EXPERIMENTAL STUDIES:

RANDOMIZATION AND COUNTERBALANCING WITH SPSS

In contrast to nonexperimental studies, true experiments have the ***potential*** to allow strong conclusions about causal relations between independent and dependent variables. If the variable manipulated by the experimenter is in fact the only variable that differs across the levels of the independent variable, then differences on the dependent variable can be attributed to the treatment or to chance (Type I error), and nondifferences can be attributed to the failure of the manipulated variable to affect the dependent measures or to chance (Type II error). Ensuring that the nominal independent variable is in fact the only difference between groups is fundamental to sound experimental design, but can be extremely challenging and difficult to realize in practice. Inadequate experimental designs result in independent variables that are correlated with other confounding variables, and this can make an experimental study a poor basis for drawing causal inferences.

A simple control vs. treatment group experiment, for example, would be flawed if any systematic factor (e.g., gender, age, intelligence, enthusiasm, desire to please the experimenter, time of day) differed for the two groups of subjects. Any such factors confounded with the treatment variable could produce an illusory treatment effect or mask a true treatment effect, just as correlated predictors similarly influence criterion variables in nonexperimental studies and render conclusions about causality difficult or impossible. Similar concerns arise for between-subject experiments with multiple levels (i.e., $k > 2$) or for factorial designs in which subjects must be assigned to combinations of treatments.

A classic example of confounding in an experimental study occurred many years ago in an effort to assess the benefits of supplementary milk for disadvantaged children. Children given supplemental milk did not differ from children not given supplemental

milk, but it turned out to have occurred because some of the nurses who were responsible for assigning children to the two conditions (supposedly by random assignment) assigned children nonrandomly, with the children receiving supplemental milk being more disadvantaged when they entered the study. The nondifference at the end of the experiment reflected the expected benefits of supplemental milk in that without the supplement disadvantaged children would have been worse off on the dependent variable(s).

Confounding also threatens the validity of within-subject experiments in which subjects are exposed to multiple conditions. To illustrate, consider a study of memory for words presented at fast or slow rates of presentation. Each subject is to study and recall several lists, some at fast rates and some at slow rates. Unless the study is carefully designed, the fast lists could differ in various ways from the slow lists (e.g., occur earlier or later in the session, contain easier or more difficult words). Any such correlated variables could produce illusory differences or mask the true effects of rate.

To reduce the probability of confounding variables and thereby strengthen causal conclusions, researchers use two related techniques: randomization and counterbalancing. We first consider randomization.

RANDOMIZATION

Randomization serves many functions in experimental studies: assigning different subjects to different levels of treatment factors or to different combinations of conditions, determining the order in which within-subjects factors are administered to subjects, selecting and ordering materials for participants (e.g., the order of words in a recall study), and so on. At the heart of all procedures is the idea of a random permutation.

Uses of Randomization

One common use of randomization is to assign subjects to experimental conditions in a between-subjects or independent groups design. This task can also be viewed as assigning treatment conditions to subjects. Imagine a simple study with an

independent variable having two levels (Treatment vs. Control). If sufficient numbers of subjects were assigned randomly to the two conditions, then the researchers could be reasonably confident that the composition of the two groups was probably equivalent with respect to a host of individual difference variables that should get allocated randomly to the two conditions (e.g., age, gender, intelligence, anxiety, and numerous other factors that could influence the dependent variable). Such equivalence strengthens the assumption that any difference in performance between the treatment and control groups arises from differences in how the experimenter treated the subjects (i.e., differences in the independent variable) and not from pre-existing differences between subjects in the two groups or other accidentally related confounding variables. Randomization similarly improves the odds that nondifferences between the groups did not occur because some confounded variable masked the treatment effect.

A second use of randomization is to randomize the order in which subjects are exposed to the different levels of a within-subject or repeated-measures manipulation. Imagine a memory study in which subjects are going to learn 5 lists of words shown at 5 different rates of presentation (1, 2, 3, 4, or 5 seconds per word). Experimenters must decide what order the five conditions will occur for each subject. One effective way to determine the order is to randomize the five conditions separately for each subject in the experiment. If researchers wanted to ensure that each condition occurred equally often first, second, and so on across subjects, then additional constraints (i.e., counterbalancing) could be used in conjunction with randomization.

Randomization can also be used to guard against secondary variables being confounded with the primary variable of interest to researchers. Researchers designing a between-subjects rate-of-presentation study, for example, might use five different lists so that the results can be generalized beyond a single set of words. Each subject would need to be randomly assigned to one of the five treatment conditions (presentation rates) and to one of the five word lists. The researchers may not be particularly interested in

differences between the lists (i.e., lists is a secondary variable), but should nonetheless ensure that each presentation rate occurred equally often with each of the lists and that each subject received a list chosen randomly (or by one of the systematic randomization procedures discussed later).

In some studies, randomization is primarily for "housekeeping." If lists of words are going to be shown to subjects, for example, researchers would probably not want to use the same order of the words for every subject. If the same order of words was used, then serial position of items would be confounded with word (i.e., only one specific word would have occurred at each position). Positions with easier words would produce higher levels of recall than positions with more difficult words. Using different orders reduces and perhaps eliminates this concern. Different orders could be determined by randomizing the words. The orders may not even be analyzed, but any conclusions about the variables examined (e.g., serial position) would no longer be tied to a specific order of the words.

Although the preceding examples involve a memory study, randomization is essential in all areas of psychology. Social psychologists who want to compare the effects of different attitude change procedures use randomization to control extraneous variables that could confound their conclusions. In between-subject designs, subjects would be assigned randomly to different treatment conditions (e.g., low, medium, or high fear conditions). In within-subject designs, the order of the conditions may be critical and would be randomized (with or without counterbalancing), as should different materials used to strengthen the generality of the findings (e.g., different attitudinal topics). Housekeeping considerations might include such things as the order in which subjects complete various rating scales (e.g., anxiety levels, intention to change,...) or the specific content used with a particular subject in an attitude change study (e.g., quitting smoking, using condoms).

Whatever the purpose being served by randomization, the underlying procedure

will depend in essence on the selection of random permutations of digits representing different aspects of the study.

Random Permutations

A permutation is a sequence or ordering of sequential digits (or other symbols), with each digit representing some aspect of the research design (e.g., an experimental condition, a specific representative of words or other kinds of material). Each of the following sequences of the numbers from 1 to 4 is a permutation: 1234, 3124, and 2413.

For k digits, there are exactly n! (n factorial) permutations, where n! = n x (n - 1) x (n - 2) ... x 2 x 1. To illustrate for k = 3, there are 3! = 3 x 2 x 1 = 6 permutations: 123, 132, 213, 231, 312, 321. One way to conceptualize this is: there are k ways to choose the first digit (1, 2, ..., k); given the first digit has been used, then there are k - 1 ways to choose the second digit, and so on until the last (k^{th}) digit. Such selection involves random sampling of the digits from 1 to k without replacement (i.e., once a digit is removed, it is not returned to the sample space).

There are several strategies for actually enumerating the possible sequences. One way is to draw (or imagine) a tree diagram with k branching points. In the example of k = 3 there are 3 paths from the origin representing the digits from 1 to 3. From each of those branches, there are two possibilities (or branches). There is a single branch for the 3rd step. This conceptualization is illustrated in Box 1.

Step 1	Step 2	Step 3	Permutation
1 <	2	3	123
	3	2	132
2 <	1	3	213
	3	1	231
3 <	1	2	312
	2	1	321

Box 1. Tree Diagram for Permutations.

It is also possible to perform the generation in Box 1 without drawing or imagining a tree. Begin with the ordered sequence of k digits (123), switch the 2nd and 3rd values (132). Then form the sequence beginning with the 2nd digit and the other

digits ordered (213), now switch the 2nd and 3rd values again (231). Then form the sequence beginning with the 3rd digit and the other digitss ordered (312), and now again switch the 2nd and 3rd digits (321). Here is this generation sequence for the 4! = 24 permutations for k = 4: 1234, 1243, 1324, 1342, 1423, 1432, 2134, 2143, 2314, 2341, 2413, 2431, 3124, 3142, 3214, 3241, 3412, 3421, 4123, 4132, 4213, 4231, 4312, 4321.

It should be obvious from these few examples that the number of permutations for k objects increases quite dramatically as k increases. Specifically, 2! = 2, 3! = 6, 4! = 24, 5! = 120, 6! = 720, 7! = 5040, and so on. Complete enumeration of the sequences becomes increasingly difficult as k increases, a point that has important implications when we examine the issue of counterbalancing later in the chapter.

A random sequence or permutation is a sequence chosen in such a way that each of the k! possible permutations has an equal chance of being chosen. Before computers became readily available, researchers used various manual or physical procedures to perform randomization. Published tables of random numbers can be used to decide about the orders of treatments and other considerations mentioned above. When only two conditions are involved, a coin-toss is reasonably random. Tossing a regular 6-sided die or various-sided Dungeons & Dragons dice is another effective way to randomize under certain conditions. It is also possible to write condition codes on slips of paper or chips, shuffle or mix them well, and use the resulting order to determine the random sequences to use. Although physical procedures are still useful in some circumstances, computer-aided methods now do much of the randomization used in experimental studies. Initially such methods are more difficult to understand than manual procedures, but computerized randomization is simpler in the long run and permits more sophisticated control.

The use of computers to perform various randomization operations requires that contemporary researchers be familiar with one or more computer methods. Sometimes special programs are written in a programming language (e.g., Basic) to perform

randomization. Randomization can also be performed using general purpose programs, such as statistical packages and data base languages. In this chapter, we examine several alternative procedures for randomization using Basic, SPSS, and SAS (another statistical package). In the case of SPSS and SAS programs, we will only show the parts of the program and output actually relevant to the randomization procedure. Other commands may be necessary to run the commands and obtain output.

RANDOMIZATION WITH SPSS

Statistical packages such as SPSS can be used to perform various types of randomization, as can spreadsheet and database programs. Here we examine how SPSS can be used to perform some of the randomization procedures described above.

Simple Randomization with SPSS

The technique used to perform randomization is SPSS is quite simple. We generate an ordered sequence of the k digits from 1 to k (or modifications of this in later examples) and then sort the sequence randomly using the capacity of SPSS to generate random numbers and to sort cases. Box 2 shows the initial commands to generate the sequence of digits from 1 to k (k = #obs in the program). The heart of the procedure is the middle three lines (from *loop o = 1 to #obs* to *end loop*). This loop is performed #obs times (6 times in our example), with o being equal to 1, 2, 3, 4, 5, 6 each successive time. The 6 was determined in the third line were #obs was set equal to 6. A variable beginning with # in SPSS is a hidden variable that does not actually become part of the data file. Note in the listing that no #obs variable was created for the cases. The data file contains the sequence from 1 to 6 (ordered at the present time). If we modified

```
*Generate ordered permutation.
INPUT PROGRAM.
COMPUTE #obs = 6.
LOOP o = 1 TO #obs.
END CASE.
END LOOP.
END FILE.
END INPUT PROGRAM.
LIST.

      O
 1.0000
 2.0000
 3.0000
 4.0000
 5.0000
 6.0000
```

Box 2. Generation of Sequence in SPSS.

the third line to *compute #obs = 100*, then the file would contain 100 cases, with their respective scores being 1, 2, 3, and so on up to 100.

Now we need to scramble the sequential cases so that we get one of the 720 (i.e., 6!) random permutations of the digits from 1 to 6. Box 3 shows how to do exactly that (new commands are italicized). The first step is to add *COMPUTE rand = UNIFORM(1)* to the commands. This creates a new variable RAND that contains pseudorandom values generated by SPSS. *UNIFORM(1)* generates random numbers between 0 and 1, with the specific values depending on the *SET SEED* = command added at the top of the file. The random values are shown in the first listing. The values are called pseudo-random because computers generate so-called random numbers in a non-random manner, but with virtually all of the properties of random numbers. The second step is to sort the file on the random values, which is what *SORT CASES BY rand* does. The second listing shows that the cases have been sorted on rand, which produces a random permutation of o (i.e., the digits from 1 to 6). This is a random permutation selected randomly out of the 720 possible permutations.

```
*Generate random permutation.
*Use seed=RANDOM in practice.
SET SEED = 1234567.
INPUT PROGRAM.
COMP #obs = 6.
LOOP o = 1 TO #obs.
END CASE.
END LOOP.
END FILE.
END INPUT PROGRAM.
COMPUTE rand = UNIFORM(1).
FORMAT rand (F6.4).
LIST.

      O   RAND
 1.0000  .6849
 2.0000  .5154
 3.0000  .8941
 4.0000  .0074
 5.0000  .1297
 6.0000  .8299

SORT CASES BY rand.
LIST.

      O   RAND
 4.0000  .0074
 5.0000  .1297
 2.0000  .5154
 1.0000  .6849
 6.0000  .8299
 3.0000  .8941
```

Box 3. Random Permutation.

This is the basis for much of what follows below, and would indeed be enough program to do many aspects of randomization. To assign multiple participants to four different conditions, for example, we could repeatedly run the program in Box 3 with

#obs = 4 (and *SET SEED = RANDOM*). As each subject appeared for the experiment, the experimenter would assign the participant to the next condition in the random sequence. As each random sequence was completed, a new sequence would be started. This form of random assignment would produce block randomization, and is discussed below.

Or imagine that the researcher had to randomize the order of 48 words for each of 12 participants. The program in Box 3 could be run 12 times with #obs = 48. Each of the numbers would denote a word (or, equivalently, the position in which a word should occur). Rather than run the program multiple times, however, the program can be modified to accommodate multiple repetitions.

Random Permutations with Replication

In the preceding example, each digit occurred only once. Most experiments would require that condition codes be produced a multiple number of times, depending on the number of participants desired in each condition. To assign 24 subjects to 6 conditions, for example, we would require 4 1s, 4 2s, and so on, ideally in a random (or constrained random) order. Box 4 shows one way to achieve this in SPSS. Now #obs represents the total

```
*Randomization with replication.
SET SEED = 1234567.
INPUT PROGRAM.
COMP #obs = 12.
LOOP o = 1 TO #obs.
END CASE.
END LOOP.
END FILE.
END INPUT PROGRAM.
COMPUTE cond = MOD(o - 1, 6) + 1.
COMPUTE rand = UNIFORM(1).
FORMAT rand (F6.4).
*  # above is number of conditions.
LIST.

      O    COND   RAND
 1.0000  1.0000  .6849
 2.0000  2.0000  .5154
 3.0000  3.0000  .8941
 4.0000  4.0000  .0074
 5.0000  5.0000  .1297
 6.0000  6.0000  .8299
 7.0000  1.0000  .1689
 8.0000  2.0000  .0552
 9.0000  3.0000  .0616
10.0000  4.0000  .8679
11.0000  5.0000  .6697
12.0000  6.0000  .2581

SORT CASES by rand.
LIST.

      O    COND   RAND
 4.0000  4.0000  .0074
 8.0000  2.0000  .0552
 9.0000  3.0000  .0616
 5.0000  5.0000  .1297
 7.0000  1.0000  .1689
12.0000  6.0000  .2581
 2.0000  2.0000  .5154
11.0000  5.0000  .6697
 1.0000  1.0000  .6849
 6.0000  6.0000  .8299
10.0000  4.0000  .8679
 3.0000  3.0000  .8941
```

Box 4. Randomization with Replications.

number of observations that we want to generate (12 in the example in Box 4). The actual condition code (COND) is generated by *COMPUTE cond = MOD(o-1, 6) + 1*, where 6 now represents the number of conditions and *MOD* represents the modulus operator in SPSS. The modulus of a number is the remainder when one number (o - 1 in our example) is divided by another number (6); for example, 8 - 1 modulus 6 = 1 (to which 1 is added to produce COND = 2. The first listing shows that this line produces the numbers from 1 to 6 in sequence, then repeats the numbers in sequence again. In fact, this line would repeatedly do this no matter how large we made #obs. Hence, we have 2 1s, 2 2s, and so one, which is what we wanted. Sorting on RAND produces the desired random sequence. If we changed #obs to 24, then we would have each code four times.

Box 5 shows a second way in which this same end could have been achieved. Rather than generating the sequence 1 to 6 twice, the *COMPUTE cond = TRUNC((o-1)/2)+1* line produces 2 1s, 2 2s, and so on, as shown in the first listing. Sorting the cases on rand again produces the desired outcome of a random order containing each of our 6 condition codes twice. If we changed #obs to 24 and the divisor for

```
*Randomization with replication #2.
SET SEED = 1234567.
INPUT PROGRAM.
COMP #obs = 12.
LOOP o = 1 TO #obs.
END CASE.
END LOOP.
END FILE.
END INPUT PROGRAM.
COMPUTE rand = UNIFORM(1).
FORMAT rand (F6.4).
COMPUTE cond = TRUNC((O-1)/2) + 1.
* 2 above=number of replications.
LIST.

      O    COND   RAND
 1.000  1.00000  .6849
 2.000  1.00000  .5154
 3.000  2.00000  .8941
 4.000  2.00000  .0074
 5.000  3.00000  .1297
 6.000  3.00000  .8299
 7.000  4.00000  .1689
 8.000  4.00000  .0552
 9.000  5.00000  .0616
10.000  5.00000  .8679
11.000  6.00000  .6697
12.000  6.00000  .2581

SORT CASES BY rand.
LIST.

      O    COND   RAND
 4.0000  2.0000  .0074
 8.0000  4.0000  .0552
 9.0000  5.0000  .0616
 5.0000  3.0000  .1297
 7.0000  4.0000  .1689
12.0000  6.0000  .2581
 2.0000  1.0000  .5154
11.0000  6.0000  .6697
 1.0000  1.0000  .6849
 6.0000  3.0000  .8299
10.0000  5.0000  .8679
 3.0000  2.0000  .8941
```

Box 5. Randomization with Replication #2.

computing cond to 4, then we would have each condition code four times, randomly scrambled.

Note an important difference between the actual sequences generated in Boxes 4 and 5, a difference that is solely due to chance. Box 4 actually produced a better outcome (although it was purely fortuitous) in that each of the 6 condition codes occurred once in the first 6 subjects and once in the last 6 subjects. This is often a desirable outcome. Note in Box 5, on the other hand, that the two occurrences of condition 4 both occurred in the first 6 rows (subjects), and the two occurrences of condition 1 occurred in the last 6 rows (subjects). This could introduce some bias into the conditions if the first 6 and last 6 subjects differ from one another in some systematic way (e.g., gender, motivation). It may be desirable to randomize in such a way as to ensure the equal distribution of conditions obtained accidentally in Box 4.

Block Randomization

Because of potential problems with simple randomization as done above, many researchers prefer to use a method called block randomization. Block randomization randomly orders the condition codes, but ensures that each condition occurs the same number of times within each block of observations. SPSS can easily do this, because we can sort cases on more than one variable.

Box 6 shows one way to do block randomization in SPSS. In essence, the modulus operator is used to generate condition codes and the truncate operator to generate block codes. These are shown ordered in the first listing. The cases are then sorted on both block and rand to produce the blocks of the 6 codes in a random order within each block. Notice in Box 6, for example, that each condition code occurs once in the first 6 cases (i.e., block = 1) and once in the last 6 cases (block = 2). This will always occur with this procedure (whereas it occurred just by chance in Box 4). Using this procedure, researchers can be quite confident that experimental condition is independent of any subject characteristics associated with the order in which participants are allocated to conditions.

In Box 6, the block size was equal to the number of conditions so that each condition occurred exactly once per block. Sometimes, researchers will want or find acceptable replications of conditions within blocks. The program in Box 7 allows for replications within blocks. The number of conditions is reduced to 4 simply to shorten the output. We have two blocks of size 8, each containing the four condition codes

```
*Block randomization.
SET SEED = 1234567.
INPUT PROGRAM.
COMP #obs = 12.
LOOP o = 1 TO #obs.
END CASE.
END LOOP.
END FILE.
END INPUT PROGRAM.
COMPUTE rand = UNIFORM(1).
FORMAT rand (F6.4).
COMPUTE block = TRUNC((O-1)/6) + 1.
*  6 above is size of block.
COMPUTE cond = MOD(O-1, 6) + 1.
*  6 above is number of conditions.
LIST.

      O   BLOCK    COND  RAND
 1.0000  1.0000  1.0000  .6849
 2.0000  1.0000  2.0000  .5154
 3.0000  1.0000  3.0000  .8941
 4.0000  1.0000  4.0000  .0074
 5.0000  1.0000  5.0000  .1297
 6.0000  1.0000  6.0000  .8299
 7.0000  2.0000  1.0000  .1689
 8.0000  2.0000  2.0000  .0552
 9.0000  2.0000  3.0000  .0616
10.0000  2.0000  4.0000  .8679
11.0000  2.0000  5.0000  .6697
12.0000  2.0000  6.0000  .2581

SORT CASES BY block rand.
LIST.

      O   BLOCK    COND   RAND
 4.0000  1.0000  4.0000  .0074
 5.0000  1.0000  5.0000  .1297
 2.0000  1.0000  2.0000  .5154
 1.0000  1.0000  1.0000  .6849
 6.0000  1.0000  6.0000  .8299
 3.0000  1.0000  3.0000  .8941
 8.0000  2.0000  2.0000  .0552
 9.0000  2.0000  3.0000  .0616
 7.0000  2.0000  1.0000  .1689
12.0000  2.0000  6.0000  .2581
11.0000  2.0000  5.0000  .6697
10.0000  2.0000  4.0000  .8679
```

Box 6. Block Randomization.

twice.

The program used in Boxes 6 and 7 is quite general. Some care is needed in determining the values for the number of observations (#obs), the block size (the divisor in the *compute block* = line, and the number of conditions specified in the *comp cond* = line. The block size should allow each condition to occur the same number of times; specifically, block size = number of conditions x number of repetitions per block. With 4 conditions and 2 replications in our example, our block size is 8. If we wanted 3 replications per block, block size would be 12, and so on. Given the block size, the total number of observations should be a multiple of the block size. In our example of 8 observations per block, the total number of observations could be 8, 16 (as in Box 7), 32, 48, and so on, depending on the total number of subjects desired, and available. If the block size was 12, then the total number of observations could be 12, 24, 36, and so on.

```
*Block randomization with.
*  replications per block.
SET SEED = 1234567.
INPUT PROGRAM.
COMP #obs = 16.
LOOP o = 1 TO #obs.
END CASE.
END LOOP.
END FILE.
END INPUT PROGRAM.
COMPUTE block = TRUNC((O-1)/8) + 1.
*  8 = size of block.
COMPUTE cond = MOD(O-1, 4) + 1.
*  4 = number of conditions.
COMPUTE rand = UNIFORM(1).
FORMAT rand (F6.4).
LIST.

      O   BLOCK    COND   RAND
 1.0000  1.0000  1.0000   .6849
 2.0000  1.0000  2.0000   .5154
 3.0000  1.0000  3.0000   .8941
 4.0000  1.0000  4.0000   .0074
 5.0000  1.0000  1.0000   .1297
 6.0000  1.0000  2.0000   .8299
 7.0000  1.0000  3.0000   .1689
 8.0000  1.0000  4.0000   .0552
 9.0000  2.0000  1.0000   .0616
10.0000  2.0000  2.0000   .8679
11.0000  2.0000  3.0000   .6697
12.0000  2.0000  4.0000   .2581
13.0000  2.0000  1.0000   .4542
14.0000  2.0000  2.0000   .8614
15.0000  2.0000  3.0000   .0728
16.0000  2.0000  4.0000   .8607

SORT CASES BY block rand.
LIST.

      O   BLOCK    COND   RAND
 4.0000  1.0000  4.0000  .0074
 8.0000  1.0000  4.0000  .0552
 5.0000  1.0000  1.0000  .1297
 7.0000  1.0000  3.0000  .1689
 2.0000  1.0000  2.0000  .5154
 1.0000  1.0000  1.0000  .6849
 6.0000  1.0000  2.0000  .8299
 3.0000  1.0000  3.0000  .8941
 9.0000  2.0000  1.0000  .0616
15.0000  2.0000  3.0000  .0728
12.0000  2.0000  4.0000  .2581
13.0000  2.0000  1.0000  .4542
11.0000  2.0000  3.0000  .6697
16.0000  2.0000  4.0000  .8607
14.0000  2.0000  2.0000  .8614
10.0000  2.0000  2.0000  .8679
```

Box 7. Replications within Blocks.

COUNTERBALANCING

Counterbalancing methods ensure that all conditions occur equally often at different stages of testing and hence provide more precise control over practice and fatigue effects than does randomization. Systematic counterbalancing of conditions is often required for within-subject designs, and is generally used in conjunction with the randomization methods described above. Consider a within-studies design to test the effects of three different drugs; that is, each animal will be tested with drug A, drug B, and drug C. The order in which drugs are administered to the animals could easily affect performance, so careful control of this factor is required. Using block randomization to determine the order of the three drugs for each animal does not guarantee that every drug will occur first, second, and third equally often. It is possible that by chance one drug would occur first or last more often than the other drugs and this difference could contaminate the effects of the drugs on performance.

Programs to perform counterbalancing tend to be more complex than those we have used to this point. Therefore, programs are not presented here for some of the techniques discussed. The concepts themselves can be understood without detailed knowledge of the programming mechanics. In its simplest form, counterbalancing ensures that every condition occurs equally often at each stage of the study. We first examine a simple method that works under certain limited conditions and then more sophisticated procedures for counterbalancing conditions.

Simple (or Complete) Counterbalancing

One simple method that works well with relatively small numbers of conditions is to use all-possible permutations of conditions. In the case of three conditions, there are $3! = 3 \times 2 \times 1 = 6$ different orders (i.e., 123, 132, 213, 231, 312, and 321). Each of these 6 orders could be used with the same number of subjects, ensuring that position and order effects do not contribute unequally to the various conditions. In practice each permutation would be assigned a number from 1 to 6 and block randomization of these

digits could be used to decide which order to assign to each subject. The combination of counterbalancing and randomization provides an elegant solution to possible confounding between independent variables and order of administration of conditions. There are also occasions when counterbalancing may be desirable for independent groups designs. Consider a study in which individual subjects are being tested at one of three different time periods (e.g., 9am, 11:30am, and 2:30pm) and there are three different conditions. Randomized counterbalancing of the conditions would ensure that each condition occurred equally often at each time interval.

Boxes 8 and 9 present an SPSS program that enumerates all permutations for k = 3 and randomizes them. The program has been separated into sections with intervening listings to better demonstrate the logic. Box 8 shows commands to generate all possible combinations of the three condition codes, labelled i1, i2, and i3, including not only permutations, but also combinations with repeated values (e.g., 133 for o = 9 and 333 for o = 27). These repetitions will later be deleted. The program works by first calculating the total number of trials required, which is k raised to the kth power. Briefly, this is because there are k

```
*Method to generate random permutations.
*Generate initial sequence of k**k trials.
INPUT PROGRAM.
COMPUTE #k = 3.
LOOP o = 1 to #k**#k.
END CASE.
END LOOP.
END FILE.
END INPUT PROGRAM.
*Generate the k variables.
DO REPEAT i = i1 i2 i3 /power = 2 1 0.
COMPUTE i = MODULUS(TRUNC((o - 1)/#k**power), #k) + 1.
END REPEAT.
FORMAT o i1 i2 i3 (f3.0).
LIST.

  o  i1  i2  i3
  1   1   1   1
  2   1   1   2
  3   1   1   3
  4   1   2   1
  5   1   2   2
  6   1   2   3
  7   1   3   1
  8   1   3   2
  9   1   3   3
 10   2   1   1
 11   2   1   2
 12   2   1   3
 13   2   2   1
 14   2   2   2
 15   2   2   3
 16   2   3   1
 17   2   3   2
 18   2   3   3
 19   3   1   1
 20   3   1   2
 21   3   1   3
 22   3   2   1
 23   3   2   2
 24   3   2   3
 25   3   3   1
 26   3   3   2
 27   3   3   3
```

Box 8. Initial parts of SPSS Permutation Program.

possibilities on each of the k trials. The first section of the program in Box 8 generates the 27 cases required for k = 3. The second section uses a somewhat impenetrable algorithm to generate the 9 repeated 1s, 2s, and 3s for i1, the 3 repeated 1s, 2s, and 3s for i2 (with the cycle repeated 3 times), and the single 1s, 2s, and 3s for i3 (with the cycle repeated 9 times). This generates all possible combinations of the three condition codes.

Only 6 of the 27 sequences are actually permutations: sequences 6, 8, 12, 16, 20, and 22. The next step is to pick out those sequences from all of the sequences. Note that these are the only sequences in which each of the condition codes occurs exactly once; that is, there are no repetitions of 1, 2, or 3.

Box 9 shows the remaining commands. The first command, *COMPUTE perm =* results in a new variable that will have the value of 1 if i1 does not equal (<>) i2, AND i1 does not equal i3, AND i2 does not equal i3; that is, perm = 1 if there are no repetitions (i.e., if the sequence is a permutation). The program then computes a new value for 0, numbering the permutations from 1 to 6. It does this using a built-in SPSS variable *$CASENUM*, which is a system variable indicating to which case each row of data belongs.

```
*Delete records with repetitions of conditions.
*  and number resulting sequences.
COMPUTE perm = (i1 <> i2) and (i1<> i3) and (i2<>i3).
SELECT IF perm = 1.
EXECUTE.
DELETE VARI perm.
COMPUTE o = $CASENUM.
LIST.

  o  i1  i2  i3
  1   1   2   3
  2   1   3   2
  3   2   1   3
  4   2   3   1
  5   3   1   2
  6   3   2   1

*Randomize sequences.
SET SEED = RANDOM.
COMPUTE rand = UNIFORM(1).
SORT CASES BY rand.
DELETE VARI rand.
LIST.

  o  i1  i2  i3
  5   3   1   2
  3   2   1   3
  2   1   3   2
  6   3   2   1
  4   2   3   1
  1   1   2   3
```

Box 9. Select and Randomize the Permutations.

The next block of commands in Box 9 randomly sets the seed value, and then shuffles the sequences using methods that we have used before (i.e., *COMPUTE rand* and *SORT CASES BY*). Only a few changes would be necessary to use this program to randomize permutations for other values of k. Appendix A shows the commands

necessary to produce random permutations for k = 4. The number of resulting permutations is 4 x 3 x 2 x 1 = 24. The modified lines are italicized in Appendix A. The trickiest element is probably the computation of perm, which requires that all possible pairs of variables be included in the calculation. Note in Appendix A that cases have been listed two to a line to allow the output to appear on a single page.

Complete counterbalancing is limited to variables with few levels because of the very large numbers of orders once more than a few conditions are involved. As just noted, 4 conditions involve 24 permutations. And it was earlier noted in the section on randomization that 5 conditions entail 120 permutations, 6 conditions entail 720 permutations, and so on. Clearly complete counterbalancing would be impractical in many experimental situations. The solution is to control order effects using certain carefully chosen permutations rather than every possible permutation. One common approach involves latin squares.

Latin Squares and Counterbalancing

Latin squares are k sequences of the digits from 1 to k such that each digit occurs exactly once in each row and once in each column (i.e., in each cell); no digits are repeated in any row or column.

A simple latin square of 5 digits is shown in Box 10. Each of the rows represents a subject and each of the columns a position or stage of testing. The entries in the cells would generally correspond to experimental conditions. Note that each condition code from 1 to 5 occurs once in each row and column. Although 720 orders would be required to completely counterbalance the 5 conditions, only 5 well-selected orders are required to ensure that each condition occurs exactly once at each stage of testing.

	Columns				
	1	2	3	4	5
Rows					
1	1	2	3	4	5
2	2	3	4	5	1
3	3	4	5	1	2
4	4	5	1	2	3
5	5	1	2	3	4

Box 10. Latin Square of Order 5.

The square in Box 10 is a systematic one and was generated by cyclical rotation of

the cell entries in the first row. All numbers were shifted once to the left and the leftmost number was cycled around to the last position. This technique can be used to create the initial square (with some caution because every possible latin square will not occur), but randomization is required before the square can be used. Note in Box 10, for example, that each condition follows exactly the same other condition for every subject (i.e., row). This would be undesirable if there were any possibility of carryover effects from one condition to another (e.g., fatigue from a particularly demanding condition).

To randomize Latin squares such as that in Box 10, three variables need to be randomized: the rows, the columns, and the treatment numbers. This could be done for the square in Box 10 by generating three permutations of the digits 1 to 5 using the procedures described in the Randomization section (e.g., SPSS). One permutation would be used to shuffle the rows. A second permutation would be used to shuffle the columns. The third permutation would be used to shuffle the treatment numbers.

Box 11 shows the result of randomizing the square in Box 10 using the three permutations shown at the top. The final latin square is shown in bold. The original row, column, and treatment numbers are shown in parentheses to demonstrate the randomization process more clearly. This latin square might be used, for example, to determine the order (Columns) in which 5 presentation rates (Treatments) are to be administered to 5 subjects (Rows). Subject 1 would receive the rates in the order 2, 5, 1, 4, and 3, and so on. The square could also represent the order (Columns) in which 5 subjects (Rows) are asked to perform 5 tasks (Treatments).

```
Row Permutation: 3 4 1 5 2
Col Permutation: 5 1 3 2 4
Trt Permutation: 3 2 5 4 1

                Column
Row   1(5)  2(1)  3(3)  4(2)  5(4)
1(3)  2(2)  5(3)  1(5)  4(4)  3(1)
2(4)  5(3)  4(4)  3(1)  1(5)  2(2)
3(1)  1(5)  3(1)  5(3)  2(2)  4(4)
4(5)  4(4)  1(5)  2(2)  3(1)  5(3)
5(2)  3(1)  2(2)  4(4)  5(3)  1(5)
```

Box 11. Randomized Latin Square of Order 5.

Mutally Orthogonal Latin Squares

There are occasions when researchers have to counterbalance several factors and several orthogonal latin squares are required. Mutally orthogonal latin squares (MOLS) are multiple latin squares of the same size in which every combination of the condition numbers is unique.

Box 12 shows two orthogonal latin squares of size 3. The first digit in each cell corresponds to the entry for one latin square and the second digit corresponds to the entry for the second latin square. Note that combinations of the two latin squares occur exactly once (i.e., 11, 12, 13, ..., 33). If the 9 combinations of 4 numbers in Box 12 were entered as 4 scores (i.e., 1 1 1 1, 1 2 2 2, 1 3 3 3, 2 1 2 3, 2 2 3 1, 2 3 1 2, 3 1 3 2, 3 2 1 3, 3 3 2 1) the four variables would correlate 0 with one another; that is they are orthogonal or independent.

	1	2	3
1	11	22	33
2	23	31	12
3	32	13	21

Box 12. Orthogonal Latin Squares.

The MOLs in Box 12 could be used in a variety of ways. For example, a social psychology researcher interested in the effect of three different kinds of messages (1 = Negative, 2 = Neutral, or 3 = Positive) on attitudes toward minorities might design a study as follows. Three different ethnic groups would be selected (e.g., 1 = Norwegians, 2 = Spaniards, or 3 = Australians). Subjects would receive a single message about each of the three groups. One message would be positive, one neutral, and one negative. Subjects would rate the ethnic groups immediately after receiving each message. It would be important to control the combinations of group, message, and position so that the message variable was independent of (i.e., orthogonal to) the other factors. Complete counterbalancing would require $3 \times 3 \times 3 = 27$ combinations of the different levels of the variables (negative Norwegians 1st, negative Norwegians 2nd, ..., positive Australians 3d). The number of subjects required for the experiment would have to be some multiple of 27 if every combination was to be used equally often.

Alternatively, the MOLs in Box 12 would control for position and ethnic group

within groups of just three subjects (vs. 27). The first subject would receive the conditions 11 = negative + Norwegian, 22 = neutral + Spaniard, 33 = positive + Australian in that order. A second subject would receive 23 = neutral + Australian, 31 = positive + Norwegian, and 12 = negative + Spaniard in that order. A third subject would receive 32 = positive + Spaniard, 13 = negative + Australian, and 21 = neutral + Norwegian in that order. Note that each message and each ethnic group occurs once in each position (1st, 2nd, or 3rd) and that each message occurs once with each ethnic group. Any effect of message type would be independent of position and ethnic group differences.

In practice, the MOLs in Box 12 would be randomized prior to use. Rows, columns, and both treatment codes would be randomized independently to produce the final square.

Limitations on MOLS. A number of factors complicate the use of MOLs. There are some cases in which orthogonal latin squares have not been found or have been demonstrated to not exist. For example, there are no orthogonal latin squares of size 6. A researcher must consider such factors in designing any study.

A second complication is that automated construction of orthogonal latin squares is possible only for certain numbers of conditions. It is known that there are t-1 mutually orthogonal latin squares for numbers that are primes (i.e., divisible only by themselves and one) or products of primes. It is also relatively straightforward to write a program to generate the t - 1 MOLs for prime-number designs using modulus arithmetic.

To illustrate the use of higher order MOLS, imagine that an industrial/organizational (I/O) or human factors psychologist is interested in characteristics that affect the readability of text on computer screens. The researcher wants to examine four factors that each have 5 levels to them: text color (white, red, green, blue, or black), background (one of five shades of grey), font style (A, B, C, D, or E), and font size (6, 8, 10, 12, or 14 points). Five highly-trained subjects are available to

rate readability, but each subject can only rate 5 conditions because of eye-strain and fluctuations in standards across multiple trials and it is impractical to test every possible combination of conditions.

A Basic program produced the MOLs shown in Box 18. The initial commands are also shown, as well as the permutations generated by the program. Each row corresponds to one subject and each column to one trial. The 5441 entry for subject 1 and trial 1 indicates black text on a level 4 grey background in font style D and 6-point size. The 5225 for trial 5 of subject 5 indicates black text on level 2 background in font style B and font size 14. Careful examination of the squares will reveal that each condition occurs exactly once for each subject, for each trial, and for each level of all other factors. If the 6 scores for each of the 25 cells were analyzed (i.e., 1 1 5 4 4 1, 1 2 4 1 3 5, ...), the six variables would all correlate 0 with one another.

```
Number of conditions (Prime!!) ? 5
Randomize (1) or Not (2)       ? 1
Matrix (m) or Column (c) output? m
COL    1    2    3    4    5
ROW
  1 5441 4135 1214 2522 3353
  2 4324 2251 5533 3415 1142
  3 3232 1423 2345 5154 4511
  4 1555 5312 3121 4243 2434
  5 2113 3544 4452 1331 5225

Permutations used.
  5   1   3   4   2
  3   4   2   5   1
  1   5   4   2   3
  3   1   2   5   4
  3   5   4   2   1
  1   4   3   2   5
```

Box 13. MOLs of Size Five.

The efficiency of the latin square design can be appreciated by considering the full-factorial alternative to the latin square design. Five factors (including trial) each having 5 levels would require $5 \times 5 \times 5 \times 5 \times 5 = 5^5 = 3{,}125$ cells for a full factorial design. Even without trial, 625 cells are required. The MOLs require only 25 cells to provide information about the main effects of each factor. Latin squares are examples of partial or fractional replication designs. The cost of this efficiency is loss of information about interactions between the factors, especially the higher-order interactions. That is, we will not have information for certain combinations of conditions that might be

particularly good (or bad).

Although squares are not as easily generated for non-prime designs, they are possible and are published in a number of sources. Box 14 shows MOLs for four conditions. The square would be randomized before being used in a study. A psychologist doing marketing research, for example, might examine sales in four stores (rows) across four successive weeks (columns) for four products (1st treatment) packaged in four differently colored boxes (2nd treatment) and with four differently sized displays (3d treatment).

111	222	334	443
244	133	421	312
323	414	142	231
432	341	213	124

Box 14. MOLs for Four Conditions.

Balanced Latin Squares

All latin squares do not ensure that every condition follows every other condition exactly once. Indeed, it is possible even in a randomized square that conditions will always follow the same condition. Examine several of the squares discussed previously to see which other conditions a particular condition tends to follow. The more different squares generated and used, the less likely this possibility.

Squares in which every condition follows every other condition exactly once are called balanced latin squares. Box 15 shows examples of balanced latin squares for designs with 4 or 6 conditions. Note in the top square that condition 4 follows condition 3 in row 1, condition 2 in row 2, condition 1 in row 3, and no prior condition in row 4. Balanced latin squares are used when researchers are concerned that conditions will have robust carryover effects, either positive or negative, such that performance in one condition will depend on the preceding condition to which subjects were exposed.

Four Conditions

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

Six Conditions

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

Box 15. Balanced Latin Squares.

As with MOLs, there are some restrictions on balanced latin squares. There are no balanced latin squares for odd numbers, for example, which means that researchers who are particularly concerned about carryover or transfer effects will plan studies with even numbers of conditions.

CONCLUSION

This chapter has introduced some of the basic ideas that underlie randomization and counterbalancing in experimental design. The primary purpose of these techniques is to increase the probability and ideally ensure that treatment variables are independent of one another and of other nuisance variables that could influence performance on the dependent variable. Not all aspects of this problem have been addressed here. For instance, cognitive researchers often must ensure that across subjects different materials are used in every possible condition. Each item in a priming study, for example, might have to occur with a neutral prime, a strong prime, and a weak prime, although for different subjects. Good counterbalancing provides the required control, in essence by generalizing the principles described here for counterbalancing treatments and order.

The identification of relevant confounding variables and the design of studies independent of those variables involve mastery of some challenging and subtle cognitive skills. As you read the research literature in different areas of psychology, attend to the kinds of confounding variables common in the area and to the ways in which experimenters identify or control the effects of confounding variables. Having such knowledge will help you to design studies that avoid the well-known errors in your chosen area of research, increasing the likelihood that your study will further our understanding of the phenomena being studied.

APPENDIX A

SPSS PROGRAM TO GENERATE PERMUTATIONS FOR K = 4

```
INPUT PROGRAM.
COMP #k = 4.
LOOP o = 1 TO #k**#k.
END CASE.
END LOOP.
END FILE.
END INPUT PROGRAM.
DO REPEAT i = i1 i2 i3 i4 /power = 3 2 1 0.
COMP i = MODULUS(TRUNC((O - 1)/#k**power), #k) + 1.
END REPEAT.
COMPUTE perm = (i1 <> i2) AND (i1<> i3) AND (i1<> i4) AND (i2<>i3) AND (i2<>i4) AND (i3<>i4).
SELECT IF perm = 1.
EXEC.
DELETE VARI perm.
COMPUTE o = $CASENUM.
FORMAT o i1 i2 i3 i4 (F3.0).
LIST.
  O  I1  I2  I3  I4
  1   1   2   3   4                    2   1   2   4   3
  3   1   3   2   4                    4   1   3   4   2
  5   1   4   2   3                    6   1   4   3   2
  7   2   1   3   4                    8   2   1   4   3
  9   2   3   1   4                   10   2   3   4   1
 11   2   4   1   3                   12   2   4   3   1
 13   3   1   2   4                   14   3   1   4   2
 15   3   2   1   4                   16   3   2   4   1
 17   3   4   1   2                   18   3   4   2   1
 19   4   1   2   3                   20   4   1   3   2
 21   4   2   1   3                   22   4   2   3   1
 23   4   3   1   2                   24   4   3   2   1

SET SEED = RANDOM.
COMP rand = UNIFORM(1).
SORT CASES BY rand.
DELETE VARI rand.
LIST.
  O  I1  I2  I3  I4
  1   1   2   3   4                   12   2   4   3   1
  5   1   4   2   3                   14   3   1   4   2
 15   3   2   1   4                    3   1   3   2   4
  2   1   2   4   3                   13   3   1   2   4
  9   2   3   1   4                   23   4   3   1   2
 20   4   1   3   2                   11   2   4   1   3
  7   2   1   3   4                    8   2   1   4   3
 24   4   3   2   1                   17   3   4   1   2
 21   4   2   1   3                   22   4   2   3   1
  6   1   4   3   2                    4   1   3   4   2
 18   3   4   2   1                   19   4   1   2   3
 16   3   2   4   1                   10   2   3   4   1
```