

%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 222 “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: 78, 82, 126, 154, 166, 176, 106, 109, and 95. Also see the examples of using this macro in the discrete choice chapter from pages 263 through 623. 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 69) of the design matrix. See pages 51 and 221, 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:

- $n \leq 448$ 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%
	<u>733</u>		<u>117,556</u>	

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 24⁸ 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		Parents	Designs	Runs		Parents	Designs	Runs	
1	1	4	2^3	20	26	36	2^{35}	4	4	52	2^{51}
1	1	6	2^{13^1}				$2^{27}3^1$				$2^{16}13^1$
1	2	8	2^{44^1}				$2^{20}3^2$				$2^{22}6^1$
1	1	9	3^4				$2^{18}3^16^1$				$4^{11}3^1$
1	1	10	2^{15^1}				$2^{16}9^1$	3	6	54	$2^{12}7^1$
4	4	12	2^{11}				$2^{13}3^26^1$				$3^{20}6^{19^1}$
			2^{43^1}				$2^{13}6^2$				$3^{18}18^1$
			2^{26^1}				$2^{10}3^86^1$	1	1	55	$5^{11}1^1$
			3^{14^1}				$2^{10}3^16^2$	5	8	56	$2^{52}4^1$
1	1	14	2^{17^1}				$2^{93}4^62$				$2^{37}4^{17^1}$
1	1	15	3^{15^1}				2^86^3				$2^{28}28^1$
2	7	16	2^{88^1}				$2^{43}16^3$				$2^{27}4^{11}14^1$
			4^5				$2^{33}96^1$				7^{18^1}
2	3	18	2^{19^1}				$2^{33}2^63$	1	1	57	$3^{11}9^1$
			3^{66^1}				$2^{23}5^62$	1	1	58	$2^{12}9^1$
4	4	20	2^{19}				$2^{21}8^1$	11	15	60	2^{59}
			2^{85^1}				$2^{13}3^63$				$2^{30}3^1$
			2^{210^1}				$3^{12}12^1$				$2^{24}6^1$
			4^{15^1}				3^76^3				$2^{23}5^1$
1	1	21	3^{17^1}				4^{19^1}				$2^{21}10^1$
1	1	22	2^{111^1}	1	1	38	$2^{11}9^1$				$2^{17}15^1$
5	8	24	$2^{20}4^1$	1	1	39	$3^{11}3^1$				$2^{15}6^{11}10^1$
			$2^{13}3^{14^1}$	5	8	40	$2^{36}4^1$				$2^{23}0^1$
			$2^{12}12^1$				$2^{25}4^{15^1}$				$3^{12}0^1$
			$2^{11}4^{16^1}$				$2^{20}20^1$				$4^{11}5^1$
			3^{18^1}				$2^{19}4^{110^1}$				$5^{11}2^1$
1	1	25	5^6				5^{18^1}	1	1	62	$2^{13}3^1$
1	1	26	2^{113^1}	3	4	42	$2^{12}21^1$	2	3	63	$3^{12}21^1$
1	2	27	3^{99^1}				$3^{11}4^1$				7^{19^1}
4	4	28	2^{27}				6^{17^1}	7	123	64	$2^{32}32^1$
			$2^{12}7^1$	4	4	44	2^{43}				$2^{54}4^{17}8^1$
			2^{214^1}				$2^{15}11^1$				$2^{54}10^84$
			4^{17^1}				$2^{22}2^1$				$4^{16}16^1$
3	4	30	2^{115^1}				4^{111^1}				$4^{14}8^3$
			3^{110^1}	2	3	45	3^915^1				4^78^6
			5^{16^1}				5^{19^1}				8^9
2	20	32	$2^{16}16^1$	1	1	46	$2^{12}3^1$	1	1	65	$5^{11}3^1$
			4^{88^1}	6	58	48	$2^{40}8^1$	3	4	66	$2^{13}3^1$
1	1	33	3^{111^1}				$2^{33}3^{18^1}$				$3^{12}2^1$
1	1	34	2^{117^1}				$2^{31}6^{18^1}$				$6^{11}1^1$
1	1	35	5^{17^1}				$2^{24}24^1$	4	4	68	2^{67}
							$3^{11}6^1$				$2^{18}17^1$
							$4^{12}12^1$				$2^{23}4^1$
				1	1	49	7^8				$4^{11}7^1$
				2	3	50	$2^{12}5^1$	1	1	69	$3^{12}3^1$
							$5^{10}10^1$	3	4	70	$2^{13}5^1$
				1	1	51	$3^{11}7^1$				$5^{11}4^1$
											$7^{11}0^1$

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

Parents	Designs	Runs		Parents	Designs	Runs		Parents	Designs	Runs	
13	15	140	2 ¹³⁹	1	1	164	2 ¹⁶³	3	13	225	3 ²⁷ 75 ¹
			2 ³⁸ 7 ¹	2	16	168	2 ¹⁶⁴ 4 ¹				5 ²⁰ 45 ¹
			2 ³⁶ 10 ¹				2 ⁸⁴ 84 ¹				15 ⁶
			2 ³⁴ 14 ¹	1	1	169	13 ¹⁴	1	1	228	2 ²²⁷
			2 ²⁷ 5 ¹ 7 ¹	1	1	171	3 ²⁸ 19 ¹	2	6	232	2 ²²⁸ 4 ¹
			2 ²⁵ 5 ¹ 14 ¹	1	1	172	2 ¹⁷¹				2 ¹¹⁶ 116 ¹
			2 ²² 35 ¹	1	2	175	5 ¹⁰ 35 ¹	1	5	234	3 ³⁰ 78 ¹
			2 ²¹ 7 ¹ 10 ¹	4	56	176	2 ¹⁶⁸ 8 ¹	1	1	236	2 ²³⁵
			2 ¹⁷ 10 ¹ 14 ¹				2 ¹⁶⁶ 4 ³	6	302	240	2 ²³² 8 ¹
			2 ²⁷ 0 ¹				2 ⁸⁸ 88 ¹				2 ²³⁰ 4 ³
			4 ¹ 35 ¹				4 ¹² 44 ¹				2 ²⁰⁵ 5 ¹ 24 ¹
			5 ¹ 28 ¹	3	24	180	2 ¹⁷⁹				2 ¹⁹⁹ 10 ¹ 24 ¹
			7 ¹ 20 ¹				3 ³⁰ 60 ¹				2 ¹²⁰ 120 ¹
1	1	141	3 ¹ 47 ¹				6 ² 30 ¹				4 ²⁰ 60 ¹
1	1	142	2 ¹ 71 ¹	2	6	184	2 ¹⁸⁰ 4 ¹	1	2	242	11 ²² 22 ¹
1	1	143	11 ¹ 13 ¹				2 ⁹² 92 ¹	2	58	243	3 ⁸¹ 81 ¹
16	1,241	144	2 ¹³⁶ 8 ¹	1	1	188	2 ¹⁸⁷				9 ²⁷ 27 ¹
			2 ¹¹⁷ 8 ¹ 9 ¹	1	4	189	3 ³⁶ 63 ¹	1	1	244	2 ²⁴³
			2 ¹¹³ 3 ¹ 24 ¹	4	726	192	2 ¹⁶⁰ 32 ¹	1	2	245	7 ¹⁰ 35 ¹
			2 ¹¹¹ 6 ¹ 24 ¹				2 ⁹⁶ 96 ¹	2	6	248	2 ²⁴⁴ 4 ¹
			2 ¹⁰³ 8 ¹ 18 ¹				4 ⁴⁸ 48 ¹				2 ¹²⁴ 124 ¹
			2 ⁷⁶ 3 ¹ 26 ⁴ 8 ¹				8 ⁸ 24 ¹	1	4	250	5 ⁵⁰ 50 ¹
			2 ⁷⁶ 3 ⁷ 4 ¹ 6 ⁵ 12 ¹	3	11	196	2 ¹⁹⁵	3	23	252	2 ²⁵¹
			2 ⁷⁵ 3 ³ 4 ¹ 6 ⁶ 12 ¹				7 ²⁸ 28 ¹				3 ⁴² 84 ¹
			2 ⁷⁴ 3 ⁴ 6 ⁶ 8 ¹				14 ⁵				6 ² 42 ¹
			2 ⁷² 72 ¹	1	5	198	3 ³⁰ 66 ¹	2	6,101	256	8 ³² 32 ¹
			2 ⁴⁴ 3 ¹¹ 12 ²	4	42	200	2 ¹⁹⁶ 4 ¹				16 ¹⁷
			2 ¹⁶ 3 ³ 6 ⁶ 24 ¹				2 ¹⁰⁰ 100 ¹	1	1	260	2 ²⁵⁹
			3 ⁴⁸ 48 ¹				5 ²⁰ 40 ¹	1	2	261	3 ²⁷ 87 ¹
			4 ³⁶ 36 ¹				10 ⁵ 20 ¹	2	16	264	2 ²⁶⁰ 4 ¹
			4 ¹¹ 12 ²	1	1	204	2 ²⁰³				2 ¹³² 132 ¹
			12 ⁷	1	1	207	3 ²⁵ 23 ¹	1	1	268	2 ²⁶⁷
1	2	147	7 ⁹ 21 ¹	4	72	208	2 ²⁰⁰ 8 ¹	1	11	270	3 ⁹⁰ 90 ¹
1	1	148	2 ¹⁴⁷				2 ¹⁹⁸ 4 ³	4	136	272	2 ²⁶⁴ 8 ¹
1	5	150	5 ¹¹ 30 ¹				2 ¹⁰⁴ 104 ¹				2 ²⁶² 4 ³
2	6	152	2 ¹⁴⁸ 4 ¹				4 ¹⁶ 52 ¹				2 ¹³⁶ 136 ¹
			2 ⁷⁶ 76 ¹	1	1	212	2 ²¹¹				4 ³² 68 ¹
1	1	153	3 ²⁵ 17 ¹	6	258	216	2 ²¹² 4 ¹	1	2	275	5 ¹¹ 55 ¹
1	1	156	2 ¹⁵⁵				2 ¹⁰⁸ 108 ¹	1	1	276	2 ²⁷⁵
6	110	160	2 ¹⁴⁴ 16 ¹				2 ¹¹³ 77 ¹ 12 ¹ 18 ¹	1	2	279	3 ³⁰ 93 ¹
			2 ¹³⁸ 4 ⁷				3 ⁷² 72 ¹	2	17	280	2 ²⁷⁶ 4 ¹
			2 ¹³³ 5 ¹ 16 ¹				3 ⁶⁶ 6 ⁵ 12 ¹ 18 ¹				2 ¹⁴⁰ 140 ¹
			2 ¹²⁷ 10 ¹ 16 ¹				6 ⁷ 36 ¹	1	1	284	2 ²⁸³
			2 ⁸⁰ 80 ¹	1	1	220	2 ²¹⁹				
			4 ¹⁶ 40 ¹	5	351	224	2 ²⁰⁸ 16 ¹				
3	61	162	3 ⁶⁵ 6 ¹ 27 ¹				2 ¹⁹³ 7 ¹ 16 ¹				
			3 ⁵⁴ 54 ¹				2 ¹⁸³ 14 ¹ 16 ¹				
			9 ¹⁸ 18 ¹				2 ¹¹² 112 ¹				
							4 ⁵⁶ 56 ¹				

Parents	Designs	Runs		Parents	Designs	Runs		Parents	Designs	Runs	
8	3,201	288	$2^{272}16^1$	1	5	342	$3^{30}114^1$	1	2	408	$2^{404}4^1$
			$2^{253}9^{116}1$	1	2	343	$7^{49}49^1$	1	1	412	2^{411}
			$2^{239}16^{118}1$	1	2	344	$2^{340}4^1$	1	5	414	$3^{48}138^1$
			$2^{144}144^1$	1	1	348	2^{347}	4	299	416	$2^{400}16^1$
			$3^{96}96^1$	1	5	350	$5^{20}70^1$				$2^{394}4^7$
			$4^{36}72^1$	1	4	351	$3^{39}117^1$				$2^{312}104^1$
			$6^{10}48^1$	4	203	352	$2^{336}16^1$				$4^{48}104^1$
			12^624^1				$2^{330}4^7$	1	1	420	2^{419}
1	1	289	17^{18}				$2^{264}88^1$	1	2	423	$3^{30}141^1$
1	1	292	2^{291}				$4^{32}88^1$	1	2	424	$2^{420}4^1$
1	5	294	$7^{18}42^1$	1	1	356	2^{355}	1	2	425	$5^{20}85^1$
1	2	296	$2^{292}4^1$	3	133	360	$2^{356}4^1$	1	1	428	2^{427}
1	4	297	$3^{39}99^1$				$3^{48}120^1$	7	10,839	432	$2^{424}8^1$
3	24	300	2^{299}				6^860^1				$2^{389}9^{124}1$
			$5^{20}60^1$	1	1	361	19^{20}				$2^{375}18^{124}1$
			10^230^1	1	2	363	$11^{11}33^1$				$3^{144}144^1$
4	70	304	$2^{296}8^1$	1	1	364	2^{363}				$4^{108}108^1$
			$2^{294}4^3$	4	150	368	$2^{360}8^1$				$6^{12}72^1$
			$2^{228}76^1$				$2^{358}4^3$				12^636^1
			$4^{16}76^1$				$2^{276}92^1$	1	1	436	2^{435}
1	5	306	$3^{48}102^1$				$4^{36}92^1$	1	2	440	$2^{436}4^1$
1	1	308	2^{307}	1	2	369	$3^{30}123^1$	3	12	441	$3^{42}7^21^1$
1	2	312	$2^{308}4^1$	1	1	372	2^{371}				$7^{14}63^1$
1	5	315	$3^{29}105^1$	1	4	375	$5^{40}75^1$				21^7
1	1	316	2^{315}	1	2	376	$2^{372}4^1$	1	1	444	2^{443}
5	725	320	$2^{288}32^1$	1	11	378	$3^{72}126^1$	4	8,598	448	$2^{416}32^1$
			$2^{274}4^{15}$	1	1	380	2^{379}				$2^{336}112^1$
			$2^{240}80^1$	3	3,252	384	$2^{320}64^1$				$4^{56}112^1$
			$4^{40}80^1$				$4^{96}96^1$				$8^{56}56^1$
			$8^{10}40^1$				$8^{16}48^1$	3	35	450	$3^{150}5^{11}30^1$
6	974	324	2^{323}	1	2	387	$3^{48}129^1$				$5^{90}90^1$
			$3^{143}12^{127}1$	1	1	388	2^{387}				15^530^1
			$3^{108}108^1$	4	28	392	$2^{388}4^1$	1	2	456	$2^{452}4^1$
			6^254^1				$2^{196}7^{28}28^1$	1	2	459	$3^{72}9^{17}1^1$
			$9^{36}36^1$				$7^{28}56^1$	1	1	460	2^{459}
			18^3				14^528^1	4	150	464	$2^{456}8^1$
1	2	325	$5^{20}65^1$	3	23	396	2^{395}				$2^{454}4^3$
1	2	328	$2^{324}4^1$				$3^{132}132^1$				$2^{348}116^1$
1	1	332	2^{331}				6^266^1				$4^{36}116^1$
1	2	333	$3^{36}111^1$	7	912	400	$2^{392}8^1$	3	17	468	2^{467}
6	487	336	$2^{328}8^1$				$2^{390}4^3$				$3^{49}52^1$
			$2^{326}4^3$				$2^{300}100^1$				6^278^1
			$2^{297}7^{124}1$				$4^{36}100^1$	1	2	472	$2^{468}4^1$
			$2^{287}14^{124}1$				$5^{80}80^1$	1	2	475	$5^{20}95^1$
			$2^{252}84^1$				10^640^1	1	1	477	$3^{37}53^1$
			$4^{36}84^1$				20^5	4	1,210	480	$2^{464}16^1$
1	2	338	$13^{26}26^1$	1	1	404	2^{403}				$2^{458}4^7$
1	1	340	2^{339}	2	60	405	$3^{81}135^1$				$2^{360}120^1$
							$9^{18}45^1$				$4^{56}120^1$

Parents	Designs	Runs		Parents	Designs	Runs		Parents	Designs	Runs	
3	8	484	2^{483}	1	2	495	$3^{42}5^133^1$	3	111	504	$2^{500}4^1$
			$11^{44}44^1$	4	78	496	$2^{488}8^1$				$2^{84}3^{84}84^1$
			22^3				$2^{486}4^3$				6^884^1
1	277	486	$9^{54}54^1$				$2^{372}124^1$	2	73,992	512	$8^{64}64^1$
1	2	488	$2^{484}4^1$				$4^{18}124^1$				$16^{32}32^1$
1	5	490	$7^{18}70^1$	3	29	500	2^{499}	1	2	513	$3^{81}9^119^1$
1	1	492	2^{491}				$5^{100}100^1$	733	117,561		
							$10^{25}50^1$				

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 958).

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 = '123456789abcdefghijklmnopqrstuvwxyza+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z';
    s2 = 'ABCDEFGHIJKLMNopqrstuvwxyz123456789+*';
    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:

```

%mktx(3 ** 4, n=3 * 3)
%latin(3)
%latin(4)

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

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

%mktx(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:

```

%mktx(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 $p = 3$:

```

%mktx(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 $p = 12$:

```

%mktx(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).

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

Next, the %MktEx macro 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 of 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 of 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 of 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.

%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					
Design	Row,Col	Current D-Efficiency	Best 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

Design	Row,Col	Current D-Efficiency	Best D-Efficiency	Notes
0	Initial	98.8933	98.8933	Ini
1	Start	80.4296		Tab,Ran
1	End	98.8567		
.				
.				
.				

Design Refinement History

Design	Row,Col	Current D-Efficiency	Best 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
anneal = <i>n1</i> < <i>n2</i> < <i>n3</i> >>	starting probability for annealing
annealfun = <i>function</i>	annealing probability function
anniter = <i>n1</i> < <i>n2</i> < <i>n3</i> >>	first annealing iteration
balance = <i>n</i>	maximum level-frequency range
big = <i>n</i> < <i>choose</i> >	size of big full-factorial design
canditer = <i>n1</i> < <i>n2</i> >	iterations for OPTEx designs
cat = <i>SAS-data-set</i>	input design catalog
detfuzz = <i>n</i>	determinants change increment
examine =I V	matrices that you want to examine
exchange = <i>n</i>	number of factors to exchange
fixed = <i>variable</i>	indicates runs that are fixed
holdouts = <i>n</i>	adds holdout observations
imlopts = <i>options</i>	IML PROC statement options
init = <i>SAS-data-set</i>	initial (input) experimental design
interact = <i>interaction-list</i>	interaction terms
iter = <i>n1</i> < <i>n2</i> < <i>n3</i> >>	maximum number of iterations
levels = <i>value</i>	assigning final factor levels
list	list of the numbers of factor levels

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

Option	Description
maxdesigns= <i>n</i>	maximum number of designs to make
maxiter= <i>n1</i> < <i>n2</i> < <i>n3</i> >>	maximum number of iterations
maxstages= <i>n</i>	maximum number of algorithm stages
maxtime= <i>n1</i> < <i>n2</i> < <i>n3</i> >>	approximate maximum run time
mintry= <i>n</i>	minimum number of rows to process
mutate= <i>n1</i> < <i>n2</i> < <i>n3</i> >>	mutation probability
mutiter= <i>n1</i> < <i>n2</i> < <i>n3</i> >>	first iteration to consider mutating
n= <i>n</i>	number of runs in the design
* options= <i>options-list</i>	binary options
optiter= <i>n1</i> < <i>n2</i> >	OPTEX iterations
order= <i>value</i>	coordinate exchange column order
out= <i>SAS-data-set</i>	output experimental design
outall= <i>SAS-data-set</i>	output data set with all designs
* outeff= <i>SAS-data-set</i>	output data set with final efficiency
outr= <i>SAS-data-set</i>	randomized output experimental design
partial= <i>n</i>	partial-profile design
repeat= <i>n1 n2 n3</i>	times to iterate on a row
reslist= <i>list</i>	constant matrix list
resmac= <i>macro-name</i>	constant matrix creation macro
restrictions= <i>macro-name</i>	restrictions macro
ridge= <i>n</i>	ridging factor
seed= <i>n</i>	random number seed
stopearly= <i>n</i>	the macro can stop early
tabiter= <i>n1</i> < <i>n2</i> >	design table initialization iterations
tabsize= <i>n</i>	orthogonal array size
target= <i>n</i>	target efficiency criterion
unbalanced= <i>n1</i> < <i>n2</i> >	unbalance initial design iterations

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

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 931).

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

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.

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 $\text{interact}=@2$ 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 $\text{interact}=@2$, 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. $\text{interact}=x1$ | $x2$ | $x3$ | $x4$ | $x5@2$) 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 $\text{mintry}=5 * n$. 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 $\text{mintry}=mt$ 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 $\text{mintry}=n$ 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(36,2¹¹,3¹²)**. 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 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 931). 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 945 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 things 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 = j2 = 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
<code>=</code>	equals	EQ
<code>^ =</code> or <code>^ =</code>	not equals	NE
<code><</code>	less than	LT
<code><=</code>	less than or equal to	LE
<code>></code>	greater than	GT
<code>>=</code>	greater than or equal to	GE
<code>&</code>	and	AND
<code> </code>	or	OR
<code>^</code> or <code>^</code>	not	NOT
<code>a <= b & b <= c</code>	range check	<code>a <= b <= c</code>

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 945 for more information about restrictions. See pages 431 and 564 for examples of restrictions. There are many examples of restrictions in the partial-profile examples starting on page 555.

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,          /* 0A catalogue 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;
%mktx(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;
%mktx(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 these 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: $(1\ 2\ 3\ 4) \Rightarrow (1\ 2\ 3\ 3)$ or a two-level factor from a three-level factor: $(1\ 2\ 3) \Rightarrow (1\ 2\ 2)$. 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= $n < \text{choose} >$

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 $1\ 2 \rightarrow -0.5\ 0.5$, $1\ 2\ 3 \rightarrow -1\ 0\ 1$.

i – centered and scaled to integers. $1\ 2 \rightarrow -1\ 1$, $1\ 2\ 3 \rightarrow -1\ 0\ 1$.

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 $r, r+1, r+2, \dots, m, 1, 2, \dots, r-1$ (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 . .
;

%mktext(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 $x_2 \times x_3$ 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 x_2 , x_3 , and x_4 , 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;

%mktext(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 `xn` 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;

%mktx(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

Design	Row,Col	Current D-Efficiency	Best 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;

%mkrtex(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

Design	Row,Col	Current D-Efficiency	Best 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

Design Number	D-Efficiency	A-Efficiency	G-Efficiency	Average Prediction Standard 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 design.

The first four choice sets are as follows:

```

-
A
S 1          x 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
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 i th 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 (j_1 , j_2 , j_3), you probably do not need to worry about this step. If you use j_1 , j_2 , or j_3 , you will need to either not execute your restrictions in this step or conditionally execute some assignment statements that set up j_1 , j_2 , and j_3 appropriately. Ordinarily for this step, `xmat` contains the candidate design, `x` contains the i th row, $j_1 = 0$; $j_2 = 0$; $j_3 = 0$; `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 i th row, j_1 , j_2 , and j_3 typically contain the column indices, i is the row number, and `try` is the zero-based cumulative row number. With `exchange=1`, j_1 exists, with `exchange=2`, j_1 and j_2 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 j_1 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 $j_1 = 1$. 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 i th row, $j_1 = 1$; $j_2 = 1$; $j_3 = 1$; `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.