Dap 3.5

Data analysis & presentation

Susan Bassein, Ph.D.

Contents

Chapter 1

Overview and invocation

1.1 Description, an example, and reading this manual

Dap is a small statistics and graphics package based on C. As of Version 3.0, dap can read SBS programs to perform basic statistical analyses, thus freeing the user from learning and using C syntax for straightforward tasks, while retaining access to the C-style graphics and statistics features provided by the original implementation (see Chapter 9.9 [Appendix IV], page 103). Dap provides core methods of data management, analysis, and graphics commonly used in statistical consulting practice. Anyone familiar with the basic syntax of C programs can learn to use the C-style features of dap quickly and easily from this manual and the examples it provides; advanced features of C are not necessary, although they are available. (Chapter 9.6 [Appendix II], page 95 provides a brief description of the C syntax needed to use those features of dap dependent on it.) Because dap processes files one line at a time, rather than reading entire files into memory, it can be used on data sets that have very large numbers of lines and/or large numbers of variables.

Dap consists of: an interactive front end that controls editing, compiling, execution, and viewing; an optional SBS-to-C-style-dap translator that can convert an SBS program into a C-style dap program; a preprocessor that facilitates data input and output by converting a C-style dap program into an ordinary C program; and a library of statistics functions and graphics functions. Typical use of dap involves the following steps:

1. Obtain, or use a text editor to create, one or more data files in acceptable formats.

2. Invoke (i.e., run) '`dap`' to create or view an SBS or C-style dap program that will read and process the data file(s) to produce tables and/or graphics. (See Section 1.2 [Invoking Dap], page 6 to find out how to run dap. If you want to use the C-style features of dap and are unfamiliar with C,

read Chapter 9.6 [Appendix II], page 95 before creating your C-style dap program.)

3. Tell dap to compile (i.e., translate into machine usable form) your SBS or C-style dap program and execute (i.e., run) your program.
4. If the previous step was successful, view your tables and/or graphics; otherwise, if there are errors in your program, you will have to edit your program further and repeat Step 3.

The following program examples illustrate the basic structure of simple SBS and C-style dap programs and how functions in the library are used in dap.

```
/* A SIMPLE SBS PROGRAM */
data;                  /* Begin data step */
 infile "two" delimiter="\t" firstobs=2;
                       /* The infile statement: identifies "two"
as
                        * the file to read from, specifies that
                        * the delimiter is a tab, i.e. the numbers
                        * in each line in "two" are separated
by a
                        * single tab character, and directs the
                        * program to skip one header line and
start
                        * data input on the second line of the
file "two".
                        */
 input x y;            /* The first number on each line of "two"
                        * will be held in x and the second number
                        * will be held in y
                        */
proc plot;             /* Run the plot procedure to display a
scatter-plot... */
 plot y * x / box;     /* ...using y for the vertical axis and
x for the
                        * horizontal and enclosing the plot in
a box
                        */


/* A SIMPLE C-STYLE DAP PROGRAM */
#include <dap.h>       /* REQUIRED: includes prototypes of
                        * statistics and graphics functions
                        */

void main()            /* REQUIRED: main runs everything */
{
```

```
/* This part of the program reads in data from an ordinary file
 * and writes it out to a dataset, which is the kind of file
 * on which the library functions can operate to perform
 * exploratory data analysis, graphical interpretation, and
 * statistical analyses.
 */
infile("two", "\t")    /* The infile statement: identifies "two"
as
                        * the file to read from and specifies
that
                        * the delimiter is a tab, i.e. the numbers
                        * in each line in "two" are separated
by a
                        * single tab character
                        */
  {                    /* Beginning of infile statement body */
  double x, y;         /* specify that x and y are variables that
                        * hold double-precision floating point
                        * numbers, i.e., numbers that are not
                        * necessarily whole numbers
                        */
  input("x y");        /* The first number on each line of "two"
                        * will be held in x and the second number
                        * will be held in y
                        */
  outset("xyset", ""); /* Create a dataset "xyset" and write all
                        * the variables to it
                        */
  skip(1);             /* The data file contains 1 header line;
                        * skip it
                        */
  while (step())       /* Read in the file, one line at a time
*/
    output();          /* and write it to the dataset "xyset"
*/
  }                    /* end of infile statement body */

/* This line runs the plot function to get a scatter-plot, then
 * displays the result graphically using the nport function; the
 * various options used are described in the section for the plot
 * and nport functions.  */
nport(plot("xyset", "x y", "", "==", NULL, NULL, 1), 1, 1);
}
```

For each C-style dap function, this manual gives:

1. **SPECIFICATION**: This is an abstract template for the function that

indicates the types of the parameters (string, i.e., array of char, double, array of double, int, or array of int), and the return value, if any; **bold face** indicates literal text, *italics* indicates parameters for which you choose your own names. The corresponding SBS template, if available, is also provided.

2. **Description**: A description of the functionality and use of the function. If there is a corresponding SBS proc, notes on its usage follow.
3. **Code fragment(s)**: These are one or more examples of calling the function, with made-up values for the parameters; for complete examples of use, see the final chapter (see Chapter 9 [Examples], page 75) or consult the index.

For the SBS templates, the following should be noted. Elements within square brackets '`[ ]`' are optional and some, although permitted, may have no effect. Elements within braces '`{ }`', separated by a vertical bar '`|`' are alternatives. Keywords may be in upper or lower case. Single and double quotes must be used as in C, i.e., use double quotes for strings (whether or not they are one character long) and use single quotes only for character values. Not all statements listed below the '`data`' or '`proc`' statements are required and multiple statements of some types may be allowed.

1.2 Invoking Dap

To run dap, you must type a command which looks like one of the following (in which the bracketed items are optional and are described below):
SPECIFICATION

dap [-k] [-d] *file1*.**c [** *file2*.**c ...] [-a** *arg1* **...]**
dap [–keep] [–debug] *file1*.**c [** *file2*.**c ...] [–args** *arg1* **...]**
dap [-k] [-d] *file1*.**sbs [** *file2*.**c ...]**
dap [–keep] [–debug] *file1*.**sbs [** *file2*.**c ...]**

Description
The following description is based on using the GNU editor Emacs to interact with dap. If you don't use Emacs, you can still use dap, but in the commands specified above, use '`daprun`' in place of '`dap`'; in that case, you will have to open your editor and/or file viewer to edit your program and view your text output.
When you type the command to run dap:

1. The editor Emacs (type the command `info emacs` for the manual) is opened for the named SBS or C-style dap file(s). Each file is created, but is empty, if it doesn't exist. When you are finished editing your file(s), save it (them), but do not exit unless you are finished running dap.

2. Emacs will have a buffer called '`Async Shell Command`'; select that buffer and you will be asked whether to compile the named SBS or C file(s) and run the resulting executable file (if the compilation is successful). If you answer '`y`', then dap runs the preprocessor on the named SBS or C file(s) to produce *file1*.dap.c [*file2*.dap.c ...]. If you answer '`q`', then dap stops running (and you can either leave Emacs running for some other purpose or exit it).

3. If the preprocessing is successful, then dap runs a C compiler on the preprocessed file(s), linking library functions as necessary, to produce *file1*.dap.

4. If the compilation is successful, then dap removes the `dap.c` file(s) (unless the '`-d`' or '`--debug`' option is present) and runs the executable *file1*.dap, passing the arguments *arg1*, ..., if any, and produces *file1*.lst file for tables and *file1*.ps for graphics, as requested. There are Emacs buffers for the `lst` and `log` files, but you must use '`Revert buffer`' in the '`File`' menu to see the most recent results. A viewer will be opened for the `ps` file, if it exists.

5. Each time a dataset or datafile is read through to the end during execution, the number of lines read from the dataset or datafile is displayed in your '`Async Shell Command`' buffer and written to the `log` file. If '`dap_outreport`' (see Chapter 9.6 [Appendix I], page 91) is positive (default: 100000), then dap reports, in your '`Async Shell Command`' buffer, each time that an additional '`dap_outreport`' lines have been written to a dataset; if your program is writing to a dataset inside an infinite loop, this helps you abort the program to avoid filling up your disk (by holding down the '`Ctrl`' key and pressing the '`C`' key *twice*).

6. If there were errors or warnings, they will be written to the `err` and you will see them displayed in your '`Async Shell Command`' buffer.

7. At this point, you can edit (and save!) your SBS or C-style dap files again or quit.

If the '`-a`' or '`--args`' option is present (C-style dap programs only), then all the arguments that follow it on the command are passed to the '`main`' in your program as '`argv`'. In all cases, the `lst` and `ps` files are removed before the first execution, but if the '`-k`' or '`--keep`' option is present, then the `lst` file is kept and appended to from one cycle to the next. (Keeping the `lst` file from one cycle to the next can be useful for exploratory data analysis, such as model building for linear regression or loglinear models, in which it is useful to have a record of a succession of analyses without running the cumulative set of analyses repeatedly.)

NOTE: If your system uses '`locale`'s, then you must set your environment variable '`LC_COLLATE`' to '`C`' in order to obtain correct sorting.

The editor invoked is /usr/bin/emacs unless the environment variable '`DAPEDITOR`' specifies a different location for Emacs. Emacs is invoked with no options unless the environment variable '`DAPEDOPTS`' is set to a single string of all the options desired. The compiler used is /usr/bin/gcc unless the environment variable '`DAPCOMPILER`' is set to a different compiler. The compiler is invoked with no options unless the environment variable '`DAPCOMPOPTS`' is set to a single string of all the options desired. The `lst` file is displayed in its entirety in its buffer unless the environment variable '`DAPPAGER`' is set to a pager (such as `more`). That pager is invoked with no options unless the environment variable '`DAPPAGEOPTS`' is set to a single string of all the options desired. The `ps` file is viewed with /usr/bin/X11/gv unless the environment variable '`DAPVIEWER`' is set to a different graphics viewer. The viewer is invoked with no options unless the environment variable '`DAPVIEWOPTS`' is set to a single string of all the options desired.

The executable *file1*.dap can be run, or debugged, on its own. If the '`-a`' or '`--args`' option was used when the source files were processed, then arguments (without the '`-a`' or '`--args`') are used on the command line. If you want to use '`gdb`' to debug *file1*.dap, include '`-g`' as one of your compiler options and use the '`-d`' or '`--debug`' option when you run dap so that the `.dap.c` files will not be removed.

Code fragment(s)

Run `dap` on the file `tweedle.c`:

```
dap tweedle.c
```

Run `dap` on the file `tweedle.sbs`:

```
dap tweedle.sbs
```

Run `dap` on the files `tweedle.c` and `dee.c`:

```
dap tweedle.c dee.c
```

Run `dap` on the files `tweedle.sbs` and `dee.c`:

```
dap tweedle.sbs dee.c
```

Run `dap` on `tweedle.c`; append to `tweedle.lst` in each cycle:

```
dap -k tweedle.c
dap --keep tweedle.c
```

Run `dap` on `tweedle.c`; pass arguments '`17`' and '`dum`' to '`main`' in `tweedle.c`:

```
dap tweedle.c -a 17 dum
dap tweedle.c --args 17 dum
```

Run `dap` on `tweedle.c`; do not remove `tweedle.dap.c` for debugging:

```
dap -d tweedle.c
dap --debug tweedle.c
```

Chapter 2

Program structure

This chapter concerns C-style dap programs only.

2.1 Reading files and datasets

This section and the next apply only to C-style dap programs. C-style dap programs are C programs with two additional statement types and additional library functions to perform statistical and graphical tasks. All C-style dap programs must contain a line

```
#include <dap.h>
```

to include the dap header file before the function '`main`' and any variable declarations, except that if the program resides in more than one `C` file, only one of those files must include `dap.h` while the others must include `dap1.h` instead. (This is to avoid multiple definitions of dap parameters.)

The two additional statement types are '`infile`' and '`inset`' statements, which are used to set up input from files and datasets and which have the following syntaxes:

SPECIFICATION

infile(*datafile*, *delimiter*) **{ ... }**
datafile, *delimiter:* string

inset(*dataset*) **{ ... }**
dataset: string

Description

The body of '`infile`' and '`inset`' statements may contain local variable declarations, an '`outset`' statement, and statements to process the input file or dataset. In addition, the body an '`infile`' statement may contain an '`input`' statement.

The local variable declarations in the body of '`infile`' and '`inset`' statements are restricted to the following types:

```
double name;
double name[number];
double name[string];
int name;
int name[number];
int name[string];
char name[number];
char name[string];
```

When the dimension of the array is given as a string, that string must appear in a '`#define`' somewhere in the file before the declaration.

A program can generate data without reading from a file. In that case, an '`infile`' statement is still required, but both arguments can be given as '`NULL`'.

The delimiter argument to an '`infile`' statement can be in one of two forms, one for delimited fields and the other for fixed-width fields. If the file to be read is simply delimited text, the the delimiter argument is a string (not a character) consisting of the single delimiter character. If the file has fixed-width fields, then the delimiter argument is an alternating sequence of any non-numerical character and numbers that specify the field widths, in order and without spaces (unless the non-numerical character is a space). For example:

```
infile("data", "x6x5x8")
or for sbs
PROC IMPORT  out = data datafile =  "Pres2007Clustered.csv"  dbms
= csv //support tab and dlm
delimiter = ',' replace ;  //support getnames=yes/no;
run ;
```

specifies three data fields of widths 6, 5, and 8, respectively.

It is inappropriate to call a dap statistics or graphics function from within the body of an '`infile`' or '`inset`' statement; results from such a call are unpredictable.

The header file `dap.h` defines parameters that control the execution of dap. For example, '`dap_maxvar`' is the maximum number of variables allowed in any dataset. To change any of these parameters, use a preprocessor '`#define`' line in your program **before** the line that includes '`dap.h`'. For example,

```
#define DAP_MAXVAR 512
#include <dap.h>
```

will set the maximum number of variables to 512 instead of the default 256. Note that such a '`#define`' line does *not* end with a semi-colon.

It is not necessary to use a '`#define`' such as the one above if you do not wish to change the values of the parameters from their defaults. However, some dap functions have limitations on how many values they can process at one time and if you have a dataset that is particularly large, then you may receive a memory allocation error message. That message will usually contain a suggestion of which parameter may need changing. The parameters available for modification are listed in Appendix I (see Chapter 9.6 [Appendix I], page 91).

Code fragment(s)

Read space-delimited data file `tweedledee` *with values for '*`name`*', '*`wages`*', and '*`tips`*' on each line and write those values to dataset* `tweedledee`*:*

```
infile("tweedledee", " ")
  {
    char name[21];
    double wages, tips;
    input("name wages tips");
    outset("tweedledee", "");
    while (step())
      output();
  }
```

Read from dataset `tweedledee`*, compute '*`income`*' as the sum of '*`wages`*' and '*`tips`*', and write to dataset* `tweedledum`*:*

```
inset("tweedledee")
  {
    double wages, tips, income;
    outset("tweedledum", "");
    while (step())
      {
        income = wages + tips;
        output();
      }
  }
```

2.2 Variables

This section applies only to C-style dap programs. The variables declared in the body of an '`infile`' or '`inset`' statement are available for reading from the specified file or dataset and for writing to the specified dataset. It is not necessary to declare all the variables in an input or output dataset; only those variables that are used explicitly in a statement in the body of the '`infile`' or '`inset`' statement (i.e., not in a quoted string as an argument to a function such as '`input`', '`outset`', or '`dap_list`'), must be declared. Avoid using variable (or function) names that begin with '`dap_`' or '`pict_`' or end with '`dap_`'. WARNING: Within an '`inset`' statement, if you declare and set the value of a variable that is already in the dataset referenced by the '`inset`' statement, values that your program gives to that variable will be overwritten on each call to '`step`'.

Data management, statistics, graphics, and other C-style dap functions take lists of variables as arguments. Such lists must always be given as strings of variable names (sometimes with other specifications included), separated by spaces (not commas). Generally, the null string (`""`, not '`NULL`') is a shorthand for a list of all the variables in the dataset and '`NULL`' is a shorthand for none of the variables in the dataset.

Arrays of int or double may be referenced, by the array name with no brackets or indices, in calls to '`input`' and '`outset`'. Individual array elements, with their bracketed indices, may also be referenced in calls to those same functions, but not in calls to '`dataset`' or '`merge`', for which arrays of int or double must be referenced by the array name, with no brackets or indices. For all other C-style dap functions, each element of an array of int or double that is to be used must appear with its bracketed index.

2.3 Output: the `err`, `log`, `lst`, and `ps` files

The user's program can write directly to the `lst` and `log` files using stream functions ('`putc`', '`fputs`', '`fprintf`', etc.) on '`dap_lst`' and '`dap_log`', respectively. The following function may be called to display a section header in the `lst` file:

SPECIFICATION

dap_head(*partv*, *npartv***)**
partv: array of int
npartv: int

Description

If *partv* is not '`NULL`', the header includes the values of the variables defining a part of the file or dataset (see Section 3.3 [Partitioning a dataset], page 18). NOTE: '`dap_head`' makes a system call to '`time`' and calls the GNU library function '`ctime`'; consequently, user-defined functions with those names will interfere with the date displayed by '`dap_head`'.

Code fragment(s)

Write header, with current title, if any, to `lst` file:

```
while (step())
  {
    ...
    dap_head(NULL, 0);
    ...
  }
```

With input dataset sorted by the variables '`state`', '`county`', and '`city`', write header, with current title, if any, and with current values of '`state`', '`county`', and '`city`', to `lst` file:

```
int fooparts[3];
...
dap_list("state county city", fooparts, 3);
while (step())
  {
    ...
```

```
        dap_head(fooparts, 3);
        ...
      }
```

A title may be specified as part of the section head or as caption in graphical output by calling:

SPECIFICATION

title(*the-title*)
thetitle: string

title "*the-title*";

Description

The string *the-title* may contain newline ('\n') characters for multi-line titles. For graphical output captions, subscripts are specified by enclosing them in vertical bars ('`|`') and superscripts are specified by enclosing them in carets ('`^`'). If a title has been specified by '`title`', then a graphics function that produces a separate page for each part of a partitioned dataset will include in the caption the value(s) of the variable(s) defining the partition.

Code fragment(s)

Set title for '`dap_head`':

```
title("Analysis of variance\nfor crop yields");
title("CFU per cm^2^ of surface area");
```

2.4 The preprocessor: dappp

The preprocessor, which is named dappp, reads *file*.c, performs the following four actions to create *file*.dap.c:

1. Process simple '`#defines`' of the form:

   `#define` *string n*

   in which *string* starts with a letter or with '`_`' and *n* is a positive integer.

2. Replace '`main`' with '`dap_main`' so that the '`main`' internal to dap can set up variables and the output files before calling '`dap_main`'.

3. Append a semicolon to each '`infile`' or '`inset`' statement header to prepare it for the C compiler.

4. Append calls to '`dap_vd`', '`dap_dl`', '`dap_il`', and '`dap_sl`' after declarations in the body of '`infile`' and '`inset`' statements.

The '`dap_vd`' function tells the internal dap routines the names and types of variables so that datasets can be processed appropriately. The '`dap_dl`', '`dap_il`', and '`dap_sl`' functions create links between the variables in the user's program and the internal dap storage to allow input and output. These functions are appended to the last line of the user's declarations so that line numbers in C compiler messages for *file*.dap.c correspond exactly to line numbers in *file*.c.

Chapter 3

Datasets

Datasets are data files with an additional first line that specifies the names and types of the variables whose values follow. The only other differences between datasets and data files are that (1) integers and double precision floating point numbers are encoded as text in a way that loses no precision and which allows sorting to be performed in the usual alphanumeric order and (2) the delimiter between fields on a line is always '`|`'. (A consequence of this is that string values cannot contain '`|`'.)

Most C-style dap data management, statistics, and graphics functions have an argument, called *partvars* in the specifications, that allows parts of a dataset to be processed separately. (SBS programs use a '`by`' statement.) To do that, first sort the dataset by the values of an ordered list of variables: each part is defined as a set of lines for which the values of all of those variables are constant. Then the same variables in the same order must be named in the *partvars* argument of the function that is subsequently used to process the sorted dataset. If no partitioning is desired, then *partvars* should be the null string (`""`).

Dap allows datasets that are not too large to be stored in memory instead of on disk in order to speed processing. To specify that a dataset is to be stored in memory, precede the name of the dataset with a '`<`' in the call to '`outset`' that creates it and in calls to functions that read it. Note that a dataset that is stored in memory will not be retained after the current dap session.

Only the last section in this chapter, '`Manage datasets`', has information relevant to SBS programs. For input for SBS programs, see See Chapter 9.9 [Appendix IV], page 103.

3.1 Input and output

SPECIFICATION

input(*variable-list*)
variable-list: string

Description

The string *variable-list* is a space-separated list of the names of the variables, in order, on each line of the data file to be read; *variable-list* may contain the names of arrays or individual array elements. Do not call '`input`' for a dataset.

Code fragment(s)

Specify that (the beginning of) each line of the input data file contains two fields whose values are to be read into the variables '`fert`' and '`yield`', in order:

```
input("fert yield");
```

SPECIFICATION

step()
Returns: int

Description

Attempts to read one line of the file or dataset specified by '`infile`' or '`inset`', respectively. Returns 1 if a line was read, 0 if EOF was reached before any characters were read; in this latter case, all input variables retain their values from the previous line. The program exits with an error message if '`step`' is called after an EOF was reached. Missing integer data is read in as 0. Data of type double that is missing or is '`.`' is set to NaN[1].

Code fragment(s)

Read all the lines of the current input dataset or data file and write them to the current output dataset:

```
while (step())
  output();
```

SPECIFICATION

skip(*nlines***)**
nlines: int

Description

Skips *nlines* lines of the input data file. This is useful for skipping header lines in a data file.

Code fragment(s)

Skip 3 lines of the input data file:

```
skip(3);
```

SPECIFICATION

outset(*dataset*, *variable-list***)**
dataset, *variable-list:* string

Description

The string *dataset* is the name of the data set to be written. The string *variable-list* is either a space-separated list of the names of the variables to be included in the output data set, a '`!`' followed by a space-separated list of the

[1] "Not a Number": see the function '`finite`' in the GNU/Linux Programmer's Manual.

names of the variables to be excluded from the output data set, or a null string (not '`NULL`'), which specifies that all variables are to be included in the output data set. The output data set always includes a variable named '`_type_`', which is a string of length 9 (includes terminating null) and which indicates the type of the observation. The string *variable-list* may contain the names of arrays, without bracketed indices, or individual array elements, but in either case the entire array is included in or excluded from the output dataset.

If the dataset is small and need not be retained after the current dap session, then *dataset* may begin with a '<' to specify that the dataset is to be stored in memory rather than on disk; this will speed processing. The string *dataset* must then begin with a '<' in calls to functions that read it. If *dataset* does not begin with a '<', then '`outset`' creates the data set in the subdirectory '`dap_sets`' of the current directory. If '`dap_sets`' does not exist, '`outset`' creates it. A call to '`outset`' must be preceded by an '`infile`' or '`inset`' statement; if no input file or dataset is desired, you may use '`NULL`' as the file specified by '`infile`' and '`NULL`' as the delimiter string.

Code fragment(s)

Name the output dataset '`tweedledee`' and specify that it will contain the values of the variables '`alice`' and '`caterpillar`' only:

```
outset("tweedledee", "alice caterpillar");
```

Name the output dataset '`tweedledum`' and specify that it will contain the values of all the variables that are either in the input dataset or are declared in the current '`infile`' or '`inset`' statement:

```
outset("tweedledum", "");
```

Name the output dataset '`tweedledum`' and specify that it will be stored in memory, not on disk, and that it will contain the values of all the variables that are either in the input dataset or are declared in the current '`infile`' or '`inset`' statement:

```
outset("<tweedledum", "");
```

SPECIFICATION

output()

Description

Writes one line of data to the output dataset specified by '`outset`'.

3.2 Positioning within a dataset

SPECIFICATION

dap_mark()

Description

Marks a position in a dataset: a call to '`dap_rewind`' rewinds the dataset to that position.

SPECIFICATION

dap_rewind()

Description

Rewinds the input dataset to the location saved by the most recent call to ‘`dap_mark`’.

3.3 Partitioning a dataset

SPECIFICATION

dap_list(*variable-list*, *partv*, *npartv*)
variable-list: string
partv: array of int
npartv: int
Returns: int

Description

Fills the array *partv* with the indices of the at most *npartv* variables whose names are in the space-separated list *variable-list*. Returns the number of variables actually listed.

Code fragment(s)

With the input dataset sorted by the values of the variables ‘`first`’ and ‘`last`’, prepare ‘`namev`’ for use in ‘`dap_head`’ or ‘`dap_newpart`’:

```
int namev[2];
dap_list("first last", namev, 2);
```

In the function ‘`foo`’, prepare ‘`varv`’ for use in ‘`dap_head`’ or ‘`dap_newpart`’:

```
foo(char *varlist)
  {
    int varv[3];
    int numvars;

    numvars = dap_list(varlist, varv, 3);
    ...
  }
```

SPECIFICATION

dap_swap()

Description

Swaps the two most recently read input file or dataset lines so that the program can complete processing of a part of that file or dataset after it detects the beginning of a new part.

SPECIFICATION

dap_newpart(*partv*, *npartv*);
partv: array of int
npartv: int
Returns: int

Description

Returns 1 if the beginning of a new part, or the end of the file or the dataset, has been reached, 0 otherwise.

A collection of variables can be used to define parts of a dataset for which all the variables in that collection are constant. For example, if a dataset '`names`' has a variable '`first`' that gives a person's first name and a variable '`last`' that gives a person's last name, then sorting the dataset by '`last`' and '`first`' will define parts, each of which contains all the records for one person (assuming no two people in the dataset have the same first and last name). The following example program fragment counts how many records belong to each person; it is assumed that the dataset has previously been sorted by '`last`' and '`first`'. (This example is for illustrative purposes only: the same task can be done in one line with the '`means`' function.)

Code fragment(s)

```
inset("names")
  {
    int nrecords;   /* count of records */
    int partv[2];   /* array of numerical ids for variables */
    int more;       /* flag: is there more input? */

    dap_list("last first", partv, 2);  /* sets up numerical ids
*/
    for (nrecords = 0, more = 1; more; nrecords++)
      {
        more = step();             /* read 1 line or end of dataset
*/
        if (dap_newpart(partv, 2)) /* test if beginning of part
                                    * defined by last and first
                                    */
          {
            dap_swap();      /* back to last line in previous
part */
            dap_head(partv, 2);  /* display header for that part
*/
            fprintf(dap_lst, "%d records\n", nrecords);
                                 /* display the number of records
*/
            dap_swap();          /* back to beginning of new part
                                  * or end of dataset
                                  */
            nrecords = 0;        /* re-initialize n for new part
*/
          }
      }
  }
```

3.4 Modifying output

SPECIFICATION

dap_save()

Description

Saves the most recently read input file or dataset line so that the values of the variables can be modified for an additional output.

SPECIFICATION

dap_rest()

Description

Restores the most recently read input file or dataset line that was saved by '`dap_save`'.

3.5 Manage datasets

SPECIFICATION

dataset(*oldname*, *newname*, *action*)
oldname, *newname*, *action:* string

proc datasets;
append {base | out}=*newname* [{data | new}=*oldname*];
change *oldname-1*=*newname-1* [... *oldname-n*=*newname-n*];
delete *filename-1* ... *filename-n*;

Description

Performs the specified action on the datasets *oldname* and, if the action is not '`REMOVE`', *newname*. The string *action* must contain one of the words '`APPEND`', '`RENAME`', '`COPY`', '`FILL`', or '`REMOVE`'.

'`APPEND`' appends the dataset *oldname* at the end of *newname*. For '`APPEND`', only those variables existing in *newname* are retained from *oldname* and if a variable in *newname* does not appear in *oldname*, the value of that variable is set to the null string if of type string or 0 if of type int or double.

'`RENAME`' changes the name of the dataset from *oldname* to *newname*.

'`COPY`' may be followed by a space-separated list of variable specifications, each of which has the form *variable-name* or *variable-name* > *new-variable-name*. In either case, only the variables listed will be retained in the copy of the dataset and in the second case, those variables will be renamed as indicated. If *variable-name* refers to an array, then no index may be specified and the entire array is retained and, if indicated, renamed.

'`FILL`' makes a copy, named *newname*, of the dataset *oldname* with the values in missing cells filled by 0.0 for variables of type double, 0 for variables of type int, and the null-string '`""`' for variables of type string. '`FILL`' must be followed by two space-separated lists which are separated by a '`:`'. The first list names the variables whose values are to be filled and the second list names the variables that classify the cells in the dataset. The dataset must be sorted by the classification variables in that order.

'`REMOVE`' deletes the dataset *oldname*.

Code fragment(s)

Append the contents of the dataset `tweedledee` to the dataset `tweedledum`:

```
dataset("tweedledee", "tweedledum", "APPEND");
```

Change the name of the dataset `tweedledee` to `tweedledum`:

```
dataset("tweedledee", "tweedledum", "RENAME");
```

Make a copy of the dataset `tweedledee` and name the new dataset `tweedledum`:

```
dataset("tweedledee", "tweedledum", "COPY");
```

Make a copy of the dataset `tweedledee` but retain the values of '`alice`' and '`mushroom`' only, and name the new dataset `tweedledum`:

```
dataset("tweedledee", "tweedledum", "COPY alice mushroom");
```

Make a copy of the dataset `tweedledee`, retain the values of '`alice`' and '`mushroom`' only, name the new dataset `tweedledum`, and change the name of '`alice`' to '`tall`' in the new dataset:

```
dataset("tweedledee", "tweedledum", "COPY alice>tall mushroom");
```

Make a copy, named `tweedledum`, of the dataset `tweedledee`, which is sorted by '`SES`', '`race`', and '`gender`', with the missing cells filled with 0.0 for the variable '`count`':

```
dataset("tweedledee", "tweedledum", "FILL count : SES race gender");
```

Remove the dataset `tweedledee`:

```
dataset("tweedledee", "", "REMOVE");
```

Chapter 4

Managing and displaying data

These functions manipulate or display the data within a dataset (not a data file).

4.1 Sorting a dataset

SPECIFICATION

> **sort(**_dataset_**,** _variable-list_**,** _modifiers_**)**
> _dataset_, _variable-list_, _modifiers:_ string
>
> proc sort [data=_dataset-name_] [out=_dataset-name_] [nodupkey];
> by [descending] _variable-1_ [... [descending] _variable-n_]

Description

Writes to _dataset_.srt the result of sorting _dataset_ in the order specified by the space-separated, ordered list of variables _variable-list_, according to _modifiers_. The original dataset is left unchanged. If _modifiers_ is the null string, then all variables are sorted in increasing order and all sorted lines are kept. Otherwise, _modifiers_ is a space separated list of '`u`' and/or an ordered string of '`i`' and '`d`' (no spaces) whose length equals the number of variables in _variable-list:_ '`u`' means retain only one record in a group of lines for which all the values of all the variables in _variable-list_ are identical; '`i`' indicates that the corresponding variable will be sorted in increasing order and '`d`' indicates decreasing order. NOTE: sort allocates enough memory for sorting small to moderately large disk files efficiently; for very large files, consider setting DAP_MAXMEM (see Chapter 9.6 [Appendix I], page 91).

Code fragment(s)

Create a copy of the dataset `people` _sorted by '_`height`_' in ascending order and by '_`weight`_' in ascending order within groups with equal '_`height`_' and name the new dataset_ `people.srt`_:_

```
sort("people", "height weight", "");
```

Create a copy of the dataset `people` *sorted by '*`height`*' in descending order and by '*`weight`*' in ascending order within each group of lines with constant '*`height`*' and name the new dataset* `people.srt`*:*

```
sort("people", "height weight", "di");
```

Create a copy of the dataset `people` *sorted by '*`last`*' in ascending order and by '*`weight`*' in ascending order within each group of lines with constant '*`height`*', but retain only one line from each group of lines with the constant '*`last`*' and '*`first`*', and name the new dataset* `people.srt`*:*

```
sort("people", "last first", "u");
```

Create a copy of the dataset `people` *sorted by '*`last`*' in descending order and by '*`weight`*' in descending order within each group of lines with constant '*`height`*', but retain only one line from each group of lines with the constant '*`last`*' and '*`first`*', and name the new dataset* `people.srt`*:*

```
sort("people", "last first", "u dd");
```

4.2 Merging datasets

SPECIFICATION

merge(*dataset1*, *variable-list1*, *dataset2*, *variable-list2*, *partvars*, *outputset***)**
dataset1, *variable-list1*, *dataset2*, *variable-list2*, *partvars*, *outputset:* string

Description

Performs a many-to-one or one-to-one merge of *dataset1* and *dataset2* and writes the result to *outputset*. (For usage in an SBS '`data`' step, see See Chapter 9.9 [Appendix IV], page 103.) A variable is renamed in the output dataset if the variable name is followed by '> *newname*'. Only the variables in *variable-list1*, renamed if requested, are included from *dataset1* and only the variables in *variable-list2*, renamed if requested, are included from *dataset2*; these two variable lists must be disjoint. Individual array elements may not be referenced in the variable lists; array names must appear without bracketed indices. If either variable list is a '`NULL`' pointer, then no variables are included from that dataset; this can be used to select observations from a dataset. If either variable list is the null string (`""`, not '`NULL`'), then all variables from that dataset are included. If either variable list contains the character '`!`' before the first variable name, then the named variables are excluded from, not included in, the output dataset. If *partvars* is not the null string or '`NULL`', then matching is based on the variables in *partvars:* for each part of the datasets *dataset1* and *dataset2* specified by the variables in *partvars*, the first observation in *dataset2* is matched with all observations from *dataset1*; both datasets must be sorted by *partvars* before merging. Otherwise, observations in the two datasets are matched one-to-one until one or both datasets are exhausted.

Code fragment(s)

Merge datasets `pretreat` *and* `posttreat` *into a new dataset '*`changes`*' by matching lines with corresponding values of '*`name`*'; retain the values of '*`name`*' and '*`preweight`*' from* `pretreat` *and the values of '*`postweight`*' from* `posttreat`*:*

```
merge("pretreat", "name preweight",
      "posttreat", "postweight", "name", "changes");
```

Merge datasets `pretreat` *and* `posttreat` *into a new dataset '*`changes`*' by matching lines with corresponding values of '*`name`*'; retain the values of '*`name`*' and '*`weight`*' from* `pretreat` *and the values of '*`weight`*' from* `posttreat`*, but change the name of '*`weight`*' from* `pretreat` *to '*`preweight`*' and the name of '*`weight`*' from* `posttreat` *to '*`postweight`*' in the dataset* `changes`*:*

```
merge("pretreat", "name weight>preweight",
      "posttreat", "weight>postweight", "name", "changes");
```

4.3 Displaying a dataset

SPECIFICATION

print(*dataset*, *variable-list***)**
dataset, *variable-list:* string

proc print [data=*dataset-name*];
var *variable-list*;

Description

Displays the values of variables in *dataset*. If *variable-list* is NULL or the null string or a string containing only tabs or only commas, the values of all variables are printed, otherwise just those of the named variables are printed. If the variables in *variable-list* are separated by spaces, then output contains value of the '`_type_`' variable and the observation number. Otherwise, the variables must be separated by tabs or commas and the value of the '`_type_`' variable and the observation number are omitted. No special handling is performed for strings containing tabs or commas. **Code fragment(s)**

Display the values of all the variables of the dataset `names`*:*

```
print("names", "");
```

Display the values, separated by tabs, of all the variables of the dataset `names`*:*

```
print("names", "\t");
```

Display the values of '`first`*' and '*`last`*' of the dataset* `names`*:*

```
print("names", "first last");
```

Display the values, separated by commas, of '`first`*', '*`middle`*', and '*`last`*' of the dataset* `names`*:*

```
print("names", "first, middle,last");
```

SPECIFICATION

table(*dataset*, *row-vars*, *col-vars*, *format*, *partvars*)
dataset, *row-vars*, *col-vars*, *format*, *partvars:* string

proc tabulate [data=*dataset-name*] [format=*width.places*];
[class *class-variable-list*;]
[var *analysis-variable*;]
table *row-variable-1* [... *row-variable-n*],
column-variable-1 [... *column-variable-n*] * *analysis-variable*
[/ {rtspace | rts}=*number*];
by *variable-list*;

Description

Displays values from *dataset* in tabular form. The values of the variables in *row-vars* label the rows hierarchically and the values of the variables in *col-vars* label the columns hierarchically except that the last variable in *col-vars* is the variable of type double whose values are displayed in the cells of the table. The string *format* is of either of the two forms, *width.precision* or *width*. The first specifies that *width* characters should be allotted for each cell and *precision* digits should be displayed to the right of the decimal point. The second only specifies the number of characters to be allotted for each cell and lets the program choose the "best" allocation of decimals. The '`table`' function does not sort the data except that it sorts the columns if an '`s`' appears before the *width.precision* specification; thus, the data must be sorted by the *partvars* (if any), *row-vars*, and *col-vars* before calling '`table`'. The *width.precision* can also be followed by a number indicating how many columns to allocate for the combined row labels, including column dividers. '`Table`' does not process the data and expects at most one value of the numerical variable for each distinct combination of the row and column variables; see function '`split`' (see Section 4.4 [Splitting and joining lines], page 27) for datasets with multiple numerical values to display.

SBS note: In '`proc tabulate`', the '`class`' and '`var`' statements have no effect and may be omitted.

Code fragment(s)

Display a table, with cells 6 characters long and values displayed to 1 decimal place, of the values of '`height`' with the rows labeled by '`last`' and '`first`' and the columns labeled by '`county`':

```
table("names", "last first", "county height", "6.1", "");
```

Display tables, one for each '`state`', with cells 6 characters long and values displayed to 1 decimal place, of the values of '`height`' with the rows labeled by '`last`' and '`first`' and the columns labeled by '`county`':

```
table("names", "last first", "county height", "6.1", "state");
```

Display a table, with cells 6 characters long and values displayed to 1 decimal place, of the values of '`height`' with the rows labeled by '`last`' and '`first`' and the columns labeled by '`county`', with '`county`' sorted in ascending order:

```
table("names", "last first", "county height", "s6.1", "");
```

4.4 Splitting and joining lines

SPECIFICATION

> **split**(*dataset*, *variable-list*, *class-value-vars*)
> *dataset*, *variable-list*, *class-value-vars:* string

Description

For each line of *dataset*, writes to *dataset*.spl one line for each variable in *variable-list*, with the value of the first variable in *class-value-vars* set to the name of that variable and the value of the second set to the value. All the variables in *variable-list* must have the same type, which will be the type of the second variable in *class-value-vars*. All variables in *dataset* except those *variable-list* appear in *dataset*.spl, in addition to the two new variables named in *class-value-vars*. This function is useful for preparing a dataset for function '`table`'.

Code fragment(s)

Create a new dataset `people.spl` *that is a copy of* `people` *except that each person's height and weight are on consecutive lines so that '*`table`*' can be called to display both values on the same line:*

```
split("people", "height weight", "statname statistic");
table("people.spl", "name", "statname statistic", "6.0", "");
```

SPECIFICATION

> **join**(*dataset*, *partvars*, *value-var*)
> *dataset*, *partvars*, *value-var:* string

Description

For consecutive lines in *dataset* distinguished by values of the last variable named in *partvars*, which must be a string variable, writes one line to *dataset*.joi. In each part of *dataset* specified by all but the last variable named in *partvars*, for each distinct value of the last variable named in *partvars*, the new dataset *dataset*.joi contains one new variable whose name is that value and whose value is set to the value of *value-var* in the corresponding line of *dataset*. *dataset* must be sorted by *partvars* and the same set of values of the last variable named in *partvars* must occur in every part defined by all the variables but the last in *partvars*. Neither the last variable name in *partvars* nor the variable *value-var* appears in *dataset*.joi. This function is useful, for example, for preparing a dataset for function '`plot`'.

Code fragment(s)

Create a new dataset `people.joi` *that is a copy of* `people` *except that each person's height and weight, which were given as values of the variable '*`statistic`*' of type double, on separate lines identified by the values '*`height`*' and '*`weight`*' of the string variable '*`statname`*', now appear as variables named '*`height`*' and '*`weight`*' on a single line:*

```
join("people", "city name statname", "statistic");
nport(plot("people.joi", "height weight", "city",
           "", NULL, NULL, 3), 3, 1);
```

4.5 Grouping lines in a dataset

SPECIFICATION

> **group(***dataset*, *variable-specification*, *partvars***)**
> *dataset*, *variable-specification*, *partvars:* string
>
> proc rank [data=*dataset-name*] [out=*dataset-name*]
> [{fraction | f | percent | p | groups=*number*}];
> by *variable-list*;
> [ranks *new-variable-list*;]
> var *variable-list*;

Description

Writes to *dataset*.grp the observations of *dataset*, grouped according to the *variable-specification*. Groups are numbered consecutively and are formed as specified by *variable-specification*. *dataset* only needs to be sorted by *partvars*, if there are any.

If *variable-specification* begins with either '`#`', '`/`', or '`%`' then for each observation, *group* computes either the number of the observation or the fraction of the total number of observations or the percent of the total number of observations, respectively. '`#`', '`/`', or '`%`' may be followed by a space and one of '`+`', '`-`', '`+1`', '`+0`', '`-1`', or '`-0`' to indicate ascending order, starting from 0, descending order, ending at 0, ascending order, starting from 1, ascending order, starting from 0, descending order, ending at 1, or descending order, ending at 0, respectively; the default is ascending order, starting from 1. This may in turn may be followed by a space-separated list of variables: only those observations of the dataset for which all of these variables have a finite (i.e., non-NaN) value are given group numbers and the remaining observations are given a group number of NaN.

If *variable-specification* does not begin with '`#`', '`/`', or '`%`', then it must be a space-separated list of *variable-name ngroups-method*, in which *ngroups-method* is either an integer followed by '`^`' (no space in between) for that number of groups of equal width from the smallest observation to the largest or an integer followed by '`#`' (no space in between), for that number of groups of (approximately) equal counts from the smallest observation to the largest. Only those observations for which all the grouping variables have finite (i.e., non-NaN) values are given group numbers and the remaining observations are given a group number of NaN. The numbering of the groups starts at 1.

In addition to all variables in *dataset*, the output dataset contains for each grouping variable, which must be of type double, a new variable whose name is formed by preceding the original variable name with an underscore ('`_`'); this new variable, which is also of type double, contains the group number of the value of the original variable. If only '`#`', '`/`', or '`%`', was requested, then the new variable is named '`_N_`'.

SBS note: In '`proc rank`', the '`ranks`' statement is ignored: the output data set always contains new variables for the ranks, as in the dap function '`group`'.

Code fragment(s)

Create a new dataset `people.grp` *that is a copy of* `people` *except that it has an additional variable named '*`_N_`*' that contains the line number, starting at 1:*

```
group("people", "#", "");
```

Create a new dataset `people.grp` *that is a copy of* `people` *except that it has an additional variable named '*`_N_`*' that contains the line number, starting at 0:*

```
group("people", "# +0", "");
```

Create a new dataset `people.grp` *that is a copy of* `people` *except that it has an additional variable named '*`_height`*' that contains a number from 1 to 5 that indicates the quintile of the value of '*`height`*':*

```
group("people", "height 5#", "");
```

Create a new dataset `people.grp` *that is a copy of* `people` *except that it has an additional variable named '*`_height`*' that contains a number from 1 to 5 that indicates which of 5 equal length intervals, starting at the smallest value of '*`height`*' and ending at the largest, contains the value of '*`height`*':*

```
group("people", "height 5^", "county");
```

4.6 Trimming extreme values

SPECIFICATION

trim(*dataset*, *trim-specification*, *partvars***)**
dataset, *trim-specification*, *partvars:* string

Description

Writes to *dataset*.trm the observations from *dataset* for which the values of all of the variables in *trim-specifications* do not lie in the upper or lower percent indicated in *trim-specifications*. The string *trim-specifications* consists of space-separated pairs of *variable-name percent*. The variables listed in *trim-specifications* must be of type double.

Code fragment(s)

Create a new dataset named `wheat.trm` *that contains only those values of '*`yield`*' that do not lie in the upper or lower 5% of all the values of '*`yield`*':*

```
trim("wheat", "yield 5", "");
```

Create a new dataset named `wheat.trm` *that contains only those lines of* `wheat` *whose value of '*`yield`*' does not lie in the upper or lower 5% of all the values of '*`yield`*' and, simultaneously, whose value of '*`nitrogen`*' does not lie in the upper or lower 10% of all the values of '*`nitrogen`*':*

```
trim("wheat", "yield 5 nitrogen 10", "");
```

Chapter 5

Statistics functions

5.1 Survey selection

SPECIFICATION

> **surveyselect**(*dataset*, *variable-specification*, *stat-list*, *partvars*)
> *dataset*, *variable-specification*, *stat-list*, *partvars:* string
>
> proc surveyselect [data=*dataset-name*] [out=*out-name*] [method=*method-name*] [n=*n-name*];

Description

survey selection on a dataset. method SRS and SYS are implemented. Method SRS take [n] values in the dataset by randomization. Method SYS take [n] values in the dataset with a systematic algorithm starting with a simple random.

```
proc surveyselect data =tPres2007 method=SRS  n = 220 out=sasPres2007
;
run ;
```

5.2 One variable statistics

SPECIFICATION

> **means**(*dataset*, *variable-specification*, *stat-list*, *partvars*)
> *dataset*, *variable-specification*, *stat-list*, *partvars:* string
>
> proc means [data=*dataset-name*] [noprint] [*statistics-list*]
> [vardf={df | wdf}];
> var *variable-list*;
> weight *variable*;
> by *variable-list*;
> output [out=*dataset-name*];

Description

Writes to *dataset*.mns the statistics requested in *stat-list* for the variables in *variable-specification*, which must be of type double. The '`means`' function does not display its results in the `lst` file; use '`print`' or '`table`' (Section 4.3 [Displaying a dataset], page 25) after calling '`means`' to display the results from *dataset*.mns. The statistics are given as values of the variables named in *variable-specification* in *dataset*.mns. The string *stat-list* is a space separated list of one or more of the following statistics:

MAX

the maximum of the variable values

MEAN

the mean of the variable values

MIN

the minimum of the variable values

N

the number of finite observations

RANGE

the maximum minus the minimum of the variable values

SD or **STD**

the sample standard deviation of the variable values (df = number of observations - 1)

SDFREQ

the weighted sample standard deviation of the weighted variable values (df = sum of weights - 1)

SEM or **STDERR**

the sample standard error of the mean of the variable values (df = number of observations - 1)

SEMFREQ

the weighted sample standard error of the mean of the variable values (df = sum of weights - 1)

STEPxxxx

a sequence of *xxxx* + 1 equally spaced numbers from the minimum to the maximum of the variable values, with xxxx an integer of at most 4 digits

SUM

the sum of the variable values

SUMWT

the sum of the weight variable values

T

the value of the t-statistic for the variable values

TPROB or **PRT**

the (two-tailed) probability that the absolute value of the t-statistic would equal or exceed the observed value

VAR

the sample variance of the variable values (df = number of observations - 1)

VARFREQ

the weighted sample variance of the variable values (df = sum of weights - 1)

VARM

the sample variance of the mean of the variable values (df = number of observations - 1)

VARMFREQ

the weighted sample variance of the variable values (df = sum of weights - 1)

For each observation in the output data set, the '`_type_`' variable names the statistic whose value is given for the requested variable. If the only statistic requested is '`N`', then *variable-list* may consist of a single variable which need not be present in the input data set. The string *variable-specification* is a space-separated list of variables except that each weight variable is preceded by an '`*`'. For example,

```
w x * u y z * v
```

indicates that each value of w and x is weighted by the value of u and that each value of y and z is weighted by the value of v.

For each variable separately, means discards values for which either the value itself or its weight value, if any, is a NaN. The number of discarded values for each variable is reported in the `log` file.

Code fragment(s)

For each distinct value of the variable '`plot`', compute and display the number of observations in dataset `wheat` and the mean and variance of the values of '`yield`' and '`height`':

```
means("wheat", "yield height", "N MEAN VAR", "plot");
table("wheat.mns", "plot", "_type_ yield", "s6.1", "");
table("wheat.mns", "plot", "_type_ height", "s6.1", "");
```

For each distinct value of the variable '`county`', compute and display the number of observations in dataset `wheat` and the mean and variance of the value of '`meanyield`', weighted by '`acres`':

```
means("wheat", "meanyield * acres", "N MEAN VAR", "county");
table("wheat.mns", "county", "_type_ meanyield", "s6.1", "");
```

For each distinct value of the pair of variables '`last`' and '`first`', compute the number of observations in dataset `names`:

```
means("names", "nrecords", "N", "last first");
```

SPECIFICATION

pctiles(*dataset*, *variable-specification*, *stat-list*, *partvars***)**
dataset, *variable-specification*, *stat-list*, *partvars:* string

proc univariate [data=*dataset-name*] [noprint] [normal] [plot];
var *variable-list*;
by *variable-list*;
weight *variable*;
output [out=*dataset-name*] [*statistics-list*];

Description

Writes to *dataset*.pct the statistics requested in *stat-list* for the variables *variable-specification*, which must be of type double. The '`pctiles`' function does not display its results in the `lst` file; use '`print`' or '`table`' (Section 4.3 [Displaying a dataset], page 25) after calling '`pctiles`' to display the results from *dataset*.pct. The statistics are given as values of the variables named in *variable-specification* in *dataset*.pct. The string *stat-list* is a space separated list of one or more of the following statistics:

MAX

the maximum

MED or **MEDIAN**

the median

MIN

the minimum

N

the number of observations

P1

the 1% point

P5

the 5% point

P10

the 10% point

P90

the 90% point

P95

the 95% point

P99

the 99% point

Q1

the first quartile

Q3

the third quartile

QRANGE

the interquartile range

RANGE

the range

Up to 9 additional percentile points can be specified in the form 'P*xxxxx*' in which '*xxxxx*' is a number, which may contain up to 5 characters, including a decimal point.

The string *variable-specification* is a space-separated list of variables except that each weight variable is preceded by an '`*`'. For example,

```
w x * u y z * v
```

indicates that each value of *w* and *x* is weighted by the value of *u* and that each value of *y* and *z* is weighted by the value of *v*.

Code fragment(s)

Compute and display the 10th, 25th, 50th (median), 75th, and 90th percentiles of '`income`' for each '`county`' in the dataset `people`:

```
pctiles("people", "income", "P10 P25 MED P75 P90", "county");
table("people.pct", "county", "_type_ income", "s7.2", "");
```

Compute and display the 37.5th, 50th (median), and 62.5th percentiles of '`income`' for each '`county`' in the dataset `people`:

```
pctiles("people", "income", "P37.5 MED P62.5", "county");
table("people.pct", "county", "_type_ income", "s7.2", "");
```

5.3 Correlations

SPECIFICATION

corr(*dataset*, *variable-list*, *partvars***)**
dataset, *variable-list*, *partvars:* string

proc corr [data=*dataset-name*] [outp=*dataset-name*] [noprint];
var *variable-list*;
by *variable-list*;

Description

Writes to *dataset*.cor the correlation statistics and their significances for all pairs of the variables, each of which must be of type double, in *variable-list*. The '`corr`' function does not display its results in the `lst` file; use '`print`' or '`table`' (Section 4.3 [Displaying a dataset], page 25) after calling '`corr`' to display the results from *dataset*.cor. The output dataset contains the variables '`_var1_`', '`_var2_`', and '`_corr_`' in addition to all the variables in *partvars*. There is one pair of correlation and significance values for each pair of numeric variables. For each line of the output data set, '`_var1_`', and '`_var2_`', contain the names of the variables being reported on and '`_corr_`' contains the number of observations, the correlation coefficient, or its significance, as indicated by the values '`N`', '`CORR`', and '`PCORR`' of the '`_type_`' variable, respectively.

Code fragment(s)

Compute and display the correlations and their signficances for each pair of the variables '`height`', '`weight`', and '`income`':

```
corr("people", "height weight income", "");
sort("people.cor", "_type_ _var1_ _var2_", "");
table("people.cor.srt", "_var1_", "_var2_ _corr_", "6.2", "_type_
");
```

5.4 Analysis of variance

SPECIFICATION

effects(*dataset*, *variable-list*, *model*, *partvars***)**
dataset, *variable-list*, *model*, *partvars:* string

proc glm [data=*dataset-name*];
class *variable-list*;
model *response-variable* = *effects-list*;
by *variable-list*;
contrast "*label*" *effect coefficient-list* [/ e=*effect*];
lsmeans *effect-list* / [e=*effect*] [alpha=*n*] {DUNNETT | TUKEY | LSD};
test h=*effect-list* e=*effect*;

Description

For *dataset*, created by *means* with '`N`', '`MEAN`', and '`VAR`', but no other statistics requested, *effects* constructs a dataset named *dataset*.con appropriate for '`ftest`' and then calls '`ftest`' on the set of contrasts specified by all the terms together in *model*. '`Ftest`' can be used on *dataset*.con to perform F-tests on the contrasts that correspond to individual terms in the model. The first variable in *variable-list* is the response variable, which must be of type double, and the remainder are the treatment variables, all of which must be of type string. The string *model* is a space-separated list of terms in the model; all possible terms not in the model are considered to be in the error, in addition to the cell variances (if any). Each term is either a treatment or a cross of two or more treatments, which is indicated by an '`*`' separated list of treatments. Nesting can be specified by including an effect in a cross term without that effect appearing as a main effect. NOTE: In the current version, 3.5, nested designs must be balanced and the levels of the nested factors must be the same for each set of levels of the factors within which they are nested; presumably, this will be corrected in later versions.

The output dataset *dataset*.con contains a variable named '`_term_`', of type int, which indicates which terms the error and contrast lines refer to: each bit in '`_term_`', with the lowest order bit corresponding to the first treatment variable, indicates whether the correspondingly numbered treatment in the model is included in the term. The error, contrast, and least-squares means terms in *dataset*.con are adjusted for missing cells, but only the error terms are orthogonalized.

SBS note: All the variables used in the *effects-list* of the '`model`' statement must be named in the '`class`' statement, as dap only allows categorical, string-valued, variables for the terms in the model. Specify crossed and nested effects using '`*`' only. A '`test`' statement automatically computes an appropriate combination of mean squares from the terms in '`e=`' error term list, if possible, to test the numerator and applies the Satterthwaite approximation. An '`lsmeans`' statement that has an '`e=`' option or that appears after a '`contrast`', a '`test`', or another *lsmeans* statement automatically runs an F-test with the same denominator. In each '`contrast`' statement, the effect to be tested must be a main effect, the coefficients must appear in the sorting order of the levels of that effect, they must be integers, and they must sum to zero. For more complicated contrasts, see See Section 9.1 [Analysis of variance examples], page 75.

Code fragment(s)

Perform an analysis of variance on the completely randomized one-way layout with response variable '`yield`' and treatment '`fert`':

```
effects("wheat.srt.mns", "yield fert", "fert", "");
```

Perform an analysis of variance (F-test on the model only) on the one-way layout in a completely randomized block design with response variable '`yield`' and treatment '`fert`', blocked by '`block`', with main effects only:

```
effects("wheat.srt.mns", "yield fert block",
                   "fert block", "");
```

Perform an analysis of variance (F-test on the model only) on the two-way layout in a completely randomized block design with response variable '`yield`' and treatments '`fert`' and '`variety`', blocked by '`block`', with main effects and treatment interaction:

```
effects("wheat.srt.mns", "yield fert variety block",
                "fert variety fert*variety block", "");
```

SPECIFICATION

ftest(*dataset*, *variable-list*, *numerator*, *denominator*, *partvars*)
dataset, *variable-list*, *numerator*, *denominator*, *partvars:* string

Description

Performs an F-test on the hypothesis that all the contrasts specified by *numerator* are zero, using the terms specified by *denominator*, and the variances of the cell means, as appropriate, as the error. The string *variable-list* must list the response variable followed by all the variables in the model and error terms.

The input dataset must have the following format: for each cell, there must be a line for the mean, the number of observations for that cell, and the variance (these three may be in any order) followed by the denominator terms and then the numerator terms. These lines are identified respectively by the following values of the '`_type_`' variable: '`MEAN`', '`N`', '`VAR`', '`ERROR`', and '`CONTR`'. Additional lines, such as '`LSMEAN`', lines, may follow. The numerator and denominator terms used for the test, which need to be adjusted for missing data (as is done by effects) but which do not need to be orthogonal, are identified by the value of the '`_term_`' variable in dataset (see '`effects`'). If *numerator* is null, then all the terms identified as '`CONTR`' are used in the test. If *denominator* is null, then the cell variances and all the terms identified as '`ERROR`' are used in the test. If *denominator* is non-null, then the cell variances are not included in the test and '`ftest`' constructs an appropriate combination of mean squares from the terms in that denominator to test the numerator and applies the Satterthwaite approximation. Note, however, that *denominator* must contain all possible terms that might be needed to construct that combination of mean squares.

The function *ftest* writes out a dataset, with the suffix of *dataset* replaced by .tst, which contains only those error, contrast, and least-squares means terms appearing in dataset that appear in the test. This file is suitable for input to *lsmeans*. At the end of that file are two lines, the first with the mean-squared error appearing as the value of the response variable and the second with the error degrees of freedom appearing as the value of the response variable. These lines are identified by the _type_ variable as '`MSERROR`' and '`ERRORDF`', respectively.

Code fragment(s)

Perform an F-test on the interaction term in the in the two-way layout in a completely randomized block design with response variable '`yield`' and treatments '`fert`' and '`variety`', blocked by '`block`' as a fixed effect, with main effects and treatment interaction:

```
ftest("wheat.srt.mns.con", "yield fert variety block",
                                       "fert*variety", "", "");
```

Perform an F-test on the interaction term in the in the two-way layout in a completely randomized block design with response variable '`yield`' and treatments '`fert`' and '`variety`', blocked by '`block`' as a random effect, with main effects and treatment interaction:

```
ftest("wheat.srt.mns.con", "yield fert variety block", "fert",
                                       "block fert*block", "");
```

SPECIFICATION

lsmeans(*dataset*, *method*, *alpha*, *variable-list*, *treat*, *partvars*, *format***)**
dataset, *method*, *variable-list*, *treat*, *partvars*, *format:* string
alpha: double

Description

For *dataset* produced by '`ftest`' whose *numerator* is the same as the *treat* of '`lsmeans`', computes the least-squares means of the levels of *treat* and reports the minimum difference significant at level *alpha* and places the comparison of them according to *method* in a dataset named *dataset-name*.lsm, and displays a table of results, the cells of which are formated according to *format* as for *table*. The available methods are '`LSD`', '`TUKEY`', and '`DUNNETT`'. For '`DUNNETT`', the first level of *treat* is taken to be the control.

Code fragment(s)

Compute and test, using the LSD method with significance level 0.05, least squares means for the one-way layout in a completely randomized block design with response variable '`yield`' and treatment '`fert`', blocked by '`block`', with main effects only; the table of results is displayed with cells 6 characters long and values to 1 decimal place:

```
lsmeans("data.srt.mns.tst", "LSD", 0.05,
                    "yield treat block", "block", "", "8.4");
```

5.5 Categorical data analysis

SPECIFICATION

freq(*dataset*, *variable-list*, *stat-list*, *partvars***)**
dataset, *variable-list*, *stat-list*, *partvars:* string

proc freq [data=*dataset-name*];
by *variable-list*;
tables *variable-1* [* *variable-2* [... * *variable-n*]] /
[out=*dataset-name*] [noprint] [nofreq] [noprecent]
[norow] [nocol] [*statistics-list*];
weight *variable*;

Description

Writes dataset *dataset*.frq with the variable '`_cell_`', of type double, set equal to the count, fraction, percent, or expected value under independence, as requested by *stat-list* for the cells with distinct values of the variables in *variable-list*. The dataset *dataset* must be sorted by *variable-list*. The '`freq`' function does not display its results in the `lst` file; use '`print`' or '`table`' (see Section 4.3 [Displaying a dataset], page 25) after calling '`freq`' to display the results from *dataset*.frq. If *variable-list* contains a count-variable, which must appear last in the list and must be preceded by an '`*`' and which must be of type double, then the count, fraction, or percent is weighted by the values of that count-variable for that cell. The string *stat-list* is a space separated list of one or more of the following statistics:

CHISQ

for 2-dimensional tables, compute Pearson's chi-squared statistic and test

CMH

for tables of dimension 3 or higher, compute the Cochran-Mantel-Haenszel statistic and test, with strata defined by the levels of all but the last two variables in *variable-list*

COLPERC

the percent of the number of observations in the column

COUNT

the number of observations

EXPECTED

for 2-dimensional tables, the expected table entry under independence, conditioned on the marginal totals

FISHER

for 2x2 tables, computes Fisher's exact test

FRACTION

the fraction of the total number of observations

NOMINAL

for 2-dimensional tables, compute the following measures of association of nominal variables and their asymptotic standard errors: uncertainty coefficients

ODDSRAT

for 2x2 tables, computes the odds ratio

ORDINAL

for 2-dimensional tables, compute the following measures of association of ordinal variables and their asymptotic standard errors: gamma, Kendall's Tau-b, and Somers' D (column on row and row on column)

PAIR

for 2-dimensional tables, compute measures of association for matched pairs and their asymptotic standard errors

PERCENT

the percent of the total number of observations

ROWPERC

the percent of the number of observations in the row

For '`CHISQ`', '`FISHER`', '`ODDSRAT`', and '`ORDINAL`', the requested statistic and, if applicable, test is displayed in the `lst` file. For '`COUNT`', '`EXPECTED`', '`FRACTION`', and '`PERCENT`', the requested '`_cell_`' values are indicated in *dataset*.frq by the corresponding value of the '`_type_`' variable. The input dataset must be sorted by the variables named in *partvars* and *variable-list*.

Code fragment(s)

For each '`state`' and '`county`' in the dataset `people`, compute and display the cell percentages for a two-way table whose rows are labeled by '`education`' and whose columns are labeled by '`education`':

```
freq("people", "education income", "PERCENT", "state county");
table("people.frq", "education", "income _cell_",
                                        "3.0", "state county");
```

For each '`county`' in the dataset `people`, compute and display the cell counts for a two-way table whose rows are labeled by '`education`' and whose columns are labeled by '`education`'; also compute and display the cell counts expected under independence and the Chi-squared statistic and its significance:

```
freq("people", "education income", "COUNT EXPECTED CHISQ", "county");
sort("people.frq", "county _type_ education income", "");
table("people.frq.srt", "education", "income _cell_",
                                        "6.0", "county _
type_");
```

For each '`state`' and '`county`' in the dataset `people`, compute and display the cell percentages within row for a two-way table whose rows are labeled by '`education`' and whose columns are labeled by '`education`':

```
freq("people", "education income", "ROWPERC", "state county");
table("people.frq", "education", "income _cell_",
                                            "3.0", "state county");
```

For the dataset `rabbits.srt`*, compute the Cochran-Mantel-Haenszel statistic and its significance for a three-way table that is stratified by '*`penicillin`*' and, within each stratum, whose rows are labeled by '*`delay`*' and whose columns are labeled by '*`response`*' and whose cell counts are given by '*`count`*'; :*

```
freq("rabbits.srt", "penicillin delay response*count", "CMH",
"");
```

SPECIFICATION

categ(*dataset*, *variable-list*, *aux-variable-list*, *expect*, *param*, *select*, *partvars*, *trace*)
dataset, *variable-list*, *aux-variable-list*, *select*, *partvars*, *trace:* string
expect: pointer to function returning double
param: array of double

Description

'`Categ`' fits the model specified by the function *expect* and the parameter selection string *select* to the data by the method of maximum likelihood, reports goodness of fit statistics from the comparison of the specified model with either a reduced model or the saturated model, writes the observed and fitted cell counts to *dataset*.cat, with the '`_type_`' variable set to '`OBS`' and '`FIT`', respectively, and the estimates and the covariance matrix to *dataset*.cov, with the '`_type_`' variable set to '`ESTIMATE`' and '`COVAR`', respectively, the '`_param1_`' and '`_param2_`' variables indicating which of the parameters named in *select* identify the row and column, and the '`_cov_`' variable containing the numerical value. The first variable named in *variable-list* is the cell count for the cells. Cells are classified by the remaining variables in *variable-list*. Additional auxiliary variables, such as the total number of observations or marginal totals, that are required for computing expected cell counts (see '`expect`' below) can be listed in *aux-variable-list*. All variables in *variable-list* and in *aux-variable-list* must be coded as double; this allows '`categ`' to work with continuous and categorical variables together. The input dataset must be sorted according to the classification variables in *variable-list* in the order that they appear there. If non-null, the trace option *trace* is passed to '`dap_maximize`' for tracing the maximum likelihood iteration.

The string *select* is a space-separated sequence of parameter names, each one optionally preceded by either a '`!`' to indicate that it is to omitted from the model or '`?`' to indicate that it should be omitted from the reduced model only. The number of parameters must equal the dimension of the *param* array and the order corresponds to the order of entries in that array. Note that the parameter names need not correspond to variables in the dataset, they are used only to identify their estimates in the output. If there are no '`?`'s, then the goodness of fit statistics computed are relative to the saturated model. If there are '`?`'s, then the goodness of fit statistics are computed relative to the reduced model obtained by setting all the '`?`'s to '`!`'s.

'`Categ`' assumes that the parameters in *param* are independent and computes the degrees of freedom as the number of cells minus the number of parameters. See '`estimate`' (see Section 5.9 [Statistics utilities], page 47) to see how to obtain statistics on the remaining parameters or contrasts. The initial values of the parameters for the maximum likelihood estimation must be supplied in the call to '`categ`'; they are not modified by '`categ`'. The function pointed to by *expect* must take two parameters, an array of parameter values, with dimension equal to the number of parameters specified by the string *select*, followed by an array of classification, and possibly auxiliary, values, with dimension equal to the number of classification and auxiliary variables in *variable-list* and *aux-variable-list*, and return the expected number of observations for the indicated cell, based on the parameter, classification, and auxiliary (if any) values supplied.

Note: '`categ`' creates or overwrites the dataset *dataset*.fil to create a dataset with no missing cells for the analysis.

Code fragment(s)

Use the user-defined function '`expect`' to fit and test a loglinear model and print a table of the observed and fitted cell counts; in the dataset `deathpen`, the variable '`n`' gives the cell counts for the three-way table classified by '`def`', '`vic`', and '`pen`' and '`param`' is an array of 7 independent parameters:

```
double expect(double param[8], double class[2]);
categ("cda262", "count income jobsat", &expect, param,
      "mu <6 6-15 15-25 VD LD MS ?Inc*Sat", "", "");
sort("cda262.cat", "income _type_ jobsat", "");
table("cda262.cat.srt", "income", "_type_ jobsat count", "6.2",
"");
```

SPECIFICATION

loglin(*dataset*, *variable-list*, *model0*, *model1*, *partvars*)
dataset, *variable-list*, *model0*, *model1*, *partvars:* string

Description

The function '`loglin`' fits the hierarchical loglinear models specified by *model0* and *model1* and compares them. The first variable named in *variable-list* gives the cell counts and must be of type double, the remainder give the classification and must be strings. The strings *model0* and *model1* consist of one or more space-separated terms, each term of which is a '`*`'-separated list of classification variables. The model specified by *model1* must incorporate at least as many effects as the one specified by *model0*. If *model0* is the null string (not NULL), then *model1* is compared to the saturated model. The dataset *dataset* must be sorted according to the partitioning variables and the classification variables in the order listed in *variable-list*. Further, if the input dataset is partitioned, the input data for each part must have the same table layout and size. The function '`loglin`' creates the dataset *dataset*.llm and calls function '`categ`', which creates the dataset *dataset*.llm.cat, which contains the observed and fitted cell counts and is copied back to *dataset*.llm.

The output in the `lst` file reports the estimate and asymptotic standard error for each of the independent parameters of the model, using zero-sum constraints to eliminate dependent parameters.

Code fragment(s)

Compare the models '`vic*pen def*vic`' and '`def*pen vic*pen def*vic`' using counts in '`n`', which gives the cell counts for the three-way table classified by '`def`', '`vic`', and '`pen`':

```
loglin("deathpen", "n def vic pen",
       "vic*pen def*vic", "def*pen vic*pen def*vic", "");
sort("deathpen.llm", "def vic _type_ pen", "");
table("deathpen.llm.srt", "def vic", "_type_ pen n", "s6.2 30",
"");
```

5.6 Linear regression

SPECIFICATION

linreg(*dataset*, *y-variable-list*, *x0-variable-list*, *x1-variable-list*, *partvars*, *x-dataset*, *level***)**
dataset, *y-variable-list*, *x0-variable-list*, *x1-variable-list*, *partvars*, *x-dataset:* string
level: double

proc reg [data=*dataset-name*] [outest=*dataset-name*];
model *response-variables* = *explanatory-variables*;
[var *variable-list*;]
[add *variable-list*;]
by *variable-list*;
[plot *y-variable* * *x-variable*;]

Description

Performs ordinary least squares linear regression for each variable in *y-variable-list* as a function of all the variables in *x0-variable-list* and *x1-variable-list*, computes t-tests on each parameter, and and tests the full model against the model with just the variables in *x0-variable-list*. The model always includes an intercept term, which is named _intercept_ and is always included implicitly in *x0-variable-list*.

Results are displayed in the `lst` file. In addition, for each of the specified values of the variables in *x0-variable-list* and *x1-variable-list*, observed and predicted values and lower and upper confidence limits for the mean at *level* for each of the variables in *y-variables* are written to *dataset*.reg in lines identified by the respective values '`OBS`', '`PRED`', '`LOWER`', and '`UPPER`' of the '`_type_`' variable. These values are given as the values of the *y-variables* in *dataset*.reg.

If *x-dataset* is non-null, specified values for the x-variables are read from that dataset; otherwise, the values of the x-variables in *dataset* are used. Finally, the estimates and covariance matrix are written to the file *dataset*.cov, with the '`_type_`' variable set to '`ESTIMATE`' and '`COVAR`', respectively, the '`_response_`' variable indicating the response variable and the '`_param1_`' and '`_param2_`' variables indicating which parameters identify the row and column, and the '`_cov_`' variable containing the numerical value.

SBS note: If you include a '`plot`' statement, then the model must contain only one response and one explanatory variable. If there is an '`add`' statement, the model will be fit as originally specified and with the additional variables and the change in R-squared will be reported. The '`var`' statement is ignored in any case.

Code fragment(s)

Perform linear regression of the response variable '`strength`' on the independent variables '`length`' and '`thickness`', based on the values in the dataset `wires`, and compare the full model with the reduced model that contains the intercept term only:

```
linreg("wires", "strength", "", "length thickness", "", NULL,
0.0);
```

Perform linear regression of the response variable '`strength`' on the independent variables '`length`' and '`thickness`', based on the values in the dataset `wires`, and compare the full model with the reduced model that contains the intercept term and '`length`' only; also compute predicted values and the endpoints of 95% confidence intervals for the mean of the response for the values of '`length`' and '`thickness`' in the dataset `steps`:

```
linreg("wires", "strength", "length", "thickness", "", "steps",
0.95);
```

5.7 Logistic regression

SPECIFICATION

logreg(*dataset*, *y-spec*, *x0-var-list*, *x1-var-list*, *partvars*, *x-dataset-name*, *level***)**
dataset, *y-spec*, *x0-var-list*, *x1-var-list*, *partvars*, *x-dataset-name:* string
level: double

proc logistic [data=*dataset-name*] [outest=*dataset-name*];
model $\{$*variable* | *events*/*trials*$\}$ = *explanatory-variables*;
by *variable-list*;

Description

Performs logistic regression for the response variable in *y-spec* as a function of all the variables in *x0-var-list* and *x1-var-list*, and tests the full model against the model with just the variables in *x0-var-list*. The model always includes an intercept term, which is named '`_intercept_`' and is always included implicitly in *x0-var-list*. Variables in *y-spec*, *x0-var-list*, and *x1-var-list* must be of type double. Two forms for the response in *y-spec* are available: *events-variable* / *trials-number*, in which *trials-number* is an explicit number (e.g., 1 for binary data), and *events-variable* / *trials-variable*, in which *events-variable* is a count of successes in the number of trials specified by *trials-variable*.

Results are displayed in the `lst` file. In addition, for each of the specified values of the variables in *x0-var-list* and *x1-var-list*, observed and predicted probabilities and lower and upper confidence limits at *level* for the probability that the *events-variable* is 1 are written to *dataset*.lgr in lines identified by the respective values '`PRED`', '`LOWER`', and '`UPPER`' of the '`_type_`' variable. These values are given as the values of the *events-variable* in *dataset*.lgr. If *x-dataset-name* is non-null, specified values for the x-variables are read from that dataset; otherwise, the values of the x-variables in *dataset* are used. Finally, the estimates and covariance matrix are written to the file *dataset*.cov, with the '`_type_`' variable set to '`ESTIMATE`' and '`COVAR`', respectively, the '`_param1_`' and '`_param2_`' variables indicating which parameters identify the row and column, and the '`_cov_`' variable containing the numerical value.

SBS note: only binary response is allowed, although the *events*/*trials* form allows that to be expressed as the result of a binomial experiment.

Code fragment(s)

Perform logistic regression with the binomial response variable '`heartdis`', with values from 0 to '`ncases`' in each observation, on the independent variable '`bloodpress`', based on the values in the dataset `heartrisk`:

```
logreg("heartrisk", "heartdis / ncases",
                   "", "bloodpress", "", NULL, 0.0);
```

Perform logistic regression with the binomial response variable '`larvae`', with values from 0 to 10 in each observation, on the independent variable '`co2`', based on the values in the dataset `biocontrol`; also computed predicted values and the endpoints of 95% confidence intervals for the mean response:

```
logreg("biocontrol", "larvae / 10",
                   "", "co2", "", "biocontrol.mns", 0.95);
```

5.8 Nonparametric analyses

SPECIFICATION

nonparam(*dataset*, *variable-list*, *partvars***)**
dataset, *variable-list*, *partvars:* string

proc npar1way [data=*dataset-name*];
class *class-variable*;
var *variable-list*;
by *variable-list*;

Description

If *variable-list* contains the name of only one variable, which must be of type double, then '`nonparam`' tests the Wilcoxon signed rank statistic on the values of that variable. If *variable-list* contains the names of two variables, then the first must be of type double and the second must be of type string. If the second variable has two levels, then '`nonparam`' tests the Wilcoxon rank sum statistic and the Kolmogorov two-sample statistic of the values of the first variable classified by the values of the second. If the second variable has more than two levels, then '`nonparam`' tests the Kruskal-Wallis statistic of the values of the first variable classified by the values of the second. The input dataset does not need to be sorted.

SBS note: In '`proc npar1way`', the '`class`' and '`var`' statements are required and the class variable must be of type string.

Code fragment(s)

For each distince value of '`sex`', perform a Wilcoxon signed rank test on the values of '`weightchange`':

```
nonparam("diettest", "weightchange", "sex");
```

Perform a Kuskal-Wallis test on the values of '`weightchange`' as classified by '`drug`':

```
nonparam("diettest", "weightchange drug", "");
```

5.9 Statistics utilities

SPECIFICATION

estimate(*dataset*, *parameters*, *definitions*, *partvars***)**
dataset, *parameters*, *definitions*, *partvars:* string

Description

'`Estimate`' prints the estimate and standard error for each of the parameters defined in *definitions*, based on the estimates and covariance matrix provided by the dataset *dataset* for the parameters named in the space-separated list *parameters*. *dataset* is typically the '`.cov`' dataset output by a function such as '`loglin`', '`categ`', '`linreg`', or '`logreg`' and must contain, in addition to the '`_type_`' variable, variables named '`_param1_`', '`_param2_`', and '`_cov_`': the first two are string variables specifying the parameters by which the estimates and covariance matrix are indexed and the last is the entry in the estimate vector or covariance matrix. For the value '`ESTIMATE`' of '`_type_`', the variable '`_param2_`' specifies the parameter. The value '`COVAR`' of '`_type_`' indicates that the value of '`_cov_`' is an entry in the covariance matrix.

The string '`definitions`' is a space-separated list of equations of the form *param = coefficient_1 param_1 ... coefficient_n param_n*, in which *param_1* through *param_n* are the names of parameters whose estimates and covariances appear in *dataset* or a previous definition and each *coefficient* is a decimal number. A '+' is optional for positive coefficients. *coefficient_1* may be omitted if it would be a '+1'.

Code fragment(s)

Use the user-defined function '`expect`' to fit and test a saturated logit model for the 4 by 2 table with 4 rootstocks, A, I, II, and U, and two levels of disease, 0 and 1. '`param`' is an array of 8 independent parameters based on the identifiability constraints lambda_A + lambda_I + lambda_II + lambda_U = 0, lambda_d0 + lambda_d1 = 0, and mu_A + mu_I + mu_II + mu_U = 0. Finally use '`estimate`' to define the missing parameter mu_U and test the contrast diff_I_U = mu_I - mu_U:

```
double expect(double param[8], double class[2]);
categ("rootdisease", "_cell_ root disease", &expect, param,
      "mu lambda_A lambda_I lambda_II lambda_d0 mu_A mu_I mu_
II", "", "");
estimate("rootdisease.cov", "mu_A mu_I mu_II",
       "mu_U = - mu_A - mu_I - mu_II diff_I_U = mu_I - mu_U",
"");
```

Chapter 6

Graphics functions

6.1 Histogram

SPECIFICATION

> **histogram(***dataset*, *variable-list*, *partvars*, *nbars*, *style*, *xfunct*, *nplots***)**
> *dataset*, *variable-list*, *partvars*, *style:* string
> *nbars*, *nplots:* int
> *xfunct:* pointer to function returning double
> *Returns:* array of '`pict`'
>
> proc chart [data=*dataset-name*];
> by *variable-list*;
> vbar *variable* / [freq=*variable*] [levels=*n*]
> [axis=[*min*] *max*]
> [type=freq | percent | pct];

Description

Constructs an array of '`pict`' structures for drawing (see Section 6.6 [Displaying the pictures], page 55) separate histograms of the values of the first variable in *variable-list*, weighted by the second variable, if present, in *variable-list*, for each distinct set of values of *partvars*. The first variable name may be followed by a string, enclosed in back quotes (`` ` ` ``), to be used instead of the variable name as the x-axis label. Each histogram has *nbars* bars. The string *style* is either the null string or contains axis specifications as for '`pict_autoaxes`' (see Section 8.2 [Axes], page 65) and can optionally specify '`EQUAL`' or '`VARIABLE`' width bars, each of which represents either the '`COUNT`' (only for equal width bars) or '`PERCENT`' or '`FRACTION`' of the observations. If *style* contains the word '`MINX`', followed without a space by a number, then the horizontal minimum point of the histogram(s) is that number. If *style* contains the word '`MAXX`', followed without a space by a number, then the horizontal maximum point of the histogram(s) is that number. If *style* contains the word '`MAXY`', followed without a space by a number, then the vertical maximum point of the histogram(s) is that number. If *style* contains the word '`ROUND`', then the right and left ends of the histogram are expanded, if necessary, to make the width of the bars rounded to 1 digit. The function pointed to by *xfunct* applies to the x-axis as in the description of '`pict_autoaxes`'. The default is equal width bars of counts. The integer *nplots* must be at least as large as the number of parts created by *partvars*. The function '`histogram`' allocates an array of '`picts`' for the histograms and axes, in which the picts for all the plots come first, followed by an equal number of '`pict`'s for the corresponding axes, and returns the starting address of that array.

Code fragment(s)

Display a histogram, in portrait orientation with a surrounding box, of 10 variable-width bars that show counts of the values of '`height`' in the dataset `people`:

```
nport(histogram("people", "height", "",
                          10, "VARIABLE ==", NULL, 1), 1, 1);
```

Display a histogram for each of 58 counties, on 58 pages in portrait orientation with axes at (0, 0), of 25 equal-width bars extending from 0 to 84 that show counts of the values of '`height`' in the dataset `people`:

```
nport(histogram("people", "height", "county",
                          25, "MIN0 MAX84", NULL, 58), 58, 1);
```

Display a histogram for each of 58 counties, on 58 pages in portrait orientation with axes at (0, 0), of 25 equal-width bars that show percentages of the values of '`height`' in the dataset `people`:

```
nport(histogram("people", "height", "county",
                          25, "PERCENT", NULL, 58), 58, 1);
```

6.2 Normal probability plot

SPECIFICATION

normal(*dataset*, *variable*, *partvars*, *nplots*)
dataset, *variable*, *partvars:* string
nplots: int
Returns: array of '`pict`'

Description

If *nplots* is greater than 0, constructs an array of '`pict`' structures for drawing (see Section 6.6 [Displaying the pictures], page 55) a q-q plot of the values of *variable* for each part of *dataset* defined by the values of *partvars*. If the number of values is at least 3 but no more than 2000, performs a Shapiro-Wilk test for normality, the results of which are reported in the lst file and, if *nplots* is greater than 0, in the caption of the '`pict`'s. Each q-q plot is prepared for display on a separate page. The integer *nplots* must be at least as large as the number of parts created by *partvars*. The function '`normal`' allocates an array of *picts* for the plots and axes, in which all the plots come first, and returns the address of that array.

Code fragment(s)

For each distinct value of '`block`', display a q-q plot of the values of '`yield`' and apply the Shapiro-Wilk test for normality:

```
nport(normal("wheat", "yield", "block", 4), 4, 1);
```

6.3 Scatter plot or line graph

SPECIFICATION

plot(*dataset*, *xyvar*, *partvars*, *style*, *xfunct*, *yfunct*, *nplots*)
dataset, *xyvar*, *partvars*, *style:* string
xfunct, *yfunct:* pointer to function returning double
nplots: int
Returns: array of '`pict`'

proc plot [data=*dataset-name*];
by *variable-list*;
plot *y-variable* * *xvariable* [/ [box]];
[...
plot *y-variable* * *xvariable* [/ [box]];]

Description

For *dataset*, constructs an array of '`pict`' structures for plotting (see Section 6.6 [Displaying the pictures], page 55) the points whose coordinates are in the two variables listed in *xyvar*. Each of those variable names may be followed by a string, enclosed in back quotes, to be used instead of the variable name as the axis label. Points are plotted in the order that they appear in the dataset and therefore must be sorted for a line graph. The default type of graph is a scatter plot with circles marking the points. To get a line graph, you must set the value of the '`pict_type`' field of the '`pict`' to the string '`LINE`'; see the description of the '`pict`' structure. If the first character of *style* is '`o`' and is not followed by an integer, then the plots for the different values of the *partvars* are all prepared to be displayed on the same pair of axes on the same page. If the first character of *style* is '`o`' and is followed by an integer *n*, then the plots are overlayed in consecutive groups of size *n*. Otherwise those plots are prepared to be displayed on different pages. The remaining characters of *style* and the functions *xfunct* and *yfunct* are axis specifications as for '`pict_autoaxes`' (see Section 8.2 [Axes], page 65). The integer *nplots* must be at least as large as the number of groups created by *partvars*. The function '`plot`' allocates an array of '`pict`'s for the plots and axes, in which all the plots come first, and returns the address of that array.

Code fragment(s)

Display a scatter plot in portrait orientation with '`height`' on the horizontal axis and '`weight`' on the vertical axis:

```
nport(plot("people", "height weight", "", "", NULL, NULL, 1),
1, 1);
```

Overlay 4 scatter plots, one for each distinct value of '`plot`', on one page in portrait orientation, with '`fert`' on the horizontal axis and '`yield`' on the vertical axis, and make the symbols be open circles, filled-in circles, open squares, and filled-in squares, respectively:

```
pict *p;

p = plot("wheat", "fert yield", "plot", "o", NULL, NULL, 4);
strcpy(p[0].pict_type, "CIRC");
strcpy(p[1].pict_type, "CIRC");
p[1].pict_fgray = 0.0;
strcpy(p[2].pict_type, "SQUA");
strcpy(p[3].pict_type, "SQUA");
p[3].pict_fgray = 0.0;
nport(p, 4, 4);
```

Display one boxed line graph per page in portrait orientation for each of the 2 distinct values of '`species`' with '`time`' on the horizontal axis and '`logpop`' on the vertical axis, with the '`exp`' function applied to the vertical coordinate labels:

```
pict *p;

p = plot("bacteria", "time logpop", "species",
                                     "==", NULL, &exp, 2);
strcpy(p[0].pict_type, "LINE");
strcpy(p[1].pict_type, "LINE");
nport(p, 2, 1);
```

Overlay 6 boxed scatter plots, one for each distinct value of '`plot`', 3 per page on 2 pages in portrait orientation, with '`fert`' on the horizontal axis and '`yield`' on the vertical axis, and make the symbols be open circles, open triangles, and open squares, respectively, on each of the 2 pages:

```
pict *p;

p = plot("wheat", "fert yield", "plot variety",
                                     "o3 ==", NULL, NULL, 6);
strcpy(p[0].pict_type, "CIRC");
strcpy(p[1].pict_type, "TRIA");
strcpy(p[2].pict_type, "SQUA");
strcpy(p[3].pict_type, "CIRC");
strcpy(p[4].pict_type, "TRIA");
strcpy(p[5].pict_type, "SQUA");
nport(p, 6, 3);
```

6.4 Plotting means

SPECIFICATION

plotmeans(*dataset*, *y-variable*, *x-variable*, *errorbars*, *style*, *partvars*, *noverlay***)**
dataset, *y-variable*, *x-variable*, *errorbars*, *style*, *partvars:* string
noverlay: int
Returns: array of '`pict`'

Description

Constructs an array of '`pict`' structures for plotting (see Section 6.6 [Displaying the pictures], page 55) the mean, with an error bar, of *y-variable* for each value of *x-variable*. Both variables must be of type double and either or both may be followed by a string, enclosed in back quotes, to be used instead of the variable name as the axis label. The height of the error bar above and below the mean can be specified to be any statistic available for the '`means`' function, optionally multiplied by a scale factor, which follows the statistic name and a space in *errorbars*. The string *style* is as in *plot*, except that overlaying is controlled by *plotmeans* and must not be specified in *style*. If *partvars* is null (""), then the array of '`pict`'s has two elements, the first for the error bars and the second for the means. If *partvars* is not null but *noverlay* is 1, then the array of '`pict`'s has those same two elements for each part of the dataset and each plot is on a separate page. If *partvars* is not null and *noverlay* is greater than 1, then the array of '`pict`'s has those same two elements for each part of the dataset but the elements are linked so that *noverlay* pictures appear on the same set of axes. NOTE: this function creates the following datasets or overwrites them if they exist: *dataset*.mns, *dataset*.err, and *dataset*.err.srt.

Code fragment(s)

For each distinct value of '`thickness`', compute the mean and standard deviation of the values of '`strength`' and display on a single page in portrait orientation the means with error bars extending the standard deviation above and below the mean:

```
nport(plotmeans("wires", "strength", "thickness", "SD",
                                                "", 0), 2, 2);
```

For each distinct value of '`thickness`', compute the mean and standard error of the mean of the values of '`strength`' and display on a single page in portrait orientation the means with error bars extending 1.96 times the standard error of the mean above and below the mean:

```
nport(plotmeans("wires", "strength", "thickness", "SEM 1.96",
                                                "", 0), 2, 2);
```

For each of 3 distinct values of '`metal`' display a separate page on which appears, for each distinct value of '`thickness`', the means of '`strength`' with error bars extending the standard deviation above and below the mean:

```
nport(plotmeans("wires", "strength", "thickness", "SD",
                                              "metal", 1),
    6, 2);
```

For each of 3 distinct values of '`metal`', display overlayed on a single page the means of '`strength`' for each distinct value of '`thickness`', with error bars extending the standard deviation above and below the mean:

```
    nport(plotmeans("wires", "strength", "thickness", "SD",
                                              "metal", 3),
    6, 6);
```

6.5 Regression plots

SPECIFICATION

plotlinreg(*dataset*, *y-variable*, *x-variable*, *style*, *partvars*, *nparts*, *level***)**
dataset, *y-variable*, *x-variable*, *style*, *partvars:* string
nparts: int
level: double
Returns: array of '`pict`'

Description

Runs *linreg* on *dataset*, allocates an array of '`pict`'s for the plots and axes for the plots of the observed data the regression line, and the curves enclosing a *level* confidence region for the predicted mean of *y-variable*, and returns the address of the array of '`pict`'s, to be displayed as *nparts* page(s) of 4 overlayed pictures each (see Section 6.6 [Displaying the pictures], page 55). Both *x-variable* and *y-variable* must be single variables and each may be followed by an axis label enclosed in pairs of '`` ` ``'. *dataset* must be partitioned by *partvars* into exactly *nparts* parts. *style* may contain axis specifications as for '`pict_autoaxes`' (see Section 8.2 [Axes], page 65). NOTE: This function creates the following datasets or overwrites them if they exist: *dataset*.mns, *dataset*.reg, and *dataset*.reg.srt.

Code fragment(s)

Display on one page a scatter plot of the data, with '`thickness`' on the horizontal axis, which will be labeled '`Thickness`', and '`strength`' on the vertical axis, while will be labeled '`Strength`', and the regression line and the curves defining a 95% confidence region for the predicted means:

```
    nport(plotlinreg("wires", "stren`Strength`", "thick`Thickness`",
                     "==", "", 1, 0.95), 4, 4);
```

For each of the 4 values of the variable '`plot`', display on a separate page a scatter plot of the data, with '`salin`' on the horizontal axis and '`yield`' on the vertical axis, the regression line, and the curves defining a 95% confidence region for the predicted means:

```
    nport(plotlinreg("pist", "yld", "sal", "==", "plot", 4, 0.95),
    16, 4);
```

SPECIFICATION

plotlogreg(*dataset*, *y-spec*, *x-variable*, *style*, *ngroups*, *partvars*, *nparts*, *level***)**
dataset, *y-spec*, *x-variable*, *style*, *partvars:* string
ngroups, *nparts:* int
level: double
Returns: array of '`pict`'

Description

Runs *logreg* on *dataset*, allocates an array of '`pict`'s for the plots and axes for plots the logistic regression curve and the curves enclosing a *level* confidence region for the predicted expectation of the response variable in *y-spec*, and returns the starting address of the array of '`pict`'s, to be displayed as *nparts* page(s) of 4 overlayed pictures each (see Section 6.6 [Displaying the pictures], page 55). Also plots the mean of the response variable in *y-spec* for the *ngroups* groups of the observations. The list *x-variable* must contain only one variable; *y-spec* is as in *logreg*. The response variable in *y-spec* and the variable in *x-variable* may be followed by axis labels enclosed in '`‘`'. *dataset* must be partitioned by *partvars* into exactly *nparts* parts. *style* may contain axis specifications as for '`pict_autoaxes`' (see Section 8.2 [Axes], page 65). NOTE: This function creates the following datasets or overwrites them if they exist: *dataset*.trl, *dataset*.trl.grp, *dataset*.trl.grp.srt, *dataset*.trl.grp.srt.mns, *dataset*.mns, *dataset*.lgr, and *dataset*.lgr.srt.

Code fragment(s)

Display on one page the mean success rate for each of 5 consecutive groups of the data, ordered by '`lab`', and the logistic regression line, and the curves defining a 95% confidence region for the predicted fraction of success:

```
nport(plotlogreg("can", "rem‘Remissions‘ / 1", "lab‘Labeling index‘",
                 "==", 5, "", 1, 0.95), 4, 4);
```

Display on one page the mean fraction of '`rem`' per '`case`' for each of 5 consecutive groups of the data, ordered by '`lab`', and the logistic regression line, and the curves defining a 95% confidence region for the predicted fraction of '`rem`' per '`case`':

```
nport(plotlogreg("can", "rem/case", "lab", "==", 5, "", 1, 0.95),
      4, 4);
```

For each of the 3 values of the variable '`hosp`', display on a separate page the mean fraction of '`rem`' per '`case`' for each of 5 consecutive groups of the data, ordered by '`lab`', and the logistic regression line, and the curves defining a 95% confidence region for the predicted fraction of '`rem`' per '`case`':

```
nport(plotlogreg("can", "rem/case", "lab", "==", 5, "hosp", 3,
0.95),
      12, 4);
```

6.6 Displaying the pictures

SPECIFICATION

nport(*p*, *nplots*, *nperpage*)
nland(*p*, *nplots*, *nperpage*)
p: array of '`pict`'
nplots, *nperpage:* int

Description

Generates an *nplots* / *nperpage* page graphics output file, in portrait or landscape orientation, from the '`pict`' array pointed to by *p* with a total of *nplots* plots, *nperpage* per page. As shown above, the first argument of a call to '`nport`' or '`nland`' is usually provided directly by the value returned from a call to a graphics function such as '`plotlogreg`'.

Code fragment(s)

Display the 8 '`pict`'s of the array '`p`' on 4 pages in portrait orientation, with 2 '`pict`'s per page:

```
pict *p;

...
nport(p, 8, 2);
```

Display '`pict`'s from '`plotlogreg`' in portrait orientation:

```
nport(plotlogreg("can", "nrem/ncase", "lab", "==", 5, "", 1, 0.95),
      4, 4);
```

Display '`pict`'s from '`plotlogreg`' in landscape orientation:

```
nland(plotlogreg("can", "nrem/ncase", "lab", "==", 5, "", 1, 0.95),
      4, 4);
```

Chapter 7

Utilities: probability and miscellaneous functions

7.1 Chi-squared distribution

SPECIFICATION

probchisq(c, *df*)
c: double
df: int
Returns: double

Description

Returns the probability that a variable distributed as chi-squared with *df* degrees of freedom has a value greater than c.

Code fragment(s)

Assign to '`p`' the probability that a Chi-squared variable with 2 degrees of freedom will have a value exceeding 3.7:

```
double p;
p = probchisq(3.7, 2);
```

SPECIFICATION

chisqpoint(p, *df*)
p: double
df: int
Returns: double

Description

Returns the value that a variable that is distributed as chi-squared with *df* degrees of freedom exceeds with probability p.

Code fragment(s)

Assign to '`c`' the point that a Chi-squared variable with 3 degrees of freedom exceeds with probability 0.05:

```
double c;
c = chisqpoint(0.05, 3);
```

7.2 F distribution

SPECIFICATION

probf(*f*, *numer-df*, *denom-df***)**
f: double
numer-df, *denom-df:* int
Returns: double

Description

Returns the probability that a variable distributed as F with *numer-df* and *denom-df* degrees of freedom has a value greater than *f*.

Code fragment(s)

Assign to '`p`' the probability that an F variable with 4 and 2 degrees of freedom will have a value exceeding 5.8:

```
double p;
p = prob(5.8, 4, 2);
```

SPECIFICATION

fpoint(*p*, *numer-df*, *denom-df***)**
p: double
numer-df, *denom-df:* int
Returns: double

Description

Returns the value that a variable that is distributed as F with *numer-df* and *denom-df* degrees of freedom exceeds with probability *p*.

Code fragment(s)

Assign to '`f`' the point that an F variable with 4 and 2 degrees of freedom exceeds with probability 0.05:

```
double f;
f = fpoint(0.05, 4, 2);
```

7.3 Normal distribution

SPECIFICATION

varnorm()
Returns: double

Description

Repeated calls return values (pseudo) independently sampled from a standard normal distribution.

Code fragment(s)

Assign to '`z`' a value sampled from a standard normal distribution:

```
    double z;
    z = varnorm();
```

SPECIFICATION

> **probz**(z)
> *z:* double
> *Returns:* double

Description

Returns the probability that a standard normal variable has a value no greater than z.

Code fragment(s)

Assign to '`p`*' the probability that a standard normal variable will have a value not exceeding 1.645:*

```
    double p;
    p = probz(1.645);
```

SPECIFICATION

> **zpoint**(p)
> *p:* double
> *Returns:* double

Description

Returns the value that a standard normal variable exceeds with probability p.

Code fragment(s)

Assign to '`z`*' the that a standard normal variable exceeds with probability 0.05:*

```
    double z;
    z = zpoint(0.05);
```

7.4 t distribution

SPECIFICATION

> **probt**(t, df)
> *t:* double
> *df:* int
> *Returns:* double

Description

Returns the probability that a variable distributed as Student's t with *df* degrees of freedom has a value greater than t.

Code fragment(s)

Assign to '`p`*' the probability that a t variable with 3 degrees of freedom will have a value exceeding 2.3:*

```
double p;
p = probt(2.3, 3);
```

SPECIFICATION

tpoint(*p*, *df*)
p: double
df: int
Returns: double

Description

Returns the value that a variable that is distributed as Student's t with *df* degrees of freedom exceeds with probability *p*.

Code fragment(s)

Assign to '`t`' the point that a t variable with 4 degrees of freedom exceeds with probability 0.05:

```
double t;
t = tpoint(0.05, 4);
```

7.5 Uniform distribution

SPECIFICATION

varunif()
Returns: double

Description

Repeated calls return values (pseudo) independently sampled from a uniform [0, 1] distribution.

Code fragment(s)

Assign to '`u`' a value sampled from a uniform [0, 1] distribution:

```
double u;
u = varunif();
```

7.6 Miscellaneous functions

SPECIFICATION

dap_bincoeff(*n*, *r*)
n, *r:* double
Returns: double

Description

Returns the binomial coefficient n C r.

SPECIFICATION

dap_maximize(*f*, *nx*, *x*, *step*, *tol*, *trace*)
f: pointer to function returning double
nx: int
x, *step*, *tol:* double
trace: string
Returns: double

Description

Maximizes (or attempts to maximize) the function *f* of *nx* variables, starting with the input point *x*. The function '`dap_maximize`' uses a simple hill-climbing algorithm, with numerically approximated partial derivatives, starting with step-size *step* and halving the step-size as necessary until it is less than *tol*. The string *trace* is either the null string (not '`NULL`') for no tracing, or '`TRACE`', for continuous tracing, or '`PAUSE`', for tracing and waiting for the user to press `Enter` at each iteration. Either '`TRACE`' or '`PAUSE`' may be followed, with no spaces, by a number to specify how many steps should be taken before each trace or pause. If convergence is not obtained in '`DAP_MAXITER`' steps Chapter 9.6 [Appendix I], page 91, that failure is reported before returning.

Code fragment(s)

Assign to '`max`' the maximum value attained by the function of 2 variables '`fun`', starting with the values stored in '`x`', with a step size of 0.01 and a tolerance in the independent variables of 0.001:

```
double fun(double x[2]);
double max;
double x[2];

max = dap_maximize(&fun, 2, x, 0.01, 0.001, "");
```

Assign to '`max`' the maximum value attained by the function of 2 variables '`fun`', starting with the values stored in '`x`', with a step size of 0.01 and a tolerance in the independent variables of 0.001 and display a trace of the iterations:

```
double fun(double x[2]);
double max;
double x[2];

max = dap_maximize(&fun, 2, x, 0.01, 0.001, "TRACE");
```

SPECIFICATION

dap_numdate(*date*)
date: string
Returns: int

Description

With *date* a date on or after January 1, 1752, in the form '`MMDDYYYY`' or '`MM/DD/YYYY`', in which the '`MM`' and the '`DD`' in the latter form can consist of a single digit, returns the number of days since December 31, 1751, otherwise returns -1.

Code fragment(s)

Assign to '`d`' the number 72143, i.e., the number of days that July 9, 1949 is later than December 31, 1751:

```
int d;
d = dap_numdate("07091949");
```

SPECIFICATION

dap_datenum(*n*, *date*)
n: int
date: string

Description

With *n* the number of days since December 31, 1751, fills *date* with the date in the form '`MMDDYYYY`'. If *n* is not positive or is too large, *date* is set to "?".

Code fragment(s)

Assign to '`date`' the string "07021761", i.e., the date 3471 days after December 31, 1751:

```
char date[9];
dap_datenum(3471, date);
```

SPECIFICATION

dap_invert(*matrix*, *rowscols*)
matrix: pointer to pointer to double
rowscols: int
Returns: int

Description

Inverts the matrix, returns 1 if non-singular, 0 otherwise. The parameter *matrix* is an array of pointers to double, each double pointed to being the first element of an array of double.

Code fragment(s)

```
double **mat;
int nonsing;
nonsing = dap_invert(mat, 3);
```

Chapter 8

Picture functions

The most common use of the information in this chapter is to modify one or more fields in the '`pict`' structure, which is described in the first section of this chapter, returned by a dap graphics function. In addition, this chapter describes low-level picture functions. Although these picture functions are not needed in most dap programs, they are useful for constructing custom graphics. The functions that add elements, shapes, or text to a picture must be followed by a call to '`pict_show`' to include those objects in the graphics output file. Complex pictures with, for example, different fonts for different pieces of text or different sized circles, may be constructed in one of two ways: either link the parts of the picture and call '`pict_show`' once, on the head of the list, or call '`pict_show`' repeatedly, after assembling each of the parts of the picture.

8.1 Pict structure and pict_init

Dap graphics functions return arrays of '`pict`'. The '`pict`' structure contains the following fields that may be modified directly to change a picture. Numerical values are in points (1/72 inch).

pict_type: string (4 letters)

This field must be one of the following strings:

LINE

Draw lines connecting successive points.

SEGM

Draw a separate segment for each successive pair of points.

IBEA

Draw an I-beam for each successive pair of points. If the horizontal coordinates of the pair of points are equal, draw a vertical I-beam; if the vertical coordinates of the pair of points are equal, draw a horizontal I-beam; otherwise, report the error and exit.

CIRC

Draw each point as a circle.

SQUA

Draw each point as a square.

TRIA

Draw each point as a triangle.

UTRI

Draw each point as a upside-down triangle.

DIAM

Draw each point as a diamond.

PATT

At the position of each point, display the pattern pointed to by *pict_patt*.

pict_dash: double
The dash length for lines if *pict_dash* > 0.0; the lines are not dashed if *pict_dash* $= 0.0$. The default is 0.0.

pict_font: string (up to 63 letters)
A string specifying the font for displayed text, if any. The default is '`Helvetica-Bold`'.

pict_fs: double
The font size for displayed text, if any. The default is 12.0.

pict_lw: double
The width for lines. The default is 0.4.

pict_r: double
The radius for circles and size for squares, triangles, upside-down triangles, and diamonds.

pict_lgray: double
The gray level for lines: 0.0 is completely black, 1.0 is completely white. The default is 0.0.

pict_fgray: double
The gray level for fill: if *pict_fgray* < 0.0, then don't fill areas; if *pict_fgray* ≥ 0.0, then fill areas, then draw the boundary lines. The default is -1.0.

pict_patt: pointer to '`pict`'
This field allows you to place a picture at each point: set *pict_patt* to the address of the '`pict`' to be displayed.

pict_next: pointer to '`pict`'
This field allows you to link pictures to be displayed on the same page.

SPECIFICATION

pict_initpict(*prev*, *p*)
prev, *p:* pointer to '`pict`'

Description
Initialize the '`pict`' pointed to by *p*, linking the '`pict`' pointed to by *prev* to it. If *prev* is '`NULL`', then no '`pict`' is linked to *p*.

Code fragment(s)
Initialize the picture structures '`p[0]`' and '`p[1]`' and link '`p[0]`' to '`p[1]`':

```
pict p[2];

pict_initpict(NULL, p);
pict_initpict(p, p + 1);
```

SPECIFICATION

pict_clearpict(*p*)
p: pointer to '`pict`'

Description
Frees up internal memory space used by *p*: use before calling '`pict_initpict`' on *p* after it has been used. Does not free *p* itself.

Code fragment(s)
Free the internal memory space used by '`p`':

```
pict p;

pict_clearpict(&p);
```

8.2 Axes

SPECIFICATION

pict_axes(*p*, *minx*, *maxx*, *xticks*, *nxticks*, *miny*, *maxy*, *yticks*, *nyticks*, *axspec*, *bpos*, *lpos*, *tpos*, *rpos*)
p: pointer to '`pict`'
xticks, *yticks:* array of '`tick`'
nxticks, *nyticks:* int
axspec: string
bpos, *lpos*, *tpos*, *rpos:* double

Description
Create axes in the '`pict`' pointed to by *p* with ticks along the x-axis specified by the array *xtick* of *nxticks* '`tick`'s and ticks along the y-axis specified by the array *yticks* of *nyticks* '`tick`'s. The ticks must be in order. The last tick is used to label the axis and the numbers of ticks, *nxticks* and *nyticks* do not include this last tick. The string *axspec* is as for *pict_autoaxes*. The doubles *bpos*, *lpos*, *tpos*, and *rpos* are the positions of the bottom or only x-axis, the left or only y-axis, the top x-axis (if any), and the right y-axis (if any). The variables *minx*, *maxx*, *miny*, and *maxy* give the endpoints of the axes.

Code fragment(s)

Creat axes in '`p`' with 11 ticks on each of the x- and y-axes as a box whose lower and upper edges are at 0.0 and 100.0, respectively, and whose left and right edges are at -10.0 and 10.0, respectively, with ticks and numbers on all sides:

```
pict p;
tick xt[12], yt[12];

pict_axes(&p, xt, 12, yt, 12, "BB", 0.0, 100.0, -10.0, 10.0);
```

SPECIFICATION

pict_autoaxes(*p*, *xlabel*, *ylabel*, *axspec*, *xfunct*, *yfunct*, *caption*, *autopos***)**
p: array of '`pict`'
xlabel, *ylabel*, *axspec*, *caption:* string
xfunct, *yfunct:* pointer to function returning double
autopos: int

Description

Create axes, as specified below, by *axspec* for the linked list of '`pict`'s whose head is pointed to by *p* and whose tail is the '`pict`' to contain the axes. Unless otherwise specified, the axes extend to the maximum and minimum values of x and y in the entire linked list of '`pict`'s (except the tail '`pict`'), but always include the origin. Unless otherwise specified, the function *pict_autoaxes* chooses an appropriate spacing for the ticks on both axes. The x- and y-axes are labeled *xlabel* and *ylabel*, respectively, and the entire picture is captioned by *caption*. The first part of the string *axspec* consists of 0, 1, or 2 characters according to whether the default is used for both axes, just the y-axis, or the axes are specified as follows, respectively. An axis specification is one of the characters '`-`', '`+`', '`0`', '`n`', '`=`', or '`#`', which indicate that the axis should be placed: on the negative end of the other axis; the positive end; at 0; not at all; or at both ends, without or with markings on the positive end. The default is '`0`'. The endpoints of the graph and the number of ticks in each direction may be specified in *axspec* after the first part of the string (and at least one space) as follows: to set the minimum point on the x-axis to a number, say '`-3.6`', use '`MINX-3.6`' (no spaces). The maximum point on the x-axis and the minimum and maximum points on the y-axis can be specified using '`MAXX`', '`MINY`', and '`MAXY`' in the same way. To set the number of tick marks on the x-axis to a number, say 14, use '`NXTICKS14`' (no spaces). The number of ticks on the y-axis can be set using '`NYTICKS`' in the same way. The number of significant figures used for the labels on the ticks can be specified using '`NXDIGITS`' and '`NYDIGITS`', as in '`NXDIGITS4`'; the default is 3 digits.

The functions *xfunct* and *yfunct* are applied to the true tick values to create the tick labels. Either one or both may be '`NULL`' to specify that the tick labels equal the tick values. If *autopos* is '`PORTRAIT`' or '`LANDSCAPE`' (integer values, not strings, defined in the header file), then *apict_utoaxes* automatically scales and positions the plot appropriately. No scaling and positioning is done if *autopos* is 0.

Code fragment(s)

Create axes in the last '`pict`' in the linked array given by '`p`' with the horizontal axis labeled "time", the vertical axis labeled "CFU", the horizontal axis at the most negative point the '`pict`'s of '`p`', the vertical axis at '`time`' = 0, the function '`exp`' used to display numerical tick values on the vertical axis, the caption "CFU per cm^2^ surface area" (with the '2' as a superscript), scaled and placed appropriately on a page in portrait orientation:

```
pict *p;

pict_autoaxes(p, "time", "CFU", "-0", NULL, &exp,
                              "CFU per cm^2^ surface area", PORTRAIT);
```

SPECIFICATION

pict_maketick(*thetick*, *position*, *label*, *length***)**
tick: pointer to '`tick`'
position, *length:* double
label: string

Description

Make the '`tick`' pointed to by *thetick* be at coordinate *position* with label *label* and of length *length*.

Code fragment(s)

Create in '`t`' a tick with position 5.0, label "25.0", and length 3.0 points:

```
tick t;

pict_maketick(&t, 5.0, "25.0", 3.0);
```

8.3 Elements

SPECIFICATION

pict_point(*p*, *x*, *y***)**
p: pointer to '`pict`'
x, *y:* double

Description

Add a point at (*x*, *y*) to the '`pict`' pointed to by *p*. Points added to the same '`pict`' appear in the order in which they are added.

Code fragment(s)

Add a point at (7.2, -4.3) to '`p`':

```
pict p;

pict_point(&p, 7.2, -4.3);
```

SPECIFICATION

pict_line(*p*, *x1*, *y1*, *x2*, *y2*)
p: pointer to '`pict`'
x1, *y1*, *x2*, *y2:* double

Description

Add a line from (*x1*, *y1*) to (*x2*, *y2*) to the '`pict`' pointed to by *p*. Lines added to the same '`pict`' appear in the order in which they are added.

Code fragment(s)

Add a line from (7.2, -4.3) to (9.1, 2.6) to '`p`':

```
pict p;

pict_line(&p, 7.2, -4.3, 9.1, 2.6);
```

8.4 Shapes

SPECIFICATION

pict_circle(*p*, *x*, *y*, *r*)
p: pointer to '`pict`'
x, *y*, *r:* double

Description

Make the '`pict`' pointed to by *p* a circle of radius *r*, centered at (*x*, *y*). NOTE: If a picture is to contain more than one circle, then those circles must be created in separate '`pict`'s that are linked together.

Code fragment(s)

Add a circle with center (7.2, -4.3) and radius 3.0 to '`p`':

```
pict p;

pict_circle(&p, 7.2, -4.3, 3.0);
```

SPECIFICATION

pict_rectangle(*p*, *x*, *y*, *xside*, *yside*)
p: pointer to '`pict`'
x, *y*, *xside*, *yside:* double

Description

Add a rectangle whose lower left corner is a (*x*, *y*) and has horizontal sides of length *xside* and vertical sides of length *yside* to the '`pict`' pointed to by *p*. NOTE: If a picture is to contain more than one rectangle but those rectangles are not to be connected, then those rectangles must be created in separate '`pict`'s that are linked together.

Code fragment(s)

Add a rectangle with lower left corner (7.2, -4.3), width 4.0, and height 4.5 to '`p`':

```
pict p;

pict_rectangle(&p, 7.2, -4.3, 4.0, 4.5)
```

SPECIFICATION

pict_hrect(*p*, *spacing*, *x*, *y*, *xside*, *yside***)**
pict_bhrect(*p*, *spacing*, *x*, *y*, *xside*, *yside***)**
p: pointer to '`pict`'
spacing, *x*, *y*, *xside*, *yside:* double

Description

Add a forward-hashed (respectively, backward-hashed) rectangle whose lower left corner is at (*x*, *y*) and has horizontal sides of length *xside* and vertical sides of length *yside* to the '`pict`' pointed to by *p*. The horizontal space between hash lines is *spacing*. NOTE: If a picture is to contain more than one rectangle but those rectangles are not to be connected, then those rectangles must be created in separate '`pict`'s that are linked together.

Code fragment(s)

Add a rectangle with lower left corner (3.5, 29.0), width 1.0, height 5.0, and with hashes sloping upwards at a spacing of 2.0 points to '`p`':

```
pict p;

pict_hrect(&p, 2.0, 3.5, 29.0, 1.0, 5.0);
```

Add a rectangle with lower left corner (3.5, 29.0), width 1.0, height 5.0, and with hashes sloping downwards at a spacing of 2.0 points to '`p`':

```
pict p;

pict_bhrect(&p, 2.0, 3.5, 29.0, 1.0, 5.0);
```

SPECIFICATION

pict_curve(*p*, *x*, *y*, *t0*, *t1*, *nsegments***)**
p: pointer to '`pict`'
x, *y:* pointer to function returning double
t0, *t1:* double
nsegments: int

Description

Add a curve to the '`pict`' pointed to by *p*. The curve is parameterized by *x* and *y*, which point to functions that take a single argument of type double and return a result of type double. If *x* is '`NULL`', then that function is taken to be the identity function. The parameter varies from *t0* to *t1* in *nsegments* steps. Curves (and lines and points) added to the same '`pict`' are connected in the order in which they are added. NOTE: If a picture is to contain more than one curve but those curves are not to be connected, then those curves must be created in separate '`pict`'s that are linked together.

Code fragment(s)

*Add to '`p`' a curve parameterized by '`cos`' and '`sin`' with parameter running from 0.0 to 2.0 * 3.14159 in 100 steps:*

```
pict p;

pict_curve(&p, &cos, &sin, 0.0, 2.0 * 3.14159, 100);
```

Add to '`p`' a graph of '`exp`' running from 0.0 to 3.0 in the horizontal direction in 100 steps:

```
pict p;

pict_curve(&p, NULL, &exp, 0.0, 3.0, 100);
```

8.5 Text

SPECIFICATION

pict_text(*p*, *text*, *x*, *y*, *angle*, *pos*)
p: pointer to '`pict`'
text, *pos:* string
x, *y*, *angle:* double

Description

Add text *text* to the '`pict`' pointed to by *p*. The string *text* may contain substrings enclosed between pairs of '`^`' for superscripts and between pairs of '`|`' for subscripts. The string *text* may also contain newline characters ('`\n`') for multi-line text. The string *pos* consists of either two or three letters that specify the position of the point with coordinates *x* and *y* relative to the text, as follows. The first character of *pos* is either '`l`', '`c`', or '`r`', for left, center, or right, respectively, and the second letter is either '`t`', '`m`', or '`b`', for top, middle, or bottom, respectively. If the optional third character of *pos* is a space, then a blank rectangle is formed to hold the text. The double *angle* is counter-clockwise rotation in degrees to be applied to text.

Code fragment(s)

Add to '`p`' the text "cm^3^ of H|2|O" (with the '3' as a superscript and the '2' as a subscript) placed horizontally with its bottom center at the point (306.0, 72.0):

```
pict p;

pict_text(&p, "cm^3^ of H|2|O", 306.0, 72.0, 0.0, "cb");
```

8.6 Transformations

SPECIFICATION

pict_translate(*p*, *x*, *y*)
p: pointer to ‘`pict`’
x, *y:* double

Description

Translate (i.e., shift) the ‘`pict`’ pointed to by *p* by *x* in the x-direction and *y* in the y-direction. If the ‘`pict`’ is linked to another ‘`pict`’, then that ‘`pict`’ is translated the same way, and so on until there are no more links.

Code fragment(s)

Translate all 4 pictures in the linked array ‘`p`’ 306.0 points to the right and 72.0 points up:

```
pict p[4];

pict_translate(p, 306.0, 72.0);
```

SPECIFICATION

pict_scale(*p*, *x*, *y*, *sx*, *sy*)
p: pointer to ‘`pict`’
x, *y*, *sx*, *sy:* double

Description

Scale the ‘`pict`’ pointed to by *p* by *sx* in the x-direction and *sy* in the y-direction, leaving the point at (*x*, *y*) fixed. If the ‘`pict`’ is linked to another ‘`pict`’, then that ‘`pict`’ is scaled the same way, and so on until there are no more links.

Code fragment(s)

Scale all 4 pictures in the linked array ‘`p`’ by 1.5 in the horizontal direction and 2.0 in the vertical direction, relative to the fixed point (306.0, 72.0):

```
pict p[4];

pict_scale(p, 306.0, 72.0, 1.5, 2.0);
```

SPECIFICATION

pict_rotate(*p*, *x*, *y*, *degrees*)
p: pointer to ‘`pict`’
x, *y*, *degrees:* double

Description

Rotate the ‘`pict`’ pointed to by *p* counter-clockwise by *degrees* around the point at (*x*, *y*). If the ‘`pict`’ is linked to another ‘`pict`’, then that ‘`pict`’ is rotated the same way, and so on until there are no more links.

Code fragment(s)

Rotate all 4 pictures in the linked array ‘`p`’ by 45.0 degrees counterclockwise around the fixed point (306.0, 400.0):

```
pict p[4];

pict_rotate(p, 306.0, 400.0, 45.0);
```

8.7 Graphics output file

A graphics output file must be started with a call to '`pict_init`' (or a function, such as '`pict_port`', that calls '`pict_init`') and ended with a call to '`pict_end`'.

SPECIFICATION

> **pict_port**(*npages*)
> **pict_land**(*npages*)
> *npages:* int

Description

Calls '`pict_init`' to set up *npages* portrait or landscape pages.

Code fragment(s)

Initialize a graphics output file for 3 pages in landscape orientation:

```
pict_land(3);
```

SPECIFICATION

> **pict_init**(*orient*, *bboxxl*, *bboxyb*, *bboxxr*, *bboxyt*, *npages*)
> *orient*, *bboxxl*, *bboxyb*, *bboxxr*, *bboxyt*, *npages:* int

Description

Initialize the graphics output file. The character (not string) *orient* is either '`l`', for landscape or '`p`' for portrait. The bounding box is specified by the *bbox* parameters and the number of pages by *npages*. '`pict_init`' is called by '`pict_portrait`' and '`pict_landscape`'.

Code fragment(s)

Initialize a graphics output file for 4 pages in portrait orientation with a bounding box with lower left corner (0, 0) and upper right corner (612, 792):

```
pict_init('p', 0, 0, 612, 792, 4);
```

SPECIFICATION

> **pict_page**()

Description

Begin new page.

SPECIFICATION

> **pict_show**(*p*)
> *p:* pointer to '`pict`'

Description

Write the picture commands to display the '`pict`' pointed to by *p* into the graphics output file. If *p* is linked to another '`pict`', then show writes out the commands for that pict and so on until there are no more links.

Code fragment(s)

Display the 4 '`pict`'s in the linked array '`p`':

```
pict p[4];

pict_show(p);
```

SPECIFICATION

pict _end()

Description

Terminate the graphics output file.

8.8 Picture datasets

SPECIFICATION

pict _save(*p*, *npicts*, *dataset***)**
p: pointer to '`pict`'
npicts: integer
dataset: string

Description

This function is useful for saving picture information generated by graphics functions, such as '`plotlogreg`', that may require lengthy execution times to generate pictures. Saved picture datasets can be restored quickly by calling '`pict_rest`'.

The function '`pict_save`' saves the picture information in the array of '`pict`'s *p* to the datasets *dataset*.picXXXX, *dataset*.ptsXXXX, and *dataset*.txtXXXX, where '`XXXX`' is the index of the '`pict`' in the array. If *npicts* is 0, then '`pict_ save`' only saves the elements of '`p`' that are linked together, starting at the first element. Otherwise, '`pict_save`' saves *npicts* '`pict`'s in the array. NOTE: graphics functions, such as '`plot`', link an extra '`pict`' for each page or overlayed '`pict`'s for the axes.

The following details are not needed for saving '`pict`' arrays generated by '`dap`' graphics functions. '`Pict_save`' calls itself to save patterns, if any, referenced in elements of *p* using *dataset*.patXXXX as the *dataset* argument. Patterns must be either single '`pict`'s or linked arrays. A link in the array *p* must always be to an element with an index larger than the element linking to it. (This condition is always satisfied by the '`pict`' arrays generated by graphics functions.)

Code fragment(s)

Save the 4 '`pict`'s in the linked array '`p`' in a picture datasets with base name '`picture`':

```
pict p[4];

pict_save(p, 0, "picture");
```

Save the 4 '`pict`'s in the unlinked array '`p`' in a picture datasets with base name '`picture`':

```
pict p[4];

pict_save(p, 4, "picture");
```

Save the 10 '`pict`'s created by '`plotlogreg`' in a picture datasets with base name '`grad`':

```
pict_save(plotlogreg("grad", "grad/1", "GPA", "==", 5, "year",
2, 0.95),
          10, "grad");
```

SPECIFICATION

pict_rest(*dataset***)**
dataset: string
Returns: pointer to '`pict`'

Description

Allocates an array of '`pict`'s and restores into that array a picture saved to *dataset* by '`pict_save`'.

Code fragment(s)

Restore a picture from picture datasets with base name '`picture`' into the array array '`p`':

```
pict *p;

p = pict_rest("picture");
```

Restore and display a picture with 2 pages, created by '`plotlogreg`', from picture datasets with base name '`grad`':

```
nport(pict_rest("grad"), 8, 4);
```

Chapter 9

Examples

This chapter contains examples to illustrate the use of dap. Data files, programs, and output for the examples are provided in a directory named `examples`.

9.1 Analysis of variance

These examples are from:

- AMD: Milliken, G.A. and Johnson, D.E. 1984. Analysis of Messy Data. Van Nostrand Reinhold: New York. 473pp.
- ED: Cochran, W.G. and Cox, G.M. 1957. Experimental Designs. John Wiley & Sons: New York. 611pp.

```
/* AMD pp. 128 - 134: unbalanced layout using SBS  */

data;
 infile "amd128.dat" firstobs=2; /* space separated, skip 1 header
line */
 length treat $ 6 block $ 6;
 input treat block y;

proc glm;
 class treat block;
 model y = treat block treat*block;
 lsmeans treat block / tukey;


/* AMD pp. 249 - 251: using SBS
 * Two factors crossed, another nested within
 * levels of one crossed factor
 */
```

```
data;  infile "amd249.dat" firstobs=2;
 length a $ 1 b $ 1 c $ 1;
 input b c a y1 y2; /* two values per cell */
 y = y1;
 output;
 y = y2;
 output;

proc glm;
 class a b c;
 model y = a b a*b c*b a*c*b;
 test h=a e=a*b;
 test h=b e=a*b b*c a*b*c;
 test h=a*b e=a*b*c;
 test h=c*b e=a*c*b;



/* AMD pp. 265 - 273 using SBS
 * Random model, unbalanced
 */

data;
 infile "amd265.dat" firstobs=2;
 length plant $ 1 site $ 1 worker $ 1;
 input plant worker site efficiency;

proc glm;
 class plant site worker;
 model efficiency = plant plant*worker plant*site plant*site*worker;
 test h=site*plant e=site*worker*plant;

proc glm;
 class plant site worker;
 model efficiency = plant plant*worker site*worker*plant;
 test h=worker*plant e=site*worker*plant;
 test h=plant e=worker*plant site*worker*plant;




/* AMD pp. 285 - 289 using SBS
 * Mixed model, balanced
 */

data;
```

```
 infile "amd285.dat" firstobs=2;
 length machine $ 1 person $ 1;
 input machine person prod1 prod2 prod3; /* 3 observations per
cell */
 productivity = prod1;
 output;
 productivity = prod2;
 output;
 productivity = prod3;
 output;

proc glm;
 class machine person;
 model productivity = machine person machine*person;
 test h=person e=machine*person;
 lsmeans machine / e=machine*person lsd;



/* AMD pp. 290 - 295 using SBS
 * Mixed model, unbalanced
 */

data;
 infile "amd290.dat" firstobs=2;
 length machine $ 1 person $ 1;
 input machine person productivity;

proc glm;
 class machine person;
 model productivity = machine person machine*person;
 test h=person e=machine*person;
 lsmeans machine / e=machine*person lsd;



/* AMD pp. 297 - 308 using SBS
 * Split plot
 */

data;
 infile "amd297.dat" firstobs=2;
 length fertilizer $ 1 block $ 1 variety $ 1;
 input block variety fertilizer yield;
```

```
proc glm;
 title "Whole plot (block, fertilizer) analysis";
 class fertilizer block variety;
 model yield = fertilizer block;
 lsmeans fertilizer / e=fertilizer*block LSD;

proc glm;
 title "Subplot (variety) analysis";
 class fertilizer block variety;
 model yield = fertilizer block variety
               fertilizer*block fertilizer*variety;



/* ED pp. 122 - 125 using SBS
 * Latin square
 */

data;
 infile "ed122.dat" firstobs=2;
 length sampler $ 1 area $ 1 order $ 1;
 input order area sampler error;

proc glm;
 class sampler area order;
 model error = sampler area order;
 lsmeans sampler / lsd;



/* ED pp. 176 using SBS and proc dap
 * Without covariate, with contrasts
 */

data muscle;
 infile "sas976.dat" dlm="\t" firstobs=3;
 length rep $ 1 time $ 1 current $ 1 number $ 1;
 input rep time current number y;

proc glm;
 class rep current time number;
 model y=rep current time number current*time current*number
         time*number current*time*number;
 contrast "curr 1 vs curr 2" current 1 -1;

/* To construct the constrast for testing "time in current 3",
 * we have to modify the muscle.srt.mns.con file produced by glm.
 */
proc dap;
{ /* start with brace to enclose everything */
  inset("muscle.srt.mns.con")
   {
     char rep[2], current[2], time[2];
     double y;
     char _type_[9]; /* N, MEAN, VAR, ERROR, CONTR, LSMEAN */
     int _term_; /* specifies term to which contrast applies */
     int more; /* to control stepping through dataset */
     double c1[4], c2[4], c3[4]; /* contrast with 3 df */
     outset("muscle.con", ""); /* datast for the F-test */
     /* set up the contrast coefficients */
     c1[0] = 1; c1[1] = 0; c1[2] = 0; c1[3] = -1;
     c2[0] = 0; c2[1] = 1; c2[2] = 0; c2[3] = -1;
     c3[0] = 0; c3[1] = 0; c3[2] = 1; c3[3] = -1;
     for (more = step(); more; )
      {
        output(); /* N, MEAN, VAR */
        step();
        output();
        step();
        output();
        for (step(); strcmp(_type_, "CONTR"); step()) /* get to
CONTR lines */
          output();
        _term_ = 4; /* bits: 1 is rep, 2 is current, 4 is time
*/
        if (!strcmp(current, "3")) /* only in current 3 */
         {
           y = c1[time[0] - '1']; /* convert time to index */
           output();
           y = c2[time[0] - '1'];
           output();
           y = c3[time[0] - '1'];
           output();
         }
        else
         {
           y = 0.0;
           output();
           output();
           output();
         }
        while (more && !strcmp(_type_, "CONTR")) /* look for the
ones we want */
           more = step();
        while (more && !strcmp(_type_, "LSMEAN")) /* get to next
cell or end */
         {
           output();
           more = step();
         }
      }
   }
  /* muscle.con only has time in numerator so don't need to specify
it */
  ftest("muscle.con", "y rep current time number", "", "", "");
}



/* AMD pp. 173 - 177:
 * missing treatment combinations
 */

data amd173;
 infile "amd173.dat" firstobs=2;
 length treat $ 2 block $ 2;
 input treat block y;

proc sort data=amd173;
 by treat block;

proc means data=amd173 N MEAN VAR noprint;
 var y;
 by treat block;
 output out=amd173.mns;

/* Now we have to create "by hand" the .con files for
 * the custom F-tests for the contrasts that are meaningful
 * in the presence of empty cells.
 */
/* The first F-test (p. 175-76) is the interaction:
 * m11 - m13 - m21 + m23 = 0 and m21 - m22 - m31 + m32 = 0
 */
proc dap;
{ /* start with a brace to enclose everything here */
  inset("amd173.mns") /* file from model statement */
   {
     char treat[3], block[3]; /* we're in C here! */
```

```
      double y;
      char _type_[9]; /* set this to CONTR */
      int _term_;     /* bits specify the effect */
      double c1[7], c2[7]; /* coeffs of the contrasts */
      int c; /* cell number */
      outset("amd173.mns.con", "treat block y _term_");
      /* cells, in sort order, are:
      /*   11       13       21       22      23       31      32
*/
      c1[0]=1;c1[1]=-1;c1[2]=-1;c1[3]= 0;c1[4]=1;c1[5]= 0;c1[6]=0;
      c2[0]=0;c2[1]= 0;c2[2]= 1;c2[3]=-1;c2[4]=0;c2[5]=-1;c2[6]=1;
      _term_ = 3; /* bit 1 for treat, bit 2 for block */
      for (c = 0; step(); c++) /* while there's another cell */
       {
         output(); /* N, MEAN, VAR */
         step();
         output();
         step();
         output();
         strcpy(_type_, "CONTR");
         y = c1[c];
         output();
         y = c2[c];
         output();
       }
   }
  ftest("amd173.mns.con", "y treat block", "treat*block", "",
"");

/* The second F-test (p. 176-77) is the treat effect:
 * m11 + m13 - m21 - m23 = 0 and m21 + m22 - m31 - m32 = 0
 */
  inset("amd173.mns") /* file from model statement */
   {
      char treat[3], block[3]; /* we're in C here! */
      double y;
      char _type_[9]; /* set this to CONTR */
      int _term_;     /* bits specify the effect */
      double c1[7], c2[7]; /* coeffs of the contrasts */
      int c; /* cell number */
      outset("amd173.mns.con", "treat block y _term_");
      /* cells, in sort order, are:
      /*   11      13       21      22       23       31      32
*/
      c1[0]=1;c1[1]=1;c1[2]=-1;c1[3]=0;c1[4]=-1;c1[5]= 0;c1[6]=
0;
```

```
      c2[0]=0;c2[1]=0;c2[2]= 1;c2[3]=1;c2[4]= 0;c2[5]=-1;c2[6]=-1;
      _term_ = 1; /* bit 1 for treat */
      for (c = 0; step(); c++) /* while there's another cell */
       {
         output(); /* N, MEAN, VAR */
         step();
         output();
         step();
         output();
         strcpy(_type_, "CONTR");
         y = c1[c];
         output();
         y = c2[c];
         output();
       }
   }
  ftest("amd173.mns.con", "y treat block", "treat", "", "");
}
```

9.2 Linear regression

```
/* Bickel, P.J. and Doksum, K.A. 1977
 * Mathematical Statistics:
 * Basic Ideas and Selected Topics
 * Holden-Day: Oakland. 493.pp.
 * Example pp. 95 - 97.
 */

data;
  infile "ms95.dat" firstobs=2;
  input soilphos plantphos;

proc reg;
 model plantphos = soilphos;
 plot phantphos * soilphos;



/* Rao, C.R. and Toutenberg, H. 1995 using SBS
 * Linear Models: Least Squares and Alternatives
 * Springer-Verlag: New York. 352 pp.
 * Example pp. 50 - 60.
 */

data;
```

```
  infile "lm50.dat" firstobs=2;
  input y x1 x2 x3 x4;

proc corr;
 var x1 x2 x3 x4 y;
 title "Correlations";

proc reg;
 model y = x4;
 title "Model building";

proc reg;
 model y = x4;
 add x1;

proc reg;
 model y = x4 x1;
 add x3;

proc reg;
 model y = x4 x1 x3;
 add x2;
```

9.3 Categorical data analysis

These examples are from CDA: Agresti, A. 1990. Categorical Data Analysis. John Wiley & Sons: New York. 558pp.

```
/* CDA pp. 49 - 50 using SBS */

data;
 infile "cda50.dat" firstobs=2;
 length income $ 5 jobsat $ 10;
 input income jobsat count;

proc freq;
 tables income * jobsat / measures chisq expected
                          norow nocol nopercent;
 weight count;


/* CDA pp. 232 - 233 using SBS */

data;
 infile "cda233.dat" firstobs=2;
 length penicillin $ 5 delay $ 4 response $ 5;
 input penicillin delay response count;

proc freq;
 tables penicillin * delay * response / norow nocol nopercent
cmh;
 weight count;



/* CDA pp. 135 - 138, 171 - 174, 176 - 177
 * Here we fit loglinear models in table 6.3 on p. 172
 */
#include <dap.h>

void main()
{
  infile("cda171.dat", " ")
    {
      char defendant[6], victim[6], penalty[4];
      double n;
      input("defendant victim penalty n");
      outset("cda171", "");
      skip(2);
      while (step())
        output();
    }

  sort("cda171", "defendant victim penalty", "");

  title("(DV, P) vs (D, V, P)");
  loglin("cda171.srt", "n defendant victim penalty",
         "victim penalty defendant", "defendant*victim penalty",
"");

  sort("cda171.srt.llm", "defendant victim _type_ penalty", "");
  table("cda171.srt.llm.srt", "defendant victim", "_type_ penalty
n",
        "s6.2 30", "");

  title("(DV, VP) vs (DV, P)");
  loglin("cda171.srt", "n defendant victim penalty",
         "defendant*victim penalty",
         "defendant*victim victim*penalty", "");
  sort("cda171.srt.llm", "defendant victim _type_ penalty", "");
  table("cda171.srt.llm.srt", "defendant victim", "_type_ penalty
n",
        "s6.2 30", "");

  title("(DV, DP, VP) vs (DV, VP)");
  loglin("cda171.srt", "n defendant victim penalty",
         "defendant*victim victim*penalty",
         "defendant*victim defendant*penalty victim*penalty",
"");
  sort("cda171.srt.llm", "defendant victim _type_ penalty", "");
  table("cda171.srt.llm.srt", "defendant victim", "_type_ penalty
n",
        "s6.2 30", "");
}



/* CDA pp. 261 - 269
 * Here we fit the logit model for linear-by-linear association
 * to Table 8.2 on page 268.
 */
#include <dap.h>

double expect(double param[8], double class[2]);

void main()
{
  infile("cda262.dat", " ")
    {
      char Income[6], JobSat[10];
      double income, jobsat, count;
      input("Income JobSat count");
      outset("cda262", "");
      skip(1);
      while (step())
        {
          /* we have to convert to double for categ */
          if (!strcmp(Income, "<6"))
            income = 0.0;
          else if (!strcmp(Income, "6-15"))
            income = 1.0;
          else if (!strcmp(Income, "15-25"))
            income = 2.0;
          else if (!strcmp(Income, ">25"))
            income = 3.0;
          if (!strcmp(JobSat, "VeryDis"))
            jobsat = 0.0;
          else if (!strcmp(JobSat, "LittleDis"))
            jobsat = 1.0;
          else if (!strcmp(JobSat, "ModSat"))
            jobsat = 2.0;
          else if (!strcmp(JobSat, "VerySat"))
            jobsat = 3.0;
          output();
        }
    }

  {
    double param[8];
    int p;

    param[0] = 1.0;
    for (p = 1; p < 8; p++)
      param[p] = 0.0;
    categ("cda262", "count income jobsat", &expect, param,
          "mu <6 6-15 15-25 VD LD MS ?Inc*Sat", "", "");
    sort("cda262.cat", "income _type_ jobsat", "");
    table("cda262.cat.srt", "income", "_type_ jobsat count", "6.2",
"");
  }
}

/* We use an independent subset of the parameters in order to
 * incorporate the zero-sum constraints. Thus, if class[0] ==
3,
 * for example, then we use the fact that lambda^{income}_{>25}
is
 * minus the sum of the other lambda^{income} parameters.
 */
double expect(double param[8], double class[2])
{
  double lx, ly;

  if (class[0] < 3.0)
    lx = param[1 + (int) class[0]];
  else
    lx = -(param[1] + param[2] + param[3]);
  if (class[1] < 3.0)
    ly = param[4 + (int) class[1]];
  else
    ly = -(param[4] + param[5] + param[6]);
  return exp(param[0] + lx + ly + param[7] * class[0] * class[1]);
}
```

9.4 Logistic regression

```
/* Agresti, A.  1990.  Categorical Data Analysis.
 * John Wiley & Sons: New York.  558pp.
 * Example pp. 87 - 89 using SBS with proc dap
 */
data cda88;
  infile "cda88.dat" firstobs=2;
  input labind ncases nremiss;

proc dap;
nport(plotlogreg("cda88", "nremiss/ncases", "labind",
                 "== MAXX40 NXTICKS5 MAXY1 NYTICKS6 NYDIGITS2
NXDIGITS1",
                 5, "", 1, 0.95), 4, 4);
```

9.5 Standard graphical output

```
/* Using plotmeans to plot means, both as symbols
 * and joined by lines, and 95% confidence intervals
 * of two groups of data together.
 */
#include <dap.h>

void main()
{
infile("standard.dat", " ")
  {
    int part;
    double x, y;

    input("x y part");
    outset("mtest", "");
    while (step())
      output();
  }

title("Means of y. Error bars are 95% confidence for means");
  {
    pict *p;
    sort("mtest", "part x", "");
    p = plotmeans("mtest.srt", "y", "x", "SEM 1.96", "part", 2);
/* p[0] and p[1] are error bars and means as points for group
1 */
    strcpy(p[1].pict_type, "TRIA");
/* p[2] and p[3] are error bars and means as points for group
2 */
    strcpy(p[3].pict_type, "SQUA");

    nport(p, 4, 4);
  }
}
```

9.6 Custom graphics

```
/* This example illustrates the construction
 * of custom graphics.  We create two distributions
 * and display them as a split histogram: the bars
 * of one distribution extend horizontally to the
 * left and the bars of the other extend
 * horizontally to the right and are shaded.
 */
#include <dap.h>

#define NBARS 10

void main()
{
 /* these variables are not in the datasets */
double min, max;  /* extremes of both
                   * distributions together
                   */
double width;     /* width of the bars */

infile("custom.dat", " ")
  {
    double x;
    int part;

    input("x part");
    outset("split", "");
    while (step())
      output();
  }

means("split", "x", "MIN MAX", ""); /* find min, max */
```

```
inset("split.mns")
  {
    double x;
    int n;
    char _type_[9];

    for (n = 0; n < 2; n++)
      {
        step();             /* and store them */
        if (!strcmp(_type_, "MIN"))
          min = x;
        else
          max = x;
      }
  }

width = (max - min) / ((double) NBARS);

inset("split")      /* compute class for each x */
  {
    double x;
    int class;

    outset("class", "x class part");
    while (step())
      {
        class = (int) floor((x - min) / width);
        if (class < 0)
          class = 0;
        else if (class > NBARS - 1)
          class = NBARS - 1;
        output();
      }
  }

sort("class", "part class", "");
/* compute counts in each class for each distribution */
means("class.srt", "count", "N", "part class");

  {
  pict p[21];  /* one pict for each class for each part
                * plus one for the axes
                */
  int pn;

  pict_initpict(NULL, p);  /* initialize the pict structs */
```

```
    for (pn = 1; pn < 21; pn++)
      pict_initpict(p + pn - 1, p + pn);

    inset("class.srt.mns")
      {
        int part, class;
        double classmin, count;

        while (step())
          {
            classmin = min + width * ((double) class);
            /* make a rectangle */
            pict_rectangle(p + NBARS * part + class,
                0.0, classmin, (part ? count : -count), width);
            /* shade the ones on the right */
            if (part)
              p[NBARS * part + class].pict_fgray = 0.8;
          }
      }
    /* set up the axes */
    pict_autoaxes(p, "Count", "X", "==", &fabs, NULL,
                            "Split histogram", 1);
    /* and make it all appear */
    pict_port(1);
    pict_page();
    pict_show(p);
    pict_end();
    }
  }
```

Appendix I: Settable parameters

The following list of settable parameters and their default values is in `dap.h`:

```
/* Parameters for variables */
DAP_MAXVAR 256  /* max number of variables in a dataset */
                /* if changed to >= 10,000, change dimstr in dap0.c
*/
DAP_NAMELEN 15  /* max length of variable names (+1 for null)
*/
DAP_INTLEN 20  /* max number of char in char representation of
int */
DAP_LISTLEN (256 * (16 + 6))
 /* max length of list of variables: dap_maxvar *
  * (max var name length + room for bracketed index)
  * This may not be entirely safe! (but most likely is)
  */
DAP_TOOLONG 10 /* max # times to print "string too long" message
*/
DAP_STRLEN 63  /* max length of some string values */

/* Parameters for tables */
DAP_MAXROWS 1024  /* max rows for table() */
DAP_MAXCOLS 64  /* max columns for table() */
DAP_MAXCLAB 128  /* max number of column labels */
DAP_MAXROWV 8  /* max number of row variables */
DAP_MAXCOLV 8  /* max number of column variables */
DAP_LABLEN 63  /* max number of non-null char in column label
*/

/* Parameters for datasets */
DAP_SETDIR "dap_sets"  /* where datasets are stored */
DAP_MAXVAL 32768  /* max number of values for some stat functions*/
DAP_MAXCELL 512  /* max number of cells in some internal tables
*/
```

```
DAP_MAXTREAT 9  /* max number of factors for ANOVA */

/* Parameters for grouping */
DAP_MAXBARS 128  /* max number of bars for histograms, grouping
*/
DAP_MAXLEV 96  /* max number of levels of a variable */

/* Parameters for I/O */
DAP_LINELEN 2047  /* max number of char for input line (+1 for
null) */
DAP_OUTREPORT 100000
   /* report multiples of this number of lines written */

/* Parameters for graphics */
DAP_MAXPTS 16384  /* max number of points in a pict */
DAP_MAXCHAR 65536  /* max number of text chars in all the picts
*/
DAP_MAXNTXT 128  /* max number of text chars in a pict */
DAP_MAXTXT 127   /* max number of chars in a single string */
DAP_MAXFONT 63   /* max number of chars in a font name */

/* Parameters for numerical algorithms */
DAP_REDTOL 1e-9 /* to tell if row is zero in reduction */
DAP_ORTHTOL 1e-9 /* to tell if row is zero in orthog */
DAP_ZEROTOL 1e-6 /* to tell if row is zero in matrix ops */
DAP_TOL 1e-8    /* for pivoting, etc. in matrix ops */
DAP_CTOL 1e-6   /* for iterative reweighted least sq (logreg)
*/
DAP_KTOL 1e-6   /* for significance of Kolmogorov statistic */
DAP_PRTOL 1e-6  /* for inverse prob functs: should disappear */
DAP_ADDTOZERO 1e-8  /* for contingency tables */
DAP_MAXITER 500 /* max number of iterations */
DAP_MAXEX1 20  /* max number of values for exact test */
DAP_MAXEX2 20  /* max number of values for exact test */
DAP_CATTOL 0.0000005 /* tolerance for convergence in categ() */

/* Parameters for memory files */
DAP_NRFILES 128  /* number of files stored in memory */
DAP_RFILESIZE 16384  /* max number of bytes in a memory file */
DAP_MAXLINES 2048  /* max number of lines in memory file:
                   * keep at dap_rfilesize / 8
                   */
DAP_MAXMEM 1048576 /* memory buffer size for sorting */
DAP_TMPDIR "dap_tmp" /* directory for temporary files for sorting
*/
```

```
/* Memory allocation tracing */
DAP_MEMTRACE NULL  /* if non-NULL, print trace of malloc and free
                    * and if address = dap_memtrace, then...
                    */
DAP_MABORT 0  /* abort on malloc */
DAP_FABORT 0  /* abort on free */
```

Appendix II: Essentials of C syntax

This appendix explains the most basic syntax of C needed for using dap. There are many books to which you can refer for more detailed or more advanced descriptions of C. Note that the GNU editor `emacs` has special provisions for editing C programs.

9.7 Variables and operations

A *variable* is a box in which one or more numbers or characters, or even more complicated objects called *structures*, are stored. You *must* use a *declaration* to indicate exactly what kind of contents the box will hold and what name you are going to use to refer to the box before you use the box to store anything or else the computer won't know what kind of information is stored in the box and how to use it. (Names must contain only letters and numbers and '`_`' and must not start with a number.) For example,

```
int n;
```

is a declaration that indicates that the box named '`n`' will hold an integer (whole number). Non-integer numbers, such as '`2.718`', are referred to as *double precision floating point* numbers, or simply as *doubles*. For example,

```
double x;
```

is a declaration that indicates that the box named '`x`' will hold a double. There is a special value called *NaN*, which stands for *Not-a-Number*, for undefined values of a double.

Numbers and characters and structures can each be grouped into ordered collections known as *arrays*; arrays of characters are called *strings*. For example, the declaration

```
char employee[21];
```

indicates that the box named '`employee`' will hold a string of 21 characters, say, the employee's name. The last character of a string of characters is usually used to hold a special character that marks the end of the string, so that '`employee`' will really be limited to 20 ordinary characters. The character that marks the end of a string is called the *null character* and is denoted '`\0`'.

You can also declare, for example,

```
double y[3];
```

to indicate a box that will hold 3 doubles. Brackets '`[`' and '`]`' also allow you to refer to the individual doubles in the array: the three elements of '`y`' are '`y[0]`', '`y[1]`', and '`y[2]`'. (The numbering of array elements always starts at 0.) Such referencing can be used for arrays of characters, integers, or structures, too.

The most common kind of structure in dap is the '`pict`' structure, which contains graphical information that is used by graphics functions; '`pict`'s are most often collected into arrays. In some cases you may not know for certain or care how many '`pict`' structures are in the array, in which case you can use a declaration like this:

```
pict *p;
```

which indicates that the graphics function that sets up the array of picts to be stored in the box named '`p`' will allow for however many picts that graphics function needs. In this case, '`p`' is usually referred to as a *pointer* to '`pict`', and in that sense, '`p`' is interpreted as the location in the computer's memory of the first '`pict`' in the array. Nevertheless, you can still refer to the '`pict`'s in the array as '`p[0]`', '`p[1]`', etc.

The pointer concept, as a location in the computer's memory, is used in other contexts, too. One use is to be able to distinguish between a string that has no interesting characters in it, that is, a string whose first character is the null character, and a string that doesn't exist at all. The former is called a *null string*, and is denoted '`""`', and the latter is called a *null pointer*, and is denoted '`NULL`'. Another use of pointers involves functions, which are described in the next section. Functions have locations in the computer's memory and can be referred to by pointers: if '`f`' is a function, then '`&f`' is a pointer to that function.

There are many operations that can be used on variables, but the most common operations are copying and arithmetic operations. For example, the *statement*

```
x = y + z;
```

computes the sum of the numbers in the boxes named '`y`' and '`z`' and copies that sum into the box named '`x`'; note that the contents of '`y`' and '`z`' are unchanged. Note that the paradoxical-looking statement

```
x = -x;
```

says to copy the negation of the contents of '`x`' back into '`x`'. Copying strings is more complicated and is described in the next section.

Two very common arithmetic operations that can be used only on '`int`' variables are '`++`', '`--`': For example, the statement

```
n++;
```

increments, i.e., increases by 1, the number stored in '`n`'. Similarly,

```
n--;
```

decrements, i.e., decreases by 1, the number stored in '`n`'.

9.8 Functions

Functions are program parts that are usually stored in *libraries* and can be used from a program. For example, a dap program could contain the statement

```
means("data", "x", "SUM", "");
```

which *calls*, i.e., uses, the function '`means`', which is stored in a library that is supplied with dap, to compute the sum of the values of the variable '`x`' in the dataset '`data`'. The comma-separated, quoted strings '`"data"`', '`"x"`', '`"SUM"`', and '`""`' are called *arguments* to the function and they tell the function what to operate on and how. This particular function will create a dataset named '`data.mns`' that will contain the computed sum.

Two functions that are not supplied with dap but that are available nonetheless in the standard C library and that are commonly used in dap programs are '`strcpy`' and '`strcmp`'. The first is used for copying strings and the second is used for comparing strings. For example, following the declaration

```
char employee[21];
```

you could write the statement

```
strcpy(employee, "Bassein, Susan");
```

which calls the '`strcpy`' function to copy the 14 characters between the quotes into the string '`employee`' and follow it with the null character, for a total of 15 characters. NOTE: An explicit string of characters must be enclosed in double quotes ("..."), not single quotes ('...'). WARNING: Serious program bugs can be caused by copying more characters into a string than are allotted in the declaration!

The '`strcmp`' function is typically used in loops and '`if`'s and is described in the next section.

Functions can also *return a value*, which means that the function can provide a value that can be copied into a variable. For example, after the declaration

```
int more;
```

you could write

```
more = step();
```

This statement calls the function '`step`', which attempts to read one line from the input data file or dataset and then returns the integer 1 or 0, if there was or was not, respectively, another line to be read from the file or dataset. (Note that although '`step`' does not require any arguments, the parentheses are still necessary, to identify it as a call to a function.) Thus, '`more`' would contain the value 1 if '`step`' successfully read another line or the value 0 if the previous call to '`step`' had read the last line in the input data file or dataset.

You can also define functions in your program. In particular, '`main`' is a function that must appear in every dap program. The *body* of the definition of a function must always be enclosed between '`{`' and '`}`'. The example program cda262.c (see Section 9.3 [Categorical data analysis examples], page 83) illustrates the definition and use of a function required by the categorical data analysis function '`categ`' to compute expected cell frequencies from parameter values.

9.9 Loops and '`if`'s

Loops allow an action to be repeated, either until some condition arises or for a fixed number of times. The most common loop in dap programs looks like this (with possible enhancements):

```
while (step())
  output();
```

This loop repeats reading a line from the input data file or dataset and writing the data to the output dataset until there are no more lines to be read. More specifically, it attempts to read a line and if there was a line to be read, it writes the data to the output dataset and attempts to read the next line; otherwise it *breaks out of the loop* and continues to the next line of the program. Note that a value of 1 (or, in fact, any non-zero value) is taken to mean *TRUE* and a value of 0 is taken to mean *FALSE*, so that these lines can be interpreted as saying, "While (i.e., as long as) it is TRUE that step has read another line, output the data from that line".

A '`for`' loop can be used to read, say, the first 100 lines of the input data file or dataset:

```
int n;

for (n = 1; n <= 100; n++)
  {
    step();
    output();
  }
```

The '`for`' statement has three parts within the parentheses, separated by semicolons: the *initialization*, '`n = 1`', which starts the loop; the *test*, '`n <= 100`', which acts as though it is inside the parentheses in a '`while`' statement; and the *increment*, '`n++`', which gets performed after each repetition of the body of the '`for`' loop. (Note that the body of the '`for`' in this case must be enclosed between '`{`' and '`}`' because, unlike the body of the '`while`' above, it contains more than one statement.)

The previous example will fail if there are fewer than 100 lines in the input data file or dataset because '`step`' will stop your program if you try to read another line of the input data file or dataset after a previous call to '`step`' has returned a 0. One better alternative would be to use an '`if`':

```
int n;

for (n = 1; n <= 100; n++)
  {
    if (step())
      output();
    else
      break;
  }
```

In this example, the '`break`' causes the program to break out of the loop when '`step`' returns a 0.

Suppose you wanted to write into the output dataset all the lines from the input data file for which the '`employee`' was named "Bassein, Susan". You could use the *logical negation* operator '`!`' with the string comparison function '`strcmp`' as follows:

```
char employee[21];

while (step())
  {
    if (!strcmp(employee, "Bassein, Susan"))
      output();
  }
```

because '`strcmp`' returns 0 if the strings are the same and a non-zero number otherwise and '`!`' turns a 0 into a 1 and a non-zero integer into 0.

The C language provides two operators that allow you to combine conditions to be tested in '`while`', '`for`', and '`if`' statements: logical *and*, '`&&`', and logical *or*, '`||`'. For example, the previous code could be modified to select only those records for "Bassein, Susan" that had a value greater than or equal to 10 for a variable named '`hours`':

```
char employee[21];
double hours;

while (step())
  {
    if (!strcmp(employee, "Bassein, Susan") && hours >= 10.0)
```

```
          output();
      }
```

Appendix III: Troubleshooting

Many things can go wrong with a program. At the simplest level, there can be an error in *syntax*, which would cause the preprocessor or the compiler to be unable to interpret what is meant by a statement. For example, a syntax error commonly made by novices is to forget to include the semi-colon that is required at the end of every declaration or statement. Another common error is to have a left brace '`{`' without a matching right brace '`}`' (or vice versa); using appropriate indentation in your program and the features of your text editor will help you find unmatched braces. If your program has a syntax error, you will get a message indicating the line number of the statement containing the error and why the preprocessor or the compiler couldn't interpret that statement. (Missing semi-colons often confuse the compiler so badly that the error message can be hard to interpret.) In that case, you simply edit your program to correct the error and compile and run it again.

If your program uses too many variables, lines that are too long, variable names or strings that are too long, etc., then you will get a message that indicates the problem during execution. You can solve that problem by setting the value of the appropriate parameter as described in Section 2.1 [Reading files and datasets], page 9.

More serious bugs will result in the program *crashing*, which means that the program will stop before completing its task, often with no output of any sort. In that case, the system will produce a file named `core`, which is a snapshot of what point the program reached when it crashed. If that happens, exit dap and use a *debugger* to examine the core file. For example, to use the GNU debugger `gdb` on the core file produced by a crash in prog.c, you would type

```
gdb prog.dap core
```

Then, typing

```
where
```

will tell you the number of the line reached by your program, and the functions that it called, when the program crashed. A debugger is a powerful and somewhat complicated program; you should read at least some of its documentation before you use it.

If a program doesn't crash but still doesn't produce output, most probably it is executing in an *infinite loop*, that is, a loop whose condition for terminating is never met. Such a situation is particularly dangerous if there is a call to '`output`' in tha loop, so that more and more lines are being written to the output dataset, which therefore runs the risk of completely filling up your disk. You can terminate a program by typing *control-C* (*twice*, if you are using the Emacs interface) which means pressing both the *`Control`* (or *`Ctrl`*) key and the *`C`* key at the same time.

Still more serious bugs will result in the program giving nonsense results. Often, you can track down such bugs by inserting calls to the dap '`print`' function at strategic points in your program, and running the program again, to display intermediate files to see where the program started to have problems. If that fails, a debugger is a good tool for tracking down unintended values of your variables.

The most serious bugs will result in the program giving sensible-looking but wrong results. To catch bugs of that sort requires a thorough understanding of approximately what your results should look like, some experience with the statistical functions you are using, and a lot of patience.

Appendix IV: SBS

This section contains miscellaneous notes and warnings regarding the use of SBS. **WARNING**: When processing *file*.sbs, dap writes (or overwrites) the file *file*.c. Note that because the syntax and options of SBS and dap graphics procedures are not very compatible, many dap graphics options are not available in SBS (which is one reason that I wrote dap in the first place); for complete access to dap graphics options (or statistical options) from SBS, use a '`proc dap`' statement (see below). In addition, you can define your own functions in a separate '`.c`' file that you include on the command line to run dap. (To declare such functions in the '`.sbs`' file, use a '`proc dap`' statement at the beginning of the file.) Note that if the program does not use a graphics proc, compiling will produce a number of warnings about unused variables, all of whose names begin and end with an underscore '`_`'; you may ignore them.

One peculiarity that SBS inherits from C through dap is that character variables that are explicitly used in a data step always need to appear in a '`length`' statement, no matter whether the data step gets its data from a '`set`', a '`merge`' or an '`infile`' statement.

Important notes regarding data steps (at least through this version of dap):

1. Comparison, assignment, and concatenation of strings must be performed using the C functions '`strcmp`' (or its variants), '`strcpy`' (or its variants), and '`strcat`' (or its variants);
2. C-style subscripting (i.e., with '`[ ]`') must be used for obtaining individual characters from a string except that in SBS, array indexes start at 1, not 0 as in C.
3. Mnemonics are not accepted for relational or logical operators (e.g., '`ne`' is not accepted for '`^=`');
4. The only statements accepted in the body of a data step, other than numerical assignment statements, are '`if-then`', '`if-then-else`', '`do`', '`do while`', '`end`', and '`output`';
5. *first*.variable is allowed by '`by`' groups, but not *last*.variable.
6. Syntax errors in the body of a data step may not be caught by the SBS-to-dap translator, but will subsequently be caught by either the dap preprocessor or the C compiler. However, the line numbers referred to by

those programs will be the line numbers of the dap program into which your SBS program was translated, which you can view by opening up file *file*.c if you ran dap on *file*.sbs.

data [*output-file-name* [({drop | keep}=*variable* [... *variable*])];
length *variable-1* $ *length-1* [... *variable-n* $ *length-n*];
infile "*infile-name*" [{delimiter | dlm}="*delimeter*"]
[firstobs=*n*];
input *variable-1* [*start-col-1* [-*end-col-1*]]
[... *variable-n* [*start-col-n* [-*end-col-n*]]];
set *dataset-name*;
merge *dataset-name-1* [({drop | keep}=*variable* [... *variable*])]
dataset-name-2 [({drop | keep}=*variable* [... *variable*])];
by *variable-list*;
{drop | keep} *variable-list*;

SBS note: for column input, it is not possible to skip columns, i.e., '`start-col-1`' must be 1 and each succeeding '`start-col`' must be one more than the preceeding '`end-col`'.

proc dap;
dap-statement

SBS note: *dap statement* can either be enclosed in braces '`{ }`' or end with a semi-colon '`;`'. Attempting to use dap graphics and SBS graphics in the same program will produce unpredictable and, perhaps, undesirable results. Note that *dap statement* is run as-is by dap and, in particular, indexes of array subscripts start at 0. The *dap statement* cannot use the '`0x`' prefix for hexidecimal numbers.

Frequently Asked Questions

9.10 I cannot find a function

Q: I cannot find a function to delete spurious zeros from my dataset without deleting anything else. What do I do?

A: You have to write some C code (see Chapter 9.6 [Appendix II], page 95) for that:

```
while (step())
  {
    if (yield > 0.0)
      output();
  }
```

9.11 I want to split a dataset

Q: I want to split a dataset based on the value of a variable. How do I do that?

A: Assume that the variable to be used is a character variable named '`part`' with 9 characters (including the terminating null) and that dataset `set` is sorted by '`part`'. The following code fragment creates a dataset for each value of '`part`' and names it that value.

```
inset("set")
  {
    char part[9];
    int partv[1];

    daplist("part", partv, 1);
    if (step())  /* to get started */
      {
        outset(part, "");
        output();
        while (step())  /* to continue */
          {
            if (dap_newpart(partv, 1))
              outset(part, "");
```

```
                output();
              }
            }
          }
```

9.12 Logistic regression reported a singular matrix

Q: Logistic regression reported a singular matrix, but my data seems OK.

A: You may need to lower the value of DAP_CTOL (see Chapter 9.6 [Appendix I], page 91), which is used to zero-out matrix entries that appear to be non-zero only as the result of round-off errors.

9.13 How do I add a line to a plot?

Q: I plotted a dataset using '`plot`' and I want to add a line to it. How do I do that without knowing the scaling and translating used by '`plot`'?

A: In your call to '`plot`', specify '`MINX`', '`MAXX`', '`MINY`', and '`MAXX`'. Don't use '`nport`' or '`nland`'; rather use '`pict_port`', '`pict_land`', or '`pict_init`' to start the graphics section of your program (and don't forget '`pict_page`', '`pict_show`', and '`pict_end`'). Then declare an array of 2 '`pict`'s and use '`pict_initpict`' to initialize them as a list. Then use '`pict_line`' on the first of these 2 picts and call '`pict_autoaxes`' with the same axis specifications (although you can set '`NXTICKS`' and '`NYTICKS`' to 0 if you wish). You can use the same axis labels, overwrite them, or leave them as null strings. Finally, call '`pict_show`'.

Here's an example:

```
#include <dap.h>

void main()
{
  infile (NULL, NULL)
    {
      double x, y;
      int n;
      outset("gtest", "");
      for (n = 0; n < 10; n++)
        {
          x = varnorm();
          y = varnorm();
          output();
        }
    }
    {
      pict *p, p1[2];

      pict_port(1);
      pict_page();

      p = plot("gtest", "x y", "", "== MINX-4 MAXX4 MINY-4 MAXY4",
               NULL, NULL, 1);
      pict_show(p);

      pict_initpict(NULL, p1);
      pict_initpict(p1, p1 + 1);
      pict_line(p1, -3.0, 2.0, 2.0, -1.0);
      pict_autoaxes(p1, "", "",
                    "== MINX-4 MAXX4 MINY-4 MAXY4 NXTICKS0 NYTICKS0",
                    NULL, NULL, "", 1);
      pict_show(p1);

      pict_end();
    }
  }
```

Index