    x29    x34    x39    x44    x49
x5    x10    x15    x20    x25    x30    x35    x40    x45    x50
```

Note that this time the names progress down the columns instead of across the rows.

The `%MktKey` macro has another type of syntax as well. You can provide the `%MktKey` macro with a list of variables as follows:

```
%mktkey(x1-x15)
```

The `%MktKey` macro produced the following line:

```
x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15
```

You can copy and paste this list to make it easier to construct the `key=` data set for the `%MktRoll` macro. The following step makes the `Key` data set:

```
data key;
   input (x1-x5) ($);
   datalines;
 x1  x2  x3  x4  x5
 x6  x7  x8  x9 x10
x11 x12 x13 x14 x15
  .   .   .   .   .
;
```

Alternatively, if you want to use precisely the `Key` data set that the `%MktKey` macro creates, you can have the `%MktRoll` macro automatically construct the `key=` data set for you by specifying the same argument in the `key=` option that you would specify in the `%MktKey` macro. In the sample code below, the first two steps are equivalent to the third step:

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

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

%MktKey Macro Options

The following option can be used with the `%MktKey` macro:

MktKey Macro Options

Option	Description
`list`	(positional) variable list or n rows and n columns
	(positional) “help” or “?” displays syntax summary

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

```
%mktkey(help)
%mktkey(?)
```

The only argument to the `%MktKey` macro is the list.

list
specifies the variable list or matrix size list. Note that the list is a positional parameter, hence it is not specified after a name and an equal sign. The list can be a variable list.

Alternatively, the list contains the number of rows followed by the number of columns, optionally followed by a `t` or `T` (for transpose). Without the `t` the names go `x1`, `x2`, `x3`, ..., across each row. With the `t` the names go `x1`, `x2`, `x3`, ..., down each column.

%MktLab Macro

The `%MktLab` autocall macro processes an experimental design, usually created by the `%MktEx` macro, and assigns the final variable names and levels. See the following pages for examples of using this macro in the design chapter: 81, 85, 109, 112, 166 and 201. Also see the following pages for examples of using this macro in the discrete choice chapter: 333, 353, 501, 538, 564, 567, 596 and 602. Additional examples appear throughout this chapter. For example, say you used the `%MktEx` macro to create a design with 11 two-level factors (with default levels of 1 and 2). The following steps create and display the design:

```
%mktex(n=12, options=nosort)

proc print noobs; run;
```

The design is as follows:

```
x1    x2    x3    x4    x5    x6    x7    x8    x9    x10    x11

 1     1     1     1     1     1     1     1     1     1      1
 1     2     2     2     1     1     2     1     1     2      2
 2     2     1     1     2     1     2     2     1     2      1
 1     1     1     2     1     2     2     2     2     2      1
 1     2     2     2     2     1     1     2     2     1      1
 1     2     1     1     2     2     2     1     2     1      2
 2     2     1     2     1     2     1     2     1     1      2
 2     1     1     2     2     1     1     1     2     2      2
 2     1     2     2     2     2     2     1     1     1      1
 1     1     2     1     2     2     1     2     1     2      2
 2     2     2     1     1     2     1     1     2     2      1
 2     1     2     1     1     1     2     2     2     1      2
```

Either the `%MktEx` macro or the `%MktLab` macro can be used to assign levels of –1 and 1 and add an intercept. You can do it directly with the `%MktEx` macro, using `levels=i int`. The value of `i` specifies centered integer levels, and `int` adds the intercept. The following step illustrates this:

```
%mktex(n=12,                       /* 12 runs                           */
       options=nosort,             /* do not sort design                */
       levels=i                    /* -1 and 1 instead of 1 and 2       */
              int)                 /* add an intercept to the design    */
```

However, if you want to change the factor names, and for more complicated relabeling of the levels, you need to use the `%MktLab` macro. This is illustrated in the following step:

```
%mktex(n=12, options=nosort)

%mktlab(data=design, values=1 -1, int=Had0, prefix=Had)

proc print noobs; run;
```

The %MktLab macro assigns levels of –1 and 1, adds an intercept named Had0, and changes the variable name prefixes from x to Had. This creates a Hadamard matrix (although, of course, the Hadamard matrix can have any set of variable names). The resulting Hadamard matrix is as follows:

```
      Had0  Had1  Had2  Had3  Had4  Had5  Had6  Had7  Had8  Had9  Had10  Had11

        1    -1    -1    -1    -1    -1    -1    -1    -1    -1     -1     -1
        1     1    -1     1    -1    -1    -1     1     1     1     -1      1
        1     1     1    -1     1    -1    -1    -1     1     1      1     -1
        1    -1     1     1    -1     1    -1    -1    -1     1      1      1
        1     1    -1     1     1    -1     1    -1    -1    -1      1      1
        1     1     1    -1     1     1    -1     1    -1    -1     -1      1
        1     1     1     1    -1     1     1    -1     1    -1     -1     -1
        1    -1     1     1     1    -1     1     1    -1     1     -1     -1
        1    -1    -1     1     1     1    -1     1     1    -1      1     -1
        1    -1    -1    -1     1     1     1    -1     1     1     -1      1
        1     1    -1    -1    -1     1     1     1    -1     1      1     -1
        1    -1     1    -1    -1    -1     1     1     1    -1      1      1
```

Alternatively, you can use the key= data set that follows to do the same thing:

```
data key;
  array Had[11];
  input Had1 @@;
  do i = 2 to 11; Had[i] = Had1; end;
  drop i;
  datalines;
1 -1
;

proc print data=key; run;
```

The key= data set is as follows:

```
      Obs  Had1  Had2  Had3  Had4  Had5  Had6  Had7  Had8  Had9  Had10  Had11

       1     1     1     1     1     1     1     1     1     1      1      1
       2    -1    -1    -1    -1    -1    -1    -1    -1    -1     -1     -1
```

The following step uses this data set to make the final design:

```
%mktlab(data=design, key=key, int=Had0)
```

The Hadamard matrix from this step (not shown) is exactly the same as the one just shown previously.

The `key=` data set contains all of the variables that you want in the design and all of their levels. This information is applied to the design, by default the one stored in a data set called RANDOMIZED, which is the default `outr=` data set name from the `%MktEx` macro. The results are stored in a new data set, FINAL, with the desired factor names and levels.

Consider the consumer food product example from page 255. The following step reads a possible design:

```
data randomized;
  input x1-x8 @@;
  datalines;
4 2 1 1 1 2 2 2 2 1 1 2 1 3 1 3 3 4 2 2 1 3 2 3 4 3 2 1 3 2 2 3 4 1 2 1
1 1 1 1 2 4 1 2 1 2 1 1 1 2 1 2 3 3 2 1 2 2 2 2 2 2 2 3 1 4 2 1 1 2 2 2
3 2 2 1 3 1 2 1 1 4 1 2 2 3 1 2 1 3 2 2 1 3 1 1 3 2 1 2 2 1 2 3 3 4 1 1
3 1 1 3 4 1 2 2 2 1 2 1 2 3 2 1 2 3 2 2 2 1 2 1 3 3 1 3 4 2 2 2 1 3 1 2
2 4 2 2 3 1 1 2 3 1 2 2 3 2 1 2 3 3 1 1 2 3 1 1 4 4 2 1 2 2 1 3 1 1 1 1
3 2 1 2 4 3 1 2 3 3 2 2 1 2 2 1 2 1 1 3 1 3 1 1 1 1 2 3
;
```

Designs created by the `%MktEx` macro always have factor names `x1`, `x2`, ..., and so on, and the levels are consecutive integers beginning with 1 (1, 2 for two-level factors; 1, 2, 3 for three-level factors; and so on). The `%MktLab` macro provides you with a convenient way to change the names and levels to more meaningful values. The data set KEY contains the variable names and levels that you ultimately want. The following step creates a `Key` data set:

```
data key;
  missing N;
  input Client ClientLineExtension ClientMicro $ ShelfTalker $
        Regional Private PrivateMicro $ NationalLabel;
  format _numeric_ dollar5.2;
  datalines;
1.29 1.39 micro Yes 1.99 1.49 micro 1.99
1.69 1.89 stove No  2.49 2.29 stove 2.39
2.09 2.39 .     .   N    N    .     N
N    N    .     .   .    .    .     .
;

%mktlab(data=randomized, key=key)

proc sort; by shelftalker; run;

proc print; by shelftalker; run;
```

The variable `Client` with 4 levels is made from `x1`, `ClientLineExtension` with 4 levels is made from `x2`, `ClientMicro` with 2 levels is made from `x3`. The `N` (for not available) is treated as a special missing value. The `Key` data set has four rows because the maximum number of levels is four. Factors with fewer than four levels are filled in with ordinary missing values. The `%MktLab` macro takes the default `data=randomized` data set from `%MktEx` and uses the rules in the `key=key` data set, to create the information in the `out=final` data set, which is shown next, sorted by the shelf talker variable.

Some of the design is as follows:

```
----------------------------- ShelfTalker=No ------------------------------

                  Client
                   Line     Client                          Private   National
Obs   Client   Extension   Micro     Regional   Private    Micro     Label

  1   $1.69      $1.39     micro      $1.99        N       micro       N
  2   $2.09        N       stove      $1.99        N       stove       N
  3   $1.69        N       micro      $1.99      $2.29     micro     $1.99
  .
  .
  .

----------------------------- ShelfTalker=Yes -----------------------------

                  Client
                   Line     Client                          Private   National
Obs   Client   Extension   Micro     Regional   Private    Micro     Label

 14     N        $1.89     micro      $1.99      $2.29     stove     $2.39
 15     N        $2.39     stove        N        $2.29     stove       N
 16     N        $1.39     stove      $1.99      $1.49     micro     $1.99
  .
  .
  .
```

This macro creates the `out=` data set by repeatedly reading and rereading the `key=` data set, one datum at a time, using the information in the `data=` data set to determine which levels to read from the `key=` data set. In this example, for the first observation, `x1`=4 so the fourth value of the first `key=` variable is read, then `x2`=2 so the second value of the second `key=` variable is read, then `x3`=1 so the first value of the third `key=` variable is read, ..., then `x8`=2 so the second value of the eighth `key=` variable is read, then the first observation is output. This continues for all observations. Unlike previous releases, the `data=` data is not required to have the default `levels=` specification (integer values beginning with 1). Other numeric values are fine and are converted to consecutive positive integers before performing the final mapping.

The following steps create the L_{36}, changes the names of the two-level factors to `two1-two11`, assigns them values the –1, 1, changes the names of the three-level factors to `thr1-thr12`, and assigns them the values –1, 0, 1:

```
%mktex(n=36, seed=420)

data key;
  array x[23] two1-two11 thr1-thr12;
  input two1 thr1;
  do i =  2 to 11; x[i] = two1; end;
  do i = 13 to 23; x[i] = thr1; end;
  drop i;
  datalines;
-1 -1
 1  0
 .  1
;

%mktlab(data=randomized, key=key)

proc print data=key noobs; var two:; run;
proc print data=key noobs; var thr:; run;

proc print data=final(obs=5) noobs; var two:; run;
proc print data=final(obs=5) noobs; var thr:; run;
```

The `Key` data set is as follows:

```
 two1   two2   two3   two4   two5   two6   two7   two8   two9   two10   two11

  -1     -1     -1     -1     -1     -1     -1     -1     -1      -1      -1
   1      1      1      1      1      1      1      1      1       1       1
   .      .      .      .      .      .      .      .      .       .       .

  thr1  thr2  thr3  thr4  thr5  thr6  thr7  thr8  thr9  thr10  thr11  thr12

   -1    -1    -1    -1    -1    -1    -1    -1    -1     -1     -1     -1
    0     0     0     0     0     0     0     0     0      0      0      0
    1     1     1     1     1     1     1     1     1      1      1      1
```

The first five rows of the design are as follows:

```
 two1   two2   two3   two4   two5   two6   two7   two8   two9   two10   two11

  -1     -1      1      1      1      1      1      1      1       1      -1
   1      1      1      1     -1      1      1     -1     -1       1       1
  -1     -1     -1      1     -1     -1     -1     -1     -1       1      -1
  -1     -1     -1     -1      1      1      1     -1     -1      -1       1
  -1      1     -1     -1     -1     -1      1      1      1       1       1
```

```
thr1  thr2  thr3  thr4  thr5  thr6  thr7  thr8  thr9  thr10  thr11  thr12

  0     0     0     1     1     1    -1     1     1     1     -1     -1
  1    -1    -1     1     0     0    -1     1     1     0      1      0
  0    -1     0     0    -1    -1    -1     0    -1     0     -1      0
  0     1     0     1     1     0     1    -1    -1    -1      1      0
  1    -1     1     1     1     1     0    -1    -1     0     -1      1
```

This following steps create a design and block it. This example shows that it is okay if not all of the variables in the input design are used. The variables `Block`, `Run`, and `x4` are just copied from the input to the output. The following steps create the design:

```
%mktex(n=18, seed=396)

%mktblock(data=design, nblocks=2, factors=x1-x4, seed=292)

data key;
   input Brand $ Price Size;
   format price dollar5.2;
   datalines;
Acme 1.49   6
Apex 1.79   8
.    1.99  12
;

%mktlab(data=blocked, key=key)

proc print; run;
```

The results are as follows:

```
Block    Run    Brand    Price    Size    x4

  1       1     Acme     $1.49     12      2
          2     Acme     $1.79      6      3
          3     Acme     $1.79      8      2
          4     Acme     $1.99      8      1
          5     Acme     $1.99     12      1
          6     Apex     $1.49      8      3
          7     Apex     $1.49     12      3
          8     Apex     $1.79      6      1
          9     Apex     $1.99      6      2

  2       1     Acme     $1.49      6      1
          2     Acme     $1.49      6      2
          3     Acme     $1.79      8      3
          4     Acme     $1.99     12      3
```

```
 5     Apex     $1.49      8       1
 6     Apex     $1.79     12       1
 7     Apex     $1.79     12       2
 8     Apex     $1.99      6       3
 9     Apex     $1.99      8       2
```

This next example illustrates using the `labels=` option. This option is more typically used with `values=` input, rather than when you construct the `key=` data set yourself, but it can be used either way. This example is from a vacation choice example. The following steps create the design:

```
%mktex(3 ** 15,                     /* 15 three-level factors            */
       n=36,                        /* 36 runs                           */
       seed=17,                     /* random number seed                */
       maxtime=0)                   /* no more than 1 iter in each phase */

%mktblock(data=randomized, nblocks=2, factors=x1-x15, seed=448)

%macro lab;
   label X1  = 'Hawaii, Accommodations'
         X2  = 'Alaska, Accommodations'
         X3  = 'Mexico, Accommodations'
         X4  = 'California, Accommodations'
         X5  = 'Maine, Accommodations'
         X6  = 'Hawaii, Scenery'
         X7  = 'Alaska, Scenery'
         X8  = 'Mexico, Scenery'
         X9  = 'California, Scenery'
         X10 = 'Maine, Scenery'
         X11 = 'Hawaii, Price'
         X12 = 'Alaska, Price'
         X13 = 'Mexico, Price'
         X14 = 'California, Price'
         X15 = 'Maine, Price';
   format x11-x15 dollar5.;
%mend;
```

```
data key;
   length x1-x5 $ 16 x6-x10 $ 8 x11-x15 8;
   input x1 & $ x6 $ x11;
   x2  = x1;    x3 = x1;    x4 = x1;    x5 = x1;
   x7  = x6;    x8 = x6;    x9 = x6;   x10 = x6;
   x12 = x11;  x13 = x11;  x14 = x11;  x15 = x11;
   datalines;
Cabin            Mountains   999
Bed & Breakfast  Lake       1249
Hotel            Beach      1499
;

%mktlab(data=blocked, key=key, labels=lab)

proc contents p; ods select position; run;
```

The variable name, label, and format information are as follows:

```
                              The CONTENTS Procedure

                           Variables in Creation Order

        #    Variable    Type    Len    Format      Label

        1    x1          Char     16                Hawaii, Accommodations
        2    x2          Char     16                Alaska, Accommodations
        3    x3          Char     16                Mexico, Accommodations
        4    x4          Char     16                California, Accommodations
        5    x5          Char     16                Maine, Accommodations
        6    x6          Char      8                Hawaii, Scenery
        7    x7          Char      8                Alaska, Scenery
        8    x8          Char      8                Mexico, Scenery
        9    x9          Char      8                California, Scenery
       10    x10         Char      8                Maine, Scenery
       11    x11         Num       8    DOLLAR5.    Hawaii, Price
       12    x12         Num       8    DOLLAR5.    Alaska, Price
       13    x13         Num       8    DOLLAR5.    Mexico, Price
       14    x14         Num       8    DOLLAR5.    California, Price
       15    x15         Num       8    DOLLAR5.    Maine, Price
       16    Block       Num       8
       17    Run         Num       8
```

%MktLab Macro Options

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

Option	Description
`help`	(positional) "help" or "?" displays syntax summary
`cfill=`*character-string*	character fill value
`data=`*SAS-data-set*	input design data set
`dolist=`*do-list*	new values using a do-list syntax
`int=`*variable-list*	name of an intercept variable
`key=`*SAS-data-set*	`Key` data set
`labels=`*macro-name*	macro that provides labels and formats
`nfill=`*number*	numeric fill value
`options=noprint`	suppress the display of the variable mappings
`out=`*SAS-data-set*	output data set with recoded design
`prefix=`*variable-prefix*	prefix for naming variables
`statements=`*SAS-code*	add extra statements
`values=`*value-list*	the new values for all of the variables
`vars=`*variable-list*	list of variable names

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

```
%mktlab(help)
%mktlab(?)
```

cfill= *character-string*

specifies the fill value in the `key=` data set for character variables. See the `nfill=` option for more information about fill values. The default is `cfill=' '`.

data= *SAS-data-set*

specifies the input data set with the experimental design, usually created by the `%MktEx` macro. The default is `data=Randomized`. The factor levels in the `data=` data set must be consecutive integers beginning with 1.

dolist= *do-list*

specifies the new values, using a do-list syntax (`n TO m <BY p>`), for example: `dolist=1 to 10` or `dolist=0 to 9`. With asymmetric designs (not all factors have the same levels), specify the levels for the largest number of levels. For example, with two-level and three-level factors and `dolist= 0 to 2`, the two-level factors is assigned levels 0 and 1, and the three-level factors is assigned levels 0, 1, and 2. Do not specify both `values=` and `dolist=`. By default, when `key=`, `values=`, and `dolist=` are all not specified, the default value list comes from `dolist=1 to 100`.

int= *variable-list*

specifies the name of an intercept variable (column of ones), if you want an intercept added to the `out=`

data set. You can also specify a variable list instead of a variable name if you would like to make a list of variables with values all one. This can be useful, for example, for creating flag variables for generic choice models when the design is going to be used as a candidate set for the `%ChoicEff` macro.

key= *SAS-data-set*

specifies the input data set with the key to recoding the design. When `values=` or `dolist=` is specified, this data set is made for you. By default, when `key=`, `values=`, and `dolist=` are all not specified, the default value list comes from `dolist=1 to 100`.

labels= *macro-name*

specifies the name of a macro that provides labels, formats, or other additional information to the `key=` data set. For a simple format specification, it is easier to use `statements=`. For more involved specifications, use `labels=`. Note that you specify just the macro name, no percents on the `labels=` option. This option is illustrated in the following step:

```
%mktex(3 ** 4, n=18, seed=205)

%macro labs;
   label x1 = 'Sploosh' x2 = 'Plumbob'
         x3 = 'Platter' x4 = 'Moosey';
   format x1-x4 dollar5.2;
   %mend;

%mktlab(data=randomized, values=1.49 1.99 2.49, labels=labs)

proc print label; run;
```

The first part of the design is as follows:

```
Obs     Sploosh     Plumbob     Platter     Moosey

  1      $2.49       $2.49       $2.49      $1.49
  2      $2.49       $2.49       $1.99      $1.99
  3      $1.49       $1.49       $1.49      $1.49
  4      $1.99       $1.99       $2.49      $2.49
  .
  .
  .
```

nfill= *number*

specifies the fill value in the `key=` data set for numeric variables. For example, when the maximum number of levels is three, the last value in the `key=` data set for numeric two-level factors should have a value of `nfill=`, which by default is ordinary missing. If the macro tries to access one of these values, it displays a warning. If you would like ordinary missing (.) to be a legitimate level, specify a different `nfill=` value and use it for the extra places in the `key=` data set.

options= *options-list*

specifies binary options. By default, none of these options are specified. Specify values after `options=`.

`noprint`
suppresses the display of the variable mappings.

out= *SAS-data-set*

specifies the output data set with the final, recoded design. The default is `out=final`. 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.

prefix= *variable-prefix*

specifies a prefix for naming variables when `values=` is specified. For example, `prefix=Var` creates variables `Var1`, `Var2`, and so on. By default, the variables are `x1`, `x2`, ... This option is ignored when `vars=` is specified.

statements= *SAS-code*

is an alternative to `labels=` that you can use to add extra statements to the `key=` data set. For a simple format specification, it is easier to use `statements=`. For more involved specifications, use `labels=`. This option is illustrated in the following step:

```
%mktex(3 ** 4, n=18, seed=205)

%mktlab(data=randomized, values=1.49 1.99 2.49,
        vars=Sploosh Plumbob Platter Moosey,
        statements=format Sploosh Plumbob Platter Moosey dollar5.2)

proc print; run;
```

The first part of the design is as follows:

```
Obs    Sploosh    Plumbob    Platter    Moosey

  1     $2.49      $2.49      $2.49     $1.49
  2     $2.49      $2.49      $1.99     $1.99
  3     $1.49      $1.49      $1.49     $1.49
  4     $1.99      $1.99      $2.49     $2.49
  .
  .
  .
```

values= *value-list*

specifies the new values for all of the variables. If all variables will have the same value, it is easier to specify `values=` or `dolist=` than `key=`. When you specify `values=`, the `key=` data set is created for you. Specify a list of levels separated by blanks. If your levels contain blanks, separate them with two blanks. With asymmetric designs (not all factors have the same levels) specify the levels for the largest number of levels. For example, with two-level and three-level factors and `values=a b c`, the two-level factors are assigned levels `'a'` and `'b'`, and the three-level factors are assigned levels `'a'`, `'b'`, and `'c'`. Do not specify both `values=` and `dolist=`. By default, when `key=`, `values=`, and `dolist=` are all not specified, the default value list comes from `dolist=1 to 100`.

vars= *variable-list*
specifies a list of variable names when `values=` or `dolist=` is specified. If `vars=` is not specified with `values=`, then `prefix=` is used.

%MktLab 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.

%MktMDiff Macro

The `%MktMDiff` autocall macro analyzes MaxDiff (maximum difference or best-worst) data (Louviere 1991, Finn and Louviere 1992). The result of the analysis is a scaling of the attributes on a preference or importance scale. In a MaxDiff study, subjects are shown sets of messages or product attributes and are asked to choose the best (or most important) from each set as well as the worst (or least important). The design consists of:

- t attributes
- b sets (or blocks) of attributes with
- k attributes in each set

These are the parameters of a balanced incomplete block design or BIBD, which can be constructed from the `%MktBIBD` macro. Note, however, that you can use designs that are produced by the `%MktBIBD` macro but do not meet the strict requirements for a BIBD. See page 1111 for more information about this. To illustrate, a researcher is interested in preference for cell phones based on the attributes of the phones. The attributes are as follows:

Camera
Flip
Hands Free
Games
Internet
Free Replacement
Battery Life
Large Letters
Applications

Subjects are shown subsets of these 9 attributes and asked to pick which is the most important when they make a cell phone choice and which is the least important. We can use the `%MktBSize` macro as follows to get ideas about how many blocks to use and how many to show at one time:

```
   %mktbsize(nattrs=9, setsize=2 to 9, nsets=1 to 20)
```

The results of this step are as follows:

```
                                                                      n
            t            k        b          r            Lambda      Total
            Number of    Set      Number     Attribute    Pairwise    Sample
            Attributes   Size     of Sets    Frequency    Frequencies Size

                 9         3        12           4             1         36
                 9         4        18           8             3         72
                 9         5        18          10             5         90
                 9         6        12           8             5         72
                 9         8         9           8             7         72
```

With 9 attributes, there are five sizes that meet the necessary but not sufficient conditions for the existence of a BIBD. Of the candidates (3, 4, 5, 6, and 8), 4 or 5 seem like good choices. (Three seems a bit small and more than 5 seems a bit big given that we only have 9 attributes.) The following step creates a BIBD with $t = 9$ attributes, shown in $b = 18$ sets of size $k = 5$:

```
%mktbibd(nattrs=9, setsize=5, nsets=18, seed=377, out=sasuser.bibd)
```

The following table, produced by the macro, shows that the design is in fact a BIBD:

```
                    Attribute by Attribute Frequencies

                          1   2   3   4   5   6   7   8   9

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

This is the attribute by attribute frequency matrix. The diagonal elements of the matrix show how often each attribute occurs. Each of the $t = 9$ attributes occurs the same number of times (10 times). Furthermore, each of the $t = 9$ attributes occurs with each of the remaining 8 attributes exactly 5 times. These two constant values, one one the diagonal and one off, show that the design is a BIBD. The design is as follows:

```
                     Balanced Incomplete Block Design

                        x1     x2     x3     x4     x5

                         1      2      4      7      9
                         6      2      9      8      4
                         5      8      2      1      7
                         7      9      1      6      3
                         9      7      6      5      2
                         3      1      5      6      8
                         4      6      3      8      2
                         6      7      4      1      8
                         7      5      2      3      4
                         1      3      8      2      7
                         3      4      1      5      9
                         2      9      6      3      1
                         8      3      9      2      5
                         9      8      7      4      3
                         5      4      8      9      1
```

```
            2     1     5     4     6
            8     5     7     9     6
            4     6     3     7     5
```

The first set or consists of attributes 1, 2, 4, 7, and 9, which are as follows:

Camera
Flip
Games
Battery Life
Applications

Subjects will choose the best and worst from this and every other set.

The `%MktMDiff` macro reads the experimental design and the data, combines them, arranges them in the right form for analysis, and performs the analysis using a multinomial logit model. The data are arrayed so that each original MaxDiff set forms two choice sets in the analysis: one positively weighted set for the best choice and one negatively weighted set for the worst choice. The input data can come in one of eight forms:

`bw`
best then worst (e.g. `b1-b18 w1-w18`), and the data are attribute numbers (range from 1 to `nattrs=`t). If the `groups=` option is specified, for example with `groups=3`, the variables are `b1-b6 w1-w6` and three observations provide the information about all of the rows in the block design.

`wb`
worst then best (e.g. `w1-w18 b1-b18`), and the data are attribute numbers (range from 1 to `nattrs=`t). If the `groups=` option is specified, for example with `groups=3`, the variables are `w1-w6 b1-b6` and three observations provide the information about all of the rows in the block design.

`bwalt`
best then worst and alternating (e.g. `b1 w1 b2 w2 ... b18 w18`), and the data are attribute numbers (range from 1 to `nattrs=`t). If the `groups=` option is specified, for example with `groups=3`, the variables are `b1 w1 b2 w2 ... b6 w6` and three observations provide the information about all of the rows in the block design.

`wbalt`
worst then best and alternating (e.g. `w1 b1 w2 b2 ... w18 b18`), and the data are attribute numbers (range from 1 to `nattrs=`t). If the `groups=` option is specified, for example with `groups=3`, the variables are `w1 b1 w2 b2 ... w6 b6` and three observations provide the information about all of the rows in the block design.

`bwpos`
best then worst (e.g. `b1-b18 w1-w18`), and the data are positions (range from 1 to `setsize=`k). If the `groups=` option is specified, for example with `groups=3`, the variables are `b1-b6 w1-w6` and three observations provide the information about all of the rows in the block design.

`wbpos`
worst then best (e.g. `w1-w18 b1-b18`), and the data are positions (range from 1 to `setsize=`k). If the `groups=` option is specified, for example with `groups=3`, the variables are `w1-w6 b1-b6` and three observations provide the information about all of the rows in the block design.

`bwaltpos`
best then worst and alternating (e.g. `b1 w1 b2 w2 ... b18 w18`), and the data are positions (range from 1 to `setsize=`k). If the `groups=` option is specified, for example with `groups=3`, the variables are `b1 w1 b2 w2 ... b6 w6` and three observations provide the information about all of the rows in the block design.

`wbaltpos`
worst then best and alternating (e.g. `w1 b1 w2 b2 ... w18 b18`), and the data are positions (range from 1 to `setsize=`k). If the `groups=` option is specified, for example with `groups=3`, the variables are `w1 b1 w2 b2 ... w6 b6` and three observations provide the information about all of the rows in the block design.

Note that in all cases, any variable names can be used. For example, the variables could be `x1-x36`. In the case of `bwalt`, the odd numbered variables will correspond to best picks and the even numbered variables will correspond to worst picks.

The following example uses the design created previously:

```
title 'Best Worst Example with Cell Phone Attributes';

data bestworst;
   input Sub $ @4 (b1-b18 w1-w18) (1.);
   datalines;
 1 188661884399349653941955342212935494
 2 765358873891388493922673644336595554
 3 782126282892848564995993447213935655
 4 481363264246399187162125415351281453
 5 787168863811878175225995382352235293
 6 787658867891878667495965442313345453
 7 788171867736888187465395344393344453
 8 788771867711888687445353445353335254
 9 188778887896878687425323443242344454
10 788778887816888687445353444343344454
11 787778877816878667442353343343235453
12 787778387711898667425321443253544594
13 767668285791988687441951364232234194
14 187168877741878687445323445216334453
15 788176487116988675492393314339579457
16 267665615733884677442193342349545564
17 481191867813938266147956244139584594
18 725778814832585185141193643296944467
19 188678863811279263445123344253934596
20 728698612719281483265755285851944597
;
```

```
%let attrlist=Camera,Flip,Hands Free,Games,Internet
,Free Replacement,Battery Life,Large Letters,Applications;

%phchoice( on )

%mktmdiff(bw, nattrs=9, nsets=18, setsize=5, attrs=attrlist,
          data=bestworst, design=sasuser.bibd)
```

The DATA step reads 36 variables with data and a subject variable, which is ignored. The descriptions of each of the t attributes are listed in comma-delimited form and stored in a macro variable. (Note that when the list is split across lines, care is taken to ensure that the next attribute description, “Free Replacement” immediately follows the comma so that it will not begin with a leading blank.) The `%PHChoice` macro is used to customize the output from PROC PHREG, which the `%MktMDiff` macro calls, to look like the output from a discrete choice procedure instead of a survival analysis procedure. The `%MktMDiff` macro begins with a positional parameter that specifies the layout of the data. Positional parameters do not begin with a keyword and an equal sign. Because of the initial `bw` specification, the data are best then worst. The variables do not alternate. The data are attribute numbers not positions. The design has `nsets=18` sets, `nattrs=9` attributes, and `setsize=5` are shown at a time. The `attrlist` macro variable contains the list of the attributes. Note that a variable name and not the value of the variable are specified. In other words, `attrlist` not `&attrlist` is specified. The SAS data set with the data is called `bestworst`, and the SAS data set with the design is called `sasuser.bibd`.

The `%MktMDiff` macro begins by displaying the following summary of the input:

```
Var Order:   Best then Worst
Alternating: Variables Do Not Alternate
Data:        Attribute Numbers (Not Positions)
Best Vars:   b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15 b16 b17 b18
Worst Vars:  w1 w2 w3 w4 w5 w6 w7 w8 w9 w10 w11 w12 w13 w14 w15 w16 w17 w18
Attributes:  Camera
             Flip
             Hands Free
             Games
             Internet
             Free Replacement
             Battery Life
             Large Letters
             Applications
```

The data consists of the best variables then the worst variables. The best variables are `b1-b18,` and the worst variables are `w1-w18`. The data are attribute numbers (1 to $t = 9$) not positions (1 to $k = 5$). Finally, the descriptions of each of the t attributes are listed.

The following table is of interest as a check of the integrity of the input data:

```
              Summary of Subjects, Sets, and Chosen and Unchosen Alternatives

                     Number of        Number of           Chosen           Not
          Pattern      Choices     Alternatives     Alternatives        Chosen

                1           36              100               20            80
```

In the aggregate data set, there is one pattern of input and it occurs 36 times (18 best choices per subject plus 18 worst choices). There are 100 alternatives (5 attributes in a set examined by 20 individuals), each of the 20 subjects chose 1 ($20 \times 1 = 20$) and did not choose 4 ($20 \times 4 = 80$).

The final table of results is as follows:

```
                      Best Worst Example with Cell Phone Attributes
                          Multinomial Logit Parameter Estimates

                                   Parameter      Standard                    Pr >
                         DF         Estimate         Error    Chi-Square     ChiSq

      Large Letters       0                0             .             .       .
      Battery Life        1         -0.52009       0.17088        9.2635    0.0023
      Free Replacement    1         -1.01229       0.17686       32.7598    <.0001
      Camera              1         -1.30851       0.18232       51.5096    <.0001
      Applications        1         -1.93107       0.18741      106.1680    <.0001
      Flip                1         -2.00892       0.18788      114.3334    <.0001
      Internet            1         -2.44592       0.18731      170.5156    <.0001
      Hands Free          1         -2.47063       0.18781      173.0569    <.0001
      Games               1         -2.86329       0.18802      231.9065    <.0001
```

The parameter estimates are arranged from most preferred to least preferred. These results are also available in a SAS data set called `parmest`.

You could change the reference level using the `classopts=` option as follows:

```
%mktmdiff(bw, nattrs=9, nsets=18, setsize=5,
          attrs=attrlist, classopts=zero='Internet',
          data=bestworst, design=sasuser.bibd)
```

The new parameter estimates table is as follows:

```
                      Best Worst Example with Cell Phone Attributes
                          Multinomial Logit Parameter Estimates

                               Parameter      Standard
                          DF    Estimate         Error    Chi-Square    Pr > ChiSq

   Large Letters           1     2.44592       0.18731      170.5156        <.0001
   Battery Life            1     1.92583       0.18533      107.9789        <.0001
   Free Replacement        1     1.43363       0.18658       59.0424        <.0001
   Camera                  1     1.13741       0.18484       37.8650        <.0001
   Applications            1     0.51485       0.18056        8.1305        0.0044
   Flip                    1     0.43700       0.18045        5.8650        0.0154
   Hands Free              1    -0.02471       0.17655        0.0196        0.8887
   Games                   1    -0.41737       0.17375        5.7702        0.0163
```

Note that the parameter estimates all change by a constant amount. If you take the original estimate for the internet parameter and subtract it from all of the original estimates, you get the new estimates.

Experimental Design for a MaxDiff Study

The experimental design for a MaxDiff study is a block design. A set of t attributes are organized into b blocks or sets of size k. Ideally, we prefer to use a balanced incomplete block design or BIBD. In a BIBD, each of the t attributes appears the same number of times, and each of the t attributes appears with every other attribute the same number of times. However, BIBDs do not exist for every combination of b, t, and k that might be of interest. Hence, in some situations we are forced to use unbalanced block designs in which each of the t attributes appears the same number of times, but not all of the $t(t-1)/2$ pairs of attributes appear the same number of times. Fortunately, while a BIBD is nice, it is not required for a MaxDiff analysis.

There is a set of necessary but not sufficient conditions for the existence of a BIBD. See page 971. You can use the `%MktBSize` macro to get lists of designs that meet these conditions for ranges of b, t, and k, for example, as follows.

```
   %mktbsize(nattrs=1 to 20, setsize=2 to 0.5 * t, nsets=t to 100)
```

This step also shows that you can restrict the sizes being considered using the values of b, t, and k. For example, this step will never consider a set size more than one half the number attributes. Nor will it consider designs with fewer sets than attributes. The results of this step are not shown.

Then you can use the `%MktBIBD` macro to find either a BIBD or an unbalanced block design, for example, as follows:

```
   %mktbibd(nattrs=12, setsize=6, nsets=22, seed=104)
```

There is never a guarantee that the `%MktBIBD` macro will find a BIBD, even when one exists. It does a computerized search using PROC OPTEX. However, it does a good job of finding small BIBDs and coming very close for larger ones—certainly close enough for most MaxDiff studies. However, just because you run the `%MktBIBD` macro and get something back does not mean it is suitable for use. Like any design, you need to examine its properties and ensure that it meets your needs. Consider, for example, the following step, which creates a design with 20 attributes shown in 16 sets of size 5:

```
%mktbibd(nattrs=20, setsize=5, nsets=16, seed=292, out=sasuser.maxdiffdes)
```

The attribute by attribute frequency matrix is as follows:

```
                      Attribute by Attribute Frequencies

           1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20

       1   4   1   1   1   1   1   1   0   0   0   1   1   1   1   1   1   1   1   1   1
       2       4   1   1   1   0   1   1   1   1   0   1   1   1   1   1   0   1   1   1
       3           4   1   1   1   1   1   1   1   1   1   0   1   1   1   1   1   0   0
       4               4   1   1   0   1   1   1   1   1   1   0   0   1   1   1   1   1
       5                   4   1   1   1   1   1   1   0   1   1   1   0   1   0   1   1
       6                       4   1   1   1   1   0   1   1   1   1   1   0   1   1   1
       7                           4   1   1   1   1   1   1   0   0   1   1   1   1   1
       8                               4   0   0   1   1   1   1   1   1   1   1   1   1
       9                                   4   0   1   1   1   1   1   1   1   1   1   1
      10                                       4   1   1   1   1   1   1   1   1   1   1
      11                                           4   1   1   1   1   1   0   1   1   1
      12                                               4   1   1   1   0   1   0   1   1
      13                                                   4   1   1   1   1   1   0   0
      14                                                       4   0   1   1   1   1   1
      15                                                           4   1   1   1   1   1
      16                                                               4   1   0   1   1
      17                                                                   4   1   1   1
      18                                                                       4   1   1
      19                                                                           4   0
      20                                                                               4
```

The design (not shown) has $bk = 16 \times 5 = 80$ entries, and each of the 20 attributes appears exactly 4 times. However, the off-diagonal attribute by attribute frequencies are not all the same, so this is not a BIBD. Furthermore, some of the attributes never appear with other attributes, so this design is not a good candidate for a MaxDiff study.

The `%MktBIBD` macro also tries to optimize the attribute by position frequencies. For this design, they are as follows:

```
                  Attribute by Position Frequencies

                             1  2  3  4  5

                          1  1  1  0  1  1
                          2  0  1  1  1  1
                          3  1  1  0  1  1
                          4  1  1  1  1  0
                          5  1  0  1  1  1
                          6  1  1  1  0  1
                          7  1  1  0  1  1
                          8  1  1  1  0  1
                          9  1  0  1  1  1
                         10  0  1  1  1  1
                         11  1  1  1  1  0
                         12  1  1  1  0  1
                         13  1  1  0  1  1
                         14  1  1  1  0  1
                         15  0  1  1  1  1
                         16  1  1  1  1  0
                         17  1  1  1  1  0
                         18  1  0  1  1  1
                         19  1  0  1  1  1
                         20  0  1  1  1  1
```

Given the b, t, and k specifications, these frequencies are optimal. There are no 2's, and there is exactly one zero in each row, that is, each attribute appears in all but one position.

The following step requests 20 blocks or sets instead of 16:

```
%mktbibd(nattrs=20, setsize=5, nsets=20, seed=292, out=sasuser.maxdiffdes)
```

The number of sets is increased by 4 since $4k = t = 20$. You have to increase b by multiples of 4 (given this k and t) so that each attribute can appear an equal number of times. The attribute by attribute frequency matrix is as follows:

```
              Attribute by Attribute Frequencies

     1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

 1   5  1  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  1
 2      5  1  1  1  1  2  1  1  1  1  1  1  1  1  1  1  1  1  1
 3         5  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2  1  1
 4            5  1  1  1  1  1  1  1  1  1  2  1  1  1  1  1  1
 5               5  1  1  1  1  1  1  1  1  1  1  1  1  1  1  2
 6                  5  1  0  1  1  1  1  1  1  1  2  1  1  2  1
 7                     5  1  1  1  1  1  1  1  1  1  1  1  1  1
 8                        5  1  1  1  1  1  1  1  2  1  1  2  1
 9                           5  1  1  1  1  1  2  1  1  1  1  1
10                              5  1  1  2  1  1  1  1  1  1  1
11                                 5  1  1  1  1  1  2  1  1  1
12                                    5  1  1  1  1  1  1  1  1
13                                       5  1  1  1  1  1  1  1
14                                          5  1  1  1  1  1  1
15                                             5  1  1  1  1  1
16                                                5  1  1  0  1
17                                                   5  1  1  1
18                                                      5  1  1
19                                                         5  1
20                                                            5
```

Now each attribute appears 5 times, and the pairwise frequencies are all 1's and 2's with two 0's. Given the *b*, *t*, and *k* specifications, these frequencies are not quite what we hoped. There are more 2's than 0's, so ideally we would like the 0's to go away. The macro finds designs with this pattern of frequencies over and over, so it is likely that a design of this size with no zero frequencies is impossible. Block designs are subject to all sorts of Sudoku-like rules—if this treatment occurs with that treatment in this block, then it cannot occur with this other treatment in that other block. Every time PROC OPTEX tries to change a 0 frequency to a 1, it changes a 1 to a 0 rather than a 2 to a 1.

The attribute by position frequencies are as follows:

```
                    Attribute by Position Frequencies

                               1  2  3  4  5

                           1   1  1  1  1  1
                           2   1  1  1  1  1
                           3   1  1  1  1  1
                           4   1  1  1  1  1
                           5   1  1  1  1  1
                           6   1  1  1  1  1
                           7   1  1  1  1  1
                           8   1  1  1  1  1
                           9   1  1  1  1  1
                          10   1  1  1  1  1
                          11   1  1  1  1  1
                          12   1  1  1  1  1
                          13   1  1  1  1  1
                          14   1  1  1  1  1
                          15   1  1  1  1  1
                          16   1  1  1  1  1
                          17   1  1  1  1  1
                          18   1  1  1  1  1
                          19   1  1  1  1  1
                          20   1  1  1  1  1
```

These are perfect. Each attribute appears in every position exactly once.

The following step requests 24 blocks or sets instead of 16 or 20:

```
   %mktbibd(nattrs=20, setsize=5, nsets=24, seed=292, out=sasuser.maxdiffdes)
```

The attribute by attribute frequency matrix is as follows:

```
                         Attribute by Attribute Frequencies

      1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

   1  6  1  1  1  1  1  1  1  1  2  2  1  2  1  2  1  1  1  2  1
   2     6  2  1  1  1  2  1  1  1  1  2  1  1  1  2  2  1  1  1
   3        6  1  2  2  1  2  1  1  1  1  1  1  1  1  1  2  1  1
   4           6  2  1  1  1  2  2  1  2  1  1  1  1  1  1  2  1
   5              6  1  2  1  1  1  1  1  2  1  1  1  1  1  1  2
   6                 6  1  1  1  1  1  1  1  1  1  2  2  1  2  2
   7                    6  1  2  2  1  1  1  2  1  1  1  1  1  1
   8                       6  1  1  2  1  1  2  1  2  2  1  1  1
   9                          6  1  1  1  2  1  1  1  1  2  1  2
  10                             6  1  1  1  1  1  2  2  1  1  1
  11                                6  2  1  1  1  1  1  2  1  2
  12                                   6  1  1  2  1  1  1  2  1
  13                                      6  2  1  1  2  1  1  1
  14                                         6  2  1  1  1  2  1
  15                                            6  1  1  2  1  2
  16                                               6  1  2  1  1
  17                                                  6  1  1  1
  18                                                     6  1  1
  19                                                        6  1
  20                                                           6
```

Now each attribute appears 6 times, and the pairwise frequencies are all 1's and 2's with no 0's.

The attribute by position frequencies are as follows:

```
                         Attribute by Position Frequencies

                                  1  2  3  4  5

                              1   1  1  2  1  1
                              2   1  2  1  1  1
                              3   1  1  1  2  1
                              4   1  1  1  1  2
                              5   2  1  1  1  1
                              6   1  1  2  1  1
                              7   1  1  1  2  1
                              8   1  1  2  1  1
                              9   1  2  1  1  1
                             10   1  2  1  1  1
                         11  1  1  1  2  1
                         12  1  1  1  1  2
                         13  2  1  1  1  1
                         14  1  2  1  1  1
                         15  2  1  1  1  1
                         16  1  1  1  2  1
                         17  2  1  1  1  1
                         18  1  1  1  1  2
                         19  1  1  1  1  2
                         20  1  1  2  1  1
```

Again, given the b, t, and k specifications, these frequencies are optimal. There are no 0's or 3's, and there is exactly one 2 in each row, that is, each attribute appears in all positions but one more than the others. Often times, we are content to use a design with patterns of frequencies like we see here, not perfect, but optimal given the design specifications.

You could run the following step to see if a BIBD is possible if you change the block size:

```
   %mktbsize(nattrs=20, setsize=5, nsets=t to 500)
```

The results are as follows:

```
                                                                 n
        t             k       b          r            Lambda         Total
        Number of     Set     Number     Attribute    Pairwise       Sample
        Attributes    Size    of Sets    Frequency    Frequencies    Size

            20          5        76          19             4          380
```

You would need 76 sets before you would have any hope of finding a BIBD. If you are willing to change the number of messages or the number of messages shown at one time, then you have more possibilities. The following step investigates:

```
   %mktbsize(nattrs=18 to 22, setsize=4 to 6, nsets=t to 500)
```

The results are as follows:

```
                                                                 n
        t             k       b          r            Lambda         Total
        Number of     Set     Number     Attribute    Pairwise       Sample
        Attributes    Size    of Sets    Frequency    Frequencies    Size

            18          4       153          34             6          612
            18          5       306          85            20         1530
            18          6        51          17             5          306
```

```
             19          4        57          12            2         228
             19          5       171          45           10         855
             19          6        57          18            5         342
             20          4        95          19            3         380
             20          5        76          19            4         380
             20          6       190          57           15        1140
             21          4       105          20            3         420
             21          5        21           5            1         105
             21          6        28           8            2         168
             22          4        77          14            2         308
             22          5       462         105           20        2310
             22          6        77          21            5         462
```

We can try to find a BIBD with 21 attributes, and 21 sets of size 5 as follows:

```
   %mktbibd(nattrs=21, setsize=5, nsets=21, seed=292, out=sasuser.maxdiffdes)
```

The attribute by attribute frequency matrix is as follows:

```
                        Attribute by Attribute Frequencies

             1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21

         1   5  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
         2      5  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
         3         5  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
         4            5  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
         5               5  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
         6                  5  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
         7                     5  1  1  1  1  1  1  1  1  1  1  1  1  1  1
         8                        5  1  1  1  1  1  1  1  1  1  1  1  1  1
         9                           5  1  1  1  1  1  1  1  1  1  1  1  1
        10                              5  1  1  1  1  1  1  1  1  1  1  1
        11                                 5  1  1  1  1  1  1  1  1  1  1
        12                                    5  1  1  1  1  1  1  1  1  1
        13                                       5  1  1  1  1  1  1  1  1
        14                                          5  1  1  1  1  1  1  1
        15                                             5  1  1  1  1  1  1
        16                                                5  1  1  1  1  1
        17                                                   5  1  1  1  1
        18                                                      5  1  1  1
        19                                                         5  1  1
        20                                                            5  1
        21                                                               5
```

The attribute by position frequencies are as follows:

```
                    Attribute by Position Frequencies

                              1  2  3  4  5

                          1   1  1  1  1  1
                          2   1  1  1  1  1
                          3   1  1  1  1  1
                          4   1  1  1  1  1
                          5   1  1  1  1  1
                          6   1  1  1  1  1
                          7   1  1  1  1  1
                          8   1  1  1  1  1
                          9   1  1  1  1  1
                         10   1  1  1  1  1
                         11   1  1  1  1  1
                         12   1  1  1  1  1
                         13   1  1  1  1  1
                         14   1  1  1  1  1
                         15   1  1  1  1  1
                         16   1  1  1  1  1
                         17   1  1  1  1  1
                         18   1  1  1  1  1
                         19   1  1  1  1  1
                         20   1  1  1  1  1
                         21   1  1  1  1  1
```

These results are perfect. The art of design is determining what ranges of parameters work for you and then finding the best design that works for what you need.

%MktMDiff Macro Options

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

Option	Description
`layout`	(positional) choice data set structure
	(positional) "help" or "?" displays syntax summary
`attrs=`*macro-variable*	macro variable with attribute descriptions
`b=`*b*	number of sets (alias for `nsets=`)
`classopts=`*options-list*	class variable options
`data=`*SAS-data-set*	input data set
`design=`*SAS-data-set*	design data set
`group=`*k*	number of groups in the `design=` data set
`k=`*k*	set size (alias for `setsize=`)
`nattrs=`*t*	number of attributes (alias for `t=`)

Option	Description
`nsets=`*b*	number of sets (alias for `b=`)
`options=nocode`	just create the OUT= data set
`options=noanalysis`	just create the OUT= and OUTCODED= data sets
`options=nosort`	do not sort the parameter estimates table
`options=nodrop`	do not drop the variable or parameter column
`options=rescale`	do `rescale=` even when dubious
`out=`*SAS-data-set*	merged data and design data set
`outcoded=`*SAS-data-set*	coded data set
`outparm=`*SAS-data-set*	parameter estimates data set
`rescale=`*value*	table of rescaled parameter estimates
`setsize=`*k*	set size (alias for `k=`)
`t=`*t*	number of attributes or messages (alias for `nattrs=`)
`vars=`*variable-list*	`data=` data set variables

Help Option

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

```
%mktmdiff(help)
%mktmdiff(?)
```

Required Options

The `layout`, `b=` or `nsets=`, `k=` or `setsize=`, `t=` or `nattrs=`, and the `design=` options must be specified. The `layout` option is a positional parameter and must be specified first.

layout

specifies a nonoptional positional parameter that indicates the structure of the choice data. Just specify a value, not `layout=`. The value has several components, you can use any case, and spaces between the pieces are optional. Values include `b`, `w`, `alt`, and `pos`. Both a `b` and a `w` must be specified. When `b` comes before `w`, the best variables come before the worst variables. Otherwise, when `w` comes before `b`, the worst variables come before the best variables. You can optionally specify `alt` as well which means that the variables alternate (e.g. `b1 w1 b2 w2 b3 w3` where the `b` variables are the best variables and the `w` variables are the worst variables). Otherwise it is expected that all of one type appear then all of the other type. By default, the data are assumed to be the numbers of the attributes that were chosen (e.g. 1 to 6 when there are a total of `nattrs=6` attributes). If instead (say with `setsize=3`), the data are 1 to 3 indicating the position of the chosen attribute, then you must specify `pos` as well.

Examples:

`bw`
(or `b w`)
best then worst, e.g. `b1-b6 w1-w6` or even `x1-x12` (if $2 \times b = 12$). While the values need to be best then worst, the variable names do not need to reflect this. The data are attribute numbers.

`wb`
(or `w b`)
worst then best, e.g. `w1-w6 b1-b6` or even `x1-x12` (if $2 \times b = 12$). While the values need to be worst then best, the variable names do not need to reflect this. The data are attribute numbers.

`bwalt`
(or `b w alt` or `alt b w` and so on)
best then worst and alternating, e.g. `b1 w1 b2 w2 ...` or even `x1-x12` (if $2 \times b = 12$). While the values need to alternate best then worst, the variable names do not need to reflect this. The data are attribute numbers.

`wbalt`
(or `w b alt` or `alt w b` and so on)
worst then best and alternating, e.g. `w1 b1 w2 b2 ...` or even `x1-x12` (if $2 \times b = 12$). While the values need to alternate worst then best, the variable names do not need to reflect this. The data are attribute numbers.

`bwpos`
(or `b w alt` or `b pos w` and so on)
best then worst, and the data are positions.

`wbpos`
(or `w b pos` or `w pos b` and so on)
worst then best, and the data are positions.

`bwaltpos`
(or `b w alt pos` or `alt b pos w` and so on)
best then worst and alternating, and the data are positions.

`wbaltpos`
(or `w b alt pos` or `alt w pos b` and so on)
worst then best and alternating, and the data are positions.

design= *SAS-data-set*

specifies the data set with the design. It is typically a BIBD in the format produced by the `out=` option of the `%MktBIBD` macro. You must specify this option. All numeric variables are assumed to contain the BIBD, unless there is exactly one extra variable, and the first or last variable is called `Group`. In that case, the group variable is ignored.

nattrs= *t*
t= *t*

specifies the number of attributes or messages. The `nattrs=` and `t=` options are aliases. This option (in one of its two forms) must be specified. The "`t`" in `t=` stands for treatments and corresponds to typical BIBD notation.

nsets= *b*
b= *b*

specifies the number of sets. The `nsets=` and `b=` options are aliases. This option (in one of its two forms) must be specified. The "`b`" in `b=` stands for blocks and corresponds to typical BIBD notation.

setsize= *k*
k= *k*
specifies the number of attributes or messages shown at one time (in a set). The `setsize=` and `k=` options are aliases. This option (in one of its two forms) must be specified. The `k=` option is named using typical BIBD notation.

Data Set Options

The `design=` option is a required data set option that is listed in the previous section. The following are optional data set options.

data= *SAS-data-set*
specifies the input data set. By default, the last data set created is used.

out= *SAS-data-set*
specifies the output data set in a form ready for coding. The default is `out=maxdiff`.

outcoded= *SAS-data-set*
specifies the output data set that has been coded by PROC TRANSREG. The default is `outcoded=coded`.

outparm= *SAS-data-set* specifies the output data set with the parameter estimates table. The default is `outparm=parmest`. If the `rescale=` option is specified, the requested results are added to this data set.

Other Options

attrs= *macro-variable*
specifies the name of a macro variable that contains a comma delimited list of attribute descriptions. Alternatively, after the name you can specify a single nonblank delimiter character. Examples: `attrs=myattrlist`, `attrs=myattrs -`, (using a dash as a delimiter). If you do not specify this option, the default attribute names are “A1”, “A2”, and so on. The macro variable must have a name that does not match any of the macro parameter names (so you cannot specify `attrs=attrs` for example). The name also must not match any macro names used internally by the macro. In the unlikely event you specify an invalid name, an error is displayed. Most names, including any name that contains '`att`' that is not '`attrs`' or '`nattrs`' is always going to work.

classopts= *options-list*
specifies options to apply to the class variable, for example: `classopts=effects`. The class variable in the final analysis data set contains the descriptions of the attributes that are specified in the `attrs=` macro variable. The default is `classopts=zero=none`. A reference cell coding is used, and the reference level is displayed with a parameter estimate of zero. If you specify a null value (`classopts=`,) then the default reference cell coding is used and the reference level is not displayed. You can control the reference level by specifying `classopts=zero=`*'reference-level'* and naming the appropriate reference level. The reference level will exactly match one of the descriptions in the `attrs=` macro variable.

group= *g*

specifies the number of blocks of choice sets. By default, when `group=` is not specified, it is assumed that the design consists of one big group. Otherwise, with n subjects in each group and `group=`g there are $n \times g$ rows of data each consisting of b/g best values and b/g worst values. The number of subjects in each group must be the same. It is also assumed that the design is sorted by the group variable. If the design is made by the `%MktBIBD` macro, this will always be the case. If the `design=` data set has $k + 1$ variables and either the first variable or the last variable is called `Group` (case is ignored), then that variable is ignored and not treated as part of the design. Otherwise, the `design=` data set is required to have k variables.

options= *options-list*

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

`nocode`
just create the OUT= data set but do not code or do the analysis.

`noanalysis`
just create the OUT= and OUTCODED= data sets but do not do the analysis.

`nosort`
do not sort the parameter estimates table by the parameter estimates.

`nodrop`
do not drop the variable or parameter column from the parameter estimates table.

`rescale`
ignore the usual restriction that `rescale=` option can only be used in the context of the default `classopts=`. If you use this option, you must ensure that you are only using the `zero=` option to change the reference level or are otherwise doing something that will not change the coding from reference cell to something else.

rescale= *value*

specifies various ways to rescale the parameter estimates. You can specify any of the following values:

`default`
ordinary parameter estimates.

`center`
parameter estimates are centered.

`p`
parameter estimates are scaled to probabilities:

$$\frac{\exp(\hat{\beta}_i)}{\sum_{j=1}^{m} \exp(\hat{\beta}_j)}$$

`p100`
parameter estimates are scaled to probabilities then multiplied by 100:

$$\frac{100 \exp(\hat{\beta}_i)}{\sum_{j=1}^{m} \exp(\hat{\beta}_j)}$$

`adjusted`
`adj`
parameter estimates are adjusted by the number of attributes in each set. First, $\hat{\boldsymbol{\beta}}$ is centered then scaled as follows:

$$\frac{\exp(\hat{\beta}_i)}{\exp(\hat{\beta}_i) + k - 1}$$

Finally, the adjusted values are rescaled to sum to 1.

`adjusted100`
`adj100`
parameter estimates are adjusted by the number of attributes in each set. First, $\hat{\boldsymbol{\beta}}$ is centered then scaled as follows:

$$\frac{\exp(\hat{\beta}_i)}{\exp(\hat{\beta}_i) + k - 1}$$

Finally, the adjusted values are rescaled to sum to 100.

`all`
the default and all rescaled values are reported.

Note that `rescale=` and `rescale=default` are equivalent. When this option is specified (or any option besides `rescale=default` is specified), an additional table is displayed with the rescaled parameter estimates. You can specify multiple values and get multiple columns in the results table. Example: `rescale=default adj100`. All specified rescalings are added to the `outparm=` data set. These rescalings were suggested by Sawtooth Software (2005, 2007).

Do not use both the `rescale=` option and the `classopts=` option unless you are only changing details of the reference cell coding. If you use these options together, you must ensure that you are only using the `zero=` option to change the reference level or are otherwise doing something that will not change the coding from reference cell to something else. In that case, you can specify `options=rescale` to allow the analysis to proceed.

vars= *variable-list*

specifies the variables in the `data=` data set that contain the data. There must be $2 \times b$ variables in this list (from `nsets=`b). The default is all numeric variables in the data set.

%MktMDiff 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.

%MktMerge Macro

The `%MktMerge` autocall macro merges a data set containing a choice design with choice data. See the following pages for examples of using this macro in the design chapter: 149 and 176. Also see the following pages for examples of using this macro in the discrete choice chapter: 325, 371, 387, 437, 522, and 529. Additional examples appear throughout this chapter. The following shows a typical example of using this macro:

```
%mktmerge(design=rolled, data=results, out=res2,
          nsets=18, nalts=5, setvars=choose1-choose18)
```

The `design=` data set comes from the `%MktRoll` macro. The `data=` data set contains the data, and the `setvars=` variables in the `data=` data set contain the numbers of the chosen alternatives for each of the 18 choice sets. The `nsets=` option specifies the number of choice sets, and the `nalts=` option specifies the number of alternatives. The `out=` option names the output SAS data set that contains the experimental design and a variable `c` that contains 1 for the chosen alternatives (first choice) and 2 for unchosen alternatives (second or subsequent choice).

When the `data=` data set contains a blocking variable, name it in the `blocks=` option. When there is blocking, it is assumed that the `design=` data set contains blocks of *nalts* × *nsets* observations. The `blocks=` variable must contain values 1, 2, ..., *n* for *n* blocks. The following example uses the `%MktMerge` macro with blocking:

```
%mktmerge(design=rolled, data=results, out=res2, blocks=form,
          nsets=18, nalts=5, setvars=choose1-choose18)
```

%MktMerge Macro Options

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

Option	Description
`help`	(positional) “help” or “?” displays syntax summary
`blocks=`1 | *variable*	blocking variable
`data=`*SAS-data-set*	input SAS data set
`design=`*SAS-data-set*	input SAS choice design data set
`nalts=`*n*	number of alternatives
`nsets=`*n*	number of choice sets
`out=`*SAS-data-set*	output SAS data set
`setvars=`*variable-list*	variables with the data
`statements=`*SAS-statements*	additional statements

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

```
%mktmerge(help)
%mktmerge(?)
```

You must specify the `design=`, `nalts=`, `nsets=`, and `setvars=` options.

blocks= 1 | *variable*

specifies either a 1 (the default) if there is no blocking or the name of a variable in the `data=` data set that contains the block number. When there is blocking, it is assumed that the `design=` data set contains blocks of *nalts* × *nsets* observations, one set per block. The `blocks=` variable must contain values 1, 2, ..., *n* for *n* blocks.

data= *SAS-data-set*

specifies an input SAS data set with data for the choice model. By default, the `data=` data set is the last data set created.

design= *SAS-data-set*

specifies an input SAS data set with the choice design. This data set could have been created, for example, with the `%MktRoll` or `%ChoicEff` macros. This option must be specified.

nalts= *n*

specifies the number of alternatives. This option must be specified.

nsets= *n*

specifies the number of choice sets. This option must be specified.

out= *SAS-data-set*

specifies the output SAS data set. If `out=` is not specified, the DATA*n* convention is used. This data set contains the experimental design and a variable `c` that contains 1 for the chosen alternatives (first choice) and 2 for unchosen alternatives (second or subsequent choice).

setvars= *variable-list*

specifies a list of variables, one per choice set, in the `data=` data set that contains the numbers of the chosen alternatives. It is assumed that the values of these variables range from 1 to *nalts*. This option must be specified.

statements= SAS-statements

specifies additional statements like `format` and `label` statements. This option is illustrated in the following step:

```
%mktmerge(design=rolled, data=results, out=res2, blocks=form,
          nsets=&n, nalts=&m, setvars=choose1-choose&n,
          statements=%str(price = input(put(price, price.), 5.);
                          format scene scene. lodge lodge.;))
```

%MktMerge 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.

%MktOrth Macro

The %MktOrth autocall macro lists some of the 100% orthogonal main-effects plans that the %MktEx macro can generate. See page 106 for an example of using this macro in the design chapter. Also see the following pages for examples of using this macro in the discrete choice chapter: 342 and 660. Additional examples appear throughout this chapter.

Mostly, you use this macro indirectly; it is called by the %MktEx macro. However, you can directly call the %MktOrth macro to see what orthogonal designs are available and decide which ones to use. The following step requests all the designs in the catalog with 100 or fewer runs and two-level through six-level factors (with no higher-level factors.)

```
   %mktorth(maxn=100, maxlev=6)
```

The macro creates data sets and displays no output except the following notes:

```
   NOTE: The data set WORK.MKTDESLEV has 347 observations and 9 variables.
   NOTE: The data set WORK.MKTDESCAT has 347 observations and 3 variables.
```

This next step generates the entire catalog of 119,852 designs[*] including over 62,000 designs in 512 runs that are not generated by default:

```
   %mktorth(maxlev=144, options=512)
```

This step might take on the order of several minutes to run.

This next step generates the catalog of approximately 57 thousand designs including designs with up to 144-level factors:

```
   %mktorth(maxlev=144)
```

This step might take on the order of several minutes to run. Unless you really want to see all of the designs, you can make the %MktOrth macro run much faster by specifying smaller values for range= or maxn= (which control the number of runs) and maxlev= (which controls the maximum number of factor levels and the number of variables in the MKTDESLEV data set) than the defaults (range=n le 1000, maxn=1000, maxlev=50). The maximum number of levels you can specify is 144.

The following step lists the first few and the last few designs in the catalog:

```
   proc print data=mktdeslev(where=(n le 12 or n ge 972));
      var design reference;
      id n; by n;
      run;
```

[*]Elsewhere in this chapter, the size of the orthogonal array catalog is reported to be 117,556. The discrepancy is due to the 2296 designs that are explicitly in the catalog and have more than 513 runs. Most are constructed from the parent array 24^8 in 576 runs (which is useful for making Latin Square designs). The rest are constructed from Hadamard matrices.

Some of the results are as follows:

```
   n             Design                 Reference

   4    2 **  3                         Hadamard

   6    2 **  1  3 **  1                Full-Factorial

   8    2 **  7                         Hadamard
        2 **  4            4 **  1      Fractional-Factorial

   9             3 **  4                Fractional-Factorial

  10    2 **  1            5 **  1      Full-Factorial

  12    2 ** 11                         Hadamard
        2 **  4  3 **  1                Orthogonal Array
        2 **  2            6 **  1      Orthogonal Array
                 3 **  1   4 **  1      Full-Factorial

 972    2 **971                         Hadamard

 976    2 **975                         Hadamard
        2 **972            4 **  1      Orthogonal Array
        2 **969            4 **  2      Orthogonal Array
        2 **968            8 **  1      Orthogonal Array
        2 **966            4 **  3      Orthogonal Array

 984    2 **983                         Hadamard
        2 **980            4 **  1      Orthogonal Array

 992    2 **991                         Hadamard
        2 **988            4 **  1      Orthogonal Array
        2 **985            4 **  2      Orthogonal Array
        2 **984            8 **  1      Orthogonal Array
        2 **982            4 **  3      Orthogonal Array
        2 **979            4 **  4      Orthogonal Array
        2 **976            4 **  5      Orthogonal Array
        2 **976           16 **  1      Orthogonal Array
        2 **973            4 **  6      Orthogonal Array
        2 **970            4 **  7      Orthogonal Array

1000    2 **999                         Hadamard
        2 **996            4 **  1      Orthogonal Array
```

In most ways, the catalog stops at 513 runs. The exceptions include: 24^8 in 576 runs, Hadamard designs up to $n = 1000$, and designs easily constructed from those Hadamard designs (by creating, 4, 8, 16, and so on level factors).

The following step displays the first few designs and variables in the MKTDESLEV data set:

```
proc print data=mktdeslev(where=(n le 12));
   var design reference x1-x6;
   id n; by n;
   run;
```

Some of the results are as follows:

```
  n          Design            Reference            x1  x2  x3  x4  x5  x6

  4  2 **  3                   Hadamard              0   3   0   0   0   0

  6  2 **  1  3 **  1          Full-Factorial        0   1   1   0   0   0

  8  2 **  7                   Hadamard              0   7   0   0   0   0
     2 **  4           4 **  1 Fractional-Factorial  0   4   0   1   0   0

  9           3 **  4          Fractional-Factorial  0   0   4   0   0   0

 10  2 **  1           5 **  1 Full-Factorial        0   1   0   0   1   0

 12  2 ** 11                   Hadamard              0  11   0   0   0   0
     2 **  4  3 **  1          Orthogonal Array      0   4   1   0   0   0
     2 **  2           6 **  1 Orthogonal Array      0   2   0   0   0   1
              3 **  1  4 **  1 Full-Factorial        0   0   1   1   0   0
```

If you just want to display a list of designs, possibly selecting on n, the number of runs, you can use the MKTDESCAT data set. However, if you would like to do more advanced processing, based on the numbers of levels of some of the factors, you can use the `outlev=mktdeslev` data set to select potential designs. You can look at the level information in MKTDESLEV and see the number of two-level factors in `x2`, the number of three-level factors in `x3`, ..., the number of fifty-level factors is in `x50`, ..., and the number of 144-level factors in `x144`. The number of one-level factors, `x1`, is always zero, but `x1` is available so you can make arrays (for example, `array x[50]`) and have `x[2]` refer to `x2`, the number of two-level factors, and so on.

Say you are interested in the design $2^5 3^5 4^1$. The following steps display some of the ways in which it is available:

```
%mktorth(maxn=100)

proc print data=mktdeslev noobs;
   where x2 ge 5 and x3 ge 5 and x4 ge 1;
   var n design reference;
   run;
```

Some of the results are as follows:

```
 n                   Design                         Reference

72    2 ** 44  3 ** 12   4 **  1                  Orthogonal Array
72    2 ** 43  3 **  8   4 **  1   6 **  1        Orthogonal Array
72    2 ** 37  3 ** 13   4 **  1                  Orthogonal Array
72    2 ** 36  3 **  9   4 **  1   6 **  1        Orthogonal Array
72    2 ** 35  3 ** 12   4 **  1   6 **  1        Orthogonal Array
.
.
.
```

The following steps illustrate one way that you can see all of the designs in a certain range of sizes:

```
%mktorth(range=12 le n le 20)

proc print; id n; by n; run;
```

The results are as follows:

```
 n                   Design                    Reference

12    2 ** 11                                  Hadamard
      2 **  4  3 **  1                         Orthogonal Array
      2 **  2            6 **  1               Orthogonal Array
               3 **  1   4 **  1               Full-Factorial

14    2 **  1            7 **  1               Full-Factorial

15             3 **  1   5 **  1               Full-Factorial

16    2 ** 15                                  Hadamard
      2 ** 12            4 **  1               Fractional-Factorial
      2 **  9            4 **  2               Fractional-Factorial
      2 **  8            8 **  1               Fractional-Factorial
      2 **  6            4 **  3               Fractional-Factorial
      2 **  3            4 **  4               Fractional-Factorial
                         4 **  5               Fractional-Factorial

18    2 **  1  3 **  7                         Orthogonal Array
      2 **  1            9 **  1               Full-Factorial
               3 **  6   6 **  1               Orthogonal Array
```

```
     20    2 ** 19                                 Hadamard
           2 **  8          5 **  1                Orthogonal Array
           2 **  2         10 **  1                Orthogonal Array
                            4 **  1   5 **  1      Full-Factorial
```

The `%MktOrth` macro can output the lineage of each design, which is the set of steps that the `%MktEx` macro uses to create it. The following steps illustrate this option:

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

proc print noobs;
   where index(design, '2 ** 11') and index(design, '3 ** 12');
   run;
```

The results are as follows:

```
                 n         Design             Reference

                36    2 ** 11  3 ** 12    Orthogonal Array

                                       Lineage

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

The design $2^{11}3^{12}$ in 36 runs starts out as a single 36-level factor, 36^1. Then 36^1 is replaced by $3^{12}12^1$. Finally, 12^1 is replaced by 2^{11} resulting in $2^{11}3^{12}$.

%MktOrth Macro Options

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

Option	Description
`help`	(positional) "help" or "?" displays syntax summary
`filter=`*n*	extra filtering of the design catalog
`maxn=`*n*	maximum number of runs of interest
`maxlev=`*n*	maximum number of levels
`options=lineage`	construct the design lineage
`options=mktex`	the macro is being called from the `%MktEx` macro
`options=mktruns`	the macro is being called from the `%MktRuns` macro
`options=parent`	lists only parent designs
`options=dups`	suppress duplicate and inferior design filtering
`options=512`	adds some designs in 512 runs
`outall=`*SAS-data-set*	output data set with all designs
`outcat=`*SAS-data-set*	design catalog data set
`outlev=`*SAS-data-set*	output data set with the list of levels
`range=`*range-specification*	number of runs of interest

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

```
%mktorth(help)
%mktorth(?)
```

filter= *n*

specifies extra design catalog filtering. Usually, you will never have to use this option. By default, `%MktOrth` filters out inferior designs. On the one hand, this process is expensive, but it also saves some time and resources by limiting the information that must be processed. Care is taken to do only the minimum amount of work to filter. However, this gets complicated. If you specify `maxlev=144`, the maximum, you will get perfect filtering of duplicates in a reasonable amount of time. `%MktOrth` uses internal and optimized constants to avoid doing extra work. Unfortunately, these constants might not be optimal for any other `maxlev=` value. This is almost never going to be a real issue. If however, you want to specify both `maxlev=` and ensure you get better filtering, specify `filter=`*n* (for some *n*) and you might get a few more inferior designs filtered out. The *n* value increases how deeply into the catalog `%MktOrth` searches for inferior designs. Larger values find more designs to exclude at a cost of greater run time. The following steps use both the default filtering and specify `filter=1000`:

```
%mktorth(maxlev=20)
%mktorth(maxlev=20, filter=1000)
```

The latter specification removes on the order of thirty more designs but at cost of much slower run time. Actually, `filter=27` is large enough for this example, and it has a negligible effect on run time, but you have to run the macro multiple times to figure that out. Values of a couple hundred or so are probably always going to be sufficient.

maxlev= *n*

specifies the maximum number of levels to consider. Specify a value n, such that $2 \le n \le 144$. The default is `maxlev=50`. This option controls the number of `x` variables in the `outlev=` data set. It also excludes from consideration designs with factors of more than `maxlev=` levels so it affects the number of rows in the output data sets. Note that specifying `maxlev=`*n* does not preclude designs with more than n-level factors from being used as parents for other designs, it just precludes the larger designs from being output. For example, with `maxlev=3`, the design $3^{12}12^1$ in 36 runs is used to make $2^{11}3^{12}$ before the $3^{12}12^1$ design is discarded. Specifying smaller values will make the macro run faster. With the maximum, `maxlev=144`, run time to generate the entire catalog can be on the order of several minutes.

maxn= *n*

specifies the maximum number of runs of interest. Specifying small numbers (e.g. $n \le 200$) will make the macro run faster.

options= *options-list*

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

`lineage`
construct the design lineage, which is the set of instructions on how the design is made.

`mktex`
specifies that the macro is being called from the `%MktEx` macro and just the `outlev=` data set is needed. The macro takes short cuts to make it run faster doing only what the `%MktEx` macro needs.

`mktruns`
specifies that the macro is being called from the `%MktRuns` macro and just the `outlev=` data set is needed. The macro takes short cuts to make it run faster doing only what `%MktRuns` needs.

`parent`
specifies that only parent designs should be listed.

`dups`
specifies that the `%MktRuns` macro should not filter out duplicate and inferior designs from the catalog. This can be useful when you are creating a data set for the `cat=` option in the `%MktEx` macro.

`512`
adds some larger designs in 512 runs with mixes of 16, 8, 4, and 2-level factors to the catalog, which gives added flexibility in 512 runs at a cost of much slower run time. This option replaces the default parent design $4^{160}32^1$ with $16^{32}32^1$.

outall= *SAS-data-set*

specifies the output data set with all designs. This data set is not created by default. This data set is like the `outlev=` data set, except larger. The `outall=` data set includes *all* of the `%MktEx` design catalog, including all of the smaller designs that can be trivially made from larger designs by dropping factors. For example, when the `outlev=` data set has `x2`=2 `x3`=2, then the `outall=` data set has that design and also `x1=2 x3=1`, `x1=1 x3=2`, and `x1=1 x2=1`. When you specify `outall=` you must also specify a reasonably small `range=` or `maxn=` value. Otherwise, the `outall=` specification will take a *long* time and create a *huge* data set, which will very likely be too large to store on your computer.

outcat= *SAS-data-set*

specifies the output data set with the catalog of designs that the `%MktEx` macro can create. The default is `outcat=MktDesCat`.

outlev= *SAS-data-set*

specifies the output data set with the list of designs and 50 (by default) more variables, `x1-x50`, which includes: `x2` – the number of two-level factors, `x3` – the number of three-level factors, and so on. The default is `outlev=MktDesLev`. The number of `x` variables is determined by the `maxlev=` option.

range= *range-specification*

specifies the number of runs of interest. Specify a range involving `n`, where `n` is the number of runs. Your range specification must be a logical expression involving `n`. Examples:
`range=n=36`

```
range=18 le n le 36
range=n eq 18 or n eq 36
```

%MktOrth 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.

The Orthogonal Array Catalog

This section contains information about the orthogonal array catalog. While this information is interesting and sometimes useful, this section can be skipped by most readers.

The `%MktOrth` macro maintains the orthogonal array catalog that the `%MktEx` and `%MktRuns` macros use. The `%MktOrth` macro instructs the `%MktEx` macro on both what orthogonal arrays exist and how to make them. In most cases, the `%MktOrth` macro is called by the `%MktEx` and `%MktRuns` macros and you never have to worry about it. However, you can use the `%MktOrth` macro to find out what designs exist in the orthogonal array catalog. This can be particularly useful for the orthogonal array specialist who seeks to understand the catalog and find candidates for undiscovered orthogonal arrays. Most designs explicitly appear in the catalog. Others are explicitly entered into the catalog, but they normally do not appear. Still others do not explicitly appear in the catalog, but the `%MktEx` macro can still figure out how to make them. These points and other aspects of the orthogonal array catalog are discussed in this section.

First, consider a design that is explicitly in the catalog. The following steps create a list of orthogonal arrays in 18 runs, display that list, then create the design using the `%MktEx` macro:

```
   %mktorth(range=n=18, options=lineage)

   data _null_;
      set;
      design = compbl(design);
      put 'Design:  ' design / 'Lineage: ' lineage /;
      run;

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

Of course the first two steps are not needed if your only goal is to make the design. The results are as follows:

```
Design:  2 ** 1 3 ** 7
Lineage: 18 ** 1 : 18 ** 1 > 3 ** 6 6 ** 1 : 6 ** 1 > 2 ** 1 3 ** 1

Design:  2 ** 1 9 ** 1
Lineage: 18 ** 1 : 18 ** 1 > 2 ** 1 9 ** 1 (parent)

Design:  3 ** 6 6 ** 1
Lineage: 18 ** 1 : 18 ** 1 > 3 ** 6 6 ** 1 (parent)
```

Three designs are listed. The third is the parent array $3^6 6^1$. The first is a child array $2^1 3^7$, which is constructed from the parent array $3^6 6^1$ by replacing the six-level factor with a two-level factor and a three-level factor. This can be seen in the lineage, which is explained in more detail later in this section. The second array is the full-factorial design $2^1 9^1$. All three of these arrays are explicitly in the catalog and can be directly produced by the `%MktEx` macro.

The catalog actually contains one more array that is created, but by default, it is discarded as inferior before the catalog is output or before `%MktEx` can use the catalog. You can use the `dups` option to see the duplicate and inferior arrays that would normally be discarded. The following steps illustrate this option:

```
%mktorth(range=n=18, options=lineage dups)

data _null_;
   set;
   design = compbl(design);
   put 'Design:  ' design / 'Lineage: ' lineage /;
   run;
```

The results are as follows:

```
Design:  2 ** 1 3 ** 7
Lineage: 18 ** 1 : 18 ** 1 > 3 ** 6 6 ** 1 : 6 ** 1 > 2 ** 1 3 ** 1

Design:  2 ** 1 3 ** 4
Lineage: 18 ** 1 : 18 ** 1 > 2 ** 1 9 ** 1 : 9 ** 1 > 3 ** 4

Design:  2 ** 1 9 ** 1
Lineage: 18 ** 1 : 18 ** 1 > 2 ** 1 9 ** 1 (parent)

Design:  3 ** 6 6 ** 1
Lineage: 18 ** 1 : 18 ** 1 > 3 ** 6 6 ** 1 (parent)
```

The new array is the second array in the list, $2^1 3^4$. It is made from the full-factorial design, $2^1 9^1$ by replacing the nine-level factor with 4 three-level factors. The resulting array, $2^1 3^4$, is inferior to $2^1 3^7$ in that the former has three fewer three-level factors than the latter. Hence, by default, it is discarded. This does not imply, however, that the $2^1 3^4$ design is a simple subset of the larger design, $2^1 3^7$. The smaller design might or might not be a subset, but typically it will not be. By default, the `%MktEx` macro operates under the assumption that orthogonal and balanced factors are interchangeable, so it always prefers arrays with more factors to arrays with fewer factors. By default, it will never produce the array $2^1 3^4$ that is based on the full-factorial design, $2^1 9^1$. It will always create $2^1 3^4$ from the first five columns of $2^1 3^7$.

The `%MktEx` macro provides you with a way to get these designs that are automatically excluded from the catalog. More generally, when there are multiple designs that meet your criteria, it gives you a way to explicitly choose which design you get. All you have to do is run the `%MktOrth` macro yourself, select which design you want, and feed just that one line of the catalog into the `%MktEx` macro. The following steps illustrate this option:

```
%mktorth(range=n=18, options=lineage dups)

data lev;
   set mktdeslev;
   where lineage ? '3 ** 4';
   run;

%mktex(2 3 ** 4,                    /* 1 two-level and 4 three-level factors*/
       n=18,                        /* 18 runs                              */
       cat=lev,                     /* OA catalog comes from lev data set   */
       out=alternative)             /* name of output design                */

%mktex(2 3 ** 4, n=18, out=default)
```

The `where` clause in the DATA step selects only one of the arrays since the other arrays that have more than 4 three-level factors have '3 ** 6' not '3 ** 4' in their lineage.

The default and the alternative (or "inferior") arrays are displayed next:

Default Array					Alternative Array				
x1	x2	x3	x4	x5	x1	x2	x3	x4	x5
1	1	1	1	1	1	1	1	1	1
1	1	2	1	3	1	1	2	3	2
1	1	3	2	3	1	1	3	2	3
1	2	1	3	1	1	2	1	3	3
1	2	2	2	2	1	2	2	2	1
1	2	3	2	1	1	2	3	1	2
1	3	1	3	2	1	3	1	2	2
1	3	2	1	2	1	3	2	1	3
1	3	3	3	3	1	3	3	3	1
2	1	1	2	2	2	1	1	1	1
2	1	2	3	1	2	1	2	3	2
2	1	3	3	2	2	1	3	2	3
2	2	1	1	3	2	2	1	3	3
2	2	2	3	3	2	2	2	2	1
2	2	3	1	2	2	2	3	1	2
2	3	1	2	3	2	3	1	2	2
2	3	2	2	1	2	3	2	1	3
2	3	3	1	1	2	3	3	3	1

If only the first three factors had been requested, the two arrays would have been the same. This is because the first three factors form a full-factorial design. The remaining factors are different in the two arrays. Both are orthogonal and balanced, however, they differ in terms of their aliasing structure. In other words, they differ in terms of which effects are confounded. The GLM procedure can be used to examine the aliasing structure of a design. It provides a list of be estimable functions. Since GLM is a modeling procedure, it requires a dependent variable. Hence, the first step is to add a dependent variable y, which can be anything for our purposes, to each design. The following steps display the aliasing structure for main effects only:

```
data d;
   set default;
   y = 1;
   run;

data i;
   set alternative;
   y = 1;
   run;

proc glm data=d;
   ods select galiasing;
   model y = x1-x5 / e aliasing;
   run; quit;

proc glm data=i;
   ods select galiasing;
   model y = x1-x5 / e aliasing;
   run; quit;
```

The results for the default design are as follows:

```
                    General Form of Aliasing Structure

                                Intercept
                                x1
                                x2
                                x3
                                x4
                                x5
```

The results for the alternative design are as follows:

```
                    General Form of Aliasing Structure

                                Intercept
                                x1
                                x2
                                x3
                                x4
                                x5
```

The two results are the same, and they show that all effects can be estimated.

Next, we will try the same thing, but this time adding two-way interactions to the model. The following steps illustrate:

```
proc glm data=d;
   ods select galiasing;
   model y = x1-x5 x1|x2|x3|x4|x5@2 / e aliasing;
   run; quit;

proc glm data=i;
   model y = x1-x5 x1|x2|x3|x4|x5@2 / e aliasing;
   run; quit;
```

Note that the `x1-x5` list could be omitted from the independent variable specification, but leaving it in serves to list the main-effect terms first. The results for the default design are as follows:

```
                    General Form of Aliasing Structure

                                Intercept
                                x1
                                x2
                                x3
                                x4
                                x5
                                x1*x2
                                x1*x3
                                x2*x3
                                x1*x4
                                x2*x4
                                x3*x4
                                x1*x5
                                x2*x5
                                x3*x5
                                x4*x5
```

All main effects and two-factor interactions are estimable.

The results for the alternative design are as follows:

```
                    General Form of Aliasing Structure

                  Intercept - 2*x2*x5 - 2*x3*x5 - 2*x4*x5
                  x1
                  x2 + 2*x2*x5 + 0.5*x3*x5 - 1.5*x4*x5
                  x3 - 1.5*x2*x5 + 2*x3*x5 + 0.5*x4*x5
                  x4 + 0.5*x2*x5 - 1.5*x3*x5 + 2*x4*x5
                  x5 + 2*x2*x5 + 2*x3*x5 + 2*x4*x5
                  x1*x2
                  x1*x3
                  x2*x3 + 0.5*x2*x5 - 0.5*x3*x5
                  x1*x4
                  x2*x4 - 0.5*x2*x5 + 0.5*x4*x5
                  x3*x4 + 0.5*x3*x5 - 0.5*x4*x5
                  x1*x5
```

The first estimable function is a function of the intercept and 3 two-way interactions. In other words, the intercept is confounded with 3 of the two-way interactions. We can estimate the intercept if the two-way interactions are zero or negligible. The second estimable function is the first factor, x1. It is not confounded with any other effect in the model. The third estimable function is a combination of the second factor, x2 and 3 two-way interactions. In most cases, lower-order terms are confounded with higher-order terms. Both sets of results assume that all three-way and higher-way interactions are zero, since they are not specified in the model. You can add all interactions to the models as follows:

```
proc glm data=d;
   ods select galiasing;
   model y = x1-x5 x1|x2|x3|x4|x5@5 / e aliasing;
   run; quit;

proc glm data=i;
   model y = x1-x5 x1|x2|x3|x4|x5@5 / e aliasing;
   run; quit;
```

The first two terms from the default design are as follows:

```
Intercept + 27*x1*x5 + 37.75*x2*x5 + 120.5*x1*x2*x5 + 45*x1*x3*x5 +
62.917*x2*x3*x5 + 200.83*x1*x2*x3*x5 + 9*x4*x5 + 63*x1*x4*x5 + 97.083*x2*x4*x5 +
269.17*x1*x2*x4*x5 + 33*x3*x4*x5 + 129*x1*x3*x4*x5 + 202.58*x2*x3*x4*x5 +
522.17*x1*x2*x3*x4*x5

x1 - 13.5*x1*x5 - 28.75*x2*x5 - 80*x1*x2*x5 - 22.5*x1*x3*x5 - 47.917*x2*x3*x5 -
133.33*x1*x2*x3*x5 - 22.5*x1*x4*x5 - 61.083*x2*x4*x5 - 159.67*x1*x2*x4*x5 -
6*x3*x4*x5 - 43.5*x1*x3*x4*x5 - 124.58*x2*x3*x4*x5 - 307.67*x1*x2*x3*x4*x5
```

Again, lower-order terms are confounded with higher-order terms. This is the nature of using anything less than a full-factorial design. You typically assume that higher-order interactions are zero or negligible and hope for the best.

At least in terms of the aliasing structure and the two-way interactions, it appears that the default design is better than the alternative design. There are no guarantees, and if the aliasing structure is ever really important to you, you should consider and evaluate the other arrays in the catalog.

The `%MktOrth` macro maintains a very large design catalog. However, the number of possible orthogonal arrays is infinite, and even for finite n, the total number starts getting very large after $n = 143$. Hence, it is impossible for the catalog to be complete. Still, there are some orthogonal designs are not in the `%MktOrth` catalog that the `%MktEx` macro still knows how to make. They are in some sense larger than the arrays that exist in the catalog, but they have a regular enough structure that they can be created without an explicit lineage from the catalog. This is illustrated in the following steps:

```
%mktorth(range=n=12*13, options=lineage)

proc print; run;

%mktex(12 13, n=12*13)
```

The catalog has only one design in $12 \times 13 = 156$ runs, and it is a Hadamard matrix with 155 two-level factors. The catalog entry is as follows:

```
                  Obs      n      Design      Reference         Lineage

                   1      156     2 **155     Hadamard     2 ** 155 (parent)
```

Still, the `%MktEx` macro can construct a full-factorial design with a twelve-level and thirteen-level factor. The following steps show the part of the catalog that contains full-factorial designs:

```
%mktorth;

proc print; where reference ? 'Full'; run;
```

The largest n in the results (not shown) is 143. The goal is for the catalog to have complete coverage up to 143 runs and good coverage beyond that. Full-factorial designs beyond 143 runs are not needed for making child arrays, so they are not included. However, `%MktEx` is capable of recognizing and making many full-factorial designs with more than 143 runs. The following steps provide another example:

```
%mktorth(range=n=1008, options=lineage)

%mktex(2 ** 1007, n=1008)
```

There are no designs with 1008 runs in the catalog, but the `%MktEx` macro recognizes that this as a Hadamard design from the Paley 1 family, and makes it using the same code that makes smaller Hadamard matrices. (More precisely, it recognizes 1008 as 4×252, and it recognizes $252 - 1$ as prime and makable with the Paley 1 construction. The final design is the Kronecker product of Hadamard matrices of order 4 and of order 252.)

There is one more class of designs that the `%MktEx` macro can find even when they are not in the catalog. In the following example, the `%MktEx` macro finds the design 29^{30} in 841 runs:

```
%mktorth(range=n=29 * 29, options=lineage)

%mktex(29 ** 30, n=29*29)
```

This is a regular fractional-factorial design (the number of runs and all factor levels are a power of the same prime, 29) that PROC FACTEX (which the `%MktEx` macro calls) can find even though it is not in the `%MktOrth` catalog.

Similarly, in 256 runs, there are designs where the factor levels are powers of two that are not in the catalog, yet the `%MktEx` macro can make them. This is illustrated in the following steps:

```
%mktorth(range=n=256, options=lineage, maxlev=64)

proc print data=mktdeslev; where x64 and x4 ge 10; run;

%mktex(4 ** 10 64, n=256)
```

The results of the PROC PRINT step (not shown) show that the design $4^{10}64^{1}$ is not in the design catalog. Nevertheless, the tools in the `%MktEx` macro (specifically, PROC FACTEX) succeed in constructing this design. (Note that mixes of 2, 4, 8, and 16 level-factors in 256 runs are completely covered by the catalog.)

In 128 runs, most designs are created directly by the `%MktEx` macro using information in the design lineage. However, some, even though they are explicitly in the catalog, are created by PROC FACTEX without using the design lineage from the catalog.

Next, we will return to the topic of design lineage. The design lineage is a set of instructions for making a design with smaller level factors from a design with higher-level factors. Previously, we saw the following design in 18 runs:

```
Design:  2 ** 1 3 ** 7
Lineage: 18 ** 1 : 18 ** 1 > 3 ** 6 6 ** 1 : 6 ** 1 > 2 ** 1 3 ** 1
```

It starts in the `%MktEx` macro as a single 18-level factor. That is what the first part of the lineage, '`18 ** 1`' specifies. Then the 18-level factor is replaced by 6 three-level factors and one six-level factors. This is specified by the lineage fragment: `18 ** 1 > 3 ** 6 6 ** 1`. Finally, the six-level factor is replaced by a two-level factor and a three level factor: `6 ** 1 > 2 ** 1 3 ** 1`. The code that makes $3^{6}6^{1}$ in 18 runs is the same code that replaces an 18-level factor in the design $3^{18}18^{1}$ in 54 runs, or replaces an 18-level factor in 72, 108, or more runs.

The following steps show the lineage for the design $2^{7}4^{5}8^{15}$ in 128 runs:

```
%mktorth(range=n=128, options=lineage)

data _null_;
   set mktdeslev;
   if x2 eq 7 and x4 eq 5 and x8 eq 15;
   design = compbl(design);
   put design / lineage /;
   run;
```

The design is shown on the first line of the following output and the lineage on the second:

```
2 ** 7 4 ** 5 8 ** 15
8 ** 16 16 ** 1 : 16 ** 1 > 4 ** 5 : 8 ** 1 > 2 ** 4 4 ** 1 : 4 ** 1 > 2 ** 3
```

The parent array is $8^{16}16^1$ not a single 128-level factor. The starting point is a single n-level factor for all arrays under 145 runs with the exception of some in 128 runs. Then the sixteen-level factor is replaced by 5 four-level factors. One of the eight-level factors is replaced by $2^4 4^1$. A four-level factor is replaced by 3 two-level factors.

The following steps show a lineage for the design 2^{31} in 32 runs:

```
%mktorth(range=n=32, options=lineage)

data _null_;
   set mktdeslev;
   if x2 eq 31;
   design = compbl(design);
   put design / lineage /;
   run;
```

The design is shown on the first line of the following output and the lineage on the second:

```
2 ** 31
32 ** 1 : 32 ** 1 > 4 ** 8 8 ** 1 : 8 ** 1 > 2 ** 4 4 ** 1 : 4 ** 1 > 2 ** 3
```

This shows that the design starts as a single 32-level factor. It is replaced by $4^8 8^1$. Next, 8^1 is replaced by $2^4 4^1$. Finally, 4^1 is replaced by 2^3. Implicit in these instructions is the fact that substitutions can occur more than once. In this case, every four-level factor is replaced by 3 two-level factors. While multiple substitutions can occur for higher-level factors, in practice, multiple substitutions usually occur for only four-level factors and sometimes six-level or eight-level factors.

%MktPPro Macro

The `%MktPPro` autocall macro makes optimal partial-profile designs (Chrzan and Elrod 1995) from block designs and orthogonal arrays.[*] See the following pages for examples of using this macro in the discrete choice chapter: 645, 650, 654, 656, 659, and 662. Additional examples appear throughout this chapter.

An incomplete block design is a list of t treatments (in this case attributes) that appear together in b blocks. Each block contains a subset ($k < t$) of the treatments. An unbalanced block design is an incomplete block design where every treatment appears with the same frequency, but pairwise frequencies (the number of times each treatment appears with each other treatment in a block) are not constant. When both treatment and pairwise frequencies are constant, the design is a balanced incomplete block design (BIBD). We can make optimal partial-profile designs with BIBDs (the best) and with unbalanced block designs as well. While you can more generally use any incomplete block design, the results will not generally be optimal.

In a certain class of partial-profile designs, a block design specifies which attributes vary in each choice set, and an orthogonal array specifies how they vary. This next example makes an *optimal* partial-profile choice design with 16 binary attributes, four of which vary at a time in 80 choice sets. The following steps create and evaluate the partial-profile design:

```
%mktex(4 2 ** 4, n=8, seed=306)

proc sort data=randomized out=randes(drop=x1);
   by x2 x1;
   run;

proc print noobs data=randes; run;

%mktbibd(b=20, nattrs=16, setsize=4, seed=104)

%mktppro(design=randes, ibd=bibd, print=f p)

%choiceff(data=chdes,              /* candidate set of choice sets         */
          init=chdes,              /* initial design                       */
          initvars=x1-x16,         /* factors in the initial design        */
          model=class(x1-x16 / sta),/* model with stdz orthogonal coding   */
          nsets=80,                /* number of choice sets                */
          nalts=2,                 /* number of alternatives               */
          rscale=                  /* relative D-efficiency scale factor   */
          %sysevalf(80 * 4 / 16),  /* 4 of 16 attrs in 80 sets vary        */
          beta=zero)               /* assumed beta vector, Ho: b=0         */
```

The results of these steps are shown following some more explanation of the process of making a partial-profile choice design using this method. Two input designs are input to the `%MktPPro` macro. The first is an orthogonal array. The orthogonal array must be a p^k subset of an array $p^k s^1$ in $p \times s$ runs with $k \leq s$. The following designs are examples of suitable orthogonal arrays:

[*]The ideas implemented in this macro are based on ideas from and discussions with Don Anderson. Don is not referenced in this book nearly as much as he should be. Much of my work can be traced to conversations with Don.

2^4 in 8 runs, selected from $2^4 4^1$ in 8 runs
3^3 in 9 runs, selected from $3^3 3^1$ in 9 runs
4^4 in 16 runs, selected from $4^4 4^1$ in 16 runs
3^6 in 18 runs, selected from $3^6 6^1$ in 18 runs

You can use the `%MktOrth` or `%MktRuns` macros to see if the orthogonal array of interest exists. The following steps list the arrays that work:

```
%mktorth(options=parent, maxlev=144)

data x(keep=n design);
   set mktdeslev;
   array x[144];
   c = 0; one = 0; k = 0;
   do i = 1 to 144;
      c + (x[i] > 0); /* how many differing numbers of levels */
      if x[i] > 1 then do; p = i; k = x[i]; end; /* p^k */
      if x[i] = 1 then do; one + 1; s = i;  end; /* s^1 */
      end;
   if c = 1 then do; c = 2; one = 1; s = p; k = p - 1; end;
   if c = 2 and one = 1 and k > 2 and s * p = n;
   design = compbl(left(design));
   run;

proc print; run;
```

The rows of the orthogonal array must be sorted into the right order. Each submatrix of s rows in this kind of orthogonal array is called a difference scheme, and submatrices 2 through p are obtained from the preceding submatrix by adding 1 (in the appropriate Galois field). You will not get the right results if you use any other kind of orthogonal array.

There are several ways to create an orthogonal array that is sorted in the right way. The preceding `%MktEx` step requests the s-level factor first, then it sorts the randomized design by the first p-level factor followed by the s-level factor. Finally, it discards the s-level factor. Alternatively, you could first request one of the p-level factors, then request the s-level factor, then request the remaining $(k-1)$ p-level factors. Then in the `out=design` data set, *after* the array is created, discard `x2`, the s-level factor. Note that you cannot drop the second factor in the `%MktEx` step by specifying `out=design(drop=x2)`, because `%MktEx` will drop the variable and then sort, and your rows will be in the wrong order. The former approach has the advantage of being less likely to produce alternatives that are constant (that is, in one alternative, all attributes are absent and in other alternatives, all attributes are present).

The orthogonal array is as follows:

```
 x2    x3    x4    x5

  1     1     2     1
  1     2     2     2
  1     1     1     2
  1     2     1     1
  2     2     1     2
  2     1     1     1
  2     2     2     1
  2     1     2     2
```

In this array, if you take the first submatrix of $s = 4$ observations and change 1 to 2 and 2 to 1, you get the second submatrix. This shows for two-level factors that the design is in the right order. More generally, when p is prime, the $i + 1$ submatrix can be made from the *ith* submatrix by adding 1 in a field with p elements. For example, with $p = 3$ and elements 1, 2, and 3, then $1 + 1 = 2$, $2 + 1 = 3$, $3 + 1 = 1$. You do not really need to worry about any of this though, because when you create the orthogonal array using the method outlined above, your array will always be in the right order.

The second input data set contains the block design. In the example above, the BIBD is as follows:

```
      Balanced Incomplete Block Design

 x1    x2    x3    x4

 15    16     5     6
  4     1    16    12
  5    13     4     7
  9     4     6     8
 14     8    13    16
 12    14     5    10
 13     9    12     3
  7     8     1    10
 16     7     3    11
  8    12    11    15
 10     3    15     4
  1     6     3    14
  2    11     4    14
 15     7    14     9
  6     2     7    12
 11     5     9     1
 13    15     2     1
 16    10     9     2
 11     6    10    13
  3     2     8     5
```

This design has `b=20` rows, so the final choice design will have 20 blocks of choice sets (in this case 20 blocks of 4 choice sets). The number of attributes is $6 = 16$, which is specified by `nattrs=16`. The number of attributes in each block is specified by the `setsize=4` option. This design specifies that in the first block of $s = 4$ choice sets, attributes 15, 16, 5, and 6 will vary; in the second block of 4 choice sets, attributes 4, 1, 16, and 12 will vary; and so on. Alternative $i = 1, \ldots\ p$, in each block of s choice sets is made from a submatrix of s runs in the orthogonal array. For example, alternative 1 of the first s choice sets is made from the first s runs in the orthogonal array, and alternative 2 is made from the next s runs in the orthogonal array, and so on. Each attribute should appear the same number of times in the block design. The following $t = 16$ by $t = 16$ attribute frequencies matrix shows how often each attribute appears with every other attribute in our design:

```
                    Attribute by Attribute Frequencies

              1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16

         1    5   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
         2        5   1   1   1   1   1   1   1   1   1   1   1   1   1   1
         3            5   1   1   1   1   1   1   1   1   1   1   1   1   1
         4                5   1   1   1   1   1   1   1   1   1   1   1   1
         5                    5   1   1   1   1   1   1   1   1   1   1   1
         6                        5   1   1   1   1   1   1   1   1   1   1
         7                            5   1   1   1   1   1   1   1   1   1
         8                                5   1   1   1   1   1   1   1   1
         9                                    5   1   1   1   1   1   1   1
        10                                        5   1   1   1   1   1   1
        11                                            5   1   1   1   1   1
        12                                                5   1   1   1   1
        13                                                    5   1   1   1
        14                                                        5   1   1
        15                                                            5   1
        16                                                                5
```

Each attribute appears 5 times and is paired with every other attribute exactly once. The fact that values above the diagonal are constant, in this case all ones, makes this block design a BIBD. We always want the diagonal elements to all be constant. The number of choice sets is s times the number of blocks (rows in the design), in this case, $4 \times 20 = 80$. The number of attributes is the maximum value in the BIBD (in this case, `nattrs=`$t = 16$). The number of attributes that vary at any one time is `setsize=`$k \leq s$. All factors have p levels, and all choice sets have p alternatives. The final report from the `%ChoicEff` macro is as follows:

```
                          Final Results

                   Design                   1
                   Choice Sets             80
                   Alternatives             2
                   Parameters              16
                   Maximum Parameters      80
                   D-Efficiency       20.0000
                   Relative D-Eff    100.0000
                   D-Error             0.0500
                   1 / Choice Sets     0.0125

           Variable                                    Standard
    n        Name       Label    Variance    DF         Error

    1        x11        x1 1       0.05       1       0.22361
    2        x21        x2 1       0.05       1       0.22361
    3        x31        x3 1       0.05       1       0.22361
    4        x41        x4 1       0.05       1       0.22361
    5        x51        x5 1       0.05       1       0.22361
    6        x61        x6 1       0.05       1       0.22361
    7        x71        x7 1       0.05       1       0.22361
    8        x81        x8 1       0.05       1       0.22361
    9        x91        x9 1       0.05       1       0.22361
   10        x101       x10 1      0.05       1       0.22361
   11        x111       x11 1      0.05       1       0.22361
   12        x121       x12 1      0.05       1       0.22361
   13        x131       x13 1      0.05       1       0.22361
   14        x141       x14 1      0.05       1       0.22361
   15        x151       x15 1      0.05       1       0.22361
   16        x161       x16 1      0.05       1       0.22361
                                             ==
                                             16
```

The variances are constant, and the relative *D*-efficiency is 100. *D*-efficiency is 20, which is 4 / 16 of the number of choice sets, 80. Since 25% of the attributes vary, *D*-efficiency is 25% of what we would expect in a full-profile generic choice design. We specified `rscale=20` so that relative *D*-efficiency could be based on the true maximum. The macro function `%sysevalf` is used to evaluate the expression `80 * 4 / 16` to get the 20. This function evaluates expressions that contain or produce floating point numbers and then returns the result. This particular expression could be evaluates with `%eval`, which does integer arithmetic, but that will not always be the case. In summary, this design is optimal for partial profiles, and it is 25% efficient relative to an optimal generic design where all attributes can vary.

The first three choice sets are as follows:

```
Set   x1   x2   x3   x4   x5   x6   x7   x8   x9   x10   x11   x12   x13   x14   x15   x16

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

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

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

We have already seen that the choice design is statistically optimal. However, it might perhaps not be optimal from a practical usage point of view. In every choice set, there is a pair of alternatives, one with three attributes present and one with only one attribute present. This might not meet your needs (or maybe it does). You always need to inspect your designs to see how well they work for your purposes. Statistical efficiency and optimality are just one part of the picture. With the OA $4^1 2^4$ in 8 runs, there is one other kind of OA that you can get by specifying other random number seeds, that will always produce a choice set (one in each block of choice sets) where all four attributes are present in one alternative and none are present in the other alternative. This might be worse. This kind of problem is less likely to be an issue when you vary more than four attributes.

You can make an optimal partial-profile choice design using an unbalanced block design (instead of a BIBD). In an unbalanced block design, the treatment frequencies are constant, but the pairwise frequencies are not constant. The following example makes a design with 6 three-level factors in 60 choice sets with three alternatives each:

```
%mktex(6 3 ** 6, n=18, seed=424)

proc sort data=randomized out=randes(drop=x1);
   by x2 x1;
   run;

proc print data=randes noobs; run;

%mktbsize(nattrs=20, setsize=6, options=ubd)

%mktbibd(b=10, nattrs=20, setsize=6, seed=104)

%mktppro(design=randes, ibd=bibd, print=f p)

%choiceff(data=chdes,               /* candidate set of choice sets         */
          init=chdes,               /* initial design                       */
          initvars=x1-x20,          /* factors in the initial design        */
          model=class(x1-x20 / sta),/* model with stdz orthogonal coding    */
          nsets=60,                 /* number of choice sets                */
          nalts=3,                  /* number of alternatives               */
          rscale=                   /* relative D-efficiency scale factor   */
          %sysevalf(60 * 6 / 20),   /* 6 of 20 attrs in 60 sets vary        */
          beta=zero)                /* assumed beta vector, Ho: b=0         */
```

The `%MktBSize` macro is used to suggest the size of a block design. Each attribute appears three times in the design and each pair will occur in the range 0 to 2 times (average 0.79). The design is optimal for this specification.

The *D*-efficiency is 18 which corresponds to the 6 / 20 of 60 choice sets. Taking this into account, relative *D*-efficiency is 100. This shows that the design is optimal for partial profiles, and is 30% efficient relative to a full-profile optimal generic design where all attributes can vary.

You could also specify `b=20` in `%MktBIBD` and `nsets=120` in ChoiceEff to make larger design. Any integer multiple of `b=10` will work.

%MktPPro Macro Options

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

Option	Description
`help`	(positional) "help" or "?" displays syntax summary
`ibd=`*SAS-data-set*	incomplete block design
`design=`*SAS-data-set*	orthogonal array
`out=`*SAS-data-set*	output partial-profile design
`print=`*print-options*	output display options
`x=`*SAS-data-set*	incomplete block design incidence matrix

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

```
%mktppro(help)
%mktppro(?)
```

You must specify either `ibd=` or `x=` but not both.

ibd= *SAS-data-set*

specifies the block design. Ideally, this design is a BIBD, but an unbalanced block design is fine. Also, any incomplete block design can be used.

design= *SAS-data-set*

specifies the orthogonal array from the `%MktEx` macro.

out= *SAS-data-set*

specifies the output choice design.

print= *print-options*
specifies both of the output display options. The default is `print=f`. Specify one or more values from the following list.

`i` incomplete block design
`f` crosstabulation of attribute frequencies
`p` partial-profile design

x= *SAS-data-set*
specifies the incidence matrix for the block design, which is a binary coding of the design.

%MktPPro 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.

%MktRoll Macro

The `%MktRoll` autocall macro constructs a choice design from a linear arrangement. See the following pages for examples of using the `%MktRoll` macro in the design chapter: 134 and 192. Also see the following pages for examples of using this macro in the discrete choice chapter: 312, 320, 357, 387, 429, 505, 546, 556, 575, 607, 617, 628 and 636. Additional examples appear throughout this chapter. The `%MktRoll` macro takes as input a SAS data set containing an experimental design with one row per choice set, the *linear arrangement*, for example, a design created by the `%MktEx` macro. This data set is specified in the `design=` option. This data set has one variable for each attribute of each alternative in the choice experiment. The output from this macro is an `out=` SAS data set is the *choice design* containing the experimental design with one row per alternative per choice set. There is one column for each different attribute. For example, in a simple branded study, `design=` could contain the variables `x1-x5` which contain the prices of each of five alternative brands. The output data set would have one factor, `Price`, that contains the price of each of the five alternatives. In addition, it would have the number (or optionally the name) of each alternative.[*]

The rules for determining the mapping between factors in the `design=` data set and the `out=` data set are contained in the `key=` data set. For example, assume that the `design=` data set contains the variables `x1-x5` which contain the prices of each of five alternative brands: Brand A, B, C, D, and E. The choice design has two factors, `Brand` and `Price`. Brand A price is made from `x1`, Brand B price is made from `x2`, ..., and Brand E price is made from `x5`. A convenient way to get all the names in a variable list like `x1-x5` is with the `%MktKey` macro. The following step creates the five names in a single column:

```
   %mktkey(5 1)
```

The `%MktKey` macro produces the following data set:

```
                                   x1

                                   x1
                                   x2
                                   x3
                                   x4
                                   x5
```

The following step creates the `Key` data set:

[*]See page 67 for an explanation of the linear arrangement of a choice design versus the arrangement of a choice design that is more suitable for analysis.

```
data key;
   input (Brand Price) ($);
   datalines;
A x1
B x2
C x3
D x4
E x5
;
```

This data set has two variables. `Brand` contains the brand names, and `Price` contains the names of the factors that are used to make the price effects for each of the alternatives. The `out=` data set will contain the variables with the same names as the variables in the `key=` data set.

The following step creates the linear arrangement with one row per choice set:

```
%mktex(3 ** 5, n=12)
```

The following step creates the choice design with one row per alternative per choice set:

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

Consider, for example, a randomized data set that contains the following row:

```
Obs    x1    x2    x3    x4    x5

  9     3     1     1     2     1
```

Then the data set SASUSER.DESIGN contains the following rows:

```
Obs    Set    Brand    Price

 41     9       A        3
 42     9       B        1
 43     9       C        1
 44     9       D        2
 45     9       E        1
```

The price for Brand A is made from `x1=3`, ..., and the price for Brand E is made from `x5=1`.

Now assume that there are three alternatives, each a different brand, and each composed of four factors: `Price`, `Size`, `Color`, and `Shape`. In addition, there is a constant alternative. First, the `%MktEx` macro is used to create a design with 12 factors, one for each attribute of each alternative. The following step creates the design:

```
%mktex(2 ** 12, n=16, seed=109)
```

The following step creates the `key=` data set:

```
data key;
   input (Brand Price Size Color Shape) ($); datalines;
          A     x1     x2   x3    x4
          B     x5     x6   x7    x8
          C     x9    x10  x11   x12
          None   .      .    .     .
;
```

It shows that there are three brands, A, B, and C, and also None.

Brand A is created from `Brand` = "A", `Price` = `x1`, `Size` = `x2`, `Color` = `x3`, `Shape` = `x4`.

Brand B is created from `Brand` = "B", `Price` = `x5`, `Size` = `x6`, `Color` = `x7`, `Shape` = `x8`.

Brand C is created from `Brand` = "C", `Price` = `x9`, `Size` = `x10`, `Color` = `x11`, `Shape` = `x12`.

The constant alternative is created from `Brand` = "None" and none of the attributes. The "`.`" notation is used to indicate missing values in input data sets. The actual values in the `Key` data set are blank (character missing).

The following step creates the design with one row per alternative per choice set:

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

Consider, for example, a randomized data set that contains the following row:

```
 Obs    x1    x2    x3    x4    x5    x6    x7    x8    x9   x10    x11    x12

   8     2     2     2     2     2     1     1     2     2     2      1      2
```

Then the data set SASUSER.DESIGN contains the following rows:

```
                  29     8    A          2        2       2        2
                  30     8    B          2        1       1        2
                  31     8    C          2        2       1        2
                  32     8    None       .        .       .        .
```

Now assume like before that there are three branded alternatives, each composed of four factors: `Price`, `Size`, `Color`, and `Shape`. In addition, there is a constant alternative. Also, there is an alternative-specific factor, `Pattern`, that only applies to Brand A and Brand C. First, the `%MktEx` macro is used to create a design with 14 factors, one for each attribute of each alternative. The following step creates the design:

```
%mktex(2 ** 14, n=16, seed=114)
```

The following step creates the key= data set:

```
data key;
   input (Brand Price Size Color Shape Pattern) ($);
   datalines;
A     x1     x2   x3    x4   x13
B     x5     x6   x7    x8     .
C     x9    x10  x11   x12   x14
None   .      .    .     .     .
;
```

It shows that there are three brands, A, B, and C, plus None.

Brand A is created from Brand = "A", Price = x1, Size = x2, Color = x3, Shape = x4, Pattern = x13.

Brand B is created from Brand = "B", Price = x5, Size = x6, Color = x7, Shape = x8.

Brand C is created from Brand = "C", Price = x9, Size = x10, Color = x11, Shape = x12, Pattern = x14.

The constant alternative is Brand = "None" and none of the attributes.

The following step creates the design with one row per alternative per choice set:

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

Consider, for example, a randomized data set that contains the following row:

```
 Obs   x1   x2   x3   x4   x5   x6   x7   x8   x9   x10   x11   x12   x13   x14

   8    2    1    1    2    1    2    1    2    1    1     1     2     1     2
```

Then the data set SASUSER.DESIGN contains the following rows:

```
 Obs    Set    Brand    Price    Size    Color    Shape    Pattern

  29      8    A          2        1       1        2         1
  30      8    B          1        2       1        2         .
  31      8    C          1        1       1        2         2
  32      8    None       .        .       .        .         .
```

Now assume we are going to fit a model with price cross-effects so we need x1, x5, and x9 (the three price effects) available in the out= data set. See pages 444 and 468 for other examples of cross-effects. The following step creates the design:

```
%mktroll(design=randomized, key=key, out=sasuser.design, alt=brand,
         keep=x1 x5 x9)
```

Now the data set also contains the three original price variables, for example, as follows:

```
   Obs    Set    Brand    Price    Size    Color    Shape    Pattern    x1    x5    x9

    29     8     A          2        1       1        2         1        2     1     1
    30     8     B          1        2       1        2         .        2     1     1
    31     8     C          1        1       1        2         2        2     1     1
    32     8     None       .        .       .        .         .        2     1     1
```

Every value in the `key=` data set must appear as a variable in the `design=` data set. The macro displays a warning if it encounters a variable name in the `design=` data set that does not appear as a value in the `key=` data set.

%MktRoll Macro Options

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

Option	Description
`help`	(positional) "help" or "?" displays syntax summary
`alt=`*variable*	variable with name of each alternative
`design=`*SAS-data-set*	input SAS data set
`keep=`*variable-list*	factors to keep
`key=`*SAS-data-set*	`Key` data set name
`key=`*rows columns* <`t`>	`Key` data set description
`options=nowarn`	suppress the variables not mentioned warning
`out=`*SAS-data-set*	output SAS data set
`set=`*variable*	choice set number variable

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

```
%mktroll(help)
%mktroll(?)
```

You must specify the `design=` and `key=` options.

alt= *variable*

specifies the variable in the `key=` data set that contains the name of each alternative. Often this is something like `alt=Brand`. When `alt=` is not specified, the macro creates a variable `_Alt_` that contains the alternative number.

design= *SAS-data-set*

specifies an input SAS data set with one row per choice set. The `design=` option must be specified.

keep= *variable-list*

specifies factors from the `design=` data set that should also be kept in the `out=` data set. This option is useful to keep terms that are used to create cross-effects.

key= *SAS-data-set* | *l*ist

specifies the rules for mapping the `design=` data set to the `out=` data set. The `key=` option must be specified. It has one of two forms. 1) The `key=` option names an input SAS data set containing the rules for mapping the `design=` data set to the `out=` data set. The structure of this data set is described in detail in the preceding examples. 2) When you want the `key=` data set to exactly match the data set that comes out of the MktKey macro, you can specify the argument to the MktKey macro directly in the `key=` option, and the `%MktRoll` macro will make the `key=key` data set for you. In other words, the following two specifications are equivalent:

```
%mktkey(3 3 t)
%mktroll(design=design, key=key,   out=frac)

%mktroll(design=design, key=3 3 t, out=frac)
```

options= *options-list*

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

`nowarn`
does not display a warning when the `design=` data set contains variables not mentioned in the `key=` data set. Sometimes this is perfectly fine.

out= *SAS-data-set*

specifies the output SAS data set. If `out=` is not specified, the DATAn convention is used.

set= *variable*

specifies the variable in the `out=` data set that will contain the choice set number. By default, this variable is named `Set`.

%MktRoll 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.

%MktRuns Macro

The `%MktRuns` autocall macro suggests reasonable sizes for experimental designs. See the following pages for examples of using this macro in the design chapter: 128, 188 and 199. Also see the following pages for examples of using this macro in the discrete choice chapter: 302, 340, 411, 415, 482, 483, 535 and 557. Additional examples appear throughout this chapter. The `%MktRuns` macro tries to find sizes in which perfect balance and orthogonality can occur, or at least sizes in which violations of orthogonality and balance are minimized. Typically, the macro takes one argument, a list of the number of levels of each factor.

For example, with 3 two-level and 4 three-level factors, you can specify either of the following:

```
%mktruns(2 2 2 3 3 3 3)

%mktruns(2 ** 3 3 ** 4)
```

The output from the macro is as follows:

```
                              Design Summary

                          Number of
                          Levels       Frequency

                              2            3
                              3            4

                   Saturated      = 12
                   Full Factorial = 648

                   Some Reasonable                      Cannot Be
                      Design Sizes       Violations     Divided By

                            36 *                  0
                            72 *                  0
                            18                    3     4
                            54                    3     4
                            12 S                  6     9
                            24                    6     9
                            48                    6     9
                            60                    6     9
                            30                    9     4 9
                            42                    9     4 9

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

```
 n    Design                                              Reference

36    2 ** 16  3 **  4                                  Orthogonal Array
36    2 ** 11  3 ** 12                                  Orthogonal Array
36    2 ** 10  3 **  8   6 **  1                        Orthogonal Array
36    2 **  9  3 **  4   6 **  2                        Orthogonal Array
36    2 **  4  3 ** 13                                  Orthogonal Array
36    2 **  3  3 **  9   6 **  1                        Orthogonal Array
72    2 ** 52  3 **  4                                  Orthogonal Array
72    2 ** 49  3 **  4   4 **  1                        Orthogonal Array
72    2 ** 47  3 ** 12                                  Orthogonal Array
72    2 ** 46  3 **  8   6 **  1                        Orthogonal Array
72    2 ** 45  3 **  4   6 **  2                        Orthogonal Array
 .
 .
 .
```

The macro reports that the saturated design has 12 runs and that 36 and 72 are optimal design sizes. The macro picks 36, because it is the smallest integer >= 12 that can be divided by 2, 3, 2×2, 2×3, and 3×3. The macro also reports 18 as a reasonable size. There are three violations with 18, because 18 cannot be divided by each of the three pairs of 2×2, so perfect orthogonality in the two-level factors will not be possible with 18 runs. Larger sizes are reported as well. The macro displays orthogonal designs that are available from the `%MktEx` macro that match your specification.

You can run PROC PRINT after the macro finishes to see every size the macro considered, for example, as follows:

```
proc print label data=nums split='-';
   label s = '00'x;
   id n;
   run;
```

The output from this step is not shown.

For 2 two-level factors, 2 three-level factors, 2 four-level factors, and 2 five-level factors specify the following:

```
%mktruns(2 2 3 3 4 4 5 5)
```

The results are as follows:

```
                                 Design Summary

                             Number of
                             Levels       Frequency

                                  2            2
                                  3            2
                                  4            2
                                  5            2

       Saturated      = 21
       Full Factorial = 14,400

       Some Reasonable                            Cannot Be
          Design Sizes       Violations     Divided By

                   120                3     9 16 25
                   180                6     8 16 25
                    60                7     8  9 16 25
                   144               15     5 10 15 20 25
                    48               16     5  9 10 15 20 25
                    72               16     5 10 15 16 20 25
                    80               16     3  6  9 12 15 25
                    96               16     5  9 10 15 20 25
                   160               16     3  6  9 12 15 25
                   192               16     5  9 10 15 20 25
                    21 S             34     2  4  5  6  8  9 10 12 15 16 20 25

              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.
```

Among the smaller design sizes, 60 or 48 look like good possibilities.

The macro has an optional keyword parameter: `max=`. It specifies the maximum number of sizes to try. Usually you will not need to specify the `max=` option. The smallest design that is considered is the saturated design. The following specification tries 5000 sizes (21 to 5020) and reports that a perfect design can be found with 3600 runs:

```
   %mktruns(2 2 3 3 4 4 5 5, max=5000)
```

The results are as follows:

```
                                    Design Summary

                                Number of
                                Levels       Frequency

                                     2            2
                                     3            2
                                     4            2
                                     5            2

                      Saturated      = 21
                      Full Factorial = 14,400

            Some Reasonable                        Cannot Be
               Design Sizes       Violations     Divided By

                       3600                0
                        720                1      25
                       1200                1       9
                       1440                1      25
                       1800                1      16
                       2160                1      25
                       2400                1       9
                       2880                1      25
                       4320                1      25
                       4800                1       9
                         21 S             34       2   4   5   6   8   9  10  12  15  16  20  25

                  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 `%MktEx` macro does not explicitly know how to make this design, however, it can usually find it or come extremely close with the coordinate exchange algorithm.

Now consider again the problem with 3 two-level and 4 three-level factors, but this time we want to estimate the interaction of two of the two-level factors. The following step runs the macro:

```
%mktruns(2 2 2 3 3 3 3, interact=1*2, options=source)
```

Since `options=source` was specified, the first part of the following output lists the sources for orthogonality violations that the macro will consider. We see that n must be divided by: 2 since `x1-x3` are two-level factors, 3 since `x4-x7` are three-level factors, 4 since `x1*x2`, `x1*x3`, and `x2*x3` interactions are specified, 6 since we have two-level and three-level factors, 8 since we have the two-way interaction of 2 two-level factors and the main-effect of an additional two-level factor, 9 since we have multiple three-level factors, and 12 since we have the two-way interaction of 2 two-level factors and the main-effect of additional three-level factors. The results are as follows:

```
                                        Failing to
                                        Divide by
                                        the Divisor
                                        Contributes
                                        this to the
                                        Violations
To Achieve Both    The Source of        Count
Orthogonality      the Divisor                           The Divisor
and Balance,                            The Number of    Comes
N Must Be          Number of            Sources of       from these
Divided By         Levels or Cells      the Divisor      Factors

        2                2                     3            1
                                                            2
                                                            3

        3                3                     4            4
                                                            5
                                                            6
                                                            7

        4                2*2                   3            1 2
                                                            1 3
                                                            2 3

        6                2*3                  12            1 4
                                                            1 5
                                                            1 6
                                                            1 7
                                                            2 4
                                                            2 5
                                                            2 6
                                                            2 7
                                                            3 4
                                                            3 5
                                                            3 6
                                                            3 7

        8                2*2*2                 1            1 2 3

        9                3*3                   6            4 5
                                                            4 6
                                                            4 7
                                                            5 6
                                                            5 7
                                                            6 7
```

```
         12             2*2*3                 4             1 2 4
                                                            1 2 5
                                                            1 2 6
                                                            1 2 7


                            Design Summary

                        Number of
                        Levels       Frequency

                             2           3
                             3           4

          Saturated      = 13
          Full Factorial = 648

          Some Reasonable                      Cannot Be
             Design Sizes       Violations     Divided By

                       72                0
                      144                0
                       36                1     8
                      108                1     8
                       48                6     9
                       96                6     9
                      120                6     9
                       60                7     8  9
                       84                7     8  9
                       13 S             33     2  3  4  6  8  9 12

         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.
```

Now we need 72 runs for perfect balance and orthogonality although the `%MktEx` design catalog is not guaranteed to contain designs with interactions.

%MktRuns Macro Options

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

Option	Description
`list`	(positional) numbers of levels of all the factors
	(positional) “help” or “?” displays syntax summary
`interact=`*interaction-list*	interaction terms
`max=`*n* $< m >$ $<$ f $>$	largest design sizes to try

Option	Description
`n=`*n*	design size to evaluate
`maxlev=`*n*	maximum number of levels
`maxoa=`*n*	maximum number of orthogonal arrays
`options=justparse`	used by other `Mkt` macros to parse the list
`options=multiple`	allow terms to be counted multiple times
`options=multiple2`	`option=multiple` and more detailed output
`options=noprint`	suppress the display of all output
`options=nosat`	suppress the saturated design from the design list
`options=source`	displays source of numbers in design sizes
`options=512`	adds some designs in 512 runs
`out=`*SAS-data-set*	data set with the suggested sizes
`outorth=`*SAS-data-set*	data set with orthogonal array list
`toobig=`*n*	specifies problem that is too big

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

```
%mktruns(help)
%mktruns(?)
```

The `%MktRuns` macro has one positional parameter, `list`, and several keyword parameters.

list

specifies a list of the numbers of levels of all the factors. For example, for 3 two-level factors specify either `2 2 2` or `2 ** 3`. Lists of numbers, like `2 2 3 3 4 4` or a *levels**number of factors* syntax like: `2**2 3**2 4**2` can be used, or both can be combined: `2 2 3**4 5 6`. The specification `3**4` means 4 three-level factors. You must specify a list. Note that the factor list is a positional parameter. This means it must come first, and unlike all other parameters, it is not specified after a name and an equal sign.

interact= *interaction-list*

specifies interactions that must be estimable. By default, no interactions are guaranteed to be estimable. Examples:

```
interact=x1*x2
interact=x1*x2 x3*x4*x5
interact=x1|x2|x3|x4|x5@2
```

The interaction syntax is in most ways like PROC GLM's and many of the other modeling procedures. It uses "`*`" for simple interactions (`x1*x2` is the interaction between `x1` and `x2`), "`|`" for main effects and interactions (`x1|x2|x3` is the same as `x1 x2 x1*x2 x3 x1*x3 x2*x3 x1*x2*x3`) and "`@`" to eliminate higher-order interactions (`x1|x2|x3@2` eliminates `x1*x2*x3` and is the same as `x1 x2 x1*x2 x3 x1*x3 x2*x3`). The specification "`@2`" creates main effects and two-way interactions. Unlike PROC GLM's syntax, some short cuts are permitted. For the factor names, you can specify either the actual variable names (for example, `x1*x2` ...) or you can just specify the factor number without the "`x`" (for example, `1*2`). You can also specify `interact=@2` for all main effects and two-way interactions omitting the `1|2|...`. The following three specifications are equivalent:

```
%mktruns(2 ** 5, interact=@2)
%mktruns(2 ** 5, interact=1|2|3|4|5@2)
%mktruns(2 ** 5, interact=x1|x2|x3|x4|x5@2)
```

max= *n* $< m >$ $<$ f $>$

specifies the maximum number of design sizes to try. By default, `max=200 2 f`. The macro tries up to *n* sizes starting with the saturated design. The macro stops trying larger sizes when it finds a design size with zero violations that is *m* times as big as a previously found size with zero violations. The macro reports the best 10 sizes. For example, if the saturated design has 10 runs, and there are zero violations in 16 runs, then by default, the largest size that the macro will consider is $32 = 2 \times 16$ runs. A third optional value of 'f' or 'F' is specified by default, which instructs the `%MktRuns` macro to not consider designs larger than full-factorials. If you specify `max=` without this option, larger designs might be considered.

maxlev= *n*

specifies the maximum number of levels to consider when generating the orthogonal array list. The default is `maxlev=50`, and the actual maximum is the max of the specified `maxlev=` value and the maximum number of levels in the factor list. Specify a value $2 \le n \le 144$.

n= *n*

specifies the design size to evaluate. By default, this option is not specified, and the `max=` option specification provides a range of design sizes to evaluate.

maxoa= *n*

specifies the maximum number of orthogonal arrays to display in the list of orthogonal arrays that the `%MktEx` macro knows how to make. By default, when no value is specified, the entire list is displayed. You can specify, for example, `maxoa=0`, `outorth=oa`, to get a data set instead of displaying this table. You could instead specify `maxoa=10` to just see the first ten arrays.

options= *options-list*

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

`justparse`
is used by other `Mkt` macros to have this macro just parse the list argument and return it as a simple list of integers.

`multiple`
specifies that a term that is required for orthogonality can be counted multiple times when counting orthogonality violations. For example, combinations of levels for the pair of variable 1 and variable 2 must have equal frequencies for orthogonality in the main effects, and if two-way interactions are required as well, then the (1, 2) pair must have equal frequencies again. By default, each combination of variables is only counted once. The difference between the single and multiple sources is the single source method just counts the places in the design where equal frequencies must occur. The multiple source method weights this count by the number of times each of these terms is important for achieving orthogonality. The results for the two methods are often highly correlated, but they can be different.

`multiple2`
specifies both `option=multiple` and more detailed output including the reason each term appears is added to the source table when `options=source multiple2` is specified.

`noprint`
suppresses the display of all output. The only output from the macro is the output data sets.

`nosat`
suppresses inclusion of the saturated design in the design list. By default, the saturated design is reported along with the reasonable design sizes, even if it is not a very reasonable size, in order to provide some context for the other numbers. If you specify `options=nosat`, then the saturated design will only be added to the list if it meets the usual criteria for being a reasonable size.

`source`
displays the source of all of the numbers in the table of reasonable design sizes.

`512`
adds some larger designs in 512 runs with mixes of 16, 8, 4, and 2-level factors to the catalog, which gives added flexibility in 512 runs at a cost of much slower run time. This option replaces the default $4^{160}32^{1}$ parent with $16^{32}32^{1}$.

out= *SAS-data-set*

specifies the name of a SAS data set with the suggested sizes. The default is `out=nums`.

outorth= *SAS-data-set*

specifies output data set with orthogonal array list. By default, this data set is not created.

toobig= *n*

is used to flag problems that are too big and take too long before they consume large quantities of resources. The default is `toobig=11400`. The number is the maximum number of levels and pairs of levels that is considered. With models with interactions, the interaction terms contribute to this number as well. A main-effects model with 150 factors is small enough ($150 \times 151/2 = 11325$) to work with the default `toobig=11400`. If you want to use the `%MktRuns` macro with larger models, you will have to specify a larger value for `toobig=`. The following step works with `toobig=` specified, but it is too large to work without it being specified:

```
%mktruns(2 ** 17, interact=@2, toobig=20000)
```

%MktRuns 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.

%Paint Macro

The `%Paint` autocall macro interpolates between colors. This macro is not a part of the experimental design or marketing research family of macros. It exists for totally different reasons. This macro has two main uses.

- You can use it with ODS. The macro can create a macro that specifies an `ods cellstyle as` statement for use in PROC TEMPLATE for coloring the background around entries in a table.
- You can use it with PROC INSIGHT. The macro can read an input `data=` data set and create an `out=` data set with a new variable `_obstat_` that contains observation symbols and colors interpolated from the `colors=` list based on the values `var=` variable. Three interpolation methods are available with the `level=` option.

This macro creates temporary work data sets named `tempdata`, `tempdat2`, `tempdat3`, and `tempdat4`.

It makes a temporary macro called "_".

When a `cellstyle` macro is not created, the paint macro by default creates an `out=` data set named `colors`.

%Paint Macro Options

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

Option	Description
`help`	(positional) "help" or "?" displays syntax summary
`values=`*do-list*	input data values
`var=`*variable-name*	input variable
`data=`*SAS-data-set*	input SAS data set
`out=`*SAS-data-set*	output SAS data set
`macro=`*macro-name*	SAS macro name
`colors=`<*color-list*> <*data-value-list*>	colors list
`level=`*measurement-level*	measurement level of the data
`symbols=`*symbols-list*	list of symbols
`format=`*SAS-format*	nominal and ordinal variable format
`order=`*summary-order*	nominal and ordinal variable order
`select=`*n*	selection state
`show=`*n*	show/hide state
`include=`*n*	include/exclude state
`label=`*n*	label/unlabel state
`rgbround=`*rounding-list*	RGB value rounding instructions
`missing=`*missing-specification*	missing value handling
`debug=`*debug-string*	debugging information to display

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

```
%paint(help)
%paint(?)
```

One of the following two options must be specified:

values= *do-list*

specifies the input data values. This option primarily exists for use with ODS. You can tell the macro to create input data set based on a `do` list provided with this option. Example: `values=0 to 10 by 0.25`.

var= *variable-name*

specifies a variable with the input data values. This option primarily exists for use with PROC INSIGHT. Use this option when there is an input `data=` data set.

All of these next options can optionally be specified:

data= *SAS-data-set*

specifies the input SAS data set (when `var=` but not `values=` is specified). This option primarily exists for use with PROC INSIGHT.

out= *SAS-data-set*

specifies the output SAS data set. The default, when `macro=` is not specified, is `out=colors`. This option primarily exists for use with PROC INSIGHT.

macro= *macro-name*

specifies the name of the SAS macro to create instead of creating an `out=` data set. This option primarily exists for use with ODS. By default, no macro is created.

colors= *<color-list> <data-value-list>*

specifies the colors list. The colors must be selected from: red, green, blue, yellow, magenta, cyan, black, white, orange, brown, gray, olive, pink, purple, violet. For other colors, specify the RGB color name (CX*rrggbb* where *rr* is the red value, *gg* is the green, and *bb* is blue, all three specified in hex, 00 to FF). When no list is specified, the default is `colors=blue magenta red`. The option `colors=red green 1 10`, interpolates between red and green, based on the values of the `var=` variable, where values of 1 or less map to red, values of 10 or more map to green, and values in between map to colors in between. The option `colors=red yellow green 1 5 10`, interpolates between red at 1, yellow at 5, and green at 10. If the data value list is omitted it is computed from the data.

level= *measurement-level*

specifies the measurement level of the data. Values include `interval`, `ordinal`, and `nominal`. The option `level=interval` (the default) interpolates on the actual values. The option `level=ordinal` interpolates on the ranks of the input values. This can even work with character variables; the ranks are just the category numbers. The option `level=nominal` specifies that there is one color or symbol per category. Only the first three characters of the measurement level are checked.

symbols= *symbols-list*

provides a list of symbols from page 323 of the Version 6 PROC INSIGHT manual. Specify integers from 1 to 8 or select from: square plus circle diamond x up down star. Note that 'triangle' is not specified with 'up' and 'down'. Equivalent examples: `symbols=1 2 3 4 5 6`, `symbols=square plus circle diamond x up`, `symbols=1 plus circle 4 5 up`. Note that when `level=interval`, only the first symbol is used. With `level=nominal` and `level=ordinal`, the first symbol is used for the first category, the second symbol is used for the second category, ..., and if there are more categories than symbols, the last symbol is reused for all subsequent categories. Extra symbols are ignored. The default first symbol is circle, and the last symbol is substituted for invalid symbols.

format= *SAS-format*

specifies the format for the values used with `level=nominal` and `level=ordinal` variables.

order= *summary-order-option*

specifies the PROC SUMMARY `order=` option for `level=ordinal` and `level=nominal`.

These next four options specify the first four columns of `_obstat_` variable from page 323 of the Version 6 PROC INSIGHT manual.

Column 5, the symbol, is specified with `symbols=`. The values must be numeric expressions including constants, variables, and arithmetic expressions. All nonzero expression results are converted to 1.

select= *n*

specifies the selection state. The value 0 means not selected. This option primarily exists for use with PROC INSIGHT.

show= *n*

specifies the show/hide state. The value 0 means hide. This option primarily exists for use with PROC INSIGHT.

include= *n*

specifies the include/exclude state. The value 0 means exclude. This option primarily exists for use with PROC INSIGHT.

label= *n*

specifies the label/unlabel state. The value 0 means unlabel. This option primarily exists for use with PROC INSIGHT.

rgbround= *rounding-list*

specifies instructions for rounding the RGB values. The first value is used to round the `colors=var` variable. Specify a positive value to have the variable rounded to multiples of that value. Specify a negative value n to have a maximum of abs(n) colors. For the other three values, specify positive values. The last three values are rounding factors for the red, green, and blue component of the color. By default, when a value is missing, there is no rounding. The default is `rgbround=-99 1 1 1`, which creates a maximum of 99 colors and rounds the RGB values to integers. You could specify larger values for the last three values to have, for examples, the color values be multiples of two or four.

missing=

specifies how missing values in the `var=` variable are handled. By default, when `missing=` is null, the observation is not shown (`show=0` is set). The specified value is a color followed by a symbol. When only one value is specified in the `symbols=` list, the symbol is optional, and the `symbols=` symbol is used.

debug= *debug-string*

specifies types of debugging information to display. The option `debug=vars` displays macro options and macro variables for debugging. The option `debug=dprint` displays intermediate data sets. The option `debug=notes` suppresses the specification of `options nonotes` during most of the macro. The option `debug=time` displays the total macro run time. The option `debug=mprint` runs the macro with `options mprint`. Create a list for more than one type of debugging. Example: `debug=vars dprint notes time mprint`.

%PHChoice Macro

The `%PHChoice` autocall macro customizes the output from PROC PHREG for choice modeling. Typically, you run the following macro once to customize the PROC PHREG output as follows:

```
%phchoice(on)
```

The macro uses PROC TEMPLATE and ODS (Output Delivery System) to customize the output from PROC PHREG. Running this code edits the templates and stores copies in SASUSER. These changes will remain in effect until you delete them. Note that these changes assume that each effect in the choice model has a variable label associated with it so there is no need to display variable names. If you are coding with PROC TRANSREG, this will usually be the case. You can submit the following step to return to the default output from PROC PHREG:

```
%phchoice(off)
```

If you ever have errors running this macro, like invalid page errors, see “Macro Errors” on page 1211. The rest of this section discusses the details of what the `%PHChoice` macro does and why. Unless you are interested in further customization of the output, you should skip to “`%PHChoice` Macro Options” on page 1177.

We are most interested in the `Analysis of Maximum Likelihood Estimates` table, which contains the parameter estimates. We can first use PROC TEMPLATE to identify the template for the parameter estimates table and then edit the template. First, let’s have PROC TEMPLATE display the templates for PROC PHREG. The `source stat.phreg` statement in the following step specifies that we want to see PROC TEMPLATE source code for the STAT product and the PHREG procedure:

```
proc template;
   source stat.phreg;
   run;
```

If we search the results for the `Analysis of Maximum Likelihood Estimates` table we find the following code, which defines the `Stat.Phreg.ParameterEstimates` table:

```
define table Stat.Phreg.ParameterEstimates;
   notes "Parameter Estimates Table";
   dynamic Confidence NRows;
   column Variable DF Estimate StdErr StdErrRatio ChiSq ProbChiSq HazardRatio
      HRLowerCL HRUpperCL Label;
   header h1 h2;

   define h1;
      text "Analysis of Maximum Likelihood Estimates";
      space = 1;
      spill_margin;
   end;
```

```
define h2;
   text Confidence BEST8. %nrstr("%% Hazard Ratio Confidence Limits");
   space = 0;
   end = HRUpperCL;
   start = HRLowerCL;
   spill_margin = OFF;
end;

define Variable;
   header = "Variable";
   style = RowHeader;
   id;
end;

define DF;
   parent = Common.ParameterEstimates.DF;
end;

define Estimate;
   header = ";Parameter;Estimate;";
   format = D10.;
   parent = Common.ParameterEstimates.Estimate;
end;

define StdErr;
   header = ";Standard;Error;";
   format = D10.;
   parent = Common.ParameterEstimates.StdErr;
end;

define StdErrRatio;
   header = ";StdErr;Ratio;";
   format = 6.3;
end;

define ChiSq;
   parent = Stat.Phreg.ChiSq;
end;

define ProbChiSq;
   parent = Stat.Phreg.ProbChiSq;
end;

define HazardRatio;
   header = ";Hazard;Ratio;";
   glue = 2;
   format = 8.3;
end;

define HRLowerCL;
   glue = 2;
   format = 8.3;
   print_headers = OFF;
end;
```

```
      define HRUpperCL;
         format = 8.3;
         print_headers = OFF;
      end;

      define Label;
         header = "Variable Label";
      end;

      col_space_max = 4;
      col_space_min = 1;
      required_space = NRows;
   end;
```

It contains header, format, spacing and other information for each column in the table. Most of this need not concern us now.[*] The template contains the following `column` statement, which lists the columns of the table:

```
      column Variable DF Estimate StdErr StdErrRatio ChiSq ProbChiSq HazardRatio
         HRLowerCL HRUpperCL Label;
```

Since we will usually have a label that adequately names each parameter, we do not need the variable column. We also do not need the hazard information. If we move the label to the front of the list and drop the variable column and the hazard columns, we get the following:

```
   column Label DF Estimate StdErr ChiSq ProbChiSq;
```

We use the `edit` statement to edit the template. We can also modify some headers. We specify the new `column` statement and the new headers. We can also modify the Summary table, which is `Stat.Phreg.CensoredSummary`, to use the vocabulary of choice models instead of survival analysis models. The code is generated by the PROC TEMPLATE step with the `source` statement. The overall header "Summary of the Number of Event and Censored Values" is changed to "Summary of Subjects, Sets, and Chosen and Unchosen Alternatives", "Total" is changed to "Number of Alternatives", "Event" is changed to "Chosen Alternatives", "Censored" is changed to "Not Chosen", and "Percent Censored" is dropped. Finally `Style=RowHeader` was specified on the label column. This sets the color, font, and general style for HTML output. The `RowHeader` style is typically used on first columns that provide names or labels for the rows. The following step contains the code that the `%phchoice(on)` macro runs:

```
   proc template;
      edit stat.phreg.ParameterEstimates;
         column Label DF Estimate StdErr ChiSq ProbChiSq;
         header h1;

         define h1;
            text "Multinomial Logit Parameter Estimates";
            space = 1;
            spill_margin;
            end;
```

[*]In fact, this is an older version of the template. It has changed. Still, this is the version that the macro was based on, so it is still shown here.

```
         define Label;
            header = " " style = RowHeader;
            end;
         end;

         edit Stat.Phreg.CensoredSummary;
            column Stratum Pattern Freq GenericStrVar Total
                   Event Censored;
            header h1;
            define h1;
               text "Summary of Subjects, Sets, "
                    "and Chosen and Unchosen Alternatives";
               space = 1;
               spill_margin;
               first_panel;
            end;

            define Freq;
              header=";Number of;Choices" format=6.0;
            end;

         define Total;
            header = ";Number of;Alternatives";
            format_ndec = ndec;
            format_width = 8;
         end;

         define Event;
            header = ";Chosen;Alternatives";
            format_ndec = ndec;
            format_width = 8;
         end;

         define Censored;
            header = "Not Chosen";
            format_ndec = ndec;
            format_width = 8;
         end;
         end;

      run;
```

The `%phchoice(off)` macro runs the following step:

```
   * Delete edited templates, restore original templates;
   proc template;
      delete Stat.Phreg.ParameterEstimates;
      delete Stat.Phreg.CensoredSummary;
      run;
```

Our editing of the multinomial logit parameter estimates table assumes that each independent variable has a label. If you are coding with PROC TRANSREG, this is true of all variables created by `class` expansions. You might have to provide labels for `identity` and other variables. Alternatively, if you want variable names to appear in the table, you can do that as follows:

```
%phchoice(on, Variable DF Estimate StdErr ChiSq ProbChiSq Label)
```

This might be useful when you are not coding with PROC TRANSREG. The optional second argument provides a list of the column names to display. The available columns are: `Variable DF Estimate StdErr StdErrRatio ChiSq ProbChiSq HazardRatio HRLowerCL HRUpperCL Label`. (`HRLowerCL` and `HRUpperCL` are confidence limits on the hazard ratio.) For very detailed customizations, you might have to run PROC TEMPLATE directly.

%PHChoice Macro Options

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

Option	Description
`onoff`	(positional) `on`, `off`, or `expb`
	(positional) "help" or "?" displays syntax summary
`column`	(positional) list of columns

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

```
%phchoice(help)
%phchoice(?)
```

The `%PHChoice` macro has two positional parameters, `onoff` and `column`. Positional parameters must come first, and unlike all other parameters, are not specified after a name and an equal sign.

onoff

`ON` specifies choice model customization.
`OFF` turns off the choice model customization and returns to the default PROC PHREG templates.
`EXPB` turns on choice model customization and adds the hazard ratio to the output.
Upper/lower case does not matter.

column

specifies an optional column list for more extensive customizations.

%PlotIt Macro

The `%PlotIt` autocall macro makes graphical scatter plots of labeled points. It is particularly designed to display raw data and results from analyses such as regression, correspondence analysis, MDPREF, PREFMAP, and MDS. However, it can make many other types of graphical displays as well. It can plot points, labeled points, vectors, circles and density. See pages 27–40 and 1231–1274, for example, plots and more about these methods. The `%PlotIt` macro is not needed nearly as much now as it was in previous SAS releases. Now, ODS Graphics automatically does much of what the `%PlotIt` macro was originally designed for, and usually, ODS Graphics does it better and more conveniently. Comparisons between the `%PlotIt` macro and ODS Graphics are as follows. The `%PlotIt` macro has a much more sophisticated algorithm for placing labels and avoiding label collisions. The `%PlotIt` macro has a few other capabilities that are not in ODS Graphics (e.g. more sophisticated color ramps or "painting" for some applications). ODS Graphics is superior to the `%PlotIt` macro in virtually every other way, and ODS Graphics has *many* capabilities that the `%PlotIt` macro does not have. The algorithm that ODS Graphics has for label placement in 9.2, while clearly nonoptimal, is good enough for many analyses.

The macro, by default, uses the last data set created. The macro creates an output Annotate data set that cannot be used as input to the macro, so you must specify `data=` if you run `%PlotIt` a second time.

By default (at least by default when a device like WIN is in effect on a PC), the `%PlotIt` macro creates a graphical scatter plot on your screen. If no graphics device has been previously specified (either directly or indirectly), you will be prompted for a device as follows:

```
      No device name has been given--please enter device name:
```

Enter your graphics device. This name is remembered for the duration of your SAS session or until you change the device. You can modify the `gopprint=` and `gopplot=` options to set default devices so that you will not be prompted. Note that all graphics options specified in a `goptions` statement (except `device=`) are ignored by default. Use the macro options `gopprint=`, `gopplot=`, `gopts2=`, and `gopts=` to set `goptions`.

To display a plot on your screen using the default `goptions` in a PC environment, specify, for example, the following:

```
   goptions device=win;
   %plotit(data=sashelp.class(drop=age sex))
```

To create a PDF file named `myplot.pdf`, suitable for printing, specify the following:

```
   ods listing style=statistical;
   %plotit(data=sashelp.class(drop=age sex),
           gopts=device=pdf, method=print, post=myplot.pdf)
```

The ODS LISTING statement is only necessary in this example if you want to set the ODS style.

To create HTML output, specify the following:

```
   goptions device=png;
   ods listing close;
   ods html body='b.html';
   %plotit(data=sashelp.class(drop=age sex))
   ods html close;
```

To create RTF output, specify the following:

```
goptions device=png;
ods listing close;
ods rtf file='myplot.rtf';
%plotit(data=sashelp.class(drop=age sex))
ods rtf close;
```

To create a PNG file, perhaps just to insert into a document, specify the following:

```
goptions device=png;
ods listing style=analysis;
%plotit(data=sashelp.class(drop=age sex), method=print, post=myplot.png)
ods listing close;
```

Use `gout=` to write the plot to a catalog.

The default color scheme and a few other options have changed with this release in ways consistent with SAS/GRAPH procedures. Like SAS/GRAPH procedures in 9.2, the appearance of the output is in part controlled by the `gstyle` system option. The following statements enable and disable this option:

```
options gstyle;
options nogstyle;
```

With `gstyle` in effect (which will typically be the default), the output will look much different (and probably better) than it looked in previous releases. In part, this is due to the defaults for several options changing. For example, with `gstyle`, the macro will use hardware fonts by default with the style instead of Swiss software fonts. Mostly, however, the differences are due to the output being at least in part controlled by the ODS style now. Specify the `nogstyle` system option to get the old appearance. Note that by default, the listing ODS destination is open with `style=listing`. That is the style that is used if you do not open any other destination or specify any other style. You can open other destinations, e.g. HTML (by default `style=default`), printer (by default `style=printer`), RTF (by default `style=rtf`), and so on by specifying them in an ODS destination statement. Other styles of particular interest include `statistical`, `analysis`, and `journal`. There are many other styles as well. Each destination has a default style, and each permits a style specification. The following statement enables the HTML destination with the `statistical` style:

```
ods html body='b.html' style=statistical;
```

If listing (with `style=listing`) and one other destination are open, the macro will run using the style from the other destination. If other destinations are open as well, the macro will only run if there is one style other than `listing`. Hence, submitting the following statements before running the macro will cause the macro to issue an error and quit:

```
ods html;
ods rtf;
ods printer;
```

The `default`, `rtf` and `printer` styles are in effect, and the macro has no basis to choose between them. However, the macro will run fine with the following destinations since there is only one style other than the `listing` style in effect, and that is `style=default`:

```
ods html body='b.html';
ods rtf style=default;
ods printer style=default;
```

When you specify a destination other than the listing destination, you can usually make everything run faster by closing the listing destination, if you are not interested in results from that destination, for example, as follows:

```
ods listing close;
```

Note that a macro such as the `%PlotIt` macro only has limited and varying access to style information. For most styles, it gets all of the information it needs, however, for some styles, that information is just not there, so you might not always get all of the right colors such as background colors.

If you do not like the default background color, specify `gopplot=cback=`*some-color*, (substituting your favorite color for some-color). This controls the area outside the graph. The area inside the graph is controlled by the `cframe=` option. You can permanently change the default by modifying the macro. Similarly, you can change `color=` and `colors=` as you see fit. The following step creates a plot with a white background inside the axes and a light gray background around the axes:

```
%plotit(data=sashelp.class(drop=age sex), cframe=white, gopts=cback=ligr)
```

If you are specifying colors, you will probably want to use the `nogstyle` option so that the style does not override the colors that you specify.

Sample Usage

For many plots, you only need to specify the `data=` and `datatype=` options. The following steps perform a simple correspondence analysis, however, PROC CORRESP and ODS Graphics can automatically make the same plot:

```
*------Simple Correspondence Analysis------;
proc corresp all data=cars outc=coor;
   tables marital, origin;
   title 'Simple Correspondence Analysis';
   run;

%plotit(data=coor, datatype=corresp)
```

The following steps perform a multiple correspondence analysis, however, PROC CORRESP and ODS Graphics can automatically make the same plot:

```
*------Multiple Correspondence Analysis------;
proc corresp mca observed data=cars outc=coor;
   tables origin size type income home marital sex;
   title 'Multiple Correspondence Analysis';
   run;

%plotit(data=coor, datatype=mca)
```

The following steps perform a multidimensional preference analysis, however, PROC PRINQUAL and ODS Graphics can automatically make the same plot:

```
*------MDPREF------;
proc prinqual data=carpref out=results n=2 replace mdpref;
   id model mpg reliable ride;
   transform ide(judge1-judge25);
   title 'Multidimensional Preference (MDPREF) Analysis';
   run;

%plotit(data=results, datatype=mdpref 2.5)
```

The vector lengths are increased by a factor of 2.5 to make a better graphical display.

The following steps perform a preference mapping vector model, however, PROC TRANSREG and ODS Graphics can automatically make the same plot:

```
*------PREFMAP, Vector Model------;
proc transreg data=results(where=(_type_ = 'SCORE'));
   model ide(mpg reliable ride)=identity(prin1 prin2);
   output tstandard=center coefficients replace out=tresult1;
   id model;
   title 'Preference Mapping (PREFMAP) Analysis - Vector';
   run;

%plotit(data=tresult1, datatype=vector 2.5)
```

Again, the vector lengths are increased by a factor of 2.5 to make a better graphical display.

The following steps perform a preference mapping ideal point model, however, PROC TRANSREG and ODS Graphics can automatically make the same plot:

```
*------PREFMAP, Ideal Point------;
proc transreg data=results(where=(_type_ = 'SCORE'));
   model identity(mpg reliable ride)=point(prin1 prin2);
   output tstandard=center coordinates replace out=tresult1;
   id model;
   title 'Preference Mapping (PREFMAP) Analysis - Ideal';
   run;

%plotit(data=tresult1, datatype=ideal, antiidea=1)
```

The `antiidea=1` option is specified to handle anti-ideal points when large data values are positive or ideal.

The following steps perform multidimensional preference analysis, however, PROC PRINQUAL and ODS Graphics can automatically make the same plot:

```
*------MDPREF, labeled vector end points------;
proc prinqual data=recreate out=rec mdpref rep;
   transform identity(sub:);
   id activity active relaxing spectato;
   title 'MDPREF of Recreational Activities';
   run;

%plotit(data=rec, datatype=mdpref2 3,
        symlen=2, vechead=, adjust1=%str(
        if _type_ = 'CORR' then do;
           __symbol = substr(activity,4);
           __ssize  = 0.7;
           activity = ' ';
           end;))
```

The `mdpref2` specification means MDPREF and label the vectors *too*. The vector lengths are increased by a factor of 3 to make a better graphical display. The `symlen=2` option specifies two-character symbols. The specification `vechead=`, (a null value) means no vector heads since there are labels. The `adjust1=` option is used to add full SAS DATA step statements to the preprocessed data set. This example processes `_type_ = 'CORR'` observations (those that contain vector the coordinates) the original variable names (`sub1`, `sub2`, `sub3`, ..., from the `activity` variable) and creates symbol values (1, 2, 3, ...) of size 0.7. The result is a plot with each vector labeled with a subject number.

The following steps create a contour plot, displaying density with color:

```
*------Bivariate Normal Density Function------;
proc iml;
   title 'Bivariate Normal Density Function';
   s = inv({1 0.25 , 0.25 1});
   m = -2.5; n = 2.5; inc = 0.05; k = 0;
   x = j((1 + (n - m) / inc) ** 2, 3, 0);
   c = sqrt(det(s)) / (2 * 3.1415);
   do i = m to n by inc;
      do j = m to n by inc;
         v = i || j; z = c * exp(-0.5 * v * s * v`);
         k = k + 1; x[k,] = v || z;
         end;
      end;
   create x from x[colname={'x' 'y' 'z'}]; append from x;
   quit;

%plotit(datatype=contour, data=x, extend=close,
         paint=z white blue magenta red)
```

The `paint=z white blue magenta red` option specifies that color interpolation is based on the variable `z`, going from white (zero density) through blue, magenta, and to red (maximum density). This color list is designed for a background of white. The option `extend=close` is used with contour plots so that the plot boundaries appear exactly at the edge of the contour data. By default, `%PlotIt` usually adds a bit of extra white space between the data and the plot boundaries which provides extra room for labels, which are not used in this example. Similar plots can be made with ODS Graphics.

The goal of this next example is to create a plot of the Fisher iris data set with each observation identified by its species. Similar and nicer plots can be made with ODS Graphics. Species name is centered at each point's location, and each species name is plotted in a different color. This scatter plot is overlaid on the densities used by PROC DISCRIM to classify the observations. There are three densities, one for each species. Density is portrayed by a color contour plot with white (the assumed background color) indicating a density of essentially zero. Yellow, orange, and red indicate successively increasing density.

The `data=` option names the input SAS data set. The `plotvars=` option names the y-axis and x-axis variables. The `labelvar=_blank_` option specifies that all labels are blank. This example does not use any of PROC PLOT's label collision avoidance code. It simply uses PROC PLOT to figure out how big to make the plot, and then the macro puts everything inside the plot independently of PROC PLOT, so the printer plot is blank. The `symlen=4` option specifies that the maximum length of a symbol value is 4 characters. This is because we want the first four characters of the species names as symbols. The `exttypes=symbol contour` option explicitly specifies that PROC PLOT will know nothing about the symbols or the contours. They are external types that are added to the graphical plot by the macro after PROC PLOT has finished. The `ls=100` option specifies a constant line size. Since no label avoidance is done, there can be no collisions, and the macro will not iteratively determine the plot size. The default line size of 65 is too small for this example, whereas `ls=100` makes a better display. The `paint=` option specifies that based on values the variable `density`, colors should be interpolated ranging from white (minimum `density`) to yellow to orange to red (maximum `density`). The `rgbtypes=contour` option specifies that the `paint=` option should apply to contour type observations.

The grid (created with the loops: `do sepallen = 30 to 90 by 0.6;` and `do petallen = 0 to 80 by 0.6;`) is not square, so for optimal results the macro must be told the number of horizontal and vertical positions. The PLOTDATA DATA step creates these values and stores them in macro variables `&hnobs` and `&vnobs`, so the specification `hnobs=&hnobs, vnobs=&vnobs`, specifies the grid size. Of course these values could have been specified directly instead of through symbolic variables. The `excolors=CXFFFFFF` option is included for efficiency. The input data consist of a large grid for the contour plot. Most of the densities are essentially zero, so many of the colors are `CXFFFFFF`, which is white, computed by `paint=`, which is the same color as the background. (See the `paint=` option, page 1204 for information about CX*rrggbb* color specifications.) Excluding them from processing makes the macro run faster and creates smaller datasets.

This example shows how to manually do the kinds of things that the `datatype=` option does for you with standard types of data sets. The macro expects the data set to contain observations of one or more types. Each type is designated by a different value in a variable, usually named `_type_`. In this example, there are four types of observations, designated by the `_type_` variable's four values, 1, 2, 3, 4, which are specified in the `types=` option. The `symtype=` option specifies the symbol types for these four observation types. The first three types of observations are `symbol` and the last type, `_type_` = 4, designates the contour observations. The first three symbols are the species names (`symbols=` values) displayed in `symfont=none` (hardware font). The last symbol is null because contours do not use symbols. The first three symbols, since they are words as opposed to a single character, are given a small size (`symsize=0.6`). A value of 1 is specified for the symbol size for contour type observations. The macro determines the optimal size for each color rectangle of the contour plot. The following steps process the data, perform the analysis, and create the plot:

```
*------Discriminant Analysis------;
data plotdata;  * Create a grid over which DISCRIM outputs densities.;
   do SepalLength = 30 to 90 by 0.6;
      h + 1; * Number of horizontal cells;
      do PetalLength = 0 to 80 by 0.6;
         n + 1; * Total number of cells;
         output;
         end;
      end;
   call symput('hnobs', compress(put(h    , best12.))); * H grid size;
   call symput('vnobs', compress(put(n / h, best12.))); * V grid size;
   drop n h;
   run;

proc discrim data=iris testdata=plotdata testoutd=plotd
             method=normal pool=no short noclassify;
   class species;
   var PetalLength SepalLength;
   title 'Discriminant Analysis of Fisher (1936) Iris Data';
   title2 'Using Normal Density Estimates with POOL=NO';
   run;

data all;
   * Set the density observations first so the scatter plot points
     are on top of the contour plot.  Otherwise the contour plot
     points will hide the scatter plot points.;
   set plotd iris(in=iris);
   if iris then do;
      _type_ = species; * unformatted species number 1, 2, 3;
      output;
      end;
   else do;
      _type_ = 4; * density observations;
      density = max(setosa,versicolor,virginica);
      output;
      end;
   run;

%plotit(data=all, plotvars=PetalLength SepalLength, labelvar=_blank_,
        symlen=4, exttypes=symbol contour, ls=100, gopplot=cback=white,
        paint=density white yellow orange red, rgbtypes=contour,
        hnobs=&hnobs, vnobs=&vnobs, excolors=CXFFFFFF,
        types  =1      2          3          4,
        symtype=symbol symbol     symbol     contour,
        symbols=Set    Vers       Virg       '',
        symsize=0.6    0.6        0.6        1,
        symfont=none   none       none       solid
        )

title;
```

How %PlotIt Works

You create a data set either with a DATA step or with a procedure. Then you run the macro to create a graphical scatter plot. This macro is not a SAS/GRAPH procedure (although it uses PROC GANNO), and in many ways, it does not behave like a typical SAS/GRAPH procedure. The `%PlotIt` macro performs the following steps.

1. The `%PlotIt` macro reads an input data set and preprocesses it. The preprocessed data set contains information such as the axis variables, the point-symbol and point-label variables, and symbol and label types, sizes, fonts, and colors. The nature of the preprocessing depends on the type of data analysis that generated the input data set. For example, if the option `datatype=mdpref` had been specified with an input data set created by PROC PRINQUAL for a multidimensional preference analysis, then the `%PlotIt` macro creates blue points for `_type_ = 'SCORE'` observations and red vectors for `_type_ = 'CORR'` observations.
2. A DATA step, using the DATA Step Graphics Interface, determines how big to make the graphical plot.
3. PROC PLOT determines where to position the point labels. By default, if some of the point label characters are hidden, the `%PlotIt` macro recreates the printer plot with a larger line and page size, and hence creates more cells and more room for the labels. Note that when there are no point labels, the printer plot might be empty. All of the information that is in the graphical scatter plot can be stored in the `extraobs=` data set. All results from PROC PLOT are written to data sets with ODS. The macro will clear existing `ods select` and `ods exclude` statements.
4. The printer plot is read and information from it, the preprocessed data set, and the extra observations data set are combined to create an Annotate data set. The label position information is read from the PROC PLOT output, and all of the symbol, size, font, and color information is extracted from the preprocessed (or extra observations) data set. The Annotate data set contains all of the instructions for drawing the axes, ticks, tick marks, titles, point symbols, point labels, axis labels, and so on. Circles can be drawn around certain locations, and vectors can be drawn from the origin to other locations.
5. The Annotate data set is displayed with the GANNO procedure. The `%PlotIt` macro does not use PROC GPLOT.

Debugging

When you have problems, try `debug=vars` to see what the macro thinks you specified. It is also helpful to specify: `debug=mprint notes`. You can also display the final Annotate data set and the preprocessing data set, for example, as follows:

```
options ls=180;
proc print data=anno uniform;
   format text $20. comment $40.;
   run;

proc print data=preproc uniform;
   run;
```

Advanced Processing

You can post-process the Annotate DATA step to change colors, fonts, undesirable placements, and so on. Sometimes, this can be done with the `adjust4=` option. Alternatively, when you specify `method=none`, you create an Annotate data set without displaying it. The data set name is by default WORK.ANNO. You can then manipulate it further with a DATA step or PROC FSEDIT to change colors, fonts, or sizes for some labels; move some labels; and so on. If the final result is a new data set called ANNO2, you can display it by running the following:

```
proc ganno annotate=anno2;
   run;
```

Notes

With `method=print`, the macro creates a file. See the `filepref=` and `post=` options and make sure that the file name does not conflict with existing names.

This macro creates variable names that begin with two underscores and assumes that these names will not conflict with any input data set variable names.

It is not feasible with a macro to provide the full range of error checking that is provided with a procedure. Extensive error checking is provided, but not all errors are diagnosed.

Not all options will work with all other options. Some combinations of options might produce macro errors or Annotate errors.

This macro might not be fully portable. When you switch operating systems or graphics devices, some changes might be necessary to get the macro to run successfully again.

Graphics device differences might also be a problem. We do not know of any portability problems, but the macro has not been tested on all supported devices.

Some styles are structured in such a way that the `%PlotIt` macro cannot extract information from them. For those styles, default colors are used instead.

This macro tries to create a plot with equated axes, where a centimeter on one axis represents the same data range as a centimeter on the other axis. The only way to accomplish this is by explicitly and jointly controlling the `hsize=`, `vsize=`, `hpos=`, and `vpos= goptions`. By default, the macro tries to ensure that all of the values work for the specific device. See `makefit=`, `xmax=`, and `ymax=`. By default the macro uses GASK to determine `xmax` and `ymax`. If you change any of these options, your axes might not be equated. Axes are equated when `vsize` × `hpos` / `hsize` × `hpos` = `vtoh`.

When you are plotting variables that have very different scales, you might need to specify appropriate tick increments for both axes to get a reasonable plot, like this for example: `plotopts=haxis=by 20 vaxis=by 5000`. Alternatively, just specifying the smaller increment is often sufficient: `plotopts= haxis= by 20`. Alternatively, specify `vtoh=`, (null value) to get a plot like PROC GPLOT's, with the window filled.

By default, the macro iteratively creates and recreates the plot, increasing the line size and the flexibility in the `placement=` list until there are no penalties.

The SAS system option `ovp` (overprint) is not supported by this macro.

%PlotIt Macro Options

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

Option	Description
`help`	(positional) "help" or "?" displays syntax summary
`adjust1=`*SAS-statements*	adjust the preprocessing data set
`adjust2=`*SAS-statements*	includes statements with PROC PLOT
`adjust3=`*SAS-statements*	extra statements for the final DATA step
`adjust4=`*SAS-statements*	extra statements for the final DATA step
`adjust5=`*SAS-statements*	extra statements for the final DATA step
`antiidea=`*n*	eliminates PREFMAP anti-ideal points
`blue=`*expression*	blue part of RGB colors
`bright=`*n*	generates random label colors
`britypes=`*type*	types to which `bright=` applies
`cframe=`*color*	color of background within the frame
`cirsegs=`*n*	circle smoothness parameter
`color=`*color*	default color
`colors=`*colors-list*	default label and symbol color list
`cursegs=`*n*	number of segments in a curve
`curvecol=`*color*	color of curve
`data=`*SAS-data-set*	input data set
`datatype=`*data-type*	data analysis that generated data set
`debug=`*values*	debugging output
`excolors=`*color-list*	excludes from the Annotate data set
`extend=`*axis-extensions*	extend the x and y axes
`extraobs=`*SAS-data-set*	extra observations data set
`exttypes=`*type*	types for `extraobs=` data set
`filepref=`*prefix*	file name prefix
`font=`*font*	default font
`framecol=`*color*	color of frame
`gdesc=`*description*	catalog description
`gname=`*name*	catalog entry
`gopplot=`*goptions*	`goptions` for plotting to screen
`gopprint=`*goptions*	`goptions` for printing
`gopts2=`*goptions*	`goptions` that are always used
`gopts=`*goptions*	additional `goptions`
`gout=`*catalog*	`proc ganno gout=` catalog
`green=`*expression*	green part of RGB colors
`hminor=`*n* | *do-list*	horizontal axis minor tick marks
`hnobs=`*n*	horizontal observations for contour plots
`hpos=`*n*	horizontal positions in graphics area
`href=`*do-list*	horizontal reference lines
`hsize=`*n*	horizontal graphics area size
`inc=`*n*	`haxis=by inc`, `vaxis=by inc`
`interpol=`*method*	axis interpolation method
`labcol=`*label-colors*	colors for the point labels
`label=`*label-statement*	`label` statement
`labelcol=`*color*	color of variable labels

Option	Description
`labelvar=`*label-variable*	point label variable
`labfont=`*label-fonts*	fonts for the point labels
`labsize=`*label-sizes*	sizes for the point labels
`ls=`*n*	how line sizes are generated
`lsinc=`*n*	increment to line size
`lsizes=`*number-list*	line sizes (thicknesses)
`makefit=`*n*	proportion of graphics window to use
`maxiter=`*n*	maximum number of iterations
`maxokpen=`*n*	maximum acceptable penalty sum
`method=`*value*	where to send the plot
`monochro=`*color*	overrides all other colors
`nknots=`*n*	number of knots option
`offset=`*n*	move symbols for coincident points
`options=border`	draws a border box
`options=close`	close up the border box and the axes
`options=diag`	draws a diagonal reference line
`options=expand`	expands the plot to fill the window
`options=noback`	do not set the frame color
`options=nocenter`	do not center
`options=noclip`	do not clip
`options=nocode`	suppress the PROC PLOT and `goptions` statements
`options=nodelete`	do not delete intermediate data sets
`options=nohistory`	suppress the iteration history table
`options=nolegend`	suppress the display of the legends
`options=noprint`	`nolegend`, `nocode`, and `nohistory`
`options=square`	the same ticks for both axes and a square plot
`options=textline`	lines overwrite text
`out=`*SAS-data-set*	output Annotate data set
`outward=none` | *char*	PLOT statement `outward=`
`paint=`*interpolation*	color interpolation
`place=`*placement*	generates a `placement=` option
`plotopts=`*options*	PLOT statement options
`plotvars=`*variable-list*	y-axis and x-axis variables
`post=`*filename*	graphics stream file name
`preproc=`*SAS-data-set*	preprocessed `data=` data set
`procopts=`*options*	PROC PLOT statement options
`ps=`*n*	page size
`radii=`*do-list*	radii of circles
`red=`*expression*	red part of RGB colors
`regdat=`*SAS-data-set*	intermediate regression results data set
`regopts=`*options*	regression curve fitting options
`regprint=`*regression-option*	regression options
`rgbround=`*RGB-rounding*	`paint=` rounding factors
`rgbtypes=`*type*	types to which `paint=` and RGB options apply
`style=`A | B	obsolete option not supported
`symbols=`*symbol-list*	plotting symbols
`symcol=`*symbol-colors*	colors of the symbols
`symfont=`*symbol fonts*	symbol fonts

Option	Description
`symlen=`*n*	length of the symbols
`symsize=`*symbol-sizes*	sizes of symbols
`symtype=`*symbol-types*	types of symbols
`symvar=`*symbol-variable*	plotting symbol variable
`tempdat1=`*SAS-data-set*	intermediate results data set
`tempdat2=`*SAS-data-set*	intermediate results data set
`tempdat3=`*SAS-data-set*	intermediate results data set
`tempdat4=`*SAS-data-set*	intermediate results data set
`tempdat5=`*SAS-data-set*	intermediate results data set
`tempdat6=`*SAS-data-set*	intermediate results data set
`tickaxes=`*axis-string*	axes to draw tick marks
`tickcol=`*color*	color of ticks
`tickfor=`*format*	tick format used by `interpol=tick`
`ticklen=`*n*	length of tick mark in horizontal cells
`titlecol=`*color*	color of title
`tsize=`*n*	default text size
`types=`*observation-types*	observations types
`typevar=`*variable*	observation types variable
`unit=in` | `cm`	`hsize=` and `vsize=` unit
`vechead=`*vector-head-size*	how to draw vector heads
`vminor=`*n* | *do-list*	vertical axis minor tick marks
`vnobs=`*n*	vertical observations for contour plots
`vpos=`*n*	vertical positions in graphics area
`vref=`*do-list*	vertical reference lines
`vsize=`*n*	vertical graphics area size
`vtoh=`*n*	PROC PLOT `vtoh=` option
`xmax=`*n*	maximum horizontal graphics area size
`ymax=`*n*	maximum vertical graphics area size

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

```
%plotit(help)
%plotit(?)
```

Note that for many analyses, the only options you need to specify are `data=`, `datatype=`, and sometimes `method=`. To specify variables to plot, specify `plotvars=`, `labelvar=`, and `symvar=`.

Overriding Options

This macro looks for a special global macro variable named `plotitop`. If it exists, its values are used to override the macro options. If you have a series of calls to the plotting macro and say, for example, that you want to route each graph to a PDF file, you can specify the following statement once:

```
%let plotitop = gopts=dev=pdf, method=print;
```

Then you can run the macro repeatedly without change. The value of the `plotitop` macro variable must consist of a name, followed by an equal sign, followed by a value. Optionally, it can continue with a comma, followed by another `name=`*value*, and so on, just like the way options are specified with the macro. Option values must not contain commas.

Destination and GOPTIONS

The options in this section specify the plot destination and SAS `goptions`. Note that with the `%PlotIt` macro, you do not specify a `goptions` statement. If you do, it will be overridden. All `goptions` (except `device=`) are specified with macro options. If you would prefer to specify your own `goptions` statement and have the macro use it, just specify or change the default for these four options to null: `gopplot=`, `gopprint=`, `gopts2=`, `gopts=`. If you use a locally installed copy of the macro, you can modify the `gopprint=` and `gopplot=` options defaults to include the devices that you typically use. Otherwise, the macro checks the `goptions` to get a device.

gopplot= *goptions*

specifies the `goptions` for directly plotting on the screen. There are no default goptions for `gopplot=`.

gopprint= *goptions*

specifies the `goptions` for printing (creating a graphics stream file). The default is `gopprint=gsfmode=replace gaccess=gsasfile gsfname=gsasfile`.

The following options show how you might modify the defaults for `gopprint=` and `gopplot=` option defaults to set default devices:

```
gopprint=gsfmode=replace gaccess=gsasfile gsfname=gsasfile device=png,
gopplot=cback=black device=win,
```

gopts= *goptions*

provides a way to specify additional `goptions` that are always used. There are no default goptions for `gopts=`. For example, to rotate to a landscape orientation with a black background color, specify `gopts=rotate cback=black`.

gopts2= *goptions*

specifies the `goptions` that are always used, no matter which `method=` is specified. The default is `gopts2=reset=goptions erase`.

method= `gplot` | `plot` | `print` | `none`

specifies where to send the plot. The default is `method=gplot`.

`gplot` – displays a graphical scatter plot on your screen using the `goptions` from `gopplot=`. The `gopplot=` option should contain the `goptions` that only apply to plots displayed on the screen.

`plot` – creates a printer plot only.

`print` – routes the plot to a graphics stream file, such as a postscript file, using the `goptions` from `gopprint=`. The `gopprint=` option should contain the `goptions` that only apply to hard-copy plots. Specify the file name with `post=`.

`none` – just creates the Annotate data set and sets up `goptions` using `gopplot=`.

Data Set and Catalog Options

These options specify the input SAS data set, output Annotate data set, and options for writing plots to files and catalogs.

data= *SAS-data-set*

specifies the input data set. The default input data set is the last data set created. You should always specify the `data=` option since the macro creates data sets that are not suitable for use as input.

filepref= *file-name-prefix*

specifies the file name prefix. The default is `filepref=sasplot`.

gdesc= *description*

specifies the name of a catalog description. This option can optionally be used with `proc ganno gout=` to provide the `description=`.

gname= *name*

specifies the name of a catalog entry. This option can optionally be used with `proc ganno gout=` to provide the `name=`.

gout= *catalog*

specifies the `proc ganno gout=` catalog. With `gout=gc.slides`, first specify: `libname gc '.'`; Then to replay, run: `proc greplay igout=gc.slides; run;` Note that replayed plots will not in general have the correct aspect ratio.

out= *SAS-data-set*

specifies the output Annotate data set. This data set contains all of the instructions for drawing the graph. The default is `out=anno`.

post= *filename*

specifies the graphics stream file name. The default name is constructed from the `filepref=` value and '`ps`' in a host-specific way.

Typical Options

These are some of the most frequently used options.

datatype= *data-type*

specifies the type of data analysis that generated the data set. This option is used to set defaults for other options and to do some preprocessing of the data.

When the data type is `corresp`, `mds`, `mca`, `row`, `column`, `mdpref`, `mdpref2`, `vector`, or `ideal`, the `label=typical` option is implied when `label=` is not otherwise specified. The default point label variable is the last character variable in the data set.

Some data types (`mdpref`, `vector`, `ideal`, `corresp`, `row`, `mca`, `column`, `mds`) expect certain observation types and set the `types=` list accordingly. For example, `mdpref` expects `_type_ = 'SCORE'` and `_type_ = 'CORR'` observations. The remaining data types do not expect any specific value of the `typevar=` variable. So if you do not get the right data types associated with the right observation types, specify `types=`, and specify the `types=` values in an order that corresponds to the order of the symbol types in the `Types Legend` table. Unlike `symtype=`, the order in which you specify `datatype=` values is irrelevant.

A null value (`datatype=`, the default) specifies no special processing, and the default plotting variables are the first two numeric variables in the data set. Specifying `corresp`, `mds`, `mca`, `row`, or `column` will set the default `plotvars` to `dim2` and `dim1`. Otherwise, when a nonnull value is specified, the default `plotvars` are `prin2` and `prin1`.

The various data types are as follows:

`datatype=column`
specifies a `proc corresp profile=column` analysis. Row points are plotted as vectors.

`datatype=contour`
draws solid color contour plots. When the number of row points is not the same as the number of column points in the grid, use `hnobs=` and `vnobs=` to specify the number of points. This method creates an `hnobs=` by `vnobs=` grid of colored rectangles. Each of the rectangles should touch all adjacent rectangles. This method works well with a regular grid of points. The `method=square` option is a good alternative when the data do not fall in a regular grid.

`datatype=corresp`
specifies an ordinary correspondence analysis.

`datatype=curve`
fits a curve through the scatter plot.

`datatype=curve2`
fits a curve through the scatter plot and tries to make the labels avoid overprinting the curve.

`datatype=function`
draws functions. Typically, no labels or symbols are drawn. This option has a similar effect to the PROC GPLOT `symbol` statement options `i=join v=none`.

`datatype=ideal`
specifies a PREFMAP ideal point model. See the `antiidea=`, `radii=`, and `cirsegs=` options.

`datatype=mca`
specifies a multiple correspondence analysis.

`datatype=mdpref`
specifies multidimensional preference analysis with vectors with blank labels. Note that `datatype=mdpref` can also be used for ordinary principal component analysis.

`datatype=mdpref2`
specifies MDPREF with vector labels (MDPREF and labels *too*).

`datatype=mds`
specifies multidimensional scaling.

`datatype=mds ideal`
specifies PREFMAP ideal point after the MDS.

`datatype=mds vector`
specifies PREFMAP after MDS.

`datatype=row`
specifies a `proc corresp profile=row` analysis. Column points are plotted as vectors.

`datatype=square`
plots each point as a solid square. The `datatype=square` option is useful as a form of contour plotting when the data do not form a regular grid. The `datatype=square` option, unlike `datatype=contour`, does not try to scale the size of the square so that each square will touch another square.

`datatype=symbol`
specifies an ordinary scatter plot.

`datatype=vector`
specifies a PREFMAP vector model.

`datatype=vector ideal`
specifies both PREFMAP vectors and ideal points.

For some `datatype=` values, a number can be specified after the name. This is primarily useful for biplot data sets produced by PROC PRINQUAL and PREFMAP data sets produced by PROC TRANSREG. This number specifies that the lengths of vectors should be changed by this amount. The number must be specified last. Examples: `datatype=mdpref 2`, `datatype=mds vector 1.5`.

The primary purpose of the `datatype=` option is to provide an easy mechanism for specifying defaults for the options in the next section (`typevar=` through `outward=`).

labelvar= *label-variable* | `_blank_`

specifies the variable that contains the point labels. The default is the last character variable in the data set. If `labelvar=_blank_` is specified, the macro will create a blank label variable.

options= *options-list*

specifies binary options. Specify zero, one, or more in any order. For example: `options=nocenter nolegend`.

`border`
`noborder`

draws a border box around the outside of the graphics area, like the border `goption`. The default depends on the style, and with typical styles, there is a border.

`close`
if a border is being drawn, perform the same adjustments on the border that are performed on the axes. This option is most useful with contour plots.

`diag`
draws a diagonal reference line.

`expand`
specifies Annotate data set post processing, typically for use with `extend=close` and contour plots. This option makes the plot bigger to fill up more of the window.

`noback`
specifies that `%PlotIt` should not set the frame color (the background color within the plot boundary).

`nocenter`
do not center. By default, when `nocenter` is not specified, `vsize=` and `hsize=` are set to their maximum values, and the `vpos=` and `hpos=` values are increased accordingly. The x and y coordinates are increased to positioning the plot in the center of the full graphics area.

`noclip`
do not clip. By default, when `noclip` is not specified, labels that extend past the edges of the plot are clipped. This option will not absolutely prevent labels from extending beyond the plot, particularly when sizes are greater than 1.

`nocode`
suppresses the display of the PROC PLOT and `goptions` statements.

`nodelete`
do not delete intermediate data sets.

`nohistory`
suppresses the display of the iteration history table.

`nolegend`
suppresses the display of the legends.

`noprint`
equivalent to `nolegend`, `nocode`, and `nohistory`.

`square`
uses the same ticks for both axes and tries to make the plot square by tinkering with the `extend=` option. Otherwise, ticks might be different.

`textline`
put text in the data set, followed by lines, so lines overwrite text. Otherwise text overwrites lines.

plotvars= *two-variable-names*

specifies the y-axis variable then the x-axis variable. To plot `dim2` and `dim3`, specify `plotvars=dim2 dim3`. The `datatype=` option controls the default variable list.

symlen= *n*

specifies the length of the symbols. By default, symbols are single characters, but the macro can center longer strings at the symbol location.

symvar= *symbol-variable* | `_symbol_`

specifies the variable that contains the plotting symbol for input to PROC PLOT. When `_symbol_` is specified, which is the default, the symbol variable is created, typically from the `symbols=` list, which might be constructed inside the macro. (Note that the variable `__symbol` is created to contain the symbol for the graphical scatter plot. The variables `_symbol_` and `__symbol` might or might not contain the same values.) Variables can be specified, and the first `symlen=` characters are used for the symbol. When a null value (`symvar=`) or a constant value is specified, the symbol from the printer plot is used (which is always length one, no matter what is specified for `symlen=`). To get PROC PLOT pointer symbols, specify `symvar='00'x`, (hex null constant). To center labels with no symbols, specify: `symbols='', place=0`.

Observation-Type List Options

Data sets for plotting can have different types of observations that are plotted differently. These options let you specify the types of observations, the variable that contains the observation types, and the different ways the different types should be plotted. For many types of analyses, these can all be handled easily with the `datatype=` option, which sets analysis-specific defaults for the list options. When you can, you should use `datatype=` instead of the list options. If you do use the list options, specify a variable, in `typevar=`, whose values distinguish the observation types. Specify the list of valid types in `types=`. Then specify fonts, sizes, and so on for the various observation types. Alternatively, you can use these options with `datatype=`. Specify lists for just those label or symbol characteristics you want to change, for example, colors, fonts or sizes.

The lists do not all have to have the same number of elements. The number of elements in `types=` determines how many of the other list elements are used. When an observation type does not match one of the `type=` values, the results are the same as if the first type is matched. If one of the other lists is shorter than the `types=` list, the shorter list is extended by adding copies of the last element to the end. Individual list values can be quoted, but quotes are not required. *Embedded blanks are not permitted. If you embed blanks, you will not get the right results.* Values of the `typevar=` variable are compressed before they are used, so for example, an `_type_` value of `'M COEFFI'` must be specified as `'MCOEFFI'`.

britypes= *type*

specifies the types to which `bright=` applies. The default is `britypes=symbol`.

colors= *colors-list*

specifies the default color list for the `symcol=` and `labcol=` options. The default depends on the style. With `gstyle`, this list is ignored and the list comes from the style. Otherwise, the default is `colors=blue red green cyan magenta orange gold lilac olive purple brown gray rose violet salmon yellow`. This is the old legacy default. Colors such as these from the default ODS style `colors=CX2A25D9 CXB2182B CX01665E CX9D3CDB CX543005 CX2597FA CX7F8E1F CXB26084 CXD17800 CX47A82A CXB38EF3 CXF9DA04` tend to look much better than the default colors. (See page 1204 for information about CX*rrggbb* color specifications.)

exttypes= *type*

specifies the types to always put in the `extraobs=` data set when they have blank labels. The default is `exttypes=vector`.

labcol= *label-colors*

specifies the colors for the point labels. The default either comes from the style or from `colors=` with `nogstyle`. This option is ignored with `gstyle`. Examples (none means hardware):

```
labcol='red'
labcol='red' 'white' 'blue'
```

labfont= *label-fonts*

specifies the fonts for the point labels. Examples (none means hardware):

```
labfont=none
labfont='swiss'
labfont='swiss' 'swissi'
```

labsize= *label-sizes*

specifies the sizes for the point labels. Examples:

```
labsize=1
labsize=1 1.5
labsize=1 0
```

rgbtypes= *type*

specifies the types to which `paint=`, `red=`, `green=`, and `blue=` apply. The default is `rgbtypes=symbol`.

symbols= *symbol-list*

specifies the plotting symbols. Symbols can be more than a single character. You must specify `symlen=n` for longer lengths. Blank symbols must be specified as `''` with no embedded blanks. Examples:

```
symbols='*'
symbols='**'
symbols='*' '+' '*' ''
symbols='NC' 'OH' 'NJ' 'NY'
```

symcol= *symbol-colors*

specifies the colors of the symbols. The default list is constructed from the `colors=` option. Examples:

```
symcol='red'
symcol='red' 'white' 'blue'
```

symtype= *symbol-types*

specifies the types of symbols. Valid types include: `symbol`, `vector`, `circle`, `contour`, and `square`. Examples:

```
symtype='symbol'
symtype='symbol' 'vector'
symtype='symbol' 'circle'
```

symfont= *symbol fonts*

specifies the symbol fonts. The font is ignored for vectors with no symbols. Examples (none means hardware):

```
symfont=none
symfont='swiss'
symfont='swiss' 'swissi'
```

symsize= *symbol-sizes*

specifies the sizes of symbols. Examples:

```
symsize=1
symsize=1 1.5
```

types= *observation-types*

specifies the observations types. Observation types are usually values of a variable like `_type_`. Embedded blanks are not permitted. Examples:

```
types='SCORE'
types='OBS' 'SUPOBS' 'VAR' 'SUPVAR'
types='SCORE'
types='SCORE' 'MCOEFFI'
```

The order in which values are specified for the other options depends on the order of the types. The default types for various `datatype=` values are given next:

```
corresp:  'VAR' 'OBS' 'SUPVAR' 'SUPOBS'
row:      'VAR' 'OBS' 'SUPVAR' 'SUPOBS'
mca:      'VAR' 'OBS' 'SUPVAR' 'SUPOBS'
column:   'VAR' 'OBS' 'SUPVAR' 'SUPOBS'
mdpref:   'SCORE' 'CORR'
vector:   'SCORE' 'MCOEFFI'
ideal:    'SCORE' 'MPOINT'
mds:      'SCORE' 'CONFIG'
```

For combinations of options, these lists are combined in order, but without repeating `'SCORE'`, for example, with `datatype=mdpref vector ideal`, the default `types=` list is: `'SCORE' 'CORR' 'MCOEFFI' 'MPOINT'`.

typevar= *variable*

specifies a variable that is looked at for the observation types. By default, this is `_type_` if it is in the input data set.

Internal Data Set Options

The macro creates one or more of these data sets internally to store intermediate results.

extraobs= *SAS-data-set*

specifies a data set used to contain the extra observations that do not go through PROC PLOT. The default is `extraobs=extraobs`.

preproc= *SAS-data-set*

specifies a data set used to contain the preprocessed `data=` data set. The default is `preproc=preproc`.

regdat= *SAS-data-set*

specifies a data set used to contain intermediate regression results for curve fitting. The default is `regdat=regdat`.

tempdat1= *SAS-data-set*

specifies a data set used to hold intermediate results. The default is `tempdat1=tempdat1`.

tempdat2= *SAS-data-set*

specifies a data set used to hold intermediate results. The default is `tempdat2=tempdat2`.

tempdat3= *SAS-data-set*

specifies a data set used to hold intermediate results. The default is `tempdat3=tempdat3`.

tempdat4= *SAS-data-set*

specifies a data set used to hold intermediate results. The default is `tempdat4=tempdat4`.

tempdat5= *SAS-data-set*

specifies a data set used to hold intermediate results. The default is `tempdat5=tempdat5`.

tempdat6= *SAS-data-set*

specifies a data set used to hold intermediate results. The default is `tempdat6=tempdat6`.

Miscellaneous Options

The following options are sometimes needed for certain situations to control the details of the plots:

antiidea= *n*

eliminates PREFMAP anti-ideal points. The TRANSREG ideal-point model assumes that small attribute ratings mean that the object is similar to the attribute and large ratings imply dissimilarity to the attribute. For example, if the objects are food and the attribute is "sweetness," then the analysis assumes that 1 means sweet and 9 is much less sweet. The resulting coordinates are usually ideal points, representing an ideal amount of the attribute, but sometimes they are anti-ideal points and need to be converted to ideal points. This option is used to specify the nature of the data (small ratings mean similar or dissimilar) and to request automatic conversion of anti-ideal points.

null value – (`antiidea=`, the default) – do nothing.

1 – reverses in observations whose `_TYPE_` contains 'POINT' when `_issq_` > 0. Specify `antiidea=1` with `datatype=ideal` for the unusual case when large data values are positive or ideal.

–1 – reverses in observations whose `_type_` contains 'POINT' when `_issq_` < 0. Specify `antiidea=-1` with `datatype=ideal` for the typical case when small data values are positive or ideal.

extend= *axis-extensions*

is used to extend the x and y axes beyond their default lengths. Specify four values, for the left, right, top, and bottom axes. If the word `close` is specified somewhere in the string, then macro moves the axes in close to the extreme data values, and the computed values are added to the specified values (if any). Sample specifications: `extend=2 2`, or `extend=3 3 -0.5 0.5`. Specifying a positive value n extends the axis n positions in the indicated direction. Specifying a negative value shrinks the axis. The defaults are in the range –2 to 2, and are chosen in an attempt to add a little extra horizontal space and make equal the extra space next to each of the four extreme ticks. When there is enough space, the horizontal axis is slightly extended by default to decrease the chance of a label slightly extending outside the plot. PROC PLOT usually puts one or two more lines on the top of the plot than in the bottom. The macro tries to eliminate this discrepancy. This option does not add any tick marks; it just extends or shrinks the ends of the axis lines. So typically, only small values should be specified. Be careful with this option and a positive `makefit=` value.

font= *font*

specifies the default font. With `nogstyle`, the default is `font=swiss`. Otherwise the font comes from the style. The annotate data set will often contain fonts of `none`, which means that a hardware font is used. This will usually look much better than a software font such as the Swiss fonts that were heavily used before the 9.2 release.

hminor= *n* | *do-list*

specifies the number of horizontal axis minor tick marks between major tick marks. A typical value is 9. The number cannot be specified when `haxis=` is specified with `plotopts=`. Alternatively, specify a DATA step `do` list. Note that with log scaling, specify $\log_{10}$'s of the data values. For example, specify `hminor=0.25 to 5 by 0.25`, with data ranging up to 10^5.

href= *do-list*

specifies horizontal-axis reference lines (which are drawn vertically). Specify a DATA step `do` list. By default, there are no reference lines.

inc= *n*

specifies `haxis=by inc` and `vaxis=by inc` values. The specified increments apply to both axes. To individually control the increments, you must specify the PLOT statement `haxis=` and `vaxis=` options on the `plotopts=` option. When you are plotting variables that have very different scales, you might need to independently specify appropriate tick increments for both axes to get a reasonable plot, like this for example: `plotopts=haxis=by 20 vaxis=by 5000`.

interpol= `ls` | `tick` | `no` | `hlog` | `vlog` | `yes`

specifies the axis interpolation method.

> `ls` – uses the least-squares method only. This method computes the mapping between data and positions using ordinary least-squares linear regression. Usually, you should not specify `interpol=ls` because slight inaccuracies can result, producing aesthetically unappealing plots.
>
> `hlog` – specifies that the x-axis is on a log scale.
>
> `no` – does not interpolate.
>
> `tick` – uses the tick mark method. This method computes the slope and intercept using tick marks and their values. Tick marks are read using the `tickfor=` format.
>
> `vlog` – specifies that the y-axis is on a log scale.
>
> `yes` – the default, interpolates symbol locations, starting with least squares but replacing them with tick-based estimates when they are available.

This option makes the symbols, vectors, and circles map to the location they would in a true graphical scatter plot, not the cell locations from PROC PLOT. This option has no effect on labels, the frame, reference lines, titles, or ticks. With `interpol=no`, plots tend to look nicer whereas `interpol=yes` plots are slightly more accurate. Note that the strategy used to interpolate can be defeated in certain cases. If the horizontal axis tick values are displayed vertically, specify `interpol=ls`. The `hlog` and `vlog` values are specified in addition to the method. For example, `interpol=yes vlog hlog`.

label= *label-statement*

specifies a `label` statement. Note that specifying the keyword `label` to begin the statement is optional. You can specify `label=typical` to request a label statement constructed with `'Dimension'` and the numeric suffix of the variable name, for example, `label dim1 = 'Dimension 1' dim2 = 'Dimension 2';` when `plotvars=dim2 dim1`. The `label=typical` option can only be used with variable names that consist of a prefix and a numeric suffix.

ls= *n* | *iterative-specification*

specifies how line sizes are generated. The default is `ls=compute search`. When the second word is `search`, the macro searches for an optimal line size. See the `place=` option for more information about searches. When the first word is `compute`, the line size is computed from the iteration number so that the line sizes are: 65 80 100 125 150 175 200. Otherwise the first word is the first linesize and with each iteration the linesize is incremented by the `lsinc=` amount. Example: `ls=65 search`.

lsinc= *n*
specifies the increment to line size in iterations when line size is not computed. The default is `lsinc=15`.

lsizes= *number-list*
specifies the line sizes (thicknesses) for frame, ticks, vectors, circles, curves, respectively. The default is `lsizes=1 1 1 1 1`.

maxiter= *n*
specifies the maximum number of iterations. The default is `maxiter=15`.

maxokpen= *n*
specifies the maximum acceptable penalty sum. The default is `maxokpen=0`. Penalties accrue when label characters collide, labels get moved too far from their symbols, or words get split.

offset= *n*
move symbols for coincident points `offset=` spaces up/down and left/right. This helps to better display coincident symbols. Specify a null value (`offset=,`) to turn off offsetting. The default is `offset=0.25`.

place= *placement-specification*
generates a `placement=` option for the plot request. The default is `place=2 search`. Specify a non-negative integer. Values greater than 13 are set to 13. As the value gets larger, the procedure is given more freedom to move the labels farther from the symbols. The generated placement list is displayed in the log. You can still specify `placement=` directly in the `plotopts=` option. This option just gives you a shorthand notation. The following list shows the correspondence between these two options:

```
place=0 --placement=((s=center))


place=1 --placement=((h=2 -2 : s=right left)
                     (v=1 * h=0 -1 to -2 by alt))


place=2 --placement=((h=2 -2 : s=right left)
                     (v=1 -1 * h=0 -1 to -5 by alt))


place=3 --placement=((h=2 -2 : s=right left)
                     (v=1 to 2 by alt * h=0 -1 to -10 by alt))


place=4 --placement=((h=2 -2 : s=right left)
                     (v=1 to 2 by alt * h=0 -1 to -10 by alt)
                     (s=center right left * v=0 1 to 2 by alt *
                      h=0 -1 to -6 by alt * l= 1 to 2))
```

and so on.

The `place=` option, along with the `ls=` option can be used to search for an optimal placement list and an optimal line size. By default, the macro will create and recreate the plot until it avoids all collisions.

The search is turned off when a `placement=` option is detected in the plot request or plot options.

If search is not specified with `place=` or `ls=`, the specified value is fixed. If search is specified with the other option, only that option's value is incremented in the search.

plotopts= *options*

specifies PLOT statement options. The `box` option is specified, even if you do not specify it. Reference lines should not be specified using the PROC PLOT `href=` and `vref=` options. Instead, they should be specified directly using the `href=` and `vref=` macro options. By default, no PLOT statement options are specified except `box`.

procopts= *options*

specifies PROC PLOT statement options. The default is `procopts=nolegend`.

tickaxes= *axis-string*

specifies the axes to draw tick marks. The default with `nogstyle` is `tickaxes=LRTBFlb`, and with a style, the default is `tickaxes=LBF`, which provides a cleaner look than the old default, more like ODS Graphics. The specification, `tickaxes=LRTBFlb`, means major ticks on left (L), right (R), top (T), and bottom (B), and the full frame (F) is to be drawn, and potentially minor tick marks on the left (l) and bottom (b). Minor ticks on the right (r) and top (t) can also be requested. To just have major tick marks on the left and bottom axes, and no full frame, specify `tickaxes=LB`. Order and spacing do not matter. `hminor=` and `vminor=` must also be specified to get minor ticks.

tickfor= *format*

specifies the tick format used by `interpol=tick`. You should change this if the tick values in the PROC PLOT output cannot be read with the default `tickfor=32.` format. For example, specify `tickfor=date7.` with dates.

ticklen= *n*

specifies the length of tick marks in horizontal cells. A negative value can be specified to indicate that only half ticks should be used, which means the ticks run to but not across the axes. The default is `ticklen=1.5`.

tsize= *n*

specifies the default text size. The default is `tsize=1`.

vminor= *n* | *do-list*

specifies the number of vertical axis minor tick marks between major tick marks. A typical value is 9. The number cannot be specified when `vaxis=` is specified with `plotopts=`. Alternatively, specify a DATA step `do` list. Note that with log scaling, specify $\log_{10}$'s of the data values. For example, specify `vminor=0.25 to 5 by 0.25`, with data ranging up to 10^5.

vref= *do-list*
specifies vertical reference lines (which are drawn horizontally). Specify a DATA step `do` list. By default, there are no reference lines.

Color Options

With `gstyle`, the symbol and point label colors are set by the style. Otherwise, the symbol and point label colors are set by the `labcol=` and `symcol=` options. The other color options are as follows:

bright= *n*
generates random label colors for `britypes=` values. In congested plots, it might be easier to see which labels and symbols go together if each label/symbol pair has a different random color. Colors are computed so that the mean RGB (red, green, blue) components equal the specified `bright=` value. The valid range is $5 \leq bright \leq 250$. 128 is a good value. Small values will produce essentially black labels and large values will produce essentially white labels, and so should be avoided. The default is a null value, `bright=`, and there are no random label colors. If you get a color table full error message, you need to specify larger values for the `rgbround=` option.

cframe= *color*
specifies the color of the background within the frame. This is analogous to the `cframe=` SAS/GRAPH option. With `gstyle`, the style color is used. This option is ignored with `gstyle`.

color= *color*
specifies the default color that is used when no other color is set. The default color is black. With `gstyle`, this option will typically have little or no effect.

curvecol= *color*
specifies the color of curves in a regression plot. The default either comes from the style or from `color=` with `nogstyle`. This option is ignored with `gstyle`.

excolors= *color-list*
excludes observations from the Annotate data set with colors in this list. For example, with a white background, to exclude all observations that have a color set to white as well as those with a computed white color, for example, from `bright=` or `paint=`, specify `excolors=white CXFFFFFF`. This is done for efficiency, to make the Annotate data set smaller. (See the `paint=` option, page 1204 for information about CX*rrggbb* color specifications.)

framecol= *color*
specifies the color of the frame. The default either comes from the style or from `color=` with `nogstyle`. This option is ignored with `gstyle`.

labelcol= *color*
specifies the color of the variable labels. The default either comes from the style or from `color=` with `nogstyle`. This option is ignored with `gstyle`.

monochro= *color*
overrides all other specified colors. By default when `monochro=` is null, this option has no effect. Typical usage: `monochro=black`. Typically, you would rarely if ever use this option, and you would only use it with `nogstyle`.

style= this option is obsolete and is no longer supported. Use OPTIONS `gstyle/nogstyle` and a `style=` specification in your ODS destination to control the style. The `nogstyle` option will make the macro run the way it did in older releases. The `gstyle` option will make the appearance sensitive to the ODS style. The new results are usually superior. However, if you want explicit color control, you will typically need to specify `nogstyle`.

tickcol= *color*
specifies the color of tick labels. The default either comes from the style or from `color=` with `nogstyle`. This option is ignored with `gstyle`.

titlecol= *color*
specifies the color of the title. The default either comes from the style or from `color=` with `nogstyle`. This option is ignored with `gstyle`.

Color Interpolation and Painting

These next options are used to create label and symbol colors using some function of the input data set variables. For example, you can plot the first two principal components on the x and y axes and show the third principal component in the same plot by using it to control the label colors. The `paint=` option gives you a simple and fairly general way to interpolate colors. The `red=`, `green=`, and `blue=` options are used together for many other types of interpolations, but these options are much harder to use. These options apply to `rgbtypes=` observations. If `red=`, `green=`, and `blue=` are not flexible enough, for example, if you need full statements, specify `red=128` (so later code will know you are computing colors) then insert the full statements you need to generate the colors using `adjust1=`.

paint= *color-interpolation-specification*
is used to interpolate between colors based on the values of a variable. The simplest specification is `paint=variable`. More generally, specify the following option:

```
paint=variable optional-color-list optional-data-value-list
```

The following color names are recognized: red, green, blue, yellow, magenta, cyan, black, white, orange, brown, gray, olive, pink, purple, violet. For other colors, specify the RGB color name. The `paint=` option applies to the observation types mentioned in the `rgbtypes=` option. Valid types include: `symbol`, `vector`, `circle`, `contour`, and `square`.

Colors can be represented as `CX`*rrggbb* where *rr* is the red value, *gg* is the green, and *bb* is blue, all three specified in hex. The base ten numbers 0 to 255 map to 00 to FF in hex. For example, white is `CXFFFFFF` (all colors at their maximum), black is `CX000000` (all colors at their minimum), red is `CXFF0000` (maximum red, minimum green and blue), blue is `CX0000FF` (maximum blue, minimum red and green), and magenta is `CXFF00FF` (maximum red and blue, minimum green). When a variable named `z` is specified with no other arguments, the default is `paint=z blue magenta red`. The option `paint=z red green 1 10` interpolates between red and green, based on the values of the variable `z`, where values of 1 or less map to red, values of 10 or more map to green, and values in between map to colors in between. The specification `paint=z red yellow green 1 5 10`, interpolates between red at `z=1`, yellow at `Z=5`, and green at `Z=10`. If the data value list is omitted, it is computed from the data.

red= *expression*
green= *expression*
blue= *expression*

specify for arithmetic expressions that produce integers in the range 0 to 255. Colors are created as follows:

```
__color = 'CX' ||
          put(%if &red   ne %then round(&red,  __roured); %else 128; ,hex2.) ||
          put(%if &green ne %then round(&green,__rougre); %else 128; ,hex2.) ||
          put(%if &blue  ne %then round(&blue, __roublu); %else 128; ,hex2.);}
```

The `__rou` variables are extracted from the second through fourth values of the `rgbround=` option. Example: `red = min(100 + (z - 10) * 3, 255)`, `blue=50`, `green=50`. Then all labels are various shades of red, depending on the value of `z`. Be aware that light colors (small red-green-blue values) do not show up well on white backgrounds and dark colors do not show up well on dark backgrounds. Typically, you will not want to use the full range of possible red-green-blue values. Computed values greater than 255 are set to 255.

rgbround= *RGB-rounding-specification*

specifies rounding factors used for the `paint=` variable and RGB values. The default is `rgbround=-240 1 1 1`. The first value is used to round the `paint=` variable. Specify a positive value to have the variable rounded to multiples of that value. Specify a negative value $-n$ to have a maximum of n colors. For the other three values, specify positive values. The last three are rounding factors used to round the values for the red, green, and blue component of the color (see `red=`). If more than 256 colors are generated, you will get the error that a color was not added because the color table is full. By default, when a value is missing, there is no rounding. Rounding the `paint=` variable is useful with contour plots.

Contour Options

Use these options with contour plots. For example, the grid for a contour plot might be generated as follows:

```
do x = -4 to 4 by 0.1;
   do y = -2 to 2 by 0.1;
      ... statements ...
      end;
   end;
```

Horizontally, there are $1+(4--4)/0.1 = 81$ horizontal points and $1+(2--2)/0.1 = 41$ vertical points, so you would specify `hnobs=81`, `vnobs=41`. By default, the square root of the number of contour type observations is used for both `hnobs=` and `vnobs=` (which assumes a square grid).

hnobs= *n*
specifies the number of horizontal observations in the grid for contour plots.

vnobs= *n*
specifies the number of vertical observations in the grid for contour plots.

Advanced Plot Control Options

You can use the these next options to add full SAS DATA step statements to strategic places in the macro, such as the PROC PLOT step, the end of the preprocessing, and last full DATA steps. These options do minor adjustments before the final plot is produced. These options allow very powerful customization of your results to an extent not typically found in procedures. However, they might require a fair amount of work and some trial and error to understand and get right.

adjust1= *SAS-statements*
specifies preprocessing SAS statements. The following variables are created in the preprocessing data set:

`__lsize` – label size
`__lfont` – label font
`__lcolor` – label color
`__ssize` – symbol size
`__sfont` – symbol font
`__scolor` – symbol color
`__stype` – symbol type
`__symbol` – symbol value
`__otype` – observation type

Use `adjust1=` to adjust these variables in the preprocessing data set. You must specify complete statements with semicolons, for example, as follows:

```
adjust1=%str(__lsize = 1.2; __lcolor = green;)

adjust1=%str(if z > 20 then do;
__scolor = 'green'; __lcolor = 'green'; end;)
```

adjust2= *SAS-statements*
specifies statements with PROC PLOT such as `format` statements. Just specify the full statement.

adjust3= *SAS-statements*
adjust4= *SAS-statements*
specify options to adjust the final Annotate data set. For example, in Swiss fonts, asterisks are not vertically centered when displayed, so `adjust3=` converts to use the SYMBOL function, so by default, `adjust3=%str(if text = '*' and function = 'LABEL' then do; style = ' '; text = 'star'; function = 'SYMBOL'; end;)`. The default for `adjust4=` is null, so you can use it to add new statements. If you add new variables to the data set, you must also include a `keep` statement.

The following illustrates using `adjust4=` to vertically display the y-axis label, like it would in PROC PLOT:

```
adjust4=%str(if angle = 90 then do; angle = 270; rotate = 90; keep rotate; end;)
```

The following option changes the size of title lines:

```
adjust4=%str(if index(comment, 'title') then size = 2;)
```

adjust5= *SAS-statements*

adds extra statements to the final DATA step that is used only for `datatype=function`. For example, to periodically mark the function with pluses, specify the following:

```
adjust5=%str(if mod(_n_,30) = 0 then do;
                  size=0.25; function = 'LABEL'; text = '+'; output; end;)
```

Other Options

The remaining options for the `%PlotIt` macro are as follows:

cirsegs= *n*

specifies a circle smoothness parameter used in determining the number of line segments in each circle. Smaller values create smoother circles. The `cirsegs=` value is approximately related to the length of the line segments that compose the circle. The default is `cirsegs=.1`.

cursegs= *n*

specifies the number of segments in a regression function curve. The default is `cursegs=200`.

debug= vars | dprint | notes | time | mprint

specifies values that control debugging output.

`dprint` – print intermediate data sets.

`mprint` – run with `options mprint`.

`notes` – do not specify `options nonotes` during most of the macro.

`time` – displays total macro run time, ignored with `options nostimer;`

`vars` – displays macro options and macro variables for debugging.

You should provide a list of names for more than one type of debugging. Example: `debug=vars dprint notes time mprint`. The default is `debug=time`.

hpos= *n*

specifies the number of horizontal positions in the graphics area.

hsize= *n*

specifies the horizontal graphics area size in `unit=` units. The default is the maximum size for the device. By default, when `options=nocenter` is not specified, `hsize=` affects the size of the plot but not the `hsize= goption`. When `options=nocenter` is specified, `hsize=` affects both the plot size and the `hsize= goption`. If you specify just the `hsize=` but not `vsize=`, the vertical size is scaled accordingly.

makefit= *n*

specifies the proportion of the graphics window to use. When the `makefit=` value is negative, the absolute value is used, and the final value might be changed if the macro thinks that part of the plot might extend over the edge. When a positive value is specified, it will not be changed by the macro. When nonnull, the macro uses GASK to determine the minimum and maximum graphics window sizes and makes sure the plot can fit in them. The macro uses `gopprint=` or `gopplot=` to determine the device. The default is `makefit=-0.95`.

nknots= *n*

specifies the PROC TRANSREG number of knots option for regression functions.

outward= none | *char*

specifies a quoted string for the PLOT statement `outward=` option. Normally, this option's value is constructed from the symbol that holds the place for vectors. Specify `outward=none` if you want to not have `outward=` specified for vectors. The `outward=` option is used to greatly increase the likelihood that labels from vectors are displayed outward—away from the origin.

ps= *n*

specifies the page size.

radii= *do-list*

specifies the radii of circles (in a DATA step do list). The unit corresponds to the horizontal axis variable. The `radii=` option can also specify a variable in the input data set when radii vary for each point. By default, no circles are drawn.

regopts= *options*

specifies the PROC TRANSREG options for curve fitting. Example: `regopts=nknots=10 evenly`.

regfun= *regression-function*

specifies the function for curve fitting. Possible values include:

`linear` – line

`spline` – nonlinear spline function, perhaps with knots

`mspline` – nonlinear but monotone spline function, perhaps with knots

`monotone` – nonlinear, monotone step function

See PROC TRANSREG documentation for more information

regprint= *regression-options*
specifies the PROC TRANSREG PROC statement options, typically display options such as:

`noprint` – no regression printed output

`short` – suppress iteration histories

`ss2` – regression results

To see the regression table, specify: `regprint=ss2 short`. The default is `regprint=noprint`.

unit= `in` | `cm`
specifies the `hsize=` and `vsize=` unit in inches or centimeters (in or cm). The default is `unit=in`.

vechead= *vector-head-size* specifies how to draw vector heads. For example, the default specification `vechead=0.1 0.025`, specifies a head consisting of two hypotenuses from triangles with sides 0.1 units long along the vector and 0.025 units on the side perpendicular to the vector. This is smaller than the default in previous releases.

vpos= *n*
specifies the number of vertical positions in the graphics area.

vsize= *n*
specifies the vertical graphics area size in `unit=` units. The default is the maximum size for the device. By default when `options=nocenter` is not specified, `vsize=` affects the size of the plot but not the `vsize= goption`. When `options=nocenter` is specified, `vsize=` affects both the plot size and the `vsize= goption`. If you specify just the `vsize=` but not `hsize=`, the horizontal size is scaled accordingly.

vtoh= *n*
specifies the PROC PLOT `vtoh=` option. The `vtoh=` option specifies the ratio of the vertical height of a typical character to the horizontal width. The default is `vtoh=2`. Do not specify values much different than 2, especially by default when you are using proportional fonts. There is no one-to-one correspondence between characters and cells and character widths vary, but characters tend to be approximately twice as high as they are wide. When you specify `vtoh=` values larger than 2, near-by labels might overlap, even when they do not collide in the printer plot. The macro uses this option to equate the axes so that a centimeter on one axis represents the same data range as a centimeter on the other axis. A null value can be specified, `vtoh=`, when you want the macro to just fill the window, like a typical GPLOT.

Smaller values give you more lines and smaller labels. The specification `vtoh=1.75` is a good alternative to `vtoh=2` when you need more lines to avoid collisions. The specification `vtoh=1.75` means 7 columns for each 4 rows between ticks (7 / 4 = 1.75). The `vtoh=2` specification means the plot will have 8 columns for each 4 rows between ticks. Note that PROC PLOT sometimes takes this value as a hint, not as a rigid specification so the actual value might be slightly different, particularly when a value other than 2.0 is specified. This is generally not a problem; the macro adjusts accordingly.

xmax= *n*
specifies the maximum horizontal size of the graphics area.

ymax= *n*
specifies the maximum vertical size of the graphics area.

Macro Error Messages

Usually, if you make a mistake in specifying macro options, the macro will display an informative message and quit. These macros go to great lengths to check their input and issue informative error messages. However, *complete* error checking like we have with procedures is impossible in macros, and on rare occasions you might get a cascade of less than helpful error messages.[*] In that case, you will have to check the input and hunt for errors. One of the more common user errors is not providing a comma between options. Sometimes, for harder errors, specifying `options mprint;` will help you locate the problem. You might get a listing with a *lot* of code, almost all of which you can ignore. Search for the error and look at the code that comes before the error for ideas about what went wrong. Once you think you know which option is involved, be sure to also check the option before and after in your macro invocation, because that might be where the problem really is.

The `%PHChoice` macro uses PROC TEMPLATE and ODS to create customized output tables. Typically, the instructions for this customization, created by PROC TEMPLATE, are stored in a file under the `sasuser` directory with a host dependent name. On some hosts, this name is `templat.sas7bitm`. On other hosts, the name is some variation of the name `templat`. Sometimes this file can be corrupted. When this happens, the macro will not run correctly, and you will see error messages including errors about invalid pages. The solution is to find the corrupt file under `sasuser` and delete it (using your ordinary operating system file deletion method). After that, this macros should run fine again. If you have run any other PROC TEMPLATE customizations, you will need to rerun them after deleting the file. For more information, see “Template Store” or “Item Store” in the SAS ODS documentation.

Sometimes, when you run the `%MktEx` macro, you might see the following error:

```
ERROR: The MKTDES macro ended abnormally.
```

This is typically caused by one or more PROC FACTEX steps failing to find the requested design. When this happens, the macro recovers and continues searching. The macro does not always know in advance if PROC FACTEX will succeed. The only way for it to find out is by trying. The macro suppresses the PROC FACTEX error messages along with most other notes and warnings that would ordinarily come out. This error can be ignored.

Other times, when you run the `%MktEx` macro, everything will seem to run fine in the entire job, but at the end of your SAS log, you will see the following message:

```
ERROR: Errors printed on page ....
```

Like before, this is typically caused by one or more PROC FACTEX steps failing and the macro recovering and continuing. While the macro can sometimes suppress error messages, SAS still knows that a procedure tried to display an error message and displays an error at the end of the log. This error can be ignored.

[*] If this happens, please contact Technical Support. See page 25 for more information. I will see if I can make the macros better handle that problem in the next release. Send all the code necessary to reproduce what you have done.