OPERATING MANUAL FOR GOSSET: A GENERAL-PURPOSE PROGRAM FOR CONSTRUCTING EXPERIMENTAL DESIGNS (SECOND EDITION).

R. H. Hardin

N. J. A. Sloane

Mathematical Sciences Research Center
AT&T Bell Laboratories
Murray Hill, New Jersey 07974 USA

ABSTRACT

This is the second edition of the operating manual for `gosset`, a flexible and powerful computer program for constructing experimental designs. Many new features have been added. Variables may be discrete or continuous (or both), discrete variables may be numeric or symbolic (or both), and continuous variables may range over a cube or a ball (or both). The variables may be required to satisfy linear equalities or inequalities, and the model to be fitted may be any low degree polynomial (e.g. a quadratic). The number of observations is specified by the user. The design may be required to include a specified set of points (so a sequence of designs can be found, each of which is optimal given that the earlier measurements have been made). The region where the model is to be fitted need not be the same as the region where measurements are to be made (so the designs can be used for interpolation or extrapolation).

The following types of designs can be requested: I-, A-, D- or E-optimal; the same but with protection against the loss of one run; or packings (when no model is available). Block designs, and designs with correlated errors can also be constructed. The algorithm is powerful enough to routinely minimize functions of 1000 variables (e.g. can find optimal or nearly optimal designs for a quadratic model involving 12 variables). An extensive library of precomputed optimal designs is included for linear and quadratic designs in the cube, ball and simplex.

The user does not have to specify starting points for the search. The user also has control over how much effort is expended by the algorithm, and can monitor the progress of the search. Applications so far include VLSI production, conductivity of diamond films, growth of protein crystals, flow through a catalytic converter, laser welding, etc.

May 12, 1993

OPERATING MANUAL FOR GOSSET: A GENERAL-PURPOSE PROGRAM FOR CONSTRUCTING EXPERIMENTAL DESIGNS (SECOND EDITION).

R. H. Hardin

N. J. A. Sloane

Mathematical Sciences Research Center
AT&T Bell Laboratories
Murray Hill, New Jersey 07974 USA

This page is intentionally blank

CONTENTS

1. Introduction

This is the second edition of the operating manual for `gosset`(*), a general-purpose program for constructing response surface designs. (This October 1992 edition is a substantial revision of the original 1991 manual, and incorporates changes made up through version 225 of `gosset`.)

Four basic steps are required to run `gosset`. First, the design is specified by a series of numbered lines, somewhat like a `BASIC` program (see Sect. 3).

Second, this "program" is compiled using the `compile` command (see Sect. 4).

Third, a moment matrix for the design is computed using the `moments` command (see Sect. 4).

Fourth, the `design` command is run to search for one or more designs of the specified type (see Sect. 4).

At the end of this, `gosset` reports the name of the file that contains the best design found. The design can be inspected using the `interp` command (Sect. 5.2), and is then ready for use.

The theory behind this program is described in [HS4]. The program also includes a large library of precomputed designs for various applications (in the file `codelib.a` — see Sect. 10). Some of the designs in this library are described in [HS1]-[HS4] and [HS7]-[HS11], and a complete description will be found in [HS5]. A number of experiments using these designs have already been run, in areas such as VLSI manufacture, conductivity of diamond films, growth of protein crystals, flow through a catalytic converter, laser welding, etc. Some of these are described in [HS1], [HS2], and we plan to discuss some of the others in a separate publication.

`gosset` also has a small but growing library of "classical" designs, `classic.a` — see Sect. 10.4.

`gosset` only constructs designs. We assume that you will use one of the standard statistical programs (such as `S` [BCW88], [CH91]) to analyze the data obtained from running the design.

The program consists of about 12000 lines of `C` code, and at present only a `UNIX`® version is available. It is anticipated that it will be distributed as part of the `S` language (cf. [BCW88], [CH91]).

This manual is enclosed with the `gosset` programs in a postscript file called `manual.ps`. To print it, type

```
lp manual.ps
```

(possibly replacing `lp` by your local printer command, e.g. `lpr`). The second page is deliberately blank.

For further information contact N. J. A. Sloane, Room 2C-376, AT&T Bell Labs, 600 Mountain Ave, Murray Hill, New Jersey 07974 USA; telephone: (908) 582 2005 (voice), (908) 582 3340 (FAX); electronic mail: njas@research.att.com .

(*) The program is named after the amateur mathematician *Thorold Gosset* (1869-1962), who was one of the first to study geometrical structures in six, seven and eight dimensions [Co73, p. 164], and his contemporary, the statistician *William Sealy Gosset* (1876-1937), who was one of the first to use statistical methods in the planning and interpretation of agricultural experiments [PK70, p. 382]. Although from our geometric viewpoint their work is related, we do not know if the paths of Thorold (Cambridge, London, lawyer) and William Sealy (Oxford, Dublin, brewer) ever crossed.

Copyright notice: The program gosset was developed at AT&T Bell Labs in 1991-1993 by R. H. Hardin and N. J. A. Sloane, and is copyright 1992, 1993 by R. H. Hardin and N. J. A. Sloane.

2. Getting started

Four simple steps are needed to find a design using gosset.

1. You specify the design with a series of numbered lines, somewhat like a BASIC program (see the example below, and Sect. 3 for more details).

2. You "compile" this "program" using the compile command (see Sect. 4.2 for more details).

3. You run the moments command (see Sect. 4.3 for more details).

4. You run the design command to search for one or more designs of the specified type (see Sect. 4.4 etc. for more details).

At the end of this, gosset reports the name of the file that contains the best design found. The design can be inspected using the interp command (Sect. 5.2), and is then ready for use.

Here is an example. Of course this illustrates only a few of the features of gosset. Many other examples will be found in Sect. 7.

Suppose we have three continuous variables x, y, z on 0 to 1, two further continuous variables u, v lying in a sphere(**) of radius 1 centered at (1,1), and a three-level numerical discrete variable d taking the values 1, 2 and 3.

In addition, there is a mixture constraint: $x + 2y + 3z = 1$, and also an inequality $d > u + x$ that must be satisfied. We wish to model the response as a full quadratic function of the variables. There are six variables, but one is eliminated by the mixture constraint, and so there are 21 unknown coefficients in the model. A minimal design will therefore require 21 measurements.

We start gosset :

```
$ gosset
please type 'cd something' to name a local directory for your work
cd testdir
```

and then type the following:

```
10 range x y z 0 1
20 sphere u v center 1 1 radius 1
30 discrete d 1 2 3
40 constraint x+2*y+3*z=1
50 constraint d>u+x
60 model (1+x+y+z+d+u+v)^2
```

This "program" specifies the design, and we compile it, compute a moments matrix for the problem, and then run 20 attempts at finding a minimal-sized design:

(**) Strictly speaking, in a circular disk. But for uniformity we shall refer to spherical regions of all dimensions (intervals, circles, disks, spheres, balls, 4-dimensional hyperspheres, etc.) as "spheres", and cubical regions of all dimensions (intervals, squares, cubes, 4-dimensional hypercubes, etc.) as "cubes". These regions always include all interior points: they are "solid" or "closed".

```
compile
moments n=1000000
design n=20
```

Each of these commands produces various lines of output which will be discussed in later sections. The output lines

```
compile done
moments done
design done
```

signify the completion of the successive stages. (See also the `wait` command, Sect. 8.20.)

We can then see the best design by typing `interp`:

```
interp

#1:d>u+x
#2:x+2*y+3*z=1

testdir/v.21.best
 d      x      y      z      u      v       #1     #2

1.0000 0.2087 0.0102 0.2569 0.7913 0.0220  0.000  0.5139
2.0000 0.0000 0.2239 0.1840 0.2225 0.7579  1.778  0.3681
3.0000 0.0000 0.0000 0.3333 0.1447 0.4819  2.855  0.6667
3.0000 0.0000 0.5000 0.0000 0.6219 0.6678  2.378  0.0000
3.0000 0.9995 0.0000 0.0002 1.9273 0.6256  0.073  0.0003
3.0000 0.0000 0.1524 0.2317 0.6570 1.9393  2.343  0.4635
3.0000 0.0000 0.0425 0.3050 1.8099 0.6383  1.190  0.6100
2.0000 0.0000 0.4670 0.0220 1.7183 0.3042  0.282  0.0440
2.0000 1.0000 0.0000 0.0000 0.2834 0.3026  0.717  0.0000
2.0000 0.5627 0.2187 0.0000 1.0348 0.9996  0.403  0.0000
3.0000 0.5133 0.0000 0.1622 0.9067 1.0514  1.580  0.3245

3.0000 1.0000 0.0000 0.0000 1.0109 1.9999  0.989  0.0000
1.0000 0.0000 0.0000 0.3333 0.2116 1.6152  0.788  0.6667
2.0000 0.3499 0.0000 0.2167 1.6501 1.7598  0.000  0.4334
3.0000 0.1726 0.3771 0.0244 1.8483 1.5295  0.979  0.0488
1.0000 0.1175 0.4413 0.0000 0.2019 0.3974  0.681  0.0000
3.0000 0.3744 0.1974 0.0769 1.0365 0.0007  1.589  0.1538
1.0000 0.0000 0.2926 0.1383 1.0000 1.4399  0.000  0.2765
1.0000 0.6904 0.0268 0.0853 0.1295 1.4921  0.180  0.1706
2.0000 0.0297 0.4851 0.0000 0.5714 1.9035  1.399  0.0000
3.0000 0.5673 0.1909 0.0170 0.0198 1.1981  2.413  0.0339
```

The 21 rows of this array specify the design. The last two columns correspond to the constraints. The precise meaning of these columns does not concern us here; as long as the numbers in these columns are not negative, the constraints are satisfied.

Unless instructed otherwise, `gosset` attempts to find an I-optimal design, that is, one for which the average prediction variance *IV* (defined in Sect 12) is minimized.

A-, D- and E-optimal designs can also be obtained, as well as designs to protect against the loss of one run, designs with correlated errors, block designs, and packings — see Sect. 4.

The average prediction variance or *IV*-value of the design we have just found can be obtained by typing `iv`, and is 0.68109.[***]

Typing

```
design n=20
```

produced a design with the minimal number of experiments. More generally we could say

```
design type=D runs=25 n=16 processors=4
```

to look for a D-optimal design with 25 measurements, taking the best of 16 attempts, and using 4 processors to do the search. (See Sect. 4.4 etc. for more details.)

Type `quit` to leave `gosset`.

3. Specifying a design

3.1. Overview: `gosset`

First it is necessary to enter `gosset`, by typing

```
$ gosset
```

(See Sect. 9 for how to install `gosset`.) The program responds by asking you to name a working subdirectory:

```
please type 'cd something' to name a local directory for your work
```

to which we might reply

```
cd testdir
```

A design is then specified by typing a series of numbered declarations in almost any order. For example

```
10 sphere x y
20 discrete d 1 2
30 range u
40 model (1+x+y+d+u)^2-d^2
```

says that there are two variables x and y that lie within a sphere (actually a circular disk, but see footnote 2) with center 0 and radius 1, a discrete numerical variable d that takes on the values 1 and 2, and a cubical variable u that lies between -1 and 1.

The model (specified in line `40`) is a quadratic that includes all products of x, y, d and u, but omits d^2 (since d takes on insufficient values to determine the coefficient of d^2).

There are three types of specification for variables: `sphere`, `range` and `discrete`. Other possible specifications are `comment`, `misc`, `constraint`, `model`,

---

(***) The *IV*-values of the 20 designs found were, in increasing order, 0.68109, 0.68277, 0.68434, 0.68563, 0.70027, 0.71198, 0.71583, 0.72016, 0.72022, 0.72095, 0.72821, 0.73592, 0.73822, 0.73850, 0.73944, 0.74861, 0.75841, 0.76005, 0.76617, 0.78907, which suggests that the design `v.21.best` is close to *I*-optimal. In fact, a later and more extensive search using 100 random starts failed to find a better design.

`set`, `use`, `start`, `samplecv`, `cmatrix`. These are described in the following sections.

Incidentally, we have found that `gosset` specifications provide a very convenient language for talking about experimental designs. A specification such as

```
10 range x y z
20 model (1+x+y+z)^2-x*z+x*y*z
design type=I runs=24
```

conveys a precise description of a design in only a few lines.

The `gosset` command has several options: `gosset -echo` causes input lines to be copied to the standard output. Other options are described in Sect. 4.2 and 4.12.

It is possible to run several different `gosset`s simultaneously in different working subdirectories.

3.2. Specifying spherical variables: `sphere`

A `sphere` specification lists all the variables that together must lie within one sphere. There may be several `sphere` lines, but separate lines specify separate spheres.

For a `sphere`, we can optionally give a `radius` or a `center`, as in

```
10 sphere center 2 -1 x y radius .5
```

The center, radius and list of variables can appear in any order in the line. The number of coordinates for a `center` must agree with the number of variables.

If we call that a contravariant specification of scaling, we may also use a covariant specification, as in

```
10 sphere 2*(x-2) 2*(y+1)
```

This produces the same result but is often more natural.

For example, if we want to avoid simultaneous extremes of pressure and temperature, we might put them in a sphere:

```
10 sphere (temperature-98.6)/10 (pressure-30)/2
```

in covariant form, scaling the units to a common radius. Expressions may be arbitrarily complicated, but have to reduce to a linear term in one unknown plus a constant.

We may combine the two forms:

```
10 sphere (temperature-98.6)/10 (pressure-30)/2 radius 2
```

if we first equalize the units of pressure and temperature and then allow them to vary a bit.

Unless specified otherwise, the `sphere` specification defaults to `radius 1` with `center` at the origin.

3.3. Specifying cubical variables: `range`

A `range` specification lists cubical variables that simply lie between two values:

```
10 range 0 1 u v w
```

where u, v and w must lie between 0 and 1. It is immaterial whether `range` variables are listed together or on separate lines.

Both covariant and contravariant forms may be used, as in

```
10 range 0 1 (u-10)*(20-10)
```

If the range limits are omitted, the variables lie between -1 and 1.

3.4. Specifying discrete variables: `discrete`

There are two kinds of `discrete` variables, *quantitative* variables taking numerical values, and *qualitative* variables taking nominal or symbolic values. For example

```
10 discrete 1 2 4.1 r s
```

specifies that *r* and *s* are quantitative variables taking the values 1, 2 and 4.1, while

```
20 discrete "glass" "plastic" material
```

specifies that *material* is a qualitative variable taking the values *glass* and *plastic*.

Equally spaced numerical values may be specified with the `#values` keyword:

```
30 discrete h i #values 101
```

defines *h* and *i* to have 101 values from -1 to 1 at intervals of .02. The covariant form can be used to rescale this.

The default specification for `discrete` is a quantitative variable taking the two values -1 and 1.

Qualitative variables that take only two levels can usually be handled as if they quantitative variables. But things are more complicated when there are three or more levels that have no numerical significance. In this case it is best to code the levels as dummy binary variables, as illustrated in Sect. 7.21 (see also p. 32 of [CH91]).

3.5. Specifying constants: `misc`

There is a `misc` type that can be used to declare constants:

```
10 misc R=.1
20 sphere radius R x y
```

Constants can be of arbitrary complexity so long as they evaluate to a constant, and may use constants previously defined. They are defined as encountered reading from left to right, except `R=(X=2^.5)/2` defines `X` to be the square root of 2, and `R` to be half this, as in the `C` language.

It is recommended (although not essential) that upper case names be used for constants.

Constants can also be defined in `sphere`, `range` and `discrete` lines wherever a constant is used:

```
10 range LOW=0 HIGH=1 u
20 range LOW HIGH v
```

These four types, `sphere`, `range`, `discrete` and `misc` are processed in the order encountered and before any other types.

3.6. Specifying constraints: `constraint`

Linear constraints are given by the `constraint` type:

```
40 constraint x+y-2*z<.8
```

or

```
40 constraint x+y-2*z<=.8
```

which limits an expression that evaluates to linear terms in the variables plus a constant.

Warning. Since all calculations are carried out in floating point arithmetic, there is no distinction between $<$ and $\le$ constraints, or between $>$ and $\ge$ constraints. The user must pay particular attention to this when dealing with inequalities involving discrete variables. For example

```
10 discrete 0 1 x y z
20 constraint x+y+z<2
```

is represented internally as $x+y+z \le 2$, and so the point $x=1,\ y=1,\ z=0$ would be permitted. If this is not what was intended,

```
20 constraint x+y+z<1.5
```

will exclude such points.

Equality constraints such as

```
110 constraint x+y+z=1
```

are special in that they reduce the dimension of the problem by one, and an internal variable must be eliminated. So there is a restriction on equality constraints, imposed to to facilitate this: at least one of the variables in an equality constraint must be a `range` variable.

Each equality constraint then reduces the internal dimension by one, and also adds a pair of inequality constraints. The latter express the bounds on the range of the eliminated variable in terms of the remaining variables. One or both of the new inequalities may show up in the compiler output.

Only one inequality sign may appear in any single constraint, and so multiple constraints must be imposed one at a time. To impose the double constraint $x<y<z$, for example, rather than typing

```
50 constraint x<y<z
```

which produces an error message, we must say

```
50 constraint x<y y<z
```

Simple constraints such as `.01<w<.05` can be imposed in `range` lines rather than `constraint` lines. (This may or may not increase the efficiency of the Monte Carlo calculation performed by the `moments` command. For if the constraint form is used, `gosset` will probably adjust the random number generator for the constraint. But removing constraints from the problem makes the search faster). Thus instead of

```
10 range 0 1 w x y z
20 constraint .01<w w<.05
```

it may be better to say

```
10 range 0 1 x y z
20 range .01 .05 w
```

so long as it's still clear what's being done.

(A disadvantage of removing constraints in this way may show up in mixtures. Reducing the range of this variable causes it to be handled differently from the others — its internal coefficient will be different — and `gosset` may then be unable to apply its special corner-simplex algorithm to a constraint such as `constraint w+x+y+z=1`.)

Constraint lines are processed next after `sphere`, `range`, `discrete` and `misc` lines, so they can appear anywhere.

3.7. Specifying the response surface: `model`

The `model` is specified symbolically, by an expression such as

```
200 model (1+x+y+z)^2-x^2
```

Here `+` means union, and `-` means "take away if it exists." We are dealing with sets, but the meaning of multiplication is intended to be obvious: form the products indicated and keep one representative of each type of term. Thus the above example defines a model containing the terms `1, x, y, z, x*y, x*z, y^2, y*z and z^2` (but not `x^2`).

Apart from some minor differences, this is also the way models are specified in the `S` language [CH91].

A full linear model would be specified by (for example)

```
200 model 1+x+y+z
```

a full quadratic model by

```
200 model (1+x+y+z)^2
```

which defines a model containing the terms `1, x, y, z, x^2, x*y, x*z, y^2, y*z and z^2`; and a model containing main-effects and interactions only by

```
200 model (1+x+y+z)^2-x^2-y^2-z^2
```

which defines a model containing the terms `1, x, y, z, x*y, x*z and y*z.`

The `1` is there to guarantee that all lower-order terms are included in the model. This is not essential, and

```
200 model (x+y+z)^2
```

would attempt to specify a model containing only the pure quadratic terms `x^2, x*y, x*z, y^2, y*z, z^2`. But this is a non-hierarchical model, and some care is needed when using such models, as we now discuss.

The model need not be hierarchical (i.e. translation-invariant), but if it is not there are two complicating factors to be kept in mind.

The first complication is that all internal variables are scaled to run from `-1` to `+1`, and the model is then expressed in terms of the internal variables. Therefore if `x` (say) is in the model, `1` should be there too, so that the model has the same functional form regardless of translations. Using the full form `(1+x+y+z)^2`, and removing high powers if necessary, guarantees that all the terms will be there.

If a model is wanted that is not translation invariant, say involving only `1` and `x^2`, `x` must be centered on zero, so that no translation is needed between it and its internal mate. (Otherwise the designs may not be optimal for the desired model.)

For this reason, `gosset` warns you if the model is not translation-invariant.

For example, to define a model involving only `x^2, y^2 and z^2`,

```
10 range x y z
20 model x^2+y^2+z^2
```

works successfully (since the default for range is `-1` to `+1`), but

```
10 range 0 1 x y z
20 model x^2+y^2+z^2
```

does not. (In the second example `x`, `y` and `z` get rescaled. In both cases, the designs will be optimal for a response surface that is symmetric about the origin in the internal variables, which is not what is intended in the second example.)

A second complication is caused by variables that appear in equality constraints. If such a variable appears in the model, the other variables in that constraint may also be forced into the model if that variable gets eliminated internally. For example

```
100 constraint x+y+z=1
200 model (1+x)^2+y+z
```

may be interpreted as `model (1+y+z)^2+y+z` (in other words `model (1+y+z)^2`), if `x` has been eliminated; or as `model (1+x)^2+x+z+z` (in other words `model (1+x)^2+z`), if `y` has been eliminated.

Since only `range` variables are eliminated from equality constraints, the user could prevent `x` from being eliminated by changing it to a `sphere` variable on a line by itself.

Note that the eliminated variable is replaced by only the linear terms in its formula. So for example

```
40 constraint x=1+y+z
50 model x
```

is the same as

```
40 constraint x=1+y+z
50 model y+z
```

not

```
50 model 1+y+z
```

Similarly,

```
40 constraint x=1
50 model x+temp+H2O
```

is the same as

```
40 constraint x=1
50 model 1+temp+H2O
```

and

```
40 constraint x=0
50 model x+temp+H2O
```

is the same as

```
40 constraint x=0
50 model temp+H2O
```

As long as we are using a hierarchical model, there is no harm in mentioning redundant variables in the model. For example, for a simple mixture with a full quadratic model,

```
10 range 0 1 w x y z
20 constraint w+x+y+z=1
30 model (1+w+x+y+z)^2
```

and

```
10 range 0 1 w x y z
20 constraint w+x+y+z=1
30 model (1+x+y+z)^2
```

both specify the same model with 10 unknown coefficients. (The first form is the recommended one.)

The specification `set` allows sets to be assigned names for use elsewhere:

```
200 set A=(x+y+z) B=(d+e)
210 model (A+B+1)^2-B^2
```

3.8. Specifying runs that must be included: `use`

The specification `use` forces a given point to appear in the solution (for example, runs made in an previous experiment, the results of which we do not want to waste). The program will then produce the best design that includes these points. For example

```
300 use x=.9 y=.3 material="plastic"
```

Inequality constraints may be violated by `use` points. It may be one of these that suggested the wisdom of the constraint!

However, *equality* constraints must be satisfied by `use` points. (This is necessary, because internally a variable has disappeared.) The constants can be algebraic expressions, which may help in carrying out the necessary calculations.

3.9. Specifying starting points for the search: `start`

`start` points are identical to `use` points, except they specify that the search for optimality should start at the given value, but need not stay there. They will be made to satisfy the inequality constraints very quickly.

3.10. Names for variables

Any variable appearing in `sphere`, `range` or `discrete` declarations can be given an alternative name, which will be used in the `interp` output. For example

```
10 sphere radius 2 ((x=temperature)-98.6)/10 ((y=pressure)-30)/2
20 constraint x<100
```

The compiler knows that `temperature` is a name and not a constant because it is undefined and appears in the right place.

`gosset` has no provision for line-continuation. Lines can be as long as you wish (with a limit of about 16000 characters and 500 fields). A carriage-return terminates the line.

`gosset` arranges the variables in the following order: `discrete`, then `range`, then `sphere`; arranged alphabetically within each group, with upper case names preceding lower case ones. Discrete variables with fewer levels precede discrete variables with more levels.

This ordering must be kept in mind if (for example) you are experimenting with the effect of changing a variable from discrete to continuous. For example, in

```
10 range m1 m2
20 discrete temp
30 range temp'
40 constraint m1+temp<=1
```

the program places the variables in the order `temp, m1, m2`; but in

```
10 range m1 m2 temp
40 constraint m1+temp<=1
```

they are placed in the order `m1, m2, temp`. So a moment matrix or design produced for the first problem won't be valid for the second problem, since the coordinates appear in a different order. Changing the name of the variable `temp` to `atemp` would solve this problem, since now the variables appear in the order `atemp, m1, m2` in both cases. Then the same `moments.h` file can be used in both cases, and designs produced for one problem are meaningful for the other.

3.11. Using different regions for measurement and modeling; interpolation and extrapolation

The region where we wish to model the response (the *modeling region*, or *region of interest*) may be different from the region where measurements can be made (the *measurement region*, or *region of operability*). In this case primed variables describe the modeling region, and unprimed variables the measurement region.

For example, to obtain optimal prediction of a quadratic model in a sphere centered at the point (10, 0), by sampling in a sphere centered at the origin, we type

```
10 sphere x y
20 sphere center 10 0 x' y'
30 model (x+y+1)^2
```

The models in the two spaces are equivalent, and either type of variable can appear in the model specification.

The following are some examples of situations where such designs may be appropriate.

(i) Discrete variables, continuous modeling region. It often happens that certain variables can only take discrete values but we wish to model the process over a continuous region. For example

```
10 discrete A 5 10 25 50
20 range A' 5 50
30 range B 100 300
40 range C 80 120
50 model (1+A+B+C)^2
```

specifies a design in which variable A can be measured only at the values 5, 10, 25, 50, while B and C can take any values in their range, and we wish to fit a quadratic model over the solid region $5 \le A \le 50$, $100 \le B \le 300$, $80 \le C \le 120$. This type of problem can be thought of as an *interpolation* design. Of course (see Sect. 12), only the I-optimality criterion makes use of this information. A-, D- and E-optimal designs ignore the modeling region.

(ii) High accuracy is wanted in part of a large operating region. We concentrate on a smaller modeling region where a simple polynomial model is appropriate (cf. Box and Draper [BD87], p. 182). For example

```
10 sphere x y z center 0 0 0 radius 10
20 sphere x' y' z' center .3 .3 .3 radius 1
30 model (1+x'+y'+z')^2
```

defines the measurement region to be a sphere of radius 10 centered at the origin, and the modeling region to be a smaller sphere centered at (.3, .3, .3); with a full quadratic model.

It may well happen that the modeling region will change after a set of measurements have been made. By including `use` lines (Sect. 3.8) in the program we can construct a sequence of designs, each of which is optimal in the new region given that certain runs have already been made.

(iii) On the other hand it may be undesirable to make measurements at the boundary points of the modeling region, in which case the measurement region is a subset of the modeling region. For example

```
10 range x' y' z' -10 10
20 range x y z -8 8
30 model (1+x'+y'+z')^2
```

defines the modeling region to be a cube with each variable ranging between -10 and 10, and the measurement region to be a slightly smaller cube inside it; again with a full quadratic model.

(iv) Extrapolation. These designs also have an obvious application to extrapolation. For example, we might have a radioactive room in which the radiation in the left half is too intense for measurements to be made. We wish to design an experiment which will enable us to estimate the radiation in the whole room from measurements made only in the right half. The following program does this.

```
10 range x' -1 1
20 range y' z' 0 1
30 range x y z 0 1
40 model (1+x'+y'+z')^2
```

A minimal design for this problem will be found in Sect. 7.14.

A similar situation arises when one of the variables is time, and we wish to design an experiment to extrapolate a response for a whole week on the basis of measurements made during one day.

(v) Astronomical applications. For example, an observer on the earth who wishes to model a response on the moon. See Sect. 7.15.

An inequality constraint applies to the primed space if *any* variable in it is primed; *equality* constraints are for both spaces at once in every case. If any primed variable appears in the program, a complete set of primed variables will be defined, using copies of the unprimed variables where nothing else is given.

If we wish, we can imagine that the primed variables were there all along, but just equal to the unprimed ones.

But if we introduce primed variables just to optimize over a continuous rather than discrete region, a very common purpose, we have to remember that inequality constraints are not duplicated automatically, and we will have to type in primed versions of them. So in this regard, the primed variables were not always there, for they require their own inequality constraints.

There are two restrictions on primed variables.

(a) If an *equality* constraint appears, there must be a variable in it of `range` type, whose image variable, primed or unprimed, must also be of `range` type. This is because equality constraints apply to both spaces, even if given for only one.

(b) The default images for variables not declared with primed versions must be consistent with the declared ones.

For example, consider:

```
10 sphere a b c
20 range a'
30 model (a+b+c+1)^2
```

This is incorrect because the implied definition of `b'` suggests that it lies in a sphere with `a'`, while `a'` has been defined to be a `range` variable. To correct it we simply give a more complete specification:

```
10 sphere a b c
20 range a'
25 sphere b' c'
30 model (a+b+c+1)^2
```

4. Running `gosset` to search for a design

4.1. Overview

After the design has been specified (see the previous section), the next three steps are to compile it, compute a moment matrix, and run the `design` program.

Compilation is performed by typing

```
compile
```

(or simply `c`). This command is described in more detail in Sect. 4.2. The program responds with (among other things) a list of the internal variables used (called $A(0)$, $A(1)$, ... — these always range from -1 to $+1$), a table showing the relations between the original variables and the internal variables, a list of the products of the internal variables that appear in the model, and the number of terms in the model, ending with

```
compile done
```

To compute a moment matrix we type (for example)

```
moments
```

which computes the exact moment matrix, if possible, and otherwise uses the default of

4000 Monte Carlo samples to estimate the matrix. This command is described in more detail in Sect. 4.3. When this computation is completed, the program reports

```
moments done
```

The `moments` command is always required, even for D-optimal designs (or packings) which do not make any use of the moment matrix.

Finally, to search for a good design we type (an elaborate example)

```
design n=10 extra=4 tiny=1e-5 steps=1000,
```

which will pick the best of ten attempts at finding a design with four more points than a minimal design, using a minimal step size of `1e-5`, and stopping each attempt after 1000 steps. A simpler version is

```
design n=10 extra=4.
```

The `design` command is described in more detail in Sect. 4.4-4.6. The program reports (among other things) the *IV*- or *D*-value, etc., of each design found, new record designs when they occur, and, when the computation is complete, a reminder of the name of the file containing the best design.

Warning. It is often tempting to ask for a design with the minimal possible number of runs, using say

```
design n=10
```

or equivalently

```
design extra=0 n=10
```

However, the user should be aware that the *IV*- or *D*-value etc. of a design usually decreases quite rapidly as the number of runs increases above the minimum number (signifying a better design). With only a few more runs it is usually possible to achieve a considerable gain in efficiency. We recommend that the user consider a range of values of `runs` before settling on the actual value to be used. The power of the program makes such experimentation very easy to carry out.

4.2. `compile` or `c`

The `compile` command reads the current program and produces `.h` files that define the search for the optimal design.

The usual sequence for finding a design with `gosset` is

```
cd dir
old
some editing of the program
compile
moments
design
interp
```

If a program is compiled successfully, a copy of it is made in `program.log` in the working directory (if it isn't there already), discarding any `start` points. A four-letter tag will be assigned at this point that will also tag the designs produced and saved in `lib.a`.

The `.h` files created by `compile` are the following:

`constraint.h`, which lists the expressions that must be nonnegative,
`define.h`, which lists the functions in the model and their derivatives,
`interp.h`, which passes information to `interp`,
`random.h`, which generates a random design point,
`renorm.h`, which puts a design point back into its range or sphere,
`regrad.h`, which zeros the normal component of the gradient of the *IV*- or *D*- etc. value, and
`samplecv.h`, which holds user-supplied code from `samplecv` or `cmatrix` program lines.

In addition, in order to optimize discrete variables, there may be parallel files to these which handle discrete variables as if they were continuous (so they have gradients, for example), namely `uconstraint.h`, `udefine.h`, `urandom.h`, `urenorm.h` and `uregrad.h`.

A note on compiling huge problems. The `compile` command has some quadratic time dependence on the number of terms in the model. If the `compile` command doesn't seem to finish, the worst of these can be turned off on a big problem. This can be done by invoking `gosset` with any of five options:

```
gosset -skipdet -skipsym -skipxmom -skipconstr -skiplinear
```

The first bypasses a check of whether the model is singular. (A singular model would occur for example if we tried to specify a quadratic model with two-level discrete variables.)

The second bypasses the search for symmetries in the design (information which can be later used by the `moments` command when averaging moments).

The third bypasses the search that checks if exact moments can be calculated.

The fourth bypasses a check for redundant constraints.

The fifth bypasses a check that linear forms are in fact linear.

However, it's probably true that if a problem is too big to compile then `gosset` is unlikely to find an optimal design. Perhaps the problem should be moved to a faster machine!

The command

```
compile | tail
```

is useful when compiling a large problem as it avoids a lot of uninteresting output.

4.3. `moments` or `m`

As discussed in Sect. 12, a moment matrix *M* is needed to compute the average prediction variance *IV* of a design.

The `moments` command must still be typed even for D-optimal or other designs that do not make any use of the moment matrix.

Each entry in this matrix is the product of a pair of functions in the model, averaged over the modeling region. In simple cases (spheres, cubes, etc.) `gosset` calculates the moments exactly (at the `compile` step), and otherwise evaluates them by a Monte Carlo method. If `gosset` is able to determine the exact moment matrix, `compile` reports

```
0 values missing from exact moments matrix
```

and otherwise tells you how many entries need to found by Monte Carlo estimation. Exact moments are used whenever possible.

The modeling region is necessarily a subset of a region Ω which is a product of spheres, cubes and a finite set, the subset being defined by the inequality and equality constraints. When using the Monte Carlo method, the `moments` program generates points with a uniform distribution over Ω. If the points satisfy all the constraints they are accepted and their contributions towards the entries of the moment matrix are recorded. Any permutation-equivalent coordinates have their moments averaged. If a coordinate is symmetric about zero, all its odd moments are set to zero.

The more samples used, the more accurate is the matrix. It is usually not too serious if the matrix is inaccurate — this corresponds to a slight change in shape of the modeling region. If the region is a sphere, this inaccuracy may impose an orientation on the space, causing the design to seek out certain directions.

The `moments` command computes the moment matrix for a program that has been compiled. The moment matrix is stored in a file `moments.h` that is used by the `design` program.

`moments` has two options. For example

```
moments time=120 n=10000
```

means sample until either 120 seconds have elapsed or 10000 samples have been taken. The default values are `time=600` and `n=4000`.

If the modeling region is simple enough that `gosset` can work out the moments exactly, without using the Monte Carlo method, it is enough to type

```
moments
```

or simply

```
m
```

This simple form of the command is often adequate, since then if the moment matrix is not known exactly the program will then use the default number of samples, 4000, to estimate it.

If the program has not changed essentially since `moments.h` was last written, i.e. nothing but `use` and `start` lines have changed, then running `moments` again will simply add to the moment matrix. So nothing is lost if `moments` is run several times: the moment matrix just gets better.

As soon as a different program is compiled, however, the moment matrix is lost and we start again. This is a good reason to keep different problems in different working directories.

`moments` runs asynchronously, which means we can do other things while it's running, such as `watch moments`, `kill moments`, or `status`. Killing `moments` just causes it to quit where it is, writing a new `moments.h` with what it has at that instant.

It may make sense to run `moments` and `design` each for a short time to see if the design seems reasonable, and then to rerun them for a longer period if all seems well.

Both `design` and `iv` will improve when the `moments` matrix improves in accuracy. And they will get worse if the reverse happens, for example if we start a new problem with a smaller Monte Carlo run. In this case, some good designs for the first problem may be lost, by being overwritten with inferior designs that appear better to a degraded `iv` command. If this is likely to happen the good designs should be saved in a separate directory.

The moment matrix for discrete variables is calculated only at the discrete values. This amounts to asking for a prediction only at the discrete values, a reasonable thing if the discrete values are inherent in the problem.

In some cases, however, this is undoubtedly not what is wanted (for example, when the discreteness is caused just by the present availability of certain supplies for the experiment), and then the modeling region should be the whole continuous space encompassed by the discrete variables.

To make the modeling region continuous over the discrete range, we use primed variables of `range` type matching the discrete variables:

```
10 discrete vial 1/4 1/3 1/2 1.0
20 range vial’ 1/4 1.0
```

(see Sect. 3.11). Then the moment matrix will reflect the `range` variable, while the experiment will use only the `discrete` values.

If the user knows the moment matrix, but `gosset` was unable to find it exactly, then it is possible to run `moments` with say `n=1000` to establish the format of the `moments.h` file, and then to edit this file to change the moments to their exact values. The ordering of terms is the same as in the compiler output, lowest order first.

If the moment matrix is improved in this way, it should be done just after `moments` finishes so that the ordering of file modification times, which `gosset` uses to determine what needs to be recompiled and what is current, remains consistent. If the artificial `moments.h` is saved elsewhere, it should be copied with `cp`, not moved with `mv`, so that its modification time is updated as well.

Sometimes `gosset` thinks that `moments.h` needs to be recomputed when it really does not (for example after one has made two mutually canceling changes to the program). To assure `gosset` that `moments.h` is still correct, it is enough to update its modification time by typing

```
!touch moments.h
```

4.4. `design`

The `design` command attempts to find the optimal design of the specified type that satisfies the conditions of the current program, using (if necessary) the moment matrix from `moments`.

The `design` command has the options

`type=`
`runs=` (or `extra=`)
`n=`
`time=`
`steps=`
`tiny=`
`start=`
`processors=`
`program=`

`design type=I` searches for an I-optimal design (see Sect. 12). `type=i` does the same thing.

`design type=A` searches for an A-optimal design (see Sect. 12). `type=a` does the same thing.

`design type=D` searches for a D-optimal design (see Sect. 12). `type=d` does the same thing.

`design type=E` searches for an E-optimal design (see Sect. 12). `type=e` does the same thing.

`design type=J` searches for a design which minimizes the *IV*-value of any of the subdesigns formed by dropping one run (see Sect. 4.8). `type=j` does the same thing.

`design type=B` searches for a design which minimizes the *A*-value of any of the subdesigns formed by dropping one run (see Sect. 4.8). `type=b` does the same thing.

`design type=C` searches for a design which minimizes the *D*-value of any of the subdesigns formed by dropping one run (see Sect. 4.8). `type=c` does the same thing.

`design type=F` searches for a design which minimizes the *E*-value of any of the subdesigns formed by dropping one run (see Sect. 4.8). `type=f` does the same thing.

`design type=P` searches for the best packing of the specified number (runs) of points in the design region (see Sect. 4.11). `type=p` does the same thing.

The default `type` is the last one used, or `type=I` if none has been specified yet.

`design runs=34` attempts to find a design with 34 points. If the number of terms in the model exceeds 34, `design` will fail, complaining of a singular initial condition. The default value for `runs` is the number of terms in the model (i.e. a minimal design).

`design extra=6` attempts to find a design with 6 more points in it than the number of terms in the model. It is equivalent to specifying `runs=p+6` (where `p` is the number of terms in the model), but lets the program calculate `p`.

`design n=20` looks for twenty designs, and will keep the best if it's better than the one residing in `v.nn.best`, where `nn` is the number of points in the design. `design n=1` is the default.

`design time=10000` will halt the design after 10000 seconds, the default, keeping the best.

`design steps=10000` will halt the design after 10000 steps, the default. When things go well, convergence takes about 500 to 1000 steps. Even if it has not converged, the design is usually quite good by then. For design types that require a series of convergences, e.g. `B, C, E, F, J`, the default value is set to 1000 steps, which applies to each convergence in the series. If the search for designs will be run unmonitored, it is a good idea to reduce the value of `steps` so that too much time isn't wasted in a single difficult design.

Recommendations. Use `steps=1000` for types `I, A, D, E, P`, and `steps=500` for types `J, B, C, F.`

If our interest in the design points is more theoretical than practical, we may prefer to let the program run to termination, for sometimes the whole solution slowly rotates through a great angle with only a slight improvement. This happens particularly with I-optimal designs in spaces containing spheres, when the `moments` matrix has imposed an orientation of the sphere through a slight inexactitude, and the program hunts out just the right orientation in the sphere.

`design tiny=1e-8`, the default, says that the smallest search step is `1e-8`. This may be too stringent, and many designs have been run with `tiny=1e-5`, which is faster. It is probably better to save `tiny=1e-8` or smaller values for polishing earlier solutions. Much smaller values have no effect, for roundoff is lurking nearby at about `1e-10`, and beyond that point nothing changes.

`design processors=1`, the default, says that only one design attempt should be run at a time. On multiprocessor machines there is a dramatic speedup if this is increased. On a single processor machine, it can be increased also, and then designs will be constructed in parallel. Each can be watched and killed if it appears unsuccessful, making room for better ones.

Multiple processor machines write solutions into files with names like `v.nn.pp` (for I-optimal designs), where `nn` is the number of points in the solution and `pp` is the processor number. The design run log for each is in `vtrace.pp`, where `watch` and `status` find it, and where the user can also find it, to see how the search is progressing.

`design program=...` specifies the optimization strategy to be used, and is described in the next subsection.

The options may be combined in any order, as in

```
design n=10 extra=4 tiny=1e-5 steps=1000,
```

which will pick the best of 10 attempts at finding an I-optimal design with 4 more points than a minimal design, using a minimal step size of `1e-5`, and stopping each attempt after 1000 steps.

The best design found is stored in an archive file called `lib.a` in the working directory. This is a standard `ar` archive that can be manipulated with the `ar` shell command. I-optimal designs in the archive are named `v.D.R.T` where `D` is the geometric or internal dimension, `R` is the number of runs or points in the solution, and `T` is the four-letter program tag. D-optimal designs are named `d.D.R.T`, A-optimal designs `a.D.R.T`, J-optimal designs `j.D.R.T`, and similarly for the other types.

`gosset` also maintains a file called `program.log`, which lists programs, dates and tags. The user may edit `program.log` if necessary. Program names are expected to start with `p.` and end with four lower-case letters. (The editing must be done from outside `gosset`, though. There is no mechanism for calling an editor from inside `gosset`.)

4.5. `design program` options for continuous variables

The `design program=` option specifies the optimization strategy for the search. `gosset` will assume reasonable defaults, and the incurious user needn't pay much attention if all goes well. This section describes the options for continuous variables, and the next section those for discretes.

If only continuous variables are present then the program attempts to optimize the design using *pattern search* (see Sect. 12). This is essentially what the `program=r*v` option does, the default when only continuous variables are present. When both continuous and discrete, or only discrete, variables are present the situation is more complicated and we have only a collection of heuristics (which do however work most of the time). These are controlled by the `program=...` option of `design`.

Programs are specified symbolically.

`r` means generate a random design satisfying all the program constraints, and the `use` and `discrete` requirements. So `design runs=100 program=r` simply places 100 points randomly in the measurement region, and serves as a random point generator for this space.

`v` means minimize *IV* by changing the continuous variables while keeping the discrete variables fixed.

`d` minimizes the *D*-value of the design by changing the continuous variables while keeping the discrete variables fixed.

`a` minimizes the *A*-value of the design by changing the continuous variables while keeping the discrete variables fixed.

`*` is a pipe command. For example `r*v` means generate a random design and then optimize it.

The `+` operator chooses whichever of the left and right sides is better, and passes it on. For example

```
design program=(r+r+r+r)*v
```

generates four random starts and optimizes the best of them, while

```
design program=(r*v+r*v+r*v+r*v)
```

generates four random starts, optimizes each one, and takes the best.

The exponentiation operator `^` iterates the left side the number of times indicated on the right, and chooses the best. For example

```
design program=r^4*v
```

and

```
design program=(r*v)^4
```

are the same as the two previous examples.

The function `1` is the identity function, and just passes on whatever is fed to it. It is a place holder in sums, to ensure that the output is never worse than the input, as in `design program=r*(1+v)*....`

`lib` searches each library in the `search` sequence for solutions that satisfy the constraints of the program, and passes on the best. For example

```
design program=lib*v
```

takes this design and optimizes it for the current problem.

If `lib` appears as an exponent, the left hand side is fed every member of each library in the search sequence that has the right dimension and number of points, regardless of whether it satisfies the constraint of the problem; and the best of the outputs is kept.

The function `check` causes the `iv` of its input to be infinitely bad if it doesn't satisfy all the constraints of the problem, so `design program=check^lib` and `design program=lib` are identical.

The second way, besides using `start` lines in the original program, to start a solution from where another left off, is to use the `start` option. For example

```
design program=v start=filename
```

passes the solution in `filename` on to the program `v` for optimization.

The most common use of this is in refining the best design found with a fairly large value of `tiny`, as in this example:

```
design runs=18 n=20 tiny=1e-5 steps=1000
wait
local best is 4sphere/v.18.best
design runs=18 program=v start=v.18.best
```

Make sure that you say `start=filename`, where `filename` is the name of an actual file. `start=` will not extract a design from a library archive for you.

Essentially, `start=` supplies an input to the left end of the `program=` sequence, a position usually occupied by `r` or `lib`.

Discrete variables are more difficult to handle and are discussed in the next section.

The following `program=` option often greatly speeds up the search. In the `program` field, the symbol _ (underscore) stands for ''whatever the current program is,'' usually the default. This makes it possible to say

```
design program=_^50 n=400 processors=8
```

and the default program will be tried 50 times before the best is chosen and the attempt terminates.

This can be much faster if each attempt is very quick. Since it normally takes a second or so after each attempt to update the library, terminate, and start another attempt, a great deal of time is saved if the number of terminations is reduced.

Presumably it is a good idea to keep the attempt run time longer than a second—in the above example, 50 iterations are done internally, and 400 instances of 50 are tried, for a total of 2000 attempts. Of course making all the iterations internally is not a good idea, since this would not share the processors, and would also conceal how the search is progressing.

Warning. You might think that it would be possible to calculate the `iv` value of an existing design by specifying its points with `use` lines, and using

```
design runs=12 program=1
```

This does not work: `program=1` ignores `use` lines. (For `use` lines instruct `program=r` to create these points and `program=v` to leave them alone. But they are not passed on to `program=1`.) Instead, simply type

```
design runs=12
```

or

```
design runs=12 program=r
```

4.6. `design program` options for discrete variables

If there are not many possibilities for the discrete coordinates of the design points then there are two plausible strategies: either chose them at random, or assign them sequentially in some fixed order, running through them all once before repeating any.

`design program=r*v` does the former, producing a random distribution of discrete values that never change again.

`design program=rj*v` does the latter, running sequentially through all possibilities for the discrete variables that satisfy the constraints, and omitting those that fail. Thus each acceptable discrete combination will appear roughly the same number of times. Again, once the discrete coordinates of the points are chosen, they remain fixed any subsequent optimization.

Both strategies are reasonable, but no optimization is made over the discrete variables, except in so far as we take the best of many attempts.

If there are many combinations of discrete variables, more than the number of points in the design, this is unsatisfactory. For example, interior discrete coordinates are usually not very good, except for central points, yet they would appear very often.

An alternative strategy is to try to optimize the discrete variables by starting them as if they were continuous `range` variables, and later moving them to nearby discrete levels as best we can.

The function `rc` generates a random start with the discrete variables taking on continuous `range` values, but satisfying all other constraints.

The function `vc` optimizes, treating the discrete variables as continuous (instead of leaving them fixed forever, as `v` does).

Thus `design program=rc*vc` produces a continuous approximation to the optimal design.

There are two problems. First, how do we convert the continuous values to nearby discrete ones, and second, what do we do about constraints that are violated when we do this?

There are three functions that change continuous variables to discrete values, `xcd1`, `xcd2` and `xcd3`, the last being preferred.

`xcd1` converts each continuous value to its nearest discrete value.

`xcd2` goes through each coordinate of each point, and tries replacing it by each discrete value in turn, choosing the one that gives the best design for the resulting solution at that point (with some of variables discretized and the rest not). This is slow, and is not much better than the third function.

`xcd3` tries only the two nearest discrete values to each continuous coordinate that needs conversion, and picks the one that gives the smallest *IV*-value.

`xcd2` and `xcd3` are both greedy algorithms, and do the best they can at the place they are. They do not try to avoid trapping themselves by their choice.

They both also traverse rows and columns in a randomly chosen order, so more than one iteration makes sense, even from the same input.

We now encounter the second problem. If any discrete variables are involved in constraints, then the discretization process may have violated these constraints. If so, we attempt to adjust both the continuous and discrete variables to correct this.

There are three functions for doing this, the last of which is preferred.

`cn1` makes a minimal adjustment of the continuous variables so as to satisfy the constraints. This may be impossible, and then `design` fails.

`cn2` generates the nearest point satisfying all the constraints, treating the discrete variables as continuous again. Then it generates a number of random legal points (using `r` or `rj`) that satisfy all the constraints and have continuous or discrete coordinates as appropriate. Finally, it determines the closest of these legal points to the continuous point that it found. In computing this distance it considers only coordinates that are involved in constraints (the others being irrelevant here). That closest legal point is the one we take, again only in the coordinates that are involved in constraints. The other coordinates are unchanged. This point is properly discrete, and satisfies all the constraints.

`cn3`, the preferred algorithm, is the same as `cn2`, except that it considers the ten nearest random points instead of only the nearest, and picks the one that gives the smallest *IV*-value at that moment (with some points modified and some not). This is also a greedy algorithm and may get trapped unawares.

If a problem has no discrete variables, `gosset` uses `design program=r*v` as the default program. It there are fewer discrete possibilities than functions in the model, `gosset` uses

```
design program=rj*v+rc*vc*xcd3
```

as the default. In other words, first the discrete coordinates are assigned sequentially and then the design is optimized with the discrete coordinates held fixed; second, an initial continuous design is chosen (with `rc`), optimized (with `vc`) and discretized (with `xcd3`). The output is the better of these two designs.

`Gosset` iterates `xcd3` four times if the number of model terms times the number of discrete variables is less than 50, three times if less than 100, and twice if less than 250. The reasoning is that an improvement is likely owing to the random ordering feature of `xcd3`, and in these ranges it's faster to iterate `xcd3` than to get the same improvement by iterating the whole design. Thus the default might be

```
design program=rj*v+rc*vc*xcd3^4
```

for a small problem.

If a problem has discrete variables involved in constraints, then the last program has `*cn3` appended to it.

Finally, if there are continuous as well as discrete variables, `*v` is appended again, to optimize the design with these (presumably well-chosen) discrete values held fixed.

Thus the longest default program that `gosset` uses is

```
design program=rj*v+rc*vc*xcd3^4*cn3*v
```

when there are discrete variables, the number of combinations is not greater than the number of functions, some discrete variables are involved in constraints, there are also continuous variables, and the problem is small. This involves three optimizations, using `v`, `vc` and `v`.

That is the default; we can try other combinations of programs if we wish.

4.7. A-, D- or E-optimal designs: `type=A`, `type=D` or `type=E`

So far we have discussed I-optimal designs. `gosset` can also be instructed to search for A-, D- and E-optimal designs, using

```
design type=A
```

(or equivalently `type=a`),

```
design type=D
```

(or equivalently `type=d`), and

```
design type=E
```

(or equivalently `type=e`), respectively.

```
design type=I
```

(or equivalently `type=i`), searches for an I-optimal design.

The default `type` is the last one used, or `type=I` if none has been specified yet.

File names for these four types of design begin with the characters `a`, `d`, `e`, and `v` respectively. (It would be more logical for file names of I-optimal designs to begin with an `i`, but for historical reasons they begin with a `v`.)

Warning: E-optimality is very new to `gosset`, and acts quite strangely — see the discussion in Sect. 12.

4.8. Designs that protect against a missing run: `type=B`, `C`, `F` or `J`

There are many situations, when testing a new product for example, where there is a reasonable chance that not all the runs will be made successfully. We have included in `gosset` the ability to construct a design that is I-, A-, D- or E-optimal given that one run will be lost. More precisely, we define the J-, B-, C- and F-values of a design to be the maximum of the I-, A-, D- or E-values of any of the designs formed by omitting one run. (These labels are alphabetically one letter away from the parent types.) Then the `design` options `type=J` (or equivalently `type=j`), `type=B` (or equivalently `type=b`), `type=C`, (or equivalently `type=c`), and `type=F` (or equivalently `type=f`), attempt to minimize the J-, B-, C- and F-values, respectively.

The design obtained has the number of runs specified, but the value given for the design is its value with the worst run omitted. An example will be found in Sect. 7.16.

`gosset` does this by adding up the values at each step for each possible subdesign, which slows it down enormously.

The subdesign values are then raised in succession to a sequence of powers such as 20, 40, 80, etc., in an attempt to equalize the contributions to the sum. The initial power is usually 20, but this is automatically randomized over a wide range half of the time. (20 seems to work best, but doesn't provide much variety by itself.) The final pass is made with the subdesign values raised to a power in the millions.

For higher dimensions, this is so slow that it's worth restricting the search to, say, the 20th power by typing

```
design runs=46 type=J program=r*j20
```

which uses only the power 20. (An extra field after a comma can be given to specify the highest power used, as in `program=r*j20,80`. The power is doubled at each stage.)

Then the best of the 20th power designs found can be used as a starting value for a single long run to optimize it:

```
design runs=46 type=J start=j.46.best
```

using the default options, or, better, starting at the 40th power:

```
design runs=46 type=J start=j.46.best program=j40,9999999
```

Some of these designs are described in more detail in [HS9].

4.9. Designs with correlated errors: `samplecv`

Normally one assumes that the errors in successive runs are independent, with mean 0 and constant variance σ^2 (cf. Eq. (1) of Sect. 12). In this case one wishes to minimize the values of

$$\text{trace } M.(X'\,X)^{-1},\ \text{trace } (X'\,X)^{-1},\ \det\,(X'\,X)^{-1},\ \text{or largest eigenvalue of } (X'X)^{-1},$$

for I-, A-, D-, or E-optimal designs, respectively (see Sect. 12). `gosset` can also handle the more general situation where the errors have mean 0 and any fixed covariance matrix C that is known in advance. Now the object is to minimize

$$\text{trace } M.(X'\ C^{-1}\ X)^{-1},\ \text{trace } (X'\ C^{-1}\ X)^{-1},\ \det\ (X'\ C^{-1}\ X)^{-1},\ \text{or}$$
$$\text{largest eigenvalue of } (X'\ C^{-1}\ X)^{-1}, \tag{1}$$

for I-, A-, D-, or E-optimal designs, respectively.

The covariance matrix C is specified by including `samplecv` lines in the program specification. These lines should contain a C program that defines the covariance matrix, as illustrated by

```
10 sphere x y
20 model (1+x+y)^2
30 samplecv double samplecv(i,j,runs) {
40 samplecv     if(i==j)return 1.;
50 samplecv     if(i/2==j/2)return 0.8;
60 samplecv     return 0.0;
70 samplecv }
```

The function `samplecv` (with the same name as the gosset command) must be of type `double`, and return the covariance of the `i`th and `j`th samples. The third argument, `runs`, is the number of runs in the current design, in case the user needs it.

This C program is copied to an include file `samplecv.h` that is included by `bpvvv.c`. Any C operations, including reading files, are possible here, and other functions may be defined, at some risk of conflict of course. What the user gives is blindly copied into `samplecv.h`. Note that C subscripts start with 0, not 1, so `i=0` is the first sample.

The function is called in natural order for initializations, but needn't be very fast because it's not called thereafter. (It may however be called more than once.)

The diagonal entries of C give the sample variances, which no longer need be equal.

The example given above defines the covariance matrix

$$C = \begin{bmatrix} 1 & .8 & 0 & 0 & 0 & . \\ .8 & 1 & 0 & 0 & 0 & . \\ 0 & 0 & 1 & .8 & 0 & . \\ 0 & 0 & .8 & 1 & 0 & . \\ 0 & 0 & 0 & 0 & 1 & . \\ . & . & . & . & . & . \end{bmatrix}$$

which describes a design used for measurements on eyes. The measurements are made in pairs (left eye, then right eye), with a correlation of .8 between measurements in a pair, successive pairs being independent. These designs are described in more detail in Sect. 7.17 and [HS10].

A complicated covariance matrix C is best read from a file. If C is stored in "myfile" in the base directory then the following `samplcv` lines accomplish the task:

```
samplecv double samplecv(i,j,n) {
samplecv  static double *a;
samplecv  int k;
samplecv  FILE *f;
samplecv  char in[100];

samplecv  if(a==0) {
samplecv      a=(double*)malloc(n*n*sizeof*a);
samplecv          f=fopen("myfile","r");
samplecv          if(a==0||f==0) {
samplecv              fprintf(stderr,"can’t setup samplecv\n");
samplecv              exit(1);
samplecv          }

samplecv          for(k=0; k<n*n&&fgets(in,sizeof in,f); k++)
        a[k]=atof(in);
samplecv      if(k!=n*n||fgets(in,sizeof in,f)) {
samplecv          fprintf(stderr,"can’t synchronize samplecv\n");
samplecv          exit(1);
samplecv              }
samplecv          fclose(f);
samplecv      }
samplecv      return a[i*n+j];
samplecv }
```

If C is instead in the working directory, replace "myfile" by (say) "workdir/myfile". "myfile" is expected to contain one number per line. It is immaterial whether the entries are arranged by rows or by columns, since C is symmetric. Comments printed on `stderr` starting with "can’t" are printed by `gosset`, which explains the slightly awkward wording used here.

4.10. Block designs: `cmatrix`

In this section we describe a method for constructing designs in which the measurements are made in blocks, and where the block effects are "nuisance parameters" in the model. Other types of block designs can also be constructed.

The following example will serve to illustrate the method.

Suppose we are modeling a response that we assume is a quadratic function of four variables x_1, x_2, x_3, x_4, that we want to make $\pi = 27$ runs in three blocks of 9 runs each, and that the effect of the block appears as an additive term in the model.

Thus the model is

$$y = \alpha_0 + \sum_{i=1}^{4} \beta_i \, x_i + \sum_{i \le j} \beta_{i,j} \, x_i \, x_j + \gamma_m + \varepsilon \,, \tag{1}$$

where α_0 is the constant term, the β_i are the coefficients we are interested in, and the γ_m are the block effects (where m is the number of the block to which this run belongs), and ε is the error term. Since we want to estimate the coefficients in the model rather than the response itself, it is appropriate here to look for a D-optimal design.

We first show how to search for this design using `gosset`, and then explain the method. This makes use of `cmatrix` lines in the program specification:

```
10 range x1 x2 x3 x4
20 model (1+x1+x2+x3+x4)^2-1
40 misc K=9
50 cmatrix double cmatrix(i,j,n) {
60 cmatrix   if(i==j)return 1-1.0/K;
70 cmatrix   if(i/K==j/K)return -1.0/K;
80 cmatrix   return 0;
90 cmatrix }

compile
moments

design type=D runs=27 n=50
```

Comments. Lines `50-90` are the same for any problem of this type in which the b blocks have equal size k. The block size k (here 9) is to be inserted in line 40. The part of the model that we want to estimate appears in line 20. We must subtract 1 from the model since of course we cannot estimate the constant term in this problem.

This and some other examples are shown in more detail in Sect. 7.18. Sect. 7.19 shows how this technique can be used to find small examples of balanced incomplete block designs.

Explanation. The `cmatrix` lines contain a `C` program that defines a certain matrix G. The `cmatrix` lines use the same format as the `samplecv` lines used in the previous section.

Then `gosset` attempts to minimize the values of

$$\text{trace } M.(X' \, G \, X)^{-1}, \text{ trace } (X' \, G \, X)^{-1}, \text{ det } (X' \, G \, X)^{-1}, \text{ or}$$
$$\text{largest eigenvalue of } (X' \, G \, X)^{-1}, \tag{2}$$

for I-, A-, D-, or E-optimal designs, respectively.

If the design is to contain b blocks, of sizes $k_1, \ldots, k_b$, for a total of $\pi = \sum k_i$ runs then G should be the matrix

$$G = I_\pi - \text{diag} \left(k_1^{-1} J_{k_1}, \ldots, k_b^{-1} J_{k_b} \right) \tag{3}$$

where I_n is an $n \times n$ identity matrix and J_n is an $n \times n$ matrix of 1's.

The lines 50-90 given above specify this G when all the block sizes k_i are equal to k, the most important case.

To justify (2), we assume for simplicity that the constant term α_0 in (1) is zero. The same result can be obtained in the general case by a limiting argument.

After the π measurements have been made, giving a vector of measurements Y, we estimate the coefficient vector β from the equation

$$Y = X \beta + Z \gamma + \varepsilon,$$

where Z is a $\pi \times b$ 0,1 matrix whose first column consists of k_1 1's followed by $\pi - k_1$ 0's, whose second column consists of k_1 0's followed by $\pi - k_2$ 1's and $\pi - k_1 - k_2$ 0's, and so on.

A standard argument (see for example [Co87], p. 337) now shows that the covariance matrix of the estimate of the coefficient vector β is $\sigma^2\ C^{-1}$, where σ is the variance of the errors in (1), and C, Fisher's information matrix, is equal to

$$X'\ X\ -\ X'\ Z\ (Z'\ Z)^{-1}\ Z'\ X$$

In our problem C simplifies to $X'\ G\ X$, with G as in (3).

Of course, after setting up the matrix G by means of the `cmatrix` lines, one then searches for the design with

```
design type=D, runs=p n=...
```

where `p` = π is the number of runs.

4.11. Packings: `type=P`

The `design` option `type=P` (or equivalently `type=p`) searches for a packing of `runs` points in the measurement region. Such designs can be used when no model is known and one simply wishes to search the space efficiently.

Of course packing problems are also of great interest in their own right—see for example the references listed on page 21 of [SPLAG]. We think this is the first computer program ever that can search for packings in general regions.

Although no model is used, it is still necessary to include a model in the `gosset` problem specification. The simplest way to do this is to specify a linear model that includes all the variables, such as

```
40 model x1+x2+x3+x4
```

It is also necessary to run the `moments` command, even though again the moment matrix is not used. The simplest way is with

```
moments n=0
```

Then, when invoked with (say)

```
design runs=20 n=200 type=P
```

the program will attempt to place 20 points in the measurement region so that the minimal distance between the points is maximized. The algorithm is a modification of that used in [HSS93] to construct spherical codes. An example is given in Sect. 7.20. The program actually attempts to minimize the P-value of the design, which we define to be the reciprocal of the minimal distance between points.

There are also some `program=` options for `type=P`. They are vestigial traces of program design, but seem worth recording. It is recommended that the reader skip the remainder of this section.

To understand these options, it is necessary to know that the program operates by moving the points (using the optimization technique described in Sect. 12) so as to minimize the sum of three potentials.

There is a repulsive potential

$$V_{rep}\ =\ \sum_{P,Q} \frac{1}{\mathrm{dist}(P,Q)\ -\ A}$$

summed over all pairs of distinct points $P,\ Q$, where A is an adjustable parameter, and two attractive potentials

$$V_{cg}\ =\ -\ \sum_{P} \frac{1}{\sqrt{1\ +\ \mathrm{dist}(P,C)^2}}$$

where C is the center of gravity of the points, and

$$V_{att} = - \sum_{P,Q} \frac{1}{\sqrt{1 + \mathrm{dist}(P,Q)^2}}$$

The program attempts to minimize the sum

$$V_{rep} + w\ V_{cg} + h\ V_{att}$$

where the parameters w and h are also adjustable. The parameter A starts at some initial value and increases in steps to the minimal distance.

The options for `program=` can be described (somewhat cryptically) as follows:

`p` starts at $A = 0$ and ends at $A = .9999999$ of the minimal distance,

`P` starts at $A = .999$ of the initial minimal distance and ends at .999999999 (this is used to continue a nonrandom solution, e.g. one produced by `xcd3`),

`pc` and `Pc` same as `p` and `P` except treat discrete variables as continuous.

`program=r*p` and `program=r*pc` are the defaults.

Two numbers following `[Pp]` or `[Pp]c` are optional values overriding the initial and final A's. E.g. `p.5,.999` overrides the difference between `p` and P.

After this the following options can appear in any order:

`w.9` set $w = .9$,

`h.9` set $h = .9$

(setting either of these parameters to 1 should reduce the diameter to about 1 when $A = 0$; the default is to choose them randomly in the range 0-10),

`s5,1.25` for the first A, do 5 steps; for the second A, do 5*1.25 steps, etc., in a geometric progression until the limit `steps` is reached,

`f.9` h drops by a factor of .9 for every A; default is 1.0,

`p` makes $h = 0$ and $w = 0$ the default; thus `pp` is pure A-polishing,

`d` means do a single run with $A = 0$ and stop; used to verify proper gravitational collapse.

An initial scaling for the potential forces is calculated from the random initial distribution of the points. The scaling is chosen so that half the pairs of points attract and half repel. If the program is unable to find such a median, an error message

error was: can't establish median

is printed. This is rare, and harmless.

4.12. Monte-Carlo problems

`gosset` is unable to do a simple Monte-Carlo evaluation of moments for, say, a ten-dimensional mixture, as in

```
10 range 0 1 a b c d e f g h i j
20 model (1+a+b+c+d+e+f+g+h+i+j)^2
30 constraint a+b+c+d+e+f+g+h+i+j=1
```

because the probability of a random point in the unit cube falling in that region is essentially zero. This happens when the mixture part of a problem has high dimension. (There should be no difficulty with a huge problem in which only a few variables appear in such constraints.)

`gosset` handles simple constraints of this type automatically, namely, those in which there is an equality or inequality constraint of the form

```
40 constraint a+b-c+d-e=1
```

with all coefficients equal in magnitude.

More complicated situations must be handled "manually" by the user.

We make a rhetorical convenience here of turning off the automatic procedure just mentioned, by using the command `gosset -skipmix`, in order to show how the user might have handled the problem himself in the case of a simple mixture. The user may someday have to set up his own programs `random.h` (and `urandom.h` if there are discrete variables in the problem) in analogous ways in analogous cases.

We emphasize that `gosset` handles this case automatically, so that what we are about to say is not needed here.

This example shows how to generate the moments of a high-dimensional simplex "manually". We run `gosset -skipmix`, and set up the design given above. Then `random.h` looks like this:

```
/*
range 0 1 a b c d e f g h i j
model (1+a+b+c+d+e+f+g+h+i+j)^2
constraint a+b+c+d+e+f+g+h+i+j=1
*/
    A(0)=2*unif()-1;
    A(1)=2*unif()-1;
    A(2)=2*unif()-1;
    A(3)=2*unif()-1;
    A(4)=2*unif()-1;
    A(5)=2*unif()-1;
    A(6)=2*unif()-1;
    A(7)=2*unif()-1;
    A(8)=2*unif()-1;
```

which puts nine random numbers uniformly distributed on -1 to $+1$ into the locations `A(0)` to `A(8)`, where `A` is a macro defined by whatever programs include the file.

To change this, first compile the `gosset` program and inspect the `random.h` and `urandom.h` files to see what they do.

The invoking programs check whether the random point satisfies all the constraints, and if not just reject it and ask for another.

We can generate a random point with a uniform distribution over our mixture region by generating nine numbers uniformly distributed on 0 to 1, sorting them, and returning their differences:

```
{double _Q[9];
extern doubsort();
_Q[0]=unif();
_Q[1]=unif();
_Q[2]=unif();
_Q[3]=unif();
_Q[4]=unif();
_Q[5]=unif();
_Q[6]=unif();
_Q[7]=unif();
_Q[8]=unif();

qsort(_Q,9,sizeof*_Q,doubsort);
A(0)=2*_Q[0]-1;
A(1)=2*(_Q[1]-_Q[0])-1;
A(2)=2*(_Q[2]-_Q[1])-1;
A(3)=2*(_Q[3]-_Q[2])-1;
A(4)=2*(_Q[4]-_Q[3])-1;
A(5)=2*(_Q[5]-_Q[4])-1;
A(6)=2*(_Q[6]-_Q[5])-1;
A(7)=2*(_Q[7]-_Q[6])-1;
A(8)=2*(_Q[8]-_Q[7])-1;
}
```

We use a leading underscore on the variables to avoid any conflicts with `gosset` variables in the including program, whose names we do not know.

We use `qsort` to sort the numbers. This command requires a function (`doubsort` in this example) that compares numbers. This is a nuisance, because the syntax won't let us define one here.

The solution is to put the function manually in the `gosset` program `quit.c`, which is shared by `moments` and `design`. Here is an example of such a function:

```
int doubsort(a,b)
double *a,*b;
{
    if(*a<*b)return -1;
    if(*a==*b)return 0;
    if(*a>*b)return 1;
}
```

Insert this into `quit.c` in the base directory, using an editor, and create a file containing the new `random.h` (but not called `random.h`, call it say `temp`).

Then enter `gosset` and compile the mixture problem. This creates `random.h` in the working directory. Overwrite `random.h` by `temp`:

```
10 range 0 1 a b c d e f g h i j
20 model (1+a+b+c+d+e+f+g+h+i+j)^2
30 constraint a+b+c+d+e+f+g+h+i+j=1
compile
!cp ../temp random.h
```

and similarly for `urandom.h` if there are discrete variables in the problem. Now run `moments` and `design` as usual.

To repeat, simple high-dimensional mixtures with coefficients that are equal in magnitude are handled automatically, but problems with high dimensions and unequal coefficients in the mixture or with several overlapping constraints require special handling. This is quite unsatisfactory from the user's point of view, but we don't see how to solve the Monte Carlo problem for general constraints.

The same problems arise if A-, D- or E-optimal designs are needed, for although those searches do not need the moments matrix, they still need to find random points in the design region to start the search.

5. Inspecting the design

5.1. Overview

The main commands for examining a design are `interp`, which produces a labeled listing of the points, and `iv`, which calculates the average prediction variance, or `dv` and `av`, which calculate the *D*- and *A*-values of the design. The `list` command (Sect. 8.8) can also be used to print a design.

5.2. `interp`

`interp filename` produces a labeled listing of the points of the design in `filename`. It also prints a column of numbers for each active constraint. These numbers are nonnegative when the constraint is satisfied, zero when the constraint is exactly satisfied, and negative when the constraint is violated. (The precise number printed is a certain linear combination of the internal variables $A(i)$ plus an appropriate constant. These linear combinations are specified in the `compiler` output.)

If `filename` is omitted and a design has just been run, `v.nn.best` is printed, where `nn` is the number of points.

If `filename` cannot be found in the working directory, `gosset` checks the base directory, and then the archive of solutions in `lib.a` in the working directory.

A `filename` of the form `../codelib.a(c.3.12.1002)` (for example) interprets that design in the archive `../codelib.a`, using the current program's interpretation scheme. (This is one of the benefits of putting problems into a standard internal form, with variables running from -1 to 1.)

If the dimensions are wrong, `interp` may fail to produce output, but it will allow constraint violations (which it announces by printing a negative number in the column corresponding to that constraint).

The `check` command (Sect. 8.2) can be used to quickly verify the legitimacy of a particular design.

It is useful to write or pipe `interp` results into shell commands for sorting, calculating the distance of points from the origin, etc., using `awk` and `sort` in the shell. If there are `nn` points in the code, piping `interp` output into `tail -nn` is an easy way to remove titles, so that the points can be sorted. For example

```
interp filename|tail -27|sort +0n +1n +2n
```

takes a 27-point design in `filename` and sorts the points into numerical order.

The design points appear in random order in `interp` output, in unnumbered lines. Of course it is easy to number the lines in a file, for example with `awk`, as in:

```
interp filename|tail -27|awk '{ print NR, $0 }'
```

`interp` has options `precision`, `fieldmin`, `use`, `start` and `invert`. The first three are illustrated by

```
interp precision=4 fieldmin=1 use=100
```

`precision` specifies the number of decimal places used to print a number of magnitude 1. The default is 4. The precision is automatically reduced as the numbers increase in magnitude.

Note. Although we use four decimal places as standard, it usually has very little effect on the efficiency of a design if the coordinates are rounded to 1 or 2 (or even 0) decimal places. Of course this depends on the type of design, but for simple linear or quadratic designs the coordinates can usually be rounded without much effect on the optimality. It is in any case easy to experiment with this. For example,

```
interp precision=1 > file1
interp invert=1 file1 > file2
iv file2
```

finds the *IV*-value of the rounded-off design.

`fieldmin` gives the minimum field width in columns. The default is 1. The field width is constrained both by the names of the variables and the size of the numbers, and `fieldmin` is sometimes helpful in adjusting the spacing.

The option `use=100` causes the points of the design to be printed in numbered `use` lines, starting at line number `100`, in a form suitable for insertion in a design program. If the line number is negative, no line numbers are used.

`start` is similar to `use`, but produces `start` lines.

The `start` option is one of two ways of starting a solution from where another left off. The other uses the `start` and `program` options of the `design` command.

The option `invert=1` causes `interp` to read an `interp` output file and produce an `interp` input file from it, as in

```
interp v.32.best > myfile
interp invert=1 myfile > newv.32.best
```

This makes it possible to edit designs in `interp` output files and produce files in `gosset`'s internal format from them. The default, which does not do this, is `invert=0`.

5.3. `iv`, `av`, `dv`, `ev`, `jv`, `bv`, `cv`, `fv`, `pv`

`iv filename` prints the average prediction variance *IV* of a design in `filename` (defined in Sect. 12).

`iv` uses the same alternatives for `filename` as `interp`. In particular, it can `iv` a whole archive at once, as in

```
iv lib.a
```

It is safer to use `check` for this, though, lest an impossibly small `iv` be achieved by violating the constraints.

`iv` depends on the moment matrix in the file `moments.h` written by the `moments` command, and on other files (particularly `interp.h`), produced by the design program.

It *might* make sense to `interp` a design from another program, in which case its `iv` would also make sense. But remember that `iv` depends on the current program and its moment matrix. Also the output is not as meaningful as it is for `interp`, since all we see is a single number.

The commands `av`, `dv`, `ev`, `jv`, `bv`, `cv`, `fv`, `pv` are similar to `iv`, but return the *A*-value, ..., *P*-value of the design(s), as defined in Sect. 12.

5.4. File names in `gosset`

The following are the file name conventions for designs used by `gosset`.

If the file name contains only one number between dots it is the number of points in the design. For example, when searching for an I-optimal design,

```
v.12.best
```

is the current best design with 12 points (and the dimension is unspecified). Similarly `v.12.0` might be the output from processor 0 for a 12-point design. The trailing 0 is not surrounded by dots, so does not count. Files such as `d.12.best`, `a.12.best`, `j.12.best`, `p.12.best`, etc., are designs produced with `type=D`, `type=A`, `type=J`, `type=P`, etc.

If there are two numbers between dots they are the internal (or geometric) dimension and the number of points. For example

```
v.2.12.aaab
```

is a (hopefully) I-optimal 2-dimensional design containing 12 runs that is associated with the program tagged by `aaab`.

Similarly, `d.2.12.aaab`, `a.2.12.aaab`, `j.2.12.aaab`, `p.2.12.aaab`, etc., are files produced with `type=D`, `type=A`, `type=J`, `type=P`, etc.

`gosset` imitates the shell's expansion of file names in an archive name. For example

```
iv ../codelib.a(s.3.16.*)
```

will return the *IV*-values for everything in `../codelib.a` that begins with `s.3.16.`.

This avoids having to guess the final component of the name of a library designs.

A question-mark matches any single character.

There is an `echo` command for inspecting the file name expansion, as in

```
echo ../codelib.a(s.3.16.*)
```

which displays all the file names in `../codelib.a` that begin with `s.3.16.`.

6. Controlling the search

6.1. Overview

The `design` command (Sect. 4.4) has a number of options which control how `gosset` searches for a design. The `status`, `watch`, `examine` and `trace` commands, described here, enable the user to monitor the progress of the search.

6.2. `status` or `s`

`status` tells what is running asynchronously, and how it is progressing. In the example

```
status
  pid  etime
27472     20 input daemon
27512      4 design.0 v 136 steps IV=0.41283948458304
```

we see that a `design` is running on processor 0, with `pid` 27512. It has been running for a wall-clock time of 4 seconds, has advanced 136 steps, and the average prediction variance is now 0.412839.... It is running subprogram `v` of `design`.

The latter piece of information comes from the file `vtrace.0`, which gives a step-by-step listing of what `design` is doing. The corresponding information about `moments` comes from a file `moments.prog`.

We can `kill design` or `moments` at any time and get their current solutions (see Sect. 8.8).

The above example also shows that an `input daemon` is running. This handles lines from the terminal, and writes them to a file, where they can be watched for by `gosset` without causing `gosset` to block when no line is present.

If `gosset` crashes, the `input daemon` may continue eating an occasional line from the terminal unless it is killed by an interrupt character.

For designs of type `B, C, E, F, J`, the `status` and `trace` commands report the quantity currently being minimized, which is only an approximation to the *B*-, *C*-, etc. value. For J-optimality, for example, the program actually minimizes the value of

$$\left\{ \frac{1}{n} \sum_{r=1}^{n} IV_r^M \right\}^{1/M}$$

for some integer M, where n is the number of runs, and IV_r is the IV-value of the design obtained by omitting the rth run. After each pass finishes, this is replaced by the true *B*-, *C*-, etc. value (in this example, by the largest IV_r).

Thus old `trace` values are exact, while the current one is only an approximation. So it may appear that the convergence gets worse at the end of a pass, when the exact value replaces the approximate one.

6.3. `watch`

`watch` does a `status` command occasionally and looks for lines that match the text it has been told to watch for.

`watch design`, `watch design.0` or `watch moments` are obvious applications of this command. When one of these lines is found in the `status` output, that line is printed.

`watch design time=30` instructs it to do this every 30 seconds, the default. The overhead is a couple of seconds, because the hidden `status` command has to read several files and count steps for its output lines. No `time` less than ten seconds will be used.

`watch design` is useful in monitoring how designs are progressing, so that we can type `kill design` or `examine design` when it is appropriate.

`watch` or `watch off` turns the `watch` mechanism off.

A common error is to type `watch designs`, which has no effect, instead of `watch design`.

6.4. `examine`

`examine design filename time=20` causes a design process (normally `design.0`) to write its current solution into a file named `filename`. The design process then continues as before.

`examine` will wait for this to happen for not more than `time=20` seconds. If it does happen, `examine` will automatically print the *IV*-, *D*-, *A*- etc. value of the result (as appropriate), and `interp` it.

If it takes longer than that, we will have to watch for the file to appear and then type `iv filename`, `dv filename`, etc., and `interp filename` ourselves. (It might take longer if the machine is heavily loaded, the problem is large or the machine is small.)

If there is more than one `design` process running at a time, then we must identify the particular one wanted, e.g. `examine design.0`; otherwise `examine` will take the first it finds.

If `filename` is omitted, `exam.file` is used, in the current working directory.

If `time` is omitted, 20 seconds is used.

A bare `examine` is accepted, and assumes all the defaults.

6.5. `trace` or `t`

`trace design.0` or `trace 0` reads through the design progress file `vtrace.0` and extracts the `design program=` names, number of steps and final *IV*- or *D*-value etc., from each stage.

It works on a design that is running or has completed. The information is the same as in the output from `watch` or `status`, but everything is listed, not just the current state.

This command is useful in analyzing what happens in the complex programs used for example with constrained discrete variables (cf. Sects. 4.6, 4.7).

`trace 0 1 2 3 4` does traces on designs from processors 0 through 4.

`trace` alone attempts to trace all the interesting processors.

6.6. Running things manually

Sometimes it is convenient to run things manually that `gosset` normally does automatically, particularly in shell scripts.

Three programs, `interp`, `vtrace`, and `vvv`, are created in the working directory when a `design` command is given within `gosset`. They correspond to the programs `interp.c`, `gv.c`, and `bpvvvmain.c` in the base directory. These compiled programs are valid for the current program and moment matrix.

`interp` reads a file and prints its interpretation, as in `gosset`. The options are specified by pairs of arguments, the first being the option, and the second the value. The options must precede the filename, but may be in any order, for example:

```
interp -precision 4 -fieldmin 1 -start 0 -use 0 filename
```

The options have the same meaning as within the corresponding `gosset` command, but are specified without an equal sign, and of course can't be expressions.

`vvv` prints the *IV*-, *D*- or *A*-values for a list of files, and has an option giving the type of evaluation, `-i`, `-d`, or `-a`. `-i` is the default. For example

```
vvv -d file1 file2 file3 file4 ...
```

would print a line for each file, giving the filename and the *D*-value. Negative values indicate a singular design, and missing lines indicate that the file can't be understood as a design.

`vtrace` is the `design` command. For example

```
vtrace -program 'r*v' -runs 24 -extra 0 -tiny 1e-8
        -lib ../codelib.a -time 999999999 -steps 10000
        filename > output
```

Note that an asterisk in the program must be quoted, since it means something to the shell.

The `filename` is a starting design if it is needed, for example if the program given is just the default `-program v`. The output design is written on standard output, and run-time diagnostics appear on standard error.

Search libraries are given in order with as many `-lib` pairs as needed.

These programs do not interact with the `gosset` environment at all, except that they match the environment where they were created.

As an example, designs can be run on an army of independent systems without running `gosset` on each one. Here is a shell script that was used with twelve workstations sharing a common disk(*):

```
#!/bin/sh
# running designs on independent systems sharing a single
# disk file system

# runs=35 36 37 38 39 40
# each host has a current design in `hostname`.x
# and a best design in `hostname`.35 ... 40
# each trial is recorded in a file "log"
# vtrace and vvv are in a working directory "4d" where
# the results will be left.

P=`hostname`
cd 4d
while :
    do
    for i in 35 36 37 38 39 40
        do
```

(*) Note that in this font it is easy to confuse ` , a backward sloping quote (like a French grave accent), and ' , a forward sloping quote (like an acute accent).

```
            # slow down an abort loop
            sleep 1

            # terminate if a file "halt" appears
            if test -r ../halt
                then exit
            fi

            # run a design, random start, I-optimal
            nice -20 vtrace -program 'r*v' -runs $i > $P.x 2>/dev/null

            # check if a minus sign in IV value (singular)
            if vvv $P.x|grep '[-]' > /dev/null
                then continue
            fi

            # make a copy if this is the first one
            if test ! -s $P.$i
                then cp $P.x $P.$i
            fi

            # record IV of run in file ../log
            echo $i `vvv $P.x` >> ../log

            # if better than previous record, update
            case `vvv $P.$i $P.x|sort +1n|sed 1q|awk '{print $1}'` in
            $P.x)
                mv $P.x $P.$i
                ;;
            *)  rm $P.x
                ;;
            esac
        done
    done
```

6.7. Changing an ongoing `gosset` job

It is possible to change the commands already given and spooled up for `gosset`.

In the base directory there is a file with a name like `12345.input`, where `12345` is the process number of `gosset`, that has in it all the spooled commands—in particular all the commands for a batch run. It is easiest to find this file with a command like `ls -lt *.input`, which gives the latest such file first.

This file can be edited. `gosset` keeps the byte offset that it will read next, and goes to that place for the next line it wants, so it is important that nothing be changed in the commands that it has already read. Anything after that point can be changed.

7. A collection of examples

In most of these examples the points of the design (the output from `interp`) have been sorted in numerical order, and sometimes blank lines inserted, in order to make them easier to understand. A number of other examples may be found in [HS6].

7.1. Three continuous variables in a cube

Three continuous variables `x`, `y`, `z`, each between `0` and `1`; a full quadratic model; find a design with 14 runs (a minimal design would contain 10 runs); take the best of 25 attempts.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd 3cube    [we create a working directory called 3cube]

10 range x y z 0 1   [only 2 lines are needed to specify the design]
20 model (1+x+y+z)^2

compile   [wait for compile done]
moments   [wait for moments done]
design runs=14 n=25   [wait for design done]
```

Here is the result:

```
local best is 3cube/v.14.best
interp

3cube/v.14.best
 x      y      z

1.0000 1.0000 0.5712
1.0000 0.5712 1.0000
1.0000 0.4288 0.0000
1.0000 0.0000 0.4288
0.6630 1.0000 0.0000
0.6630 0.0000 1.0000
0.4742 0.5000 0.5000
0.4742 0.5000 0.5000
0.4742 0.5000 0.5000

0.1708 1.0000 1.0000
0.1708 0.0000 0.0000
0.0000 1.0000 0.0000
0.0000 0.5000 0.5000
0.0000 0.0000 1.0000
```

To see the *IV*-value we can type

```
iv
     3cube/v.14.best       0.406494756891
```

Note that the program has cleverly discovered that it should make three runs close to (but not exactly at) the center of the cube.

Note also that the design is slightly asymmetrical. It is very common for optimal designs to have this property.

This is one of many situations where we think we know the best design (see [8]). This design is contained in the built-in library (Sect. 9.1), and can be retrieved as shown in the next example.

7.2. Same, but search the built-in library

Same as previous example, but search the built-in library for the design. This library contains a large collection of optimal or nearly optimal designs for many applications (see Sect. 9.1).

```
10 range x y z 0 1
20 model (1+x+y+z)^2

compile   [wait for compile done]
moments   [wait for moments done]
search ../codelib.a    [could be omitted as this is the default library]
design runs=14 program=lib   [wait for design done]
```

The design as follows (again after being sorted):

```
interp

3cube/v.14.best
 x      y      z

1.0000 1.0000 0.5712
1.0000 0.5712 1.0000
1.0000 0.4288 0.0000
1.0000 0.0000 0.4288
0.6630 1.0000 0.0000
0.6630 0.0000 1.0000
0.4742 0.5000 0.5000
0.4742 0.5000 0.5000
0.4742 0.5000 0.5000

0.1708 1.0000 1.0000
0.1708 0.0000 0.0000
0.0000 1.0000 0.0000
0.0000 0.0000 1.0000
0.0000 0.5000 0.5000
```

This is identical (at least to four decimal places) to the one found in the previous example, which shows the power of the algorithm. However, the huge library doesn't keep many significant figures, and if we wanted to obtain greater precision in this design (for the purpose of some theoretical investigation) we could optimize it further using the `design` command

```
design runs=14 program=lib*v
```

instead of the command shown. Of course the precision given is more than enough for any ordinary practical application.

7.3. Four continuous variables in a sphere

Four continuous variables `w`, `x`, `y`, `z`, in a sphere of radius 1 centered at the origin; a full quadratic model; find a design with 18 runs (a minimal design would contain 15 runs); take the best of 10 attempts.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd 4sphere    [we create a working directory called 4sphere]

10 sphere w x y z
20 model (1+w+x+y+z)^2

compile    [wait for compile done]
moments    [wait for moments done]
design runs=18 n=10    [wait for design done]
```

Here is the result:

```
local best is 4sphere/v.18.best
interp

4sphere/v.18.best
  w       x       y       z

 0.8608 -0.2553  0.3639  0.2478
 0.8097  0.0276 -0.5721 -0.1273
 0.5322  0.7577  0.0885 -0.3672
 0.3553 -0.1688  0.5302 -0.7511
 0.2962  0.6439 -0.0329  0.7047
 0.2402 -0.9016 -0.2298 -0.2769
 0.1229 -0.1007 -0.8642  0.4774
 0.0855 -0.5236  0.0203  0.8474
 0.0648  0.2571  0.9209  0.2857
 0.0004 -0.0007  0.0011  0.0008
 0.0004 -0.0007  0.0011  0.0008

-0.0212  0.0359 -0.5513 -0.8332
-0.3423 -0.6735  0.6514  0.0702
-0.3804  0.6819 -0.6236 -0.0382
-0.4612  0.7075  0.4249 -0.3259
-0.6879 -0.4793 -0.5442  0.0292
-0.7440 -0.1474  0.2142 -0.6155
-0.7593  0.1897  0.1201  0.6108

iv
   4sphere/v.18.best        0.564133087128
```

(An essentially identical design could have been obtained from the built-in library — see Sect. 9.1 or the previous example.)

7.4. Same but make three runs at center point

Same as previous example but insist that three runs be made at the center point. To do this we simply add three `use` lines to the program.

```
10 sphere w x y z
20 model (1+w+x+y+z)^2
30 use w=0 x=0 y=0 z=0
40 use w=0 x=0 y=0 z=0
50 use w=0 x=0 y=0 z=0

compile   [wait for compile done]
moments   [wait for moments done]
design runs=18 n=10   [wait for design done]
```

Here is the result:

```
local best is 4sphere/v.15.best
interp

4sphere/v.15.best
  w       x       y       z

 0.7437 -0.6277  0.2099 -0.0935
 0.7306  0.1352 -0.2180  0.6328
 0.5931  0.1318 -0.6908 -0.3921
 0.5834  0.3536  0.4029 -0.6102
 0.3360  0.2138  0.8215  0.4081
 0.1854  0.9733 -0.0792  0.1095
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000

-0.0034 -0.7718 -0.6339 -0.0496
-0.0789 -0.5820  0.0884  0.8045
-0.1065 -0.3913 -0.0092 -0.9140
-0.2295  0.1775 -0.8123  0.5059
-0.2588 -0.6047  0.7459 -0.1053
-0.4718  0.4695 -0.5279 -0.5275
-0.5142  0.4502  0.6024 -0.4124
-0.5800  0.4183  0.2310  0.6598
-0.9291 -0.3456 -0.1307 -0.0160

iv
   4sphere/v.18.best       0.569833239470
```

7.5. Six 2-level discrete factors

Six discrete factors `a`, `b`, `c`, `d`, `e`, `f`, each taking the values `-1` or `+1`; a first-order interaction model containing the 22 terms `1, a,...,f, a*b,...,e*f`; find a minimal design with 22 runs; take the best of 20 attempts.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd 6factor    [we create a working directory called 6factor]

10 discrete a b c d e f
20 model (1+a+b+c+d+e+f)^2-a^2-b^2-c^2-d^2-e^2-f^2

compile   [wait for compile done]
moments   [wait for moments done]
design n=20   [wait for design done]
```

Here is the result:

```
interp

6factor/v.22.best
 a  b  c  d  e  f

 1  1  1  1  1  1
 1  1  1  1 -1 -1
 1  1  1 -1  1 -1
 1  1  1 -1 -1  1
 1  1 -1  1  1 -1
 1  1 -1  1 -1  1
 1  1 -1 -1  1  1
 1 -1  1  1  1 -1
 1 -1  1  1 -1  1
 1 -1  1 -1  1  1
 1 -1 -1  1  1  1
 1 -1 -1 -1 -1 -1

-1  1  1  1  1 -1
-1  1  1  1 -1  1
-1  1  1 -1  1  1
-1  1 -1  1  1  1
-1  1 -1 -1 -1 -1
-1 -1  1  1  1  1
-1 -1  1 -1 -1 -1
-1 -1 -1  1 -1 -1
-1 -1 -1 -1  1 -1
-1 -1 -1 -1 -1  1

iv
   6factor/v.22.best       1.151666666667
```

This design is similar to a classical fractional factorial design, except that it has 22 runs. One nice thing about `gosset` is that when a fractional factorial design is the best, it will usually find it, but when no fractional factorial design will fit the situation, it will still find an optimal design.

7.6. Three 4-level and two 2-level discrete variables

Three numerical discrete variables `a`, `b`, `c`, each taking the values 0, 1, 2, 3, and two numerical discrete variables `x`, `y`, taking the values 0 and 1; a full quadratic model containing all 19 meaningful terms (`1, a, b, c, x, y, a^2, a*b, a*c, a*x, a*y, b^2, b*c, b*x, b*y, c^2, c*x, c*y, x*y`, but of course omitting `x^2, y^2`); find a design with 24 runs (a minimal design would contain 19 runs); take the best of 100 attempts.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd 5disc    [we create a working directory called 5disc]

10 discrete a b c 0 1 2 3
20 discrete x y 0 1
30 model (1+a+b+c+x+y)^2-x^2-y^2

compile   [wait for compile done]
moments   [wait for moments done]
design runs=24 n=100   [wait for design done]
```

Here is the result:

```
interp

5disc/v.24.best
x y a b c

1 1 3 0 0
1 1 3 3 3
1 1 2 1 1
1 1 0 0 3
1 1 0 3 0
1 0 3 0 3
1 0 3 2 2
1 0 3 3 0
1 0 1 0 0
1 0 1 3 3
1 0 0 1 1
```

```
0 1 3 0 1
0 1 3 0 3
0 1 3 3 0
0 1 1 2 0
0 1 1 2 3
0 1 0 0 0
0 1 0 3 2
0 0 3 1 0
0 0 3 3 3
0 0 2 3 1
0 0 1 0 2
0 0 0 1 3
0 0 0 3 0

iv
   4sphere/v.24.best       0.754165586219
```

7.7. Same, but run as a batch job

Since the previous example involves a large number of attempts, it may be more convenient to run it as a batch job.

To do this we might create a file called `job5` containing the lines

```
cd 5disc
10 discrete a b c 0 1 2 3
20 discrete x y 0 1
30 model (1+a+b+c+x+y)^2-x^2-y^2
compile
moments
wait
design runs=24 n=100
wait
interp
quit
```

and run it with

```
gosset -echo <job5 &
```

or

```
gosset -echo <job5 >report &
```

The latter places the output in a file `report`.

7.8. A simple mixture design

Four continuous variables `a`, `b`, `c`, `d` forming a mixture: `a+b+c+d=1`; a full quadratic model; find a design with 12 runs (a minimal design would have 10 runs); take the best of 20 attempts.

This is the first example we have encountered where `gosset` does not know the exact moment matrix of the modeling region, and must therefore use Monte Carlo methods to evaluate this matrix. In the `moments` command we ask for 1000000 samples to be used (see the discussion in Sect. 4.3). This takes only a few seconds on a fast machine.

Remark. If the user knows the exact moment matrix, then, after running the `moments` command, the `moments.h` file can be edited to replace the approximate entries by the exact values.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd 4mix    [we create a working directory called 4mix]

10 range 0 1 a b c d
20 constraint a+b+c+d=1
30 model (a+b+c+d+1)^2

compile   [wait for compile done]
moments n=1000000   [wait for moments done]
design runs=12 n=20   [wait for design done]
```

Here is the result:

```
local best is 4mix/v.12.best
interp

#1:a+b+c+d=1

4mix/v.12.best
 a      b      c      d      #1

1.0000 0.0000 0.0000 0.0000 2.0000
0.5026 0.4974 0.0000 0.0000 1.0052
0.5001 0.0000 0.4999 0.0000 1.0002
0.5001 0.0000 0.0000 0.4999 1.0002
0.2959 0.1100 0.2978 0.2963 0.5919
0.2154 0.3550 0.2137 0.2159 0.4308

0.0000 1.0000 0.0000 0.0000 0.0000
0.0000 0.4974 0.0000 0.5026 0.0000
0.0000 0.4970 0.5030 0.0000 0.0000
0.0000 0.0000 1.0000 0.0000 0.0000
0.0000 0.0000 0.5002 0.4998 0.0000
0.0000 0.0000 0.0000 1.0000 0.0000

iv
      4mix/v.12.best       0.394290949649
```

Note that the last column is related to the constraint #1, and is not part of the design.

We have discovered that many of these mixture designs have a simple structure: see [HS8].

7.9. A constrained mixture

Five continuous variables `a`, `b`, `c`, `d`, `e` forming a mixture: `a+b+c+d+e=1`; with the additional constraints `d<.2`, `.01<e<.05`; a full quadratic model; find a minimal design with 15 runs; take the best of 20 attempts.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd 5mix    [we create a working directory called 5mix]

10 range 0 1 a b c
20 range 0 .2 d
30 range .01 .05 e
40 constraint a+b+c+d+e=1
50 model (a+b+c+d+e+1)^2

compile    [wait for compile done]
moments n=1000000    [wait for moments done]
design runs=15 n=20    [wait for design done]
```

Here is the result:

```
local best is 5mix/v.15.best
interp

#1:a+b+c+d+e=1

5mix/v.15.best
  a      b      c      d       e        #1

 0.9900 0.0000 0.0000 0.00000 0.01000  1.9800
 0.8757 0.0000 0.0012 0.07317 0.05000  1.7513
 0.7766 0.0000 0.0000 0.20000 0.02342  1.5532
 0.4855 0.4819 0.0000 0.00000 0.03268  0.9709
 0.4701 0.0000 0.4438 0.06230 0.02377  0.9403
 0.4300 0.4413 0.0144 0.10426 0.01000  0.8599
 0.2676 0.2445 0.2379 0.20000 0.05000  0.5351
 0.1599 0.1300 0.6602 0.00000 0.05000  0.3198
 0.0819 0.1077 0.6004 0.20000 0.01000  0.1637

 0.0000 0.9900 0.0000 0.00000 0.01000  0.0000
 0.0000 0.8564 0.0228 0.07076 0.05000  0.0000
 0.0000 0.7747 0.0000 0.20000 0.02534  0.0000
 0.0000 0.4689 0.4501 0.05701 0.02393  0.0000
 0.0000 0.0000 0.9900 0.00000 0.01000  0.0000
 0.0000 0.0000 0.8214 0.13825 0.04032  0.0000

iv
      5mix/v.15.best       0.597757217245
```

Note that the last column is related to the constraint #1, and is not part of the design.

7.10. Linear (or "Plackett-Burman" type) designs

7.10.1. Continuous cube

Five continuous variables in the range -1 to $+1$; full linear model; find a minimal design with 6 runs; take the best of 10 attempts.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd 5lin    [we create a working directory called 5lin]

10 range  a b c d e
20 model 1+a+b+c+d+e

compile   [wait for compile done]
moments   [wait for moments done]
design n=10   [wait for design done]
```

Here is the result:

```
interp

5lin/v.6.best
  a       b       c       d       e

 1.0000  1.0000  1.0000  1.0000  1.0000
-0.2417 -1.0000 -1.0000  1.0000  1.0000
-1.0000 -0.2417  1.0000  1.0000 -1.0000
-1.0000  1.0000 -0.2417 -1.0000  1.0000
 1.0000  1.0000 -1.0000 -0.2417 -1.0000
 1.0000 -1.0000  1.0000 -1.0000 -0.2417

iv
    4sphere/v.6.best       0.533098135455
```

We have found similar designs for linear models containing 2 through 24 variables. When the number of variables is 1 less than a multiple of 4 we recover the familiar Plackett-Burman-Rao (or Hadamard) designs.

For 2, 4 and 6 variables the following (presumably) I-optimal linear designs were obtained.

2 variables:

```
2lin/v.3.best
  a       b

 1.0000  1.0000
-1.0000  0.4391
 0.4391 -1.0000

iv
    2lin/v.3.best       0.666365890955
```

4 variables:

```
4lin/v.5.best
  a  b  c  d

  1  1  1  1
  1 -1 -1 -1
 -1  1 -1 -1
 -1 -1  1 -1
 -1 -1 -1  1

iv
    4lin/v.5.best       0.518518518519
```

6 variables:

```
6lin/v.7.best
  a  b  c  d  e  f

  1  1  1  1  1  1
  1  1  1 -1 -1 -1
  1 -1 -1  1  1 -1
  1 -1 -1 -1 -1  1
 -1  1 -1  1 -1  0
 -1  1 -1 -1  1  0
 -1 -1  1  0  0  0

iv
    6lin/v.7.best       0.520833333333
```

7.10.2. Two-level variables

Five two-level variables taking the values -1 and $+1$; full linear model; find a minimal design with 6 runs; take the best of 10 attempts.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd 5lin    [we create a working directory called 5lin]

10 discrete  a b c d e
20 model 1+a+b+c+d+e

compile    [wait for compile done]
moments    [wait for moments done]
design n=10    [wait for design done]
```

Here is the result:

```
interp

5lin/v.6.best
 a  b  c  d  e

 1  1  1 -1  1
 1 -1  1  1 -1
 1 -1 -1 -1 -1
-1  1 -1  1 -1
-1 -1  1 -1 -1
-1 -1 -1  1  1

iv
    5lin/v.6.best       1.200000000000
```

Again, when the number of variables is 1 less than a multiple of 4 we recover the familiar Plackett-Burman-Rao (or Hadamard) designs.

7.10.3. Plackett-Burman design with 24 runs

`gosset` has trouble finding Plackett-Burmann designs with more than 19 variables (this is the most difficult type of problem for the program — there are now over 400 coordinates in the design, all discrete), so we have included a number of such designs in the built-in library `codelib.a`.

This example shows how to obtain a Plackett-Burman design with 23 variables (i.e. a Hadamard matrix of order 24) from the library.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd PB    [we create a working directory called PB]

10 discrete a b c d e f g h i j k l m n o p q r s t u v w
```

Remember that `w` is the 23rd letter of the alphabet! In general you would type `x1 x2` etc. to `xm` here, when looking for a Plackett-Burman design with `m` variables, or a Hadamard matrix of order `m+1`.

```
20 model  1+a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w

compile   [wait for compile done]
moments   [wait for moments done]
design type=D program=lib   [wait for design done]
```

Here is the result:

```
interp precision=0

PB/d.24.best
 a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w
```

```
   1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
   1  1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1  1 -1  1 -1
   1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1  1 -1  1 -1  1
   1  1 -1  1 -1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1
   1  1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1
   1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1  1 -1  1 -1  1  1
   1 -1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1  1 -1
   1 -1  1 -1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1
   1 -1  1 -1 -1  1  1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1 -1 -1 -1 -1
   1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1
   1 -1 -1  1  1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1 -1 -1 -1 -1  1 -1
   1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1  1 -1  1 -1  1  1  1

  -1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1  1 -1  1
  -1  1  1 -1  1 -1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1 -1
  -1  1  1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1
  -1  1 -1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1  1
  -1  1 -1  1 -1 -1  1  1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1 -1 -1 -1
  -1  1 -1 -1  1  1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1 -1 -1 -1 -1  1
  -1 -1  1  1 -1  1 -1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1
  -1 -1  1  1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1 -1 -1 -1 -1  1 -1  1
  -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1 -1 -1
  -1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1 -1
  -1 -1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1
  -1 -1 -1 -1 -1  1 -1  1 -1 -1  1  1 -1 -1  1  1 -1  1 -1  1  1  1  1

dv
   PB/d.24.best       0.041666666667
iv
   PB/d.24.best       1.000000000000
```

A Hadamard matrix of order 24 is then obtained by adjoining an initial column of 1's to this array. As a check, note that the *D*-value of a Plackett-Burman design with $n - 1$ variables (or a Hadamard matrix of order n) is $1/n$. Here it is $1/24 = 0.041666666667$. The *IV*-value is always 1.

7.11. A D-optimal example, correcting an error in a *Technometrics* design

A recent issue of *Technometrics* contains a design that is asserted (without proof) to be D-optimal ([STW91], Table 10). We tested it and were easily able to find a better design. (There is an unfortunate error in the caption to that table —the column headings should be read `B D r s A C`. But even with this correction the design is still not D-optimal.)

The `gosset` specification for the design is

```
10 discrete A B C D r s
20 model A+B+C+D+r+s+A*(B+C+D)+r*s+(A+B+C+D)*(r+s)

design type=D runs=22
```

The design found by `gosset` has a very simple structure. There are three layers of points (in six dimensions): the North pole

`+ + + + + +`, (here + means `+1`, - means `-1`),

fifteen points which lie at the midpoints of the edges of a 5-simplex:

`+ + + + - -`, where the - signs can be anywhere,

and the six vertices of another 5-simplex:

`+ - - - - -`, where the + can be anywhere.

We conjecture that our design is both D-optimal and I-optimal. It has D-value 0.048400422226, compared with 0.0485385421 for the (corrected) design in [STW91]. The improvement is quite dramatic when we examine det $X' X$. For our design,

$$\det X' X = 2^{64} 3^6 5 7 = 470668675040699209482240,$$

whereas for the published design

$$\det X' X = 2^{62} 3^6 7 19 = 447135241288664249008128.$$

The improvement is

$$23533433752034960474112 = 2.353 \; 10^{22}.$$

The design found by `gosset` has roughly the shape of a "haystack", and we have found that such designs are apparently optimal, for a model consisting of all main effects and all interactions, over a fairly wide range of dimensions (although not for all dimensions). We plan to publish these results elsewhere.

7.12. D-optimality: a maximal determinant problem of Galil

It is a classical problem to find the $k \times n$ matrix X, with entries $+1$ and -1, for which det $X' X$ is maximized. Galil [Ga85] mentions that the smallest unsolved case of this problem is $k = 19$, $n = 16$. The `gosset` formulation of this problem is

```
10 discrete x1 x2 ... x15
20 model 1+x1+x2+...+x15

design runs=19 type=D
```

(Actually `type=I` gives the same result.)

The following matrix is the best of 4000 attempts, and has

$$\det X' X = 154473467218808012800 = 2^{54} 5^2 7^3,$$

which we conjecture is maximal.

1	−1	−1	−1	−1	−1	1	−1	1	1	−1	−1	1	1	1	−1
1	−1	−1	−1	−1	1	1	1	−1	−1	−1	−1	1	−1	1	1
1	−1	−1	1	−1	1	1	1	−1	−1	1	−1	−1	1	1	1
1	−1	−1	1	1	−1	−1	1	1	1	1	1	1	1	−1	1
1	−1	−1	1	1	1	−1	−1	1	−1	−1	1	−1	−1	1	−1
1	−1	1	−1	−1	1	1	1	1	−1	−1	1	1	−1	−1	1
1	−1	1	−1	1	−1	1	−1	−1	1	1	1	−1	−1	1	1
1	−1	1	−1	1	1	−1	−1	−1	−1	1	−1	1	1	−1	−1
1	−1	1	1	−1	−1	−1	1	−1	1	−1	−1	−1	−1	−1	−1

1	1	−1	−1	−1	−1	−1	−1	1	−1	1	−1	−1	−1	−1	1
1	1	−1	−1	−1	1	−1	1	−1	1	1	1	1	−1	1	−1
1	1	−1	−1	1	−1	1	1	−1	−1	−1	1	−1	1	−1	−1
1	1	−1	1	−1	1	1	1	−1	1	1	1	−1	1	−1	−1
1	1	−1	1	1	1	1	−1	−1	1	−1	−1	1	−1	−1	1
1	1	1	−1	1	1	−1	1	1	1	−1	−1	−1	1	1	1
1	1	1	1	−1	−1	−1	−1	−1	−1	−1	1	1	1	1	1
1	1	1	1	−1	1	1	−1	1	−1	1	−1	1	1	−1	−1
1	1	1	1	−1	1	1	−1	1	1	−1	1	−1	1	1	1
1	1	1	1	1	−1	1	1	1	−1	1	−1	1	−1	1	−1

This and several similar designs we have found are discussed further in [HS11].

7.13. Measuring region discrete, modeling region continuous

Quadratic model; 3 variables in cube; 15 runs; make measurements only at vertices and center of cube; fit model over whole cube; find I-optimal design.

In such a simple example it doesn't matter much (for the purposes of constructing the design) whether we model over the discrete measurement set or over the whole cube. This example is just intended to demonstrate the use of `gosset` in a case where the measurement region and modeling region are different.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd 3cube    [we create a working directory called 3cube]

10 discrete x y z -1 0 1    [unprimed variables: measurement region]
20 range x' y' z'    [matching primed variables: modeling region]
30 model (1+x'+y'+z')^2    [the primes could be omitted in the model]

compile    [wait for compile done]
moments    [wait for moments done]
design runs=15 n=25    [wait for design done]
```

Here is the result:

```
interp precision=0 | sort +0nr +1nr

3cube/v.15.best
  x   y   z

  1   1   1
  1   1  -1
  1   0   0
  1  -1   1
  1  -1  -1
  0   1   0
  0   0   1
  0   0   0
  0   0  -1
  0  -1   0

 -1   1   1
 -1   1  -1
 -1   0   0
 -1  -1   1
 -1  -1  -1

iv
   4sphere/v.15.best       0.367592592593
```

7.14. Estimating a response in a room from measurements made only in one half

This example was described in Sect. 3.11. We give here a minimal design for a quadratic model.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd room    [we create a working directory called room]

10 range x' -1 1
20 range y' z' 0 1
30 range x y z 0 1
40 model (1+x'+y'+z')^2

compile   [wait for compile done]
moments   [wait for moments done]
design n=10   [wait for design done]
```

Here is the result:

```
interp

room/v.10.best
 x      y      z

1.0000 0.7105 0.1384
1.0000 0.1380 0.7979
0.5455 0.0024 0.0000
0.5401 0.6632 1.0000
0.4997 1.0000 0.0169
0.4850 0.2774 0.5679
0.0000 1.0000 0.8187
0.0000 0.6397 0.0000
0.0000 0.2653 1.0000
0.0000 0.0000 0.1005

iv
   room/v.10.best       5.417618900322
```

7.15. Estimating a response on the moon from measurements made on earth

This is an extreme example of a situation where the measurement region (the earth) is disjoint from the modeling region (the moon). We want a minimal design.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd moon    [we create a working directory called moon]

10 sphere radius 4000 x y z    [unprimed variables: measurement region]
20 sphere radius 1000 center 240000 0 0 x' y' z'    [primed: modeling region]
30 model (1+x+y+z)^2

compile   [wait for compile done]
moments   [wait for moments done]
design n=10   [wait for design done]
```

Here is the result:

```
interp

moon/v.10.best
     x     y      z

   3997  154.5  -46.7
   3997 -154.0   47.5
   -159 -102.3 -240.8
   -162  -60.8  235.7
   -178  111.7    6.3
   -180  207.5  -23.7
   -186 -157.8   24.9
  -3990 -130.6 -244.6
  -3991  -27.4  259.2
  -3997  160.3  -12.1

iv
   moon/v.10.best 5.337e6
```

Similarly, the following design gives an optimal way to model the temperature of the sun from measurements made on (and in) the earth.

```
10 sphere x y z radius 4000
20 sphere x’ y’ z’ radius 1e6 center 92e6 0 0
30 model (1+x+y+z)ˆ2

interp

sun/v.10.best
     x     y      z

   3993  131.5 -194.0
   3986  324.8   54.7
   3973 -441.4  134.3
    194   59.7  315.2
    182 -159.8 -344.1
    157  169.0  -39.6
    149  294.0  -90.0
    114 -376.3  164.4
  -3989   56.0  290.9
  -3989  -56.0 -294.1

iv
     sun/v.10.best 1.169e17
```

This is essentially the same as the "moon" design.

7.16. A design that guards against loss of one run

Five variables in the cube; estimate intercept and main effects; 10 runs; I-optimal; protect against possible loss of one run (cf. Sect. 4.8).

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd guard    [we create a working directory called guard]

10 range a b c d e
20 model 1+a+b+c+d+e

compile   [wait for compile done]
moments   [wait for moments done]

design runs=10 type=J n=40   [wait for design done]
```

Here is the result:

```
guard/j.10.best
  a       b       c       d       e

 1.0000  1.0000  1.0000 -1.0000  1.0000
 1.0000  1.0000 -1.0000  1.0000 -1.0000
 1.0000 -1.0000  1.0000  1.0000  1.0000
 1.0000 -1.0000  1.0000 -1.0000 -1.0000
 1.0000 -1.0000 -1.0000 -1.0000  1.0000
-1.0000  1.0000  1.0000  1.0000 -1.0000
-1.0000  1.0000 -1.0000 -1.0000 -1.0000
-1.0000  1.0000 -1.0000  1.0000  1.0000
-1.0000 -1.0000  1.0000 -1.0000  1.0000
-1.0000 -1.0000 -1.0000  1.0000 -1.0000

iv j.10.best
         guard/j.10.best       0.285185185185

jv j.10.best
         guard/j.10.best       0.356481481481
```

This design is one of a family we have discovered with the property that all the subdesigns have equal *IV*-values. In this example the *IV*-value of every subdesign is 5/4 times that of the parent design.

This design should be compared with the I-optimal design which has a slightly better *IV*-value but a much worse *J*-value:

```
interp v.10.best

guard/v.10.best
  a       b       c       d       e

 1.0000  1.0000  1.0000 -1.0000 -1.0000
 1.0000  1.0000 -1.0000 -1.0000  1.0000
 1.0000 -1.0000  1.0000  1.0000 -1.0000
 1.0000 -1.0000 -1.0000  1.0000  1.0000
-1.0000  1.0000  1.0000  1.0000  1.0000
-1.0000  1.0000 -1.0000  1.0000 -1.0000
-1.0000  1.0000 -1.0000 -1.0000  1.0000
-1.0000 -1.0000  1.0000  1.0000 -1.0000
-1.0000 -1.0000  1.0000 -1.0000  1.0000
-1.0000 -1.0000 -1.0000 -1.0000 -1.0000

iv
          guard/v.10.best        0.284722222222

jv
          guard/v.10.best        0.451388888889
```

Designs of this type are described in more detail in [HS9].

7.17. An eye-measurement experiment, illustrating a design with correlated errors

In this experiment we measure a response of the eye which we assume is a quadratic function of two variables in a circle. The measurements are to be made in pairs, and the errors in successive measurements are assumed to have the following correlation matrix C:

$$C = \begin{bmatrix} 1 & .8 & 0 & 0 & 0 & . \\ .8 & 1 & 0 & 0 & 0 & . \\ 0 & 0 & 1 & .8 & 0 & . \\ 0 & 0 & .8 & 1 & 0 & . \\ 0 & 0 & 0 & 0 & 1 & . \\ . & . & . & . & . & . \end{bmatrix}$$

To pass this matrix to `gosset` we use `samplecv` lines, as described in Sect. 4.9.

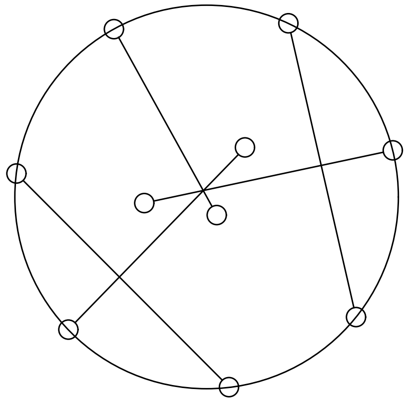
We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd eyes   [we create a working directory called eyes]

10 sphere x y
20 model (1+x+y)^2
30 samplecv double samplecv(i,j,runs) {
40 samplecv     if(i==j)return 1.;
50 samplecv     if(i/2==j/2)return 0.8;
60 samplecv     return 0.0;
70 samplecv }

compile   [wait for compile done]
moments   [wait for moments done]
design runs=10 n=100   [wait for design done]
```

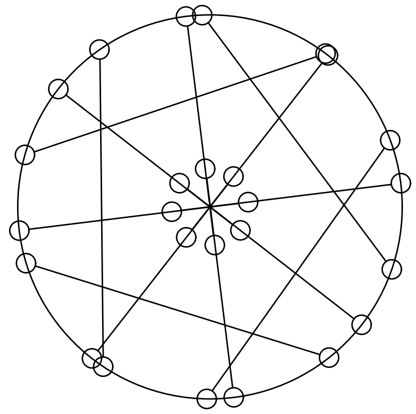
Here is the result:

```
interp

eyes/v.10.best
  x       y

-0.4837  0.8753  [the pairs of points have been separated by spaces]
 0.0525 -0.0950

 0.9701  0.2427
-0.3248 -0.0330

 0.2008  0.2575
-0.7217 -0.6922

-0.9927  0.1206
 0.1165 -0.9932

 0.7789 -0.6271
 0.4262  0.9046

iv
   eyes/v.10.best       0.268925239197
```

If these points are plotted in a circle, and each successive pair of points is joined by a straight line, an interesting picture is obtained.



The designs with around 26 runs are even prettier. For example:



Designs of this type are described in more detail in [HS10].

7.18. Block designs

Please see the discussion of block designs in Sect. 4.10.

7.18.1. A second-order response surface design in two blocks

Two variables, quadratic model, two blocks of size 5. This is an example from Atkinson and Donev [AD92], p. 159, Fig. 14.1 (a). The examples in Figs. 14.1 (b) - (d) of [AD92] could also be obtained by changing the `cmatrix` lines.

```
10 discrete x y -1 0 1
20 model (1+x+y)^2-1
30 misc K=5
40 cmatrix double cmatrix(i,j,n) {
50 cmatrix   if(i==j)return 1-1.0/K;
60 cmatrix   if(i/K==j/K)return -1.0/K;
70 cmatrix   return 0;
80 cmatrix }

design type=D runs=10 n=30
```

Here is the result:

```
interp

block/d.10.best
  x       y

 0.0000  1.0000
-1.0000 -1.0000
 1.0000 -1.0000
-1.0000  0.0000
 1.0000  0.0000

 1.0000  1.0000
 1.0000 -1.0000
-1.0000  1.0000
 0.0000 -1.0000
 0.0000  0.0000

dv
      block/d.10.best       0.263245109095
```

7.18.2. A design with four blocks of size 10

This problem was proposed to us by Jolene Splett and her colleagues at the National Institute of Standards and Technology in Boulder, Colorado, for possible use in studying the performance of an etching machine for superconducting chips.

Find a D-optimal design with 40 runs in four blocks of size 10; for a quadratic function of four variables in a cube; the first block should consist of the 2^{4-1} fractional factorial design with `abcd = +1`, supplemented by two measurements at the origin; the second block should consist of the 2^{4-1} fractional factorial design with `abcd = -1`, supplemented by two measurements at the origin; the problem is to find an optimal choice for the two remaining blocks of size 10.

(The star points `2,0,0,0` etc., would only provide one more block—compare Table 15.3 of Box and Draper [BD87]—and in any case lie outside the cube. See also the following example)

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd NIST   [we create a working directory called NIST]

10 range a b c d
20 model (1+a+b+c+d)^2-1
40 misc K=10
50 cmatrix double cmatrix(i,j,n) {
60 cmatrix   if(i==j)return 1-1.0/K;
70 cmatrix   if(i/K==j/K)return -1.0/K;
80 cmatrix   return 0;
90 cmatrix }
```

```
100 use a=1 b=1 c=1 d=1
103 use a=1 b=1 c=-1 d=-1
105 use a=1 b=-1 c=1 d=-1
106 use a=1 b=-1 c=-1 d=1
109 use a=-1 b=1 c=1 d=-1
110 use a=-1 b=1 c=-1 d=1
112 use a=-1 b=-1 c=1 d=1
115 use a=-1 b=-1 c=-1 d=-1
116 use a=0 b=0 c=0 d=0
117 use a=0 b=0 c=0 d=0

201 use a=1 b=1 c=1 d=-1
202 use a=1 b=1 c=-1 d=1
204 use a=1 b=-1 c=1 d=1
207 use a=1 b=-1 c=-1 d=-1
208 use a=-1 b=1 c=1 d=1
211 use a=-1 b=1 c=-1 d=-1
213 use a=-1 b=-1 c=1 d=-1
214 use a=-1 b=-1 c=-1 d=1
218 use a=0 b=0 c=0 d=0
219 use a=0 b=0 c=0 d=0

compile   [wait for compile done]
moments   [wait for moments done]
design runs=40 n=500 processors=8   [wait for design done]
```

Here is the result (with the blocks separated by spaces):

```
interp

NIST/d.40.best
  a       b       c       d

 1.0000  1.0000  1.0000  1.0000
 1.0000  1.0000 -1.0000 -1.0000
 1.0000 -1.0000  1.0000 -1.0000
 1.0000 -1.0000 -1.0000  1.0000
-1.0000  1.0000  1.0000 -1.0000
-1.0000  1.0000 -1.0000  1.0000
-1.0000 -1.0000  1.0000  1.0000
-1.0000 -1.0000 -1.0000 -1.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000
```

```
 1.0000  1.0000  1.0000 -1.0000
 1.0000  1.0000 -1.0000  1.0000
 1.0000 -1.0000  1.0000  1.0000
 1.0000 -1.0000 -1.0000 -1.0000
-1.0000  1.0000  1.0000  1.0000
-1.0000  1.0000 -1.0000 -1.0000
-1.0000 -1.0000  1.0000 -1.0000
-1.0000 -1.0000 -1.0000  1.0000
 0.0000  0.0000  0.0000  0.0000
 0.0000  0.0000  0.0000  0.0000

-1.0000 -1.0000 -1.0000 -1.0000
 1.0000  0.0347  1.0000 -1.0000
-1.0000 -0.0249 -0.0690 -1.0000
-1.0000  1.0000 -0.0423  0.0800
 1.0000 -1.0000 -1.0000  0.0118
-1.0000 -1.0000  1.0000 -0.0286
 1.0000  1.0000 -0.1663  1.0000
-1.0000  0.0711  1.0000  1.0000
 0.0171 -1.0000 -1.0000  1.0000
 0.0816  1.0000 -1.0000 -1.0000

 1.0000  1.0000  1.0000 -0.0996
 1.0000  0.1444 -1.0000  0.0074
-1.0000  0.1768 -1.0000  0.1923
-0.0332 -1.0000  1.0000 -1.0000
 1.0000 -1.0000  1.0000  1.0000
-0.0881  1.0000  1.0000  1.0000
 1.0000 -1.0000 -0.1237 -1.0000
-1.0000  0.0653  1.0000 -1.0000
-1.0000 -1.0000 -0.0450  1.0000
-1.0000  1.0000  0.0998 -1.0000

dv
       NIST/d.40.best        0.056825974897
```

7.18.3. The central composite design in three blocks

As a test of the program, we decided to see if it would find the classical central composite design (as in [BD87], p. 514), starting from two blocks of 9 = 8 + 1 points each. The result was a surprise. We posed the problem as follows.

Find a D-optimal design with 27 runs in three blocks of size 9; for a quadratic function of four variables in a sphere of radius 2; given that the first block should consist of the 2^{4-1} fractional factorial design with `abcd = +1`, supplemented by a measurement at the origin; the second block should consist of the 2^{4-1} fractional factorial design with `abcd = -1`, also supplemented by a measurements at the origin; the problem is to find an optimal choice for the third block of size 9.

The first `gosset` formulation of this problem that we tried was

```
10 sphere a b c d radius 2
20 model (1+a+b+c+d)^2-1
30 misc K=9
40 cmatrix double cmatrix(i,j,n) {
50 cmatrix   if(i==j)return 1-1.0/K;
60 cmatrix   if(i/K==j/K)return -1.0/K;
70 cmatrix   return 0;
80 cmatrix }

90 use a=1 b=1 c=1 d=1
100 use a=1 b=1 c=-1 d=-1
110 use a=1 b=-1 c=1 d=-1
120 use a=1 b=-1 c=-1 d=1
130 use a=-1 b=1 c=1 d=-1
140 use a=-1 b=1 c=-1 d=1
150 use a=-1 b=-1 c=1 d=1
160 use a=-1 b=-1 c=-1 d=-1
170 use a=0 b=0 c=0 d=0

180 use a=1 b=1 c=1 d=-1
190 use a=1 b=1 c=-1 d=1
200 use a=1 b=-1 c=1 d=1
210 use a=1 b=-1 c=-1 d=-1
220 use a=-1 b=1 c=1 d=1
230 use a=-1 b=1 c=-1 d=-1
240 use a=-1 b=-1 c=1 d=-1
250 use a=-1 b=-1 c=-1 d=1
260 use a=0 b=0 c=0 d=0

design type=D runs=27 n=20
```

Here is the result.

```
NIST/d.27.best
  a      b      c      d

 1.000  1.000  1.000  1.000
 1.000  1.000 -1.000 -1.000
 1.000 -1.000  1.000 -1.000
 1.000 -1.000 -1.000  1.000
-1.000  1.000  1.000 -1.000
-1.000  1.000 -1.000  1.000
-1.000 -1.000  1.000  1.000
-1.000 -1.000 -1.000 -1.000
 0.000  0.000  0.000  0.000
```

```
 1.000  1.000  1.000 -1.000
 1.000  1.000 -1.000  1.000
 1.000 -1.000  1.000  1.000
 1.000 -1.000 -1.000 -1.000
-1.000  1.000  1.000  1.000
-1.000  1.000 -1.000 -1.000
-1.000 -1.000  1.000 -1.000
-1.000 -1.000 -1.000  1.000
 0.000  0.000  0.000  0.000

 2.000  0.000  0.000  0.000
-2.000  0.000  0.000  0.000
 0.000  2.000  0.000  0.000
 0.000 -2.000  0.000  0.000
 0.000  0.000  0.750  1.854
 0.000  0.000 -0.750  1.854
 0.000  0.000  1.995 -0.137
 0.000  0.000 -1.995 -0.137
 0.000  0.000  0.000 -2.000

dv
      nist/d.27.best       0.529987240185
```

The program has cleverly placed five points of the last block at the vertices of a pentagon, instead of at the origin and the vertices of a square! This gives a slightly better D-value than the central composite design.

So we told it to place one point of the last block at the origin (see line `270`), and this time it immediately found the star points plus center.

```
10 sphere a b c d radius 2
20 model (1+a+b+c+d)^2-1
30 misc K=9
40 cmatrix double cmatrix(i,j,n) {
50 cmatrix   if(i==j)return 1-1.0/K;
60 cmatrix   if(i/K==j/K)return -1.0/K;
70 cmatrix   return 0;
80 cmatrix }

90 use a=1 b=1 c=1 d=1
100 use a=1 b=1 c=-1 d=-1
110 use a=1 b=-1 c=1 d=-1
120 use a=1 b=-1 c=-1 d=1
130 use a=-1 b=1 c=1 d=-1
140 use a=-1 b=1 c=-1 d=1
150 use a=-1 b=-1 c=1 d=1
160 use a=-1 b=-1 c=-1 d=-1
170 use a=0 b=0 c=0 d=0
```

```
180 use a=1 b=1 c=1 d=-1
190 use a=1 b=1 c=-1 d=1
200 use a=1 b=-1 c=1 d=1
210 use a=1 b=-1 c=-1 d=-1
220 use a=-1 b=1 c=1 d=1
230 use a=-1 b=1 c=-1 d=-1
240 use a=-1 b=-1 c=1 d=-1
250 use a=-1 b=-1 c=-1 d=1
260 use a=0 b=0 c=0 d=0
270 use a=0 b=0 c=0 d=0

design type=D runs=27 n=20
```

Here is the result.

```
nist/d.27.best
  a      b      c      d

 1.000  1.000  1.000  1.000
 1.000  1.000 -1.000 -1.000
 1.000 -1.000  1.000 -1.000
 1.000 -1.000 -1.000  1.000
-1.000  1.000  1.000 -1.000
-1.000  1.000 -1.000  1.000
-1.000 -1.000  1.000  1.000
-1.000 -1.000 -1.000 -1.000
 0.000  0.000  0.000  0.000

 1.000  1.000  1.000 -1.000
 1.000  1.000 -1.000  1.000
 1.000 -1.000  1.000  1.000
 1.000 -1.000 -1.000 -1.000
-1.000  1.000  1.000  1.000
-1.000  1.000 -1.000 -1.000
-1.000 -1.000  1.000 -1.000
-1.000 -1.000 -1.000  1.000
 0.000  0.000  0.000  0.000

 0.000  0.000  0.000  0.000
 2.000  0.000  0.000  0.000
-2.000  0.000  0.000  0.000
 0.000  2.000  0.000  0.000
 0.000 -2.000  0.000  0.000
 0.000  0.000  2.000  0.000
 0.000  0.000 -2.000  0.000
 0.000  0.000  0.000  2.000
 0.000  0.000  0.000 -2.000

dv
      NIST/d.27.best       0.531792968585
```

7.19. A balanced incomplete block design

`gosset` can find small examples of balanced incomplete block designs. In fact, they don't have to be balanced, and you can specify arbitrary block sizes (see Sect. 4.10). In the following example we look for 7 blocks of size 3 with 7 variables that take the values 0 and 1, in other words we construct the classical BIBD with

```
v=7, b=7, r=3, k=3
```

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd blockdir    [we create a working directory called blockdir]

10 range a b c d e f g 0 1
20 model a+b+c+d+e+f+g-1
30 constraint a+b+c+d+e+f+g=1
40 misc K=3
50 cmatrix double cmatrix(i,j,n) {
60 cmatrix   if(i==j)return 1-1.0/K;
70 cmatrix   if(i/K==j/K)return -1.0/K;
80 cmatrix   return 0;
90 cmatrix }

compile   [wait for compile done]
moments   [wait for moments done]
design runs=21 n=100 type=D processors=8   [wait for design done]
```

Here is the result:

```
interp

#1:a+b+c+d+e+f+g=1

blockdir/d.21.best
a b c d e f g #1

0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0

0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0

0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 2
0 0 0 0 1 0 0 0
```

```
0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 2
0 0 0 0 0 0 1 0

1 0 0 0 0 0 0 2
0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0

0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 1 0 0 0 0 0

0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0

dv
       nist/d.21.best        0.148187952243
```

Note that the last column is related to the constraint #1, and is not part of the design.

We have separated the blocks by blank lines. If we simply record which variables appear in each block, we obtain the array

```
0011010
0000111
1010100
1001001
1100010
0110001
0101100
```

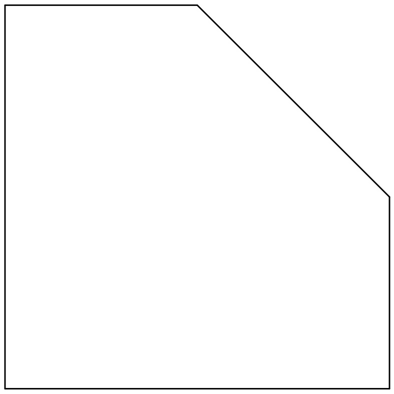
which is the BIBD we were seeking. Note that each row and column contains three 1's, and every two rows or columns meet in a single 1.

7.20. Packings

Please see the discussion of packing problems in Sect. 4.11.

7.20.1. Packing 12 points in an irregular region

Find the best packing of twelve points in the following region:



We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd pack    [we create a working directory called pack]

10 range x y
20 constraint x+y<=1
30 model x+y

compile   [wait for compile done]
moments n=0   [wait for moments done]
design type=P runs=12 n=1000 [wait for design done]
```

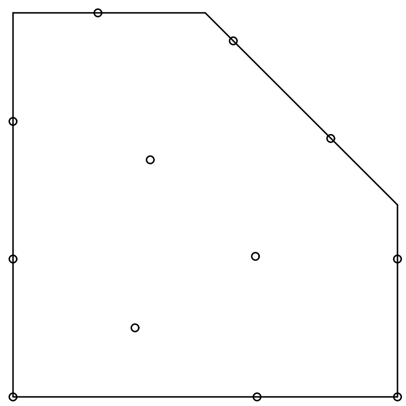
Here is the best packing found:

```
local best is pack/p.12.best
interp

#1:x+y<=1

pack/p.12.best
   x        y        #1

 1.0000 -0.2822  0.282
 1.0000 -1.0000  1.000
 0.6536  0.3464  0.000
 0.2699 -1.0000  1.730
 0.2621 -0.2678  1.006
 0.1461  0.8539  0.000
-0.2851  0.2342  1.051
-0.3649 -0.6410  2.006
-0.5567  1.0000  0.557
-1.0000  0.4355  1.564
-1.0000 -0.2822  2.282
-1.0000 -1.0000  3.000
```

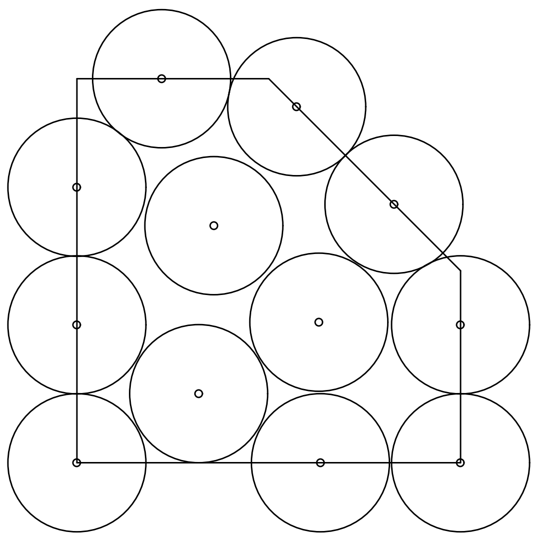
```
    pv
       pack/p.12.best        1.393224113714
```

Note that the last column is related to the constraint #1, and is not part of the design.

The minimal distance between the points is `1/pv` = 0.7178. The points are as follows:



If we draw circles of radius one-half the minimal distance around the points, we see that they do not overlap:



The picture shows that one of the points can be moved a small amount without changing the minimal distance. This is a common phenomenon in these packings.

7.20.2. Packing 20 points in a cube

Find the best packing of 20 points in a cube.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd packdir    [we create a working directory called packdir]

10 range a b c 0 1
20 model a+b+c

compile   [wait for compile done]
moments   [wait for moments done]
design type=P runs=20 n=2000 processors=8   [wait for design done]
```

Here is the result:

```
interp

packdir/p.20.best
 a      b      c

1.0000 1.0000 0.1436
1.0000 0.6077 0.5359
0.6077 1.0000 0.5359
1.0000 0.4641 0.0000
0.4641 1.0000 0.0000
1.0000 0.3039 1.0000
0.3039 1.0000 1.0000
1.0000 0.0000 0.5000
0.0000 1.0000 0.5000
0.9855 0.9855 1.0000

0.6961 0.0000 0.0000
0.0000 0.6961 0.0000
0.5359 0.0000 1.0000
0.0000 0.5359 1.0000
0.5230 0.5230 0.8103
0.5036 0.5036 0.2510
0.3923 0.0000 0.4641
0.0000 0.3923 0.4641
0.0166 0.0166 0.0000
0.0000 0.0000 0.8564

pv
   packdir/p.20.best       1.802577515476
```

The minimal distance between the points is 1/1.80257 = .554761 .

7.21. One quantitative and three qualitative variables

This is a design for studying consumer response to a new product. It was suggested by a question from our colleague Ramnath Lakshmi-Ratan.

There is one quantitative variable, `price`, and three qualitative variables. The first qualitative variable, `color`, takes four values, red, blue, green and yellow. We represent `color` by four binary variables, `CR`, `CB`, `CG`, `CY`, taking values `0` or `1`, with the constraint that their sum must be `1`. The second qualitative variable, `weight`, takes three values, light, medium or heavy. We represent `weight` by three binary variables, `WL`, `WM`, `WH`, again with the constraint that their sum must be `1`. Finally, the third qualitative variable, `material`, takes three values, cotton, nylon or wool. We represent `material` by three binary variables, `MC`, `MN`, `MW`, again with the constraint that their sum must be `1`. We seek a design with 24 runs.

We use the following model for the dollar value the consumer assigns to the specimen:

$$y = \beta_0 + \beta_{CR}\ CR + \beta_{CB}\ CB + \beta_{CG}\ CG + \beta_{CY}\ CY + \beta_{WL}\ WL + \cdots + \beta_{MW}\ MW + \varepsilon,$$

This is a "part-worth" model, in which the β_i are the dollar values the consumer gives to the individual features.

Alternatively, we may regard the coefficients β_{CR}, β_{CB}, β_{CG}, β_{CY} as being proportional to the preferential probabilities the consumer assigns to the four colors, β_{WL}, β_{WM}, β_{WH} as being proportional to the preferential probabilities the consumer assigns to the three weights, etc.

When running `gosset` we make the binary variables `CR`, `CB`, etc. continuous on 0, 1 rather than discrete, in order to simplify the search process. Since the binary variables appear only linearly in the model, `gosset` usually places them at their ends of their ranges in any case. Printing the design with `precision=0` rounds off the design points to the nearest integers.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd testdir    [we create a working directory called testdir]

10 range CR CB CG CY WL WM WH MC MN MW 0 1
20 constraint CR+CB+CG+CY=1
30 constraint WL+WM+WH=1
40 constraint MC+MN+MW=1
50 range price 1 5
60 model 1+CR+CB+CG+CY+WL+WM+WH+MC+MN+MW+price+price^2

compile    [wait for compile done]
moments n=1000000    [wait for moments done]

design type=D runs=24 n=100 [wait for design done]
```

Here is the result:

```
interp precision=0

#1:CR+CB+CG+CY=1
#2:WL+WM+WH=1
#3:MC+MN+MW=1

testdir/d.24.best
CB CG CR CY MC MN MW WH WL WM price #1 #2 #3
```

```
1  0  0  0  1  0  0  1  0  0   5   2  2  2
1  0  0  0  1  0  0  0  0  1   1   2  2  0
1  0  0  0  0  1  0  0  1  0   3   2  0  0
1  0  0  0  0  1  0  0  0  1   1   2  0  0
1  0  0  0  0  0  1  0  1  0   5   2  0  0
1  0  0  0  0  0  1  0  0  1   3   2  0  0
0  1  0  0  1  0  0  0  1  0   1   0  2  0
0  1  0  0  1  0  0  0  0  1   3   0  2  0
0  1  0  0  0  1  0  1  0  0   3   0  0  2
0  1  0  0  0  1  0  1  0  0   1   0  0  2
0  1  0  0  0  0  1  0  1  0   1   0  0  0
0  1  0  0  0  0  1  0  0  1   5   0  0  0

0  0  1  0  1  0  0  1  0  0   5   0  2  2
0  0  1  0  1  0  0  0  1  0   3   0  2  0
0  0  1  0  1  0  0  0  1  0   1   0  2  0
0  0  1  0  0  1  0  0  0  1   5   0  0  0
0  0  1  0  0  0  1  1  0  0   3   0  0  2
0  0  1  0  0  0  1  0  1  0   3   0  0  0
0  0  0  1  1  0  0  1  0  0   5   0  2  2
0  0  0  1  1  0  0  0  1  0   3   0  2  0
0  0  0  1  0  1  0  0  1  0   5   0  0  0
0  0  0  1  0  1  0  0  0  1   3   0  0  0
0  0  0  1  0  0  1  1  0  0   1   0  0  2
0  0  0  1  0  0  1  0  0  1   1   0  0  0

dv
   testdir/d.24.best        0.066400227362
```

Note that the last three columns are related to the constraints, and are not part of the design.

However, if the qualitative variables appear in the model in a more complicated way, that formulation of the problem doesn't work, because some of the test points end up with values like `CR = .6, CB = .4`, which is not meaningful.

This difficulty can be avoided by the following trick. This example shows the same problem, except that we have included the interactions of `color` and `weight` in the model (see line `70`). The trick is to declare three of the four `color` variables to be discrete and the fourth one continuous, while constraining their sum to be 1; and similarly for `weight` and `material`.

We enter `gosset`:

```
gosset
please type 'cd something' to name a local directory for your work
cd testdir

10 discrete  CB CG CY  WM WH  MN MW 0 1
20 range 0 1 CR WL MC
30 constraint CR+CB+CG+CY=1
40 constraint WL+WM+WH=1
50 constraint MC+MN+MW=1
60 range price 1 5
70 model 1+CR+CB+CG+CY+WL+WM+WH+MC+MN+MW+price+price^2
          +(CR+CB+CG+CY)*(WL+WM+WH)

compile   [wait for compile done]
moments n=1000000   [wait for moments done]

design type=D runs=24 n=100 [wait for design done]
```

Here is the result:

```
interp precision=2

#1:CR+CB+CG+CY=1
#2:WL+WM+WH=1
#3:MC+MN+MW=1

testdir/d.24.best

 CB   CG   CY   MN   MW   WH   WM   CR   MC   WL  price  #1   #2   #3

1.00 0.00 0.00 1.00 0.00 0.00 0.00 0.00 0.00 1.00   3.4 0.00 0.00 2.00
1.00 0.00 0.00 0.00 1.00 1.00 0.00 0.00 0.00 0.00   1.0 0.00 0.00 0.00
1.00 0.00 0.00 0.00 1.00 0.00 1.00 0.00 0.00 0.00   1.0 0.00 0.00 0.00
1.00 0.00 0.00 0.00 0.00 1.00 0.00 0.00 1.00 0.00   3.3 0.00 2.00 0.00
1.00 0.00 0.00 0.00 0.00 0.00 1.00 0.00 1.00 0.00   5.0 0.00 2.00 0.00
1.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 1.00 1.00   1.0 0.00 2.00 2.00
0.00 1.00 0.00 1.00 0.00 0.00 1.00 0.00 0.00 0.00   3.4 0.00 0.00 0.00
0.00 1.00 0.00 1.00 0.00 0.00 0.00 0.00 0.00 1.00   5.0 0.00 0.00 2.00
0.00 1.00 0.00 0.00 1.00 1.00 0.00 0.00 0.00 0.00   5.0 0.00 0.00 0.00
0.00 1.00 0.00 0.00 1.00 0.00 0.00 0.00 0.00 1.00   1.0 0.00 0.00 2.00
0.00 1.00 0.00 0.00 0.00 1.00 0.00 0.00 1.00 0.00   2.5 0.00 2.00 0.00
0.00 1.00 0.00 0.00 0.00 0.00 1.00 0.00 1.00 0.00   1.0 0.00 2.00 0.00
```

```
0.00 0.00 1.00 1.00 0.00 1.00 0.00 0.00 0.00 0.00   1.0 0.00 0.00 0.00
0.00 0.00 1.00 1.00 0.00 0.00 0.00 0.00 0.00 1.00   1.0 0.00 0.00 2.00
0.00 0.00 1.00 0.00 1.00 1.00 0.00 0.00 0.00 0.00   3.4 0.00 0.00 0.00
0.00 0.00 1.00 0.00 1.00 0.00 1.00 0.00 0.00 0.00   2.7 0.00 0.00 0.00
0.00 0.00 1.00 0.00 0.00 0.00 1.00 0.00 1.00 0.00   5.0 0.00 2.00 0.00
0.00 0.00 1.00 0.00 0.00 0.00 0.00 0.00 1.00 1.00   5.0 0.00 2.00 2.00
0.00 0.00 0.00 1.00 0.00 1.00 0.00 1.00 0.00 0.00   5.0 2.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00 0.00 1.00 1.00 0.00 0.00   1.0 2.00 0.00 0.00
0.00 0.00 0.00 1.00 0.00 0.00 0.00 1.00 0.00 1.00   2.5 2.00 0.00 2.00
0.00 0.00 0.00 0.00 1.00 0.00 1.00 1.00 0.00 0.00   3.4 2.00 0.00 0.00
0.00 0.00 0.00 0.00 1.00 0.00 0.00 1.00 0.00 1.00   5.0 2.00 0.00 2.00
0.00 0.00 0.00 0.00 0.00 1.00 0.00 1.00 1.00 0.00   2.5 2.00 2.00 0.00

dv
   testdir/d.24.best       0.076783183582
```

Note that the last three columns are related to the constraints, and are not part of the design.

The one slight disadvantage is that the range variables are listed after the discrete variables:

```
CB    CG    CY    MN    MW    WH    WM    CR    MC    WL   price
```

But this can easily be corrected by passing the `interp` output through `awk`.

7.22. Orthogonal arrays

`gosset` can be used to construct orthogonal arrays as long as the number of levels is not too large. Orthogonal arrays, even mixed arrays, with two and three levels present no difficulty.

Suppose we want an orthogonal array in which the first r variables have 3 levels and the remaining s variables have 2 levels (either r or s may be zero), for a total of $n = r + s$ variables. The trick is use the `gosset` specification

```
10 discrete x1 x2 ... xr -1 0 1
20 range x1' x2' ... xr'
30 discrete y1 y2 ... ys -1 1
40 range y1' y2' ... ys'
50 model (1+x1)^2+...+(1+xr)^2+y1+...+ys
```

In general, if the ith variable x_i can take q_i levels, the model should be

$$\sum_i (1 + x_i)^{q_i - 1} . \tag{1}$$

However, as already mentioned, this works best when the q_i are at most 3.

(**Question**. Is it always true that an orthogonal array is defined by the property of being an I-optimal design for the model of Eq. (1), with the ith variable restricted to values from the set $\{-1, -1 + 2/(q_i - 1), \ldots, 1 - 2/(q_i - 1), 1\}$, and with the modeling region equal to the cube?)

For example, let us construct the $OA(9, 4, 3, 2)$, with 9 runs, 4 variables, 3 levels, strength 2 and index 1. (This array is also equivalent to the tetracode. See [SPLAG, p. 81], [Ra71, p. 21], [Ta87, p. 1153].)

We enter gosset:

```
gosset
please type 'cd something' to name a local directory for your work
cd testdir    [we create a working directory called testdir]

10 discrete x1 x2 x3 x4 -1 0 1
20 range x1' x2' x3' x4'
30 model (1+x1)^2+(1+x2)^2+(1+x3)^2+(1+x4)^2

compile   [wait for compile done]
moments   [wait for moments done]

design runs=9 n=16 [wait for design done]
```

Here is the result:

```
interp precision=0

testdir/v.9.best
 x1  x2  x3  x4

  0   0   0   0
  1   1   1   0
  0   1  -1   1
 -1   0   1   1
  1  -1   0   1
 -1  -1  -1   0
  0  -1   1  -1
  1   0  -1  -1
 -1   1   0  -1

iv
   testdir/v.9.best 0.733333333333
```

8. Other gosset commands

The commands in this section are displayed as if prompt % had been invoked (see Sect. 8.12).

8.1. cd

cd changes directory to a new working directory.

Unlike the shell cd, there is no hierarchy. Everything is from the base directory where gosset started. For example

```
% cd temp
% cd work
```

changes the directory to ./temp and then to ./work, whereas the shell would have required cd temp and cd ../work

If the directory does not exist and can be created, it will be. If it exists and there are files

```
esse.program
esse.program2
esse.program3
```

in it, `gosset` will tell you about the `old` command. These files are versions of the old program saved by the last successful `compile`.

`cd` resets `search` to `codelib.a`.

Warning: when you change directories, you take the current program with you. To see the current program, type `list` (or `l`). This should always be done after a `cd` command, unless you immediately use `old`.

8.2. `check`, `acheck`, `dcheck`, `echeck`, `icheck`, `bcheck`, `ccheck`, `fcheck`, `jcheck`, `pcheck`

`check filename` checks whether `filename` is a design that satisfies all the conditions of the current program — that variables belong to the appropriate regions and satisfy the constraints, and that all points mentioned in `use` lines are present.

If all is well, the *IV*-value of the solution is printed. If not, a line starting with `BAD` and the violation is printed.

If an archive is given as `filename`, every design in it that has the appropriate dimension is checked.

`check ../codelib.a|grep -v BAD` is a useful way to see what the `gosset` library has of interest for the problem. (This will exclude designs that do not satisfy the requirements.)

`acheck`, `dcheck`, etc., are similar, except that if all the conditions of the current program are satisfied, they print the A-, D-, etc. values of `filename`. `icheck` is the same as `check`.

8.3. `cleanup` and `cleanup all`

`cleanup` removes files from the current directory that it thinks are not likely to be valuable. This includes files of the form `v.*.*`, `d.*.*`, etc., `vtrace.*` and `*.h`.

What is left is the current program; a file called `program.log` that lists old programs, dates and tags; `lib.a`, the tagged best solutions for the various programs; and any files created by the user that do not match the delete patterns. At the present time the `cleanup` commands are

```
rm -f x/*.h x/*.o x/vtrace x/vvv x/random x/interp x/moments
    x/moments.1 x/moments.2 x/moments.prog x/vtrace.*
rm -f x/[abcdefjpv].*.*
rm -f x/exam.file x/a.out x/core
```

But this list may be extended as more types are added. However, a file name all in capital letters will never be deleted.

Warning: make sure that any precious files of your own in the working directory have names that are all in capitals, otherwise `cleanup` may delete them. For example, after laboriously typing in that suspicious-looking design from the latest issue of *Technometrics* in order to check if it really is D-optimal, as claimed, call it D.6.22.TECH. If you call it d.6.22.tech it will be removed next time you run `cleanup`.

`cleanup all` must be issued in the base directory, not a working directory—in other words after calling `gosset`, but before typing `cd something`. It then runs `cleanup` in every subdirectory that contains a file called `esse.program`. Such a file exists in any working directory that has had a successfully compiled program.

It would be unwise to have files named `esse.program` in private directories hanging off the `gosset` base directory! And the warning given above about file names now applies to every working directory.

8.4. `comment`

Any text can be included in a `gosset` program on a line beginning with `comment`. For example

```
10 range a1 a2 a3 a4 a5
20 model 1+a1+a2+a3+a4+a5
30 comment Design for Dave Doehlert
40 comment Run this with: design runs=25 type=D
```

Any change to a comment line is deemed a change to the program, and gets a new four-letter tag.

8.5. `compiler`

`compiler` changes the name of the `C` compiler used. For example

```
% compiler lcc
```

changes the compiler from "cc" to "lcc." The default is "cc".

Sometimes the optimizer in the `C` compiler is very slow, and compiling takes longer than searching for the design. If so, the flags given to the `C` compiler, in particular the flag `-O` that tells it to produce optimized code, can be changed or omitted, in order to speed things up. The commands

```
% cflags none
% cflags -O
% cflags
```

instruct `gosset` respectively to omit the optimization flag, to put it back, and to report the current flags.

There are several variables in the programs that aren't always used (for example when there are no constraints). These may cause annoying warning messages with some old `C` compilers. Such messages can usually be suppressed by typing

```
% compiler cc -w
```

Warning: for low-dimensional D-optimal designs, the optimizer in the current SGI `C` compiler produces faulty code —see the example in Sect. 11.9. The command

```
cflags none
```

can be used to turn off the optimizer, which is the least reliable part of most `C` compilers.

Recommendations. On a Cray X-MP or Y-MP, use

```
% compiler scc
```

On an SGI machine, use

```
% compiler lcc
% cflags -w
```

8.6. delete

`delete` deletes all or some of the lines in the current program. For example

```
% delete
% delete 10
% delete 10 50
```

deletes everything, line 10, and lines 10 to 50, respectively.

The numbers can be expressions! This shows the utility of a good parser.

Single lines can be deleted also by typing the line number and a carriage return:

`% 20` (carriage return)

deletes line 20.

8.7. get

`get filename` fetches `filename`, usually from an archive somewhere, and places it in the current working directory. This is essentially a shell `ar x` command, except that it uses the same `filename` defaults as `iv`, `interp` and `check`. For example

```
% get real.a(v.5.19.aaab)
```

retrieves the design `v.5.19.aaab` from the archive `real.a` in the current directory, and places it in a file `v.5.19.aaab`, also in the current directory.

There is no `put`, since the shell `ar` command can be used for this purpose, as illustrated in

```
% !ar r lib.a E.15.16.d*
```

or

```
% !ar r ../codelib.a E.15.16.d*
```

which would add the designs `E.15.16.d*` to the libraries mentioned.

8.8. kill

`kill moments` or `kill design` causes either `moments` or `design` to stop and write out its current best answer as if it had finished normally.

`kill` does this internally by creating a file `xxxx.quit`, where `xxxx` is the `pid` of the `moments` or `design` process to be stopped. These two programs check for such a file every major loop, or every five seconds, whichever is longer, and will terminate when they see it. (`design` also reads the file `xxxx.quit`, and if there is text in the first line, it writes out its current best to the file of that name, and then continues normally. The command `examine` uses this feature.)

`kill design` can be used either to kill all designs currently running, or just one of them.

`kill design` kills all designs running; the `design` command stops completely.

`kill design.0` stops only the single process `design.0`, as shown by `watch` or `status`, and `design` will continue (after processing the result from the killed process) if there are other designs running or waiting to be run.

If `design` was started with the default `processors=1`, then `design.0` will be the only design running at the time, but there may be other designs waiting to be run.

If `design` was started with several processors, many `design` jobs will be running concurrently, with different trailing digits. `kill` stops the one named and the others continue.

8.9. `list` or `l`

`list` gives a listing of something.

`list` alone gives a listing of the current program.

`list all` gives a listing of all the saved programs in `program.log` in the working directory.

`list dif` prints only lines not in the previous program in `program.log`.

`list aaax` prints a listing of the program tagged `aaax` in `program.log`.

`list v.3.14.best` prints a general $A(0)$, $A(1)$, ..., listing of the design `v.3.14.best`.

`list v.3.best` prints a general $A(0)$, $A(1)$, ..., listing of the solution in `v.3.best`, guessing the dimensions of the problem from the name and number of lines.

`list lib.a(v.2.10.aaax)` prints a general listing of a file in an archive, in this case `lib.a` in the working directory.

The `interp` command prints the design in a better format than `list`, but requires a recompilation for each new problem.

It is often useful to pipe `list` output into a shell command. For example

```
% list all | grep '^p'
```

prints only the lines with program tag and date.

8.10. `old`

`old` deletes the currently edited program, if any, and reads in the last successfully compiled program in the working directory.

`old` with a four-character argument such as `aaax` reads in the saved program with that tag from `program.log`, where every unique successfully compiled program is saved in sequence. The same tag, starting with `aaaa`, appears in the names of designs saved in `lib.a` in the working directory, to aid in matching designs and programs.

`old` and `old aaax` behave slightly differently in one way, though: `old` with no argument recovers the complete program, whereas `old` with a tag recovers the program minus any `start` lines (these are deemed inessential to tagging).

Programs are usually short enough that they need not be saved. What is important is to keep track of which designs are associated with which programs, and the tags help with that.

Restarting a job. After typing `old`, you can run `design` right away, without running `compile` and `moments` first, provided they were run last time and the problem specification has not been changed.

The user may of course save or recover programs using the file redirection commands. For example

```
% write > filename
% list > filename
```

will write copies of the current program, compiled successfully or not, onto any file, without and with line numbers, respectively. Similarly

```
% delete
% <# filename
% < filename
```

will recover them.

8.11. Redirected output using > and |

Any command can have `>filename` or `|command` appended to it, and the output will go to a file, named relative to the working directory, or to a shell command.

The redirected output is everything that is printed until the next command is parsed, including asynchronous output from `moments` or `design`. (This is different from the shell's interpretation of > and `|`.)

Examples:

```
list >file1
interp >file2
iv >file3
interp | mail njas
interp | sort +0nr +1nr +2nr | mail njas
```

8.12. `prompt`

`prompt` sets a prompt string that appears after each command has finished, similar to the regular shell `$` prompt. For example

```
prompt %
```

will print a % at the beginning of each input line.

`prompt` with no argument turns it off, which may be necessary in editing to avoid ruining the margins on the terminal.

The `prompt` character is prefixed by `m` or `d` when `design` or `moments` are running asynchronously.

8.13. `quit` or `q`

`quit` exits `gosset`. The working directory and its contents are left unchanged. The last program successfully compiled will be the `old` program when work is resumed in that directory.

8.14. Reading a file using < and <#

The `gosset` commands `<filename` and `<#filename` take input from a file named relative to the working directory.

<# puts a line number in front of each line read, starting with 10 more than the last line in the current program, and incrementing by 10. These lines will therefore be placed in the current program.

< indifferently takes input from the file as if it came from a terminal, so these lines

may be commands or program lines.

Commands which execute asynchronously, namely `moments` and `design`, must be followed by a `wait` command if they are invoked with `<`.

8.15. `renum`

`renum` renumbers the lines in the current program. For example the three commands

```
% renum
% renum 100
% renum 100 20
```

would renumber starting at 10 with an increment of 10, starting at 100 with an increment of 10, or starting at 100 with an increment of 20, respectively.

8.16. `search`

`search` is used to specify a library search path for the `lib` feature of the `program` option in the `design` command.

When invoked with no arguments, `search` lists the current search sequence, which by default is `../codelib.a`.

When invoked with arguments, the arguments are made the new search sequence. The most common use of this is

```
% search ../codelib.a lib.a
```

which will search both the standard library that comes with `gosset` and the local archive of working solutions.

`search` is reset by `cd` (see Sect. 8.1).

8.17. The shell escape `!`

A line beginning with `!` is passed to the shell for execution, preceded by `cd working-directory;`.

There are two severe restrictions.

First, one cannot access standard input, because `gosset` has closed it in favor of an `input daemon`, and nothing is attached to the shell. So one cannot run an editor this way.

Second, a kill-interrupt from the terminal won't kill just the `!` command but everything else as well. If the command does need to be killed, this must be done from another window via its `pid`.

8.18. `tag`

Each successfully compiled program is assigned a `tag`, a four-letter sequence starting at `aaaa`.

`tag` tells what the program tag of the current program would be if it were compiled.

If the program is identical to an earlier program, except for `start` lines, the `tag` will be the `tag` of the previous program. Otherwise, it is the first unused `tag`, starting with `aaaa`. Programs must be exactly identical (including blanks, line order and comments).

8.19. `version`

The command `version` tells which version of `gosset` you are using.

8.20. `wait`

`wait` causes no further input to be read from the terminal until `design` or `moments` finishes. It has no effect if neither is in progress.

This is useful when several commands are to be run in sequence, as in a batch job.

It is also useful if we wish to search for a design over lunch (say), for example with the commands

```
% compile
% moments n=999999999 time=30*60
% design extra=6 n=20
```

which call for a half hour to be spent computing the moment matrix and then 20 attempts to be made to find a design. We would type

```
% compile
% moments n=999999999 time=30*60
% wait
% design extra=6 n=20
```

Otherwise `design` would try to start just after `moments` started, and we would return to find the complaint `moments is running` on our terminal (and the `design` command not even begun).

8.21. `unwait`

`gosset` reads ahead whenever it's waiting at a `wait` command, to see if an `unwait` has appeared. If so, it skips everything up to the `unwait` command, stops waiting, and starts reading commands again from that point.

Whatever `gosset` was waiting for continues running.

8.22. Macros in `gosset`

It is possible to use the `m4` macro processor `UNIX®` command to define new `gosset` commands. For example(*):

```
$ m4 -e|gosset
define(go,‘design runs=$1 n=$2 processors=8 steps=50 type=I’)
old
```

Then

```
go(20,8)
```

is equivalent to

```
design runs=20 n=8 processors=8 steps=50 type=I
```

Multiple line input can be obtained by not closing the ′ in the `define` statement. Newlines are included up to the terminating ′.

(*) Note that in this font it is easy to confuse ‘ , a backward sloping quote (like a French grave accent), and ’ , a forward sloping quote (like an acute accent).

9. Installing gosset

System administrators please note: if you are installing `gosset` for multiple users, a slightly different `cc` command is needed — see Section 9.2 below.

9.1. Installing a private copy of gosset

We first choose a base directory for `gosset`. Subdirectories are created off this for work on individual problems. We change directory to the base directory, extract everything in `codemart.cpio` :

```
$ cd basedirectory
$ cpio -icu < codemart.cpio
```

and compile several files:

```
$ cc M*.c -o gosset -lm
rm */vtrace */vvv */moments */interp
```

(The latter command is needed only if you already had a copy of `gosset`. It removes some out-of-date files.) If you are installing `gosset` for multiple users a different `cc` command is required — see the next section. It is necessary to use the right path to `codemart.cpio`, of course, if it is not in the base directory. At Bell Labs a copy of this file is in

```
/n/alice/usr/rhh/codemart/codemart.cpio
```

If you are not at Bell Labs, we may have sent you a compressed file. This was probably created by typing

```
compress <codemart.cpio|uuencode stuff.Z | mail youraddress
```

When you receive this file, place it in the base directory, naming it (say) `temp`. You then type (in the base directory)

```
uudecode < temp
```

Alternatively, you can do this directly in `mail`:

```
mail | uudecode
```

Either of these magically produces a file called stuff.Z. You then type

```
zcat stuff.Z | cpio -icum
cc M*.c -o gosset -lm
rm */vtrace */vvv */moments */interp
```

To find a design, start by entering `gosset` :

```
$ gosset
```

The program responds by asking you to name a working subdirectory:

```
please type 'cd something' to name a local directory for your work
```

At this point you will probably enter the specifications for a new design — see Sects. 2, 3, ...

Incidentally, if you are at Bell Labs, the file

```
/n/alice/usr/rhh/codemart/changes
```

contains brief comments about recent updates to the program.

9.2. Installing `gosset` for multiple users

It is possible to keep `gosset` and the code library `codelib.a` anywhere, even on another machine if your operating system permits this to be expressed as a pathname.

Instead of the above, use for example

```
cc -DGOSSETSRC=\"/u/g/sources\"
        -DCODELIB=\"/u/g/codes\" M*.c -lm -o gosset
```

which will create a version of `gosset` that expects the source files to be in `/u/g/sources` and the code library to be in `/u/g/codes`. The escapes are necessary on the quotes, because they are part of the definition.

Then the file `gosset` can be moved anywhere at all, say into `/usr/local/bin` and be executed from anywhere. The source files should be moved to the specified place using(*)

```
mv ‘cat files‘ /u/g/sources
```

Be warned, however: working directories will be created by `gosset` wherever the user is, instead of from a single base directory. So the user must exercise some discipline when invoking `gosset`, to avoid chaos in the directory structure.

It is useful to remember that `gosset` working directories can be located by looking for a file named `esse.program`:

```
find . -name esse.program -print
```

10. The built-in libraries of designs

The file `codelib.a`, included with `gosset`, contains a large library of designs, most of which we have computed using `gosset` on a variety of high-speed machines during the past two years. Usually these are the best of several thousand attempts, and appear to be optimal or very close to optimal. To save space this file can be deleted if it is not required. Its contents are described in Sect. 10.1.

There is also a small but growing library of "classical" designs, `classic.a`, which is described in Sect. 10.4.

10.1. Designs in `codelib.a`

At the present time `codelib.a` contains designs of the following kinds.

Linear or main-effect designs. Designs for k variables and n runs with a linear model

$$\beta_0 + \sum_{i=1}^{k} \beta_i x_i ,$$

containing $k+1$ unknown coefficients. When k is 1 less than a multiple of 4 these are Plackett-Burman-Rao (or Hadamard) designs.

Quadratic or second-order designs. Designs for k variables and n runs with a full quadratic model

$$\beta_0 + \sum_{i=1}^{k} \beta_i x_i + \sum_{i=1}^{k} \sum_{j=i}^{k} \beta_{ij} x_i x_j ,$$

(*) Note that in this font it is easy to confuse ‘ , a backward sloping quote (like a French grave containing $(k+1)(k+2)/2$ unknown coefficients.

Interaction designs. Designs for k discrete variables taking two levels -1 and $+1$, with n runs, for the main-effects plus interactions model

$$\beta_0 + \sum_{i=1}^{k} \beta_i x_i + \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \beta_{ij} x_i x_j ,$$

containing $1 + k(k+1)/2$ unknown coefficients. The design shown in Sect. 7.5 is of this type.

Cubic or third-order designs. Designs for k variables and n runs with a full cubic model.

Mixture designs. Designs for k variables adding to 1 and n runs, with a linear or quadratic model. (See Sect. 7.8 for an example.)

Orthogonal arrays. Orthogonal arrays $OA(M, n, q, t)$, with M runs, n variables, q levels and strength t. (Section 7.22 discusses the use of `gosset` to construct orthogonal arrays.)

Designs to protect against the loss of a run. Linear designs for cube, sphere and simplex of `type=J`. (See Sect. 4.8.)

Remarks.

(1) We have discovered a closed form for optimal linear designs in the sphere of types A, D, I and J — see [HS9]. Since these designs would take up a lot of space in `codelib.a`, only token examples have been placed there.

However, a program called `optlinsphere.c`, included with `gosset`, can be compiled and run to generate any optimal linear sphere code as needed. (These designs could even be put into `codelib.a` using the `ar` command.)

For example, to get the optimal design for 5 dimensions and 7 runs, we first compile `optlinsphere.c`, if that hasn't been done already, in the base directory:

```
$ cc optlinsphere.c -lm -o optlinsphere
```

Then in `gosset`, in the working directory, we set up the 5-dimensional linear sphere program, say from executing `!../genlib.sh Ls.5`:

```
10 sphere a1 a2 a3 a4 a5
20 model 1+a1+a2+a3+a4+a5
30 comment optimize type=i
40 comment Available in closed form!
50 comment See manual, optlinsphere.c
```

We can then generate and interpret the design, calling it `v.5.7.opt`, say:

accent), and `'` , a forward sloping quote (like an acute accent).

```
!../optlinsphere v.5.7.opt
interp v.5.7.opt

x/v.5.7.opt
  a1      a2      a3      a4      a5

-0.7071 -0.7071  0.0000  0.0000  0.0000
 0.9659 -0.2588  0.0000  0.0000  0.0000
-0.2588  0.9659  0.0000  0.0000  0.0000
 0.0000  0.0000 -0.5774 -0.5774 -0.5774
 0.0000  0.0000  0.9623 -0.1925 -0.1925
 0.0000  0.0000 -0.1925  0.9623 -0.1925
 0.0000  0.0000 -0.1925 -0.1925  0.9623

iv v.5.7.opt
         x/v.5.7.opt       0.654761902840
```

This is to be compared with the result of running `design runs=7 n=10`:

```
interp v.7.best

x/v.7.best
  a1      a2      a3      a4      a5

 0.5409 -0.0217 -0.6122 -0.5745  0.0462
 0.4292  0.4143  0.7006 -0.3688 -0.1314
 0.1968 -0.5408 -0.3381  0.5235 -0.5296
-0.6260  0.1265 -0.3625 -0.1547  0.6609
-0.6622  0.2510  0.0459 -0.1569 -0.6868
-0.0949 -0.7537  0.5490 -0.0378  0.3466
 0.2162  0.5244  0.0174  0.7691  0.2940

iv v.7.best
          x/v.7.best       0.654761892311
```

whose `iv` value differs only in roundoff, and which does not have such a pretty orientation.

(2) For $k = 1$ of course the sphere and cube coincide. For large values of k the sphere seems a better region to use than the cube, in view of the fact that the vertices of a high-dimensional cube are much further from the center than mid-points of the faces (or, to put it another way, high-dimensional cubes are "star-shaped" and are probably not what the experimenter has in mind).

(3) The geometrical structure of many of these designs is discussed in [HS1], [HS2], [HS4], [HS7], and a fairly complete listing of them is given in [HS5].

The following subsections give a complete list of the parameters of the designs in `codelib.a` at the present time.

For each family of designs the heading specifies the region where the points lie; whether the design is continuous or discrete; the model; the optimality criterion used; and the prefix used to identify these designs in the library. The first few lines in each subsection give a sample `gosset` program corresponding to one of these designs. (These were produced by `genlib.sh` — see Sect. 10.3.)

Then follows a list of the values of the geometrical dimensions of the designs and the numbers of runs. The geometrical dimension is usually the same as the number of variables, except for mixtures when it is one less.

10.1.1. Sphere; continuous; linear; I-optimal; prefix=`Ls`

```
10 sphere a1 a2 a3
20 model 1+a1+a2+a3
30 comment optimize type=i
40 comment Available in closed form!
50 comment See manual, optlinsphere.c
```

```
dim=2 runs=3
dim=3 runs=4
dim=4 runs=5
dim=5 runs=6
dim=6 runs=7
dim=7 runs=8
dim=8 runs=9
dim=9 runs=10
dim=10 runs=11

dim=11 runs=12
dim=12 runs=13
dim=13 runs=14
dim=14 runs=15
```

10.1.2. Cube; continuous; linear; I-optimal; prefix=`Lc`

```
10 range a1 a2 a3
20 model 1+a1+a2+a3
30 comment optimize type=i
```

```
dim=1 runs=2 - 18 20 24 28 32
dim=2 runs=3 - 20 24 28 32
dim=3 runs=4 - 20 24 28 32
dim=4 runs=5 - 21 24 28 32
dim=5 runs=6 - 22 24 28 32
dim=6 runs=7 - 24 28 32
dim=7 runs=8 - 24 28 32
dim=8 runs=9 - 25 28 32
dim=9 runs=10 - 26 28 32
dim=10 runs=11 - 28 32
```

```
dim=11 runs=12 - 28 32
dim=12 runs=13 - 29 32
dim=13 runs=14 - 30 32
dim=14 runs=15 - 32
dim=15 runs=16
dim=16 runs=17
dim=17 runs=18
dim=18 runs=19
dim=19 runs=20
dim=20 runs=21
```

10.1.3. Cube; discrete; linear; D-optimal; prefix=`E`

For `dim` of the form $4a - 1$ these are Plackett-Burman designs (or Hadamard matrices).

```
10 discrete a1 a2 a3
20 model 1+a1+a2+a3
30 comment optimize type=d
```

```
dim=2 runs=3
dim=3 runs=4
dim=4 runs=5
dim=5 runs=6
dim=6 runs=7
dim=7 runs=8
dim=8 runs=9
dim=9 runs=10
dim=10 runs=11

dim=11 runs=12
dim=12 runs=13 - 23
dim=13 runs=14 - 24
dim=14 runs=15 - 25
dim=15 runs=16
dim=16 runs=17
dim=17 runs=18
dim=18 runs=19
dim=19 runs=20
dim=20 runs=21

dim=23 runs=24
dim=27 runs=28
dim=31 runs=32
```

All five Hadamard matrices of order 16 are included. Their names are `E.15.16.d1` through `E.15.16.d5`.

10.1.4. Simplex; continuous; linear; I-optimal; prefix=`Lm`

```
10 range 0 1 a1 a2 a3 a4
20 model 1+a1+a2+a3+a4
30 constraint a1+a2+a3+a4=1
40 comment optimize type=i
50 comment Optimum just cycles vertices.
```

```
dim=1 runs=2 - 10
dim=2 runs=3 - 11
dim=3 runs=4 - 12
dim=4 runs=5 - 13
dim=5 runs=6 - 14
dim=6 runs=7 - 15
dim=7 runs=8 - 16
dim=8 runs=9 - 17
dim=9 runs=10 - 18
dim=10 runs=11 - 19

dim=11 runs=12 - 20
dim=12 runs=13 - 21
dim=13 runs=14 - 22
```

10.1.5. Cube; discrete; interaction; I-optimal; prefix=`d`

```
10 discrete a1 a2 a3
20 model (1+a1+a2+a3)^2-a1^2-a2^2-a3^2
30 comment optimize type=i
```

```
dim=2 runs=4 5 6 7 8
dim=3 runs=7 8 9 10 11
dim=4 runs=11 12 13 14 15
dim=5 runs=16 17 18 19 20
dim=6 runs=22 23 24 25 26
dim=7 runs=29 30 31 32 33
dim=8 runs=37 38 39 40 41
dim=9 runs=46 47 48 49 50
dim=11 runs=67 68 69 70 71
dim=12 runs=79 80 81 82 83
```

10.1.6. Sphere; continuous; quadratic; I-optimal; prefix=`s`

```
10 sphere a1 a2 a3
20 model (1+a1+a2+a3)^2
30 comment optimize type=i
```

```
dim=2 runs=6 - 24
dim=3 runs=10 - 36
dim=4 runs=15 - 31
dim=5 runs=21 - 33 41 42
dim=6 runs=28 - 44
dim=7 runs=36 - 56
dim=8 runs=45 - 72
dim=9 runs=55 - 63
dim=10 runs=66 - 75

dim=11 runs=78 - 87
dim=12 runs=91 - 100
dim=13 runs=105 - 114
dim=14 runs=120 - 128
```

For `dim=1`, use `prefix=c`.

10.1.7. Sphere; continuous; quadratic; D-optimal; prefix=`Ds`

```
10 sphere a1 a2 a3
20 model (1+a1+a2+a3)^2
30 comment optimize type=d

     dim=3 runs=10 - 23 30 40 50
     dim=4 runs=15 - 20 25 35 45 55 75 95 115
```

10.1.8. Cube; continuous; quadratic; I-optimal; prefix=`C`

```
10 range a1 a2 a3
20 model (1+a1+a2+a3)^2
30 comment optimize type=i

     dim=1 runs=3 - 30
     dim=2 runs=6 - 40
     dim=3 runs=10 - 30 50
     dim=4 runs=15 - 22 50
     dim=5 runs=21 - 29 31 50
     dim=6 runs=28 - 36 42 50
     dim=7 runs=36 - 44 50 54
     dim=8 runs=45 - 53 67
     dim=9 runs=55 - 63
     dim=10 runs=66 - 74

     dim=11 runs=78 - 86
     dim=12 runs=91 - 99
     dim=13 runs=105 - 113
     dim=14 runs=120 - 128
```

10.1.9. Cube; continuous; quadratic; D-optimal; prefix=`Dc`

```
10 range a1 a2 a3
20 model (1+a1+a2+a3)^2
30 comment optimize type=d

     dim=1 runs=3 - 30
         dim=2 runs=6 - 40
     dim=3 runs=10 - 30 40 50 60 80
     dim=4 runs=15 - 23 25 35 45 50 65 85
```

10.1.10. Cube; discrete; quadratic; I-optimal; prefix=`t`

```
10 discrete -1 0 1 a1 a2 a3
20 range a1' a2' a3'
30 model (1+a1+a2+a3)^2
40 comment optimize type=i

     dim=1 runs=3 - 30
     dim=2 runs=6 - 40
     dim=3 runs=10 - 30 50
     dim=4 runs=15 16 17 18 19 20 21 22 23 50
     dim=5 runs=21 22 23 24 25 26 27 28 29 31 50
     dim=6 runs=28 29 30 31 32 33 34 35 36 42 50
     dim=7 runs=36 37 38 39 40 41 42 43 44 50 54
     dim=8 runs=45 46 47 48 49 50 51 52 53 67
     dim=9 runs=55 56 57 58 59 60 61 62 63
     dim=10 runs=66 67 68 69 70 71 72 73 74

     dim=11 runs=78 79 80 81 82 83 84 85 86
     dim=12 runs=91 92 93 94 95 96 97 98 99
     dim=13 runs=105 106 107 108 109 110 111 112 113
     dim=14 runs=120 121 122 123 124 125 126 127 128
```

10.1.11. Cube; discrete; quadratic; D-optimal; prefix=`Dt`

```
10 discrete -1 0 1 a1 a2 a3
20 model (1+a1+a2+a3)^2
30 comment optimize type=d

     dim=1 runs=3 - 30
         dim=2 runs=6 - 40
         dim=3 runs=10 - 30 50
         dim=4 runs=15 - 23 50
         dim=5 runs=21 22
```

10.1.12. Sphere; continuous; cubic; I-optimal; prefix=`s3`

```
10 sphere a1 a2 a3
20 model (1+a1+a2+a3)^3
30 comment optimize type=i
```

```
dim=2 runs=10 - 30
dim=3 runs=20 - 52
dim=4 runs=35 - 67
dim=5 runs=56 - 88
```

10.1.13. Cube; continuous; cubic; I-optimal; prefix=`c3`

```
10 range a1 a2 a3
20 model (1+a1+a2+a3)^3
30 comment optimize type=i
```

```
dim=1 runs=4 - 20
dim=2 runs=10 - 30
dim=3 runs=20 - 32
dim=4 runs=35 - 43
```

10.1.14. Simplex; continuous; quadratic; I-optimal; prefix=`M`

```
10 range 0 1 a1 a2 a3 a4
20 model (1+a1+a2+a3+a4)^2
30 constraint a1+a2+a3+a4=1
40 comment optimize type=i
```

```
dim=1 runs=3 - 11
dim=2 runs=6 - 22
dim=3 runs=10 - 26
dim=4 runs=15 - 31
dim=5 runs=21 - 29
dim=6 runs=28 - 36
dim=7 runs=36 - 44
dim=8 runs=45 - 54
dim=9 runs=55 - 63 65
dim=10 runs=66 - 74 77

dim=11 runs=78 - 86 90
dim=12 runs=91 - 99 104
dim=13 runs=105 - 113 119
```

10.1.15. Orthogonal arrays

M	n	q	t
9	4	3	2
100	4	10	2
16	5	4	2
25	6	5	2
144	7	12	2
49	8	7	2
64	9	8	2
81	10	9	2
121	12	11	2
169	14	13	2
256	17	16	2

10.1.16. Sphere; continuous; linear; J-optimal; prefix=`Lsj`

```
10 sphere a1 a2 a3
20 model 1+a1+a2+a3
30 comment optimize type=j
40 comment Available in closed form!
50 comment See manual, optlinsphere.c
```

```
dim=2 runs=5
dim=3 runs=6
dim=4 runs=7
dim=5 runs=8
dim=6 runs=9
dim=7 runs=10
dim=8 runs=11
dim=9 runs=12
dim=10 runs=13

dim=11 runs=14
dim=12 runs=15
dim=13 runs=16
dim=14 runs=17
```

10.1.17. Cube; continuous; linear; J-optimal; prefix=`Lcj`

```
10 range a1 a2 a3
20 model 1+a1+a2+a3
30 comment optimize type=j
```

```
dim=1 runs=3 - 18 20 24 28 32
dim=2 runs=4 - 20 24 28 32
dim=3 runs=5 - 20 24 28 32
dim=4 runs=6 - 21 24 28 32
dim=5 runs=7 - 22 24 28 32
dim=6 runs=8 - 24 28 32
dim=7 runs=9 - 24 28 32
dim=8 runs=10 - 25 28 32
dim=9 runs=11 - 26 28 32
dim=10 runs=12 - 28 32

dim=11 runs=13 - 28 32
dim=12 runs=14 - 29 32
dim=13 runs=15 - 30 32
dim=14 runs=16 - 32
```

10.1.18. Simplex; continuous; linear; J-optimal; prefix=`Lmj`

```
10 range 0 1 a1 a2 a3 a4
20 model 1+a1+a2+a3+a4
30 constraint a1+a2+a3+a4=1
40 comment optimize type=j
```

```
dim=2 runs=4 - 35
dim=3 runs=5 - 36
dim=4 runs=6 - 37
dim=5 runs=7 - 38
dim=6 runs=8 - 39
dim=7 runs=9 - 40
dim=8 runs=10 - 41
dim=9 runs=11 - 42
dim=10 runs=12 - 43

dim=11 runs=13 - 44
dim=12 runs=14 - 45
dim=13 runs=15 - 46
```

10.2. Accessing the libraries

The libraries `codelib.a`, `classic.a` (see Sect. 10.4) and `lib.a` are archive files, and can be accessed in several ways.

(i) With the `ar` shell command. For example,

```
$ cd testdir                    [change to working directory]
$ ar t lib.a                    [list table of contents of lib.a]
$ ar p lib.a v.3.14.aaaa        [print v.3.14.aaaa from lib.a]
$ ar d lib.a v.3.14.aaaa        [delete v.3.14.aaaa from lib.a]
$ ar r lib.a V.3.14.NICE        [add V.3.14.NICE to lib.a]
```

(ii) With

```
search ../codelib.a    [the default, so not strictly necessary]
design runs=... program=lib
```

to search the library for an appropriate design, as illustrated in Example 7.2.

(iii) With `check ../codelib.a | grep -v BAD`, which checks to see which designs in `codelib.a` satisfy the conditions of the current program (see Sect. 8.2).

(iv) With

```
list ../codelib.a(c.2.6.1002)
```

or

```
list ../codelib.a(c.2.6.*)
```

(see Sect. 8.9), which produces a listing of the design `c.2.6.1002`.

(v) With

```
get ../codelib.a(c.2.6.1002)
```

or

```
get ../codelib.a(c.2.6.*)
```

(see Sect. 8.7), which puts a copy of the design `c.2.6.1002` in the current working directory.

To add designs to a library, use the shell `ar` command (see (i) and Sect. 8.7). The authors would also like to hear about other designs that should be added to `codelib.a` or `classic.a` (use the electronic mail address given at the end of Sect 1).

The `codelib.a` designs are each the best of many attempts; the number of attempts is the final number in the design name.

This important piece of information is also a nuisance, in that it makes the exact filename unknown in advance. One way to find the filename of, say, the 3-dimensional 10-point cubical design, `c.3.10.something`, is

```
list ../codelib.a | grep c.3.10
rw-r--r--1089/0    159 Aug  1 22:27 1991 c.3.10.2049
```

which responds with the archive index entries that match the pattern. Evidently this code is the best of 2049 attempts.

Alternatively, one can use an asterisk in the filename, to match any string of characters (see Sect. 5.4).

10.3. The `genlib.sh` command to explain a `codelib.a` design

The shell script `genlib.sh` (called from the main directory, not from inside `gosset`), generates a `gosset` program corresponding to any prefix appearing in the design library `codelib.a`. For example,

```
$ genlib.sh d.4
10 discrete a1 a2 a3 a4
20 model (1+a1+a2+a3+a4)^2-a1^2-a2^2-a3^2-a4^2
30 comment optimize type=i
```

tells us that the designs beginning with `d.4` are conjecturally I-optimal two-level full interaction designs with four variables.

10.4. A library of classical designs

The archive file `classic.a` included with `gosset` contains a small library of "classical" designs from the literature. At the present time it includes the following designs.

In the case of quadratic designs that lie on the surface of the sphere, only one copy of the center point has been included (so as to make the design nonsingular). We expect that you will add more copies of the center point yourself, possibly using one of our designs as a guide.

`CL.3.13.ICOS`. Center and 12 vertices of regular icosahedron.

`CL.3.15.CC`. Central composite design, consisting of center and 14 vertices of regular rhombic dodecahedron.

`CL.4.25.CC`. Central composite design, consisting of center and 24 vertices of regular 24-cell.

`CL.3.13.USD`. Uniform shell design ([Do70], DK72]) consisting of center and 12 vertices of regular cuboctahedron.

`CL.4.21.USD`. Uniform shell design ([Do70], DK72]) consisting of center and 20 minimal vectors of A_4 lattice.

`CL.3.10.ROQ`. Roquemore [Ro76] design 310 consisting of 10 points in a 3-dimensional sphere (normalized to have radius 1).

`CL.3.11.ROQA` and `CL.3.11.ROQB`. Roquemore [Ro76] designs 311A and 311B consisting of 11 points in a 3-dimensional sphere (normalized to have radius 1).

`CL.4.16.ROQA`, `CL.4.16.ROQB`, `CL.4.16.ROQC`. Roquemore [Ro76] designs 416A, 416B, 416C consisting of 16 points in a 4-dimensional sphere (normalized to have radius 1).

Roquemore's design 628A, consisting of a center point plus the well-known 27-point spherical 4-design associated with the Schlafli polytope, appears in `codelib.a` as `s.6.28.14`.

`CL.2.4.FACT`, `CL.3.8.FACT`, `CL.4.16.FACT`, `CL.5.32.FACT`, `CL.6.64.FACT`, `CL.7.128.FACT`, `CL.8.256.FACT`, full factorial designs consisting of the 2^d vertices of the d-dimensional cube.

`CL.2.6.NOTZ`. Notz [No82] design consisting of 6 points in a square.

`CL.5.21.NOTZ`. Notz [No82] design consisting of 21 points in a 5-dimensional cube.

Plackett-Burman designs are described elsewhere in the manual (see the index).

11. Error messages

The following are some of the commonest error messages.

11.1. Spaces inside an expression

```
10 range x y z
20 constraint x + y + z < 1
30 model (x+y+z+1)^2
compile
20 constraint x + y + z < 1
ERROR   bad expression +
```

Explanation: the spaces in `x + y + z` have led the compiler to think that + is a separate expression. Solution: change line 20 and recompile:

```
20 constraint x+y+z<1
compile
```

11.2. Omission of multiplication signs

```
10 range 0 10 w x y z
20 constraint w+2x+3y+4z<=5
30 model (w+x+y+z+1)^2
compile
20 constraint w+2x+3y+4z<=5
ERROR   bad expression w+2x+3y+4z<=5
```

Explanation: multiplication signs (*'s) were omitted. Solution: change line 20 and recompile:

```
20 constraint w+2*x+3*y+4*z<=5
compile
```

11.3. Forgetting to specify a model

```
10 range w x y z
20 constraint w+x+y+z=1
compile
ERROR   no model declared
```

Explanation: no model was specified. Solution: specify a model and recompile:

```
30 model (w+x+y+z+1)^2
compile
```

11.4. A spelling error

```
10 range 0 10 w x y z
20 constraint w+2x+3y+4z<=5
30 model (w+x+y+z+1)^2
complie
 'complie' invalid command
```

Solution: type the correct command:

```
compile
```

or (safer!) use the abbreviation:

```
c
```

11.5. Not enough runs in design

```
10 discrete Mp 32 42 52
20 range Mp' 32 52
30 discrete Tp 3 5 7
40 range Tp' 3 7
50 discrete Tf 80 100 120
60 range Tf' 80 120
70 discrete Na .8 1. 1.2
80 range Na' .8 1.2
90 discrete Ts 150 250 350
100 range Ts' 150 350
110 model (1+Mp+Tp+Tf+Na+Ts)^2

compile
— —
21 functions in model
compile done
moments
— —
moments done
design runs=12 n=10
— —
1/10 finish design of testdir/v.12.0, IV=-0.500000000000
error was:
can't find nonsingular random start after 100 iterations
```

Explanation: since there are 21 terms in the model, the smallest possible design has 21 runs. Solution: kill this job and look for a larger design:

```
kill design
aborting designs
— —
design done
design runs=24 n=10
— —
```

11.6. Experiment region too small

The following example illustrates what happens when the experiment region does not contain enough space to build a design of the required type:

```
10 discrete 0 15 20 substrate
20 discrete 0 10 20 additive
30 range 0 20 additive'
40 range treatment
50 discrete coating
60 range speed
70 constraint substrate+additive<21
80 constraint substrate+treatment<19
90 model (substrate+additive+treatment+coating+speed+substrate^2+1)^2
   -substrate*additive-substrate^2*additive-coating^2
   -substrate^3-substrate^4

compile
— —
moments
— —
design extra=3 n=10
— —
1/10 finish design of testdir/v.25.0, IV=-0.500000000000
error was:
can't find nonsingular random start after 100 iterations
```

Explanation: the constraints are too restrictive and no design of the specified type is possible. Solution: enlarge the design space by relaxing a constraint:

```
80 constraint substrate+treatment<20
compile
```

11.7. Attempting to call an editor in `gosset`

```
vi program.log
       'vi program.log' invalid command
```

Explanation: to issue a shell command, it is necessary to prefix it with "`!`". But even this will not work, as we now see.

11.8. Attempting to call an editor from `gosset` via a shell escape

```
!vi program.log
        cd testdir; vi program.log
[Using open mode]
Open and visual must be used interactively
!
```

Explanation: for reasons explained in Sect. 8.17, it is not possible to call an editor from inside `gosset`, even with a shell escape. Solution: call `vi` from another window; or leave `gosset`, edit `program.log`, and return to `gosset`.

11.9. An error produced by a faulty C optimizer

On some SGI machines the optimizer option of the *C* compiler causes the `design type=D` program to malfunction. (The search exits immediately instead of improving the solution.) This can cause very serious errors when trying to find D-optimal designs.

The following example illustrates this behavior. This example can also be used as a test case in case you suspect a similar problem has arisen with your computer. Or you can try setting `cflags none`, as explained below, rerunning the job, and seeing if you get a dramatic improvement in the *D*-value of the best design. If so, the optimizer of your C-compiler is suspect.

(The code for D-optimizing uses explicit two-dimensional arrays, which seem to offer the C compiler increased opportunity for error; the other types use internal one-dimensional arrays and are not subject to this particular error.)

```
10 range b 0 .4
20 range c 0 .5
30 range a .1 .75
40 constraint a+b+c=1
50 constraint a+b>.5
60 constraint a+b<1
70 model (1+a+b+c)^2

compile
— —
moments
— —
design type=D runs=9 n=24
— —

interp

#1:a+b+c=1

bug/d.9.best
 a      b      c      #1

0.7132 0.0038 0.2830 0.1133
0.6935 0.2808 0.0257 0.1739
0.6553 0.0566 0.2881 0.2913
0.5474 0.1826 0.2700 0.6234
0.5164 0.0410 0.4426 0.7186
0.4248 0.3207 0.2544 1.0006
0.3974 0.3995 0.2031 1.0850
0.3861 0.2554 0.3585 1.1197
0.1992 0.3925 0.4083 1.6948

dv
        bug/d.9.best       0.584133240148
```

This is not a minimum. One clue is that there are no points on the boundary of the region, which is extremely unlikely in any type of optimal design!

Explanation: the `-O` optimizer option in the `C` compiler has produced faulty code. Solution: turn off the compiler optimization by typing

```
cflags none
```

(see Sect. 8.5), before typing `compile`, and rerun the job.

For reference, the following is a conjecturally D-optimal design for this problem.

```
interp

#1:a+b+c=1

bug/d.9.best
 a      b      c       #1

0.7500 0.2498 0.0002  0.0000
0.7500 0.0000 0.2500  0.0000
0.6000 0.4000 0.0000  0.4615
0.5000 0.0000 0.5000  0.7692
0.4923 0.2256 0.2822  0.7931
0.3774 0.4000 0.2226  1.1464
0.3214 0.1786 0.5000  1.3188
0.1000 0.4000 0.5000  2.0000
0.1000 0.4000 0.5000  2.0000

dv
        bug/d.9.best        0.293685297348
```

We have had very little trouble with compilers, but such errors occasionally still occur. See Sect. 8.5 for some recommendations.

12. Optimality criteria

The theory underlying this program is described in [HS4]. `gosset` uses several different optimality criteria (see Sect. 4).

I-optimal designs. For this type of problem the program attempts to find a design with the smallest possible value of the *average* or *integrated prediction variance IV*, defined as follows.

Suppose the variables in the model are $x_1, \ldots, x_k$, and the response surface is

$$y = \sum_{(i_1, \ldots, i_k) \in S} \beta_{i_1 \ldots i_k} \, x_1^{i_1} \cdots x_k^{i_k} + \varepsilon \,, \tag{1}$$

where S is a set of p k-tuples and ε is an error term. Suppose also that the design consists of n points

$$x = (x_{1j}, \ldots, x_{kj}) \,,$$

where $j = 1, \ldots, n$. Let X be the corresponding expanded design matrix, of size $n \times p$, with a row for each design point. The row of X corresponding to the design point x is

$$f(x) = (\quad \ldots \quad , \; x_{1j}^{i_1} \cdots x_{kj}^{i_k} \; , \ldots \;) \, , \tag{2}$$

containing p terms, one for each k-tuple in S. The prediction variance at an arbitrary point x is given by the formula

$$\text{var } \hat{y}(x) = \sigma^2 f(x)(X'X)^{-1} f(x)' \, ,$$

if the errors ε are independent with mean 0 and variance σ^2 (the prime indicating transposition). The average prediction variance (Box and Draper [BD59], [BD63]) is

$$IV = \int \frac{1}{\sigma^2} \text{ var } \hat{y}(x) \, d\mu(x) \, ,$$

where the integral is over the region in which the response surface is to be fitted, and μ is uniform measure on this region. In other words, IV is the expected prediction variance, given that the sample prediction variance is 1. IV could also be called the *I-efficiency* of the design. Then

$$IV = \int f(x)(X'X)^{-1} f(x)' d\mu(x) = \text{trace } \{M(X'X)^{-1}\} \, ,$$

where

$$M = \int f(x)' f(x) \, d\mu(x)$$

is the moment matrix of the same region. The `moments` command evaluates M, and the `design` command then attempts to minimize IV. The `iv` command evaluates IV for a particular design.

The `iv`- (or IV-) value of a design is related to the I-value as defined in [HS4] by the formula

$$I - \text{value (of [HS4])} = \text{runs} \cdot iv$$

If only continuous variables appear in the design then the program minimizes the value of IV using the *pattern search* optimization strategy of Hooke and Jeeves [HJ61], as described by Beightler et al. [BPW79] (see Sect. 4.4 above). Discrete variables are handled as described in Sect. 4.5 and 4.6.

A-optimal designs. The program attempts to minimize the A-value of the design, which is

$$\text{trace } \{(X'X)^{-1}\} \, .$$

(This quantity, like the D-value, does not mention the moment matrix.) The `av` command calculates the A-value of a design.

The `av`- (or A-) value of a design is related to the A-value as defined in [HS4] by the formula

$$A - \text{value (of [HS4])} = \text{runs} \cdot av$$

D-optimal designs. The program attempts to minimize

$$\det \{(X' X)^{-1}\} ,$$

an expression which does not mention the moment matrix. (This is one reason why in general we prefer I-optimality to D-optimality.) The `dv` command calculates the *D*-value of a design, which we define to be

$$\{ \det X' X\}^{-1/n} ,$$

where n is the number of runs.

The `dv`- (or *D*-) value of a design is related to the *D*-value as defined in [HS4] by the formula

$$D - \text{value (of [HS4])} = \text{runs} \cdot dv$$

E-optimal designs. The program attempts to minimize the *E*-value of the design, which we define to be the

$$\text{largest eigenvalue of } (X' X)^{-1} .$$

(This also does not mention the moment matrix.) The `ev` command calculates the *E*-value of a design.

E-optimal designs are found by minimizing trace $(X' X)^{-p}$, where p is an increasingly large power. p starts at a number that is usually 20 by default, and doubles at every convergence. Other ranges can be specified as with `type=J` designs — see Sect. 4.8 — by specifying one or two numbers for the range of exponents, as in

```
design type=E program=r*e20,400
```

which limits the doubling from 20 to 400. Every number is internally reduced to the next lower power of two.

Saying `design type=E program=r*e1` finds an A-optimal design, but then evaluates it by the largest eigenvalue rather than the A-value.

Warning: E-optimality is very new to `gosset`, and acts quite strangely; it seems (like A-optimality) to be strongly affected by the coordinate system, and also often requires a small value of `steps=` to speed convergence. The idea of minimizing the trace of a high power of $X' X$ may be misguided in the first place. Sometimes starting with a low power works better, sometimes a higher power. Sometimes there are local minima with respect to p, which end the p search sequence. Use of the `trace` command (see Sect. 6.5) will show what happened for the last design. E-optimality is also noticeably slower than I-, A-, or D-optimality.

The final E-value given for the design is the actual largest eigenvalue, not an approximation to it.

Block designs. The optimality criteria are given in Eq. (2) of Sect. 4.10.

Correlated sample errors. The optimality criteria are given in Eq. (1) of Sect. 4.9.

B-optimal designs. The program attempts to minimize the *B*-value of the design, which we define to be the largest *A*-value of any of the designs formed by omitting one run from the original design. (See Sect. 4.8.)

C-optimal designs. The program attempts to minimize the *C*-value of the design, which we define to be the largest *D*-value of any of the designs formed by omitting one run from the original design. (See Sect. 4.8.)

F-optimal designs. The program attempts to minimize the *F*-value of the design, which we define to be the largest *E*-value of any of the designs formed by omitting one run from the original design. (See Sect. 4.8.)

J-optimal designs. The program attempts to minimize the *J*-value of the design, which we define to be the largest *IV*-value of any of the designs formed by omitting one run from the original design. (See Sect. 4.8.)

Packings. The program attempts to minimize the *P*-value of the design, which we define to be the reciprocal of the minimal distance between design points. (See Sect. 4.11.)

13. Acknowledgements

During the course of this work we have benefited from discussions with many of our colleagues at Bell Labs. In particular we should like to thank Rob Calderbank, Anne Freeny, Colin Mallows, Vijay Nair and Daryl Pregibon for their advice. Anne Freeny also provided us with many valuable comments on this manual. Several of the examples were run at the suggestion of Dave Doehlert of The Experiment Strategies Foundation in Seattle. We are especially grateful to Dave, since it was his letter to one of us in October 1990 that got us started on this work. We also thank Art Owen of Stanford University for discussions on using `gosset` to construct orthogonal arrays.

14. Appendix: list of `gosset` commands and keywords

Name	Meaning	Section
`acheck`	check if design satisfies current conditions, print *A*-value	8.2
`all`	`list all` lists all saved programs	8.9
`av`	compute *A*-value of design	5.3, 12
`bcheck`	check if design satisfies current conditions, print *B*-value	8.2
`bv`	compute *B*-value of design	5.3, 12
`ccheck`	check if design satisfies current conditions, print *C*-value	8.2
`cd`	`gosset` version of change directory command	8.1
`center`	center of sphere	3.2
`cflags`	report or change flags for `C` compiler	8.5
`check`	check if design satisfies current conditions, print *IV*-value	8.2
`cleanup`	remove unwanted files	8.3
`comment`	comment lines	8.4
`compile` or `c`	compile current program	4.2
`compiler`	change `C` compiler	8.5
`constraint`	specify constraints	3.6
`dcheck`	check if design satisfies current conditions, print *D*-value	8.2
`delete`	delete lines from program	8.6
`design` or `d`	search for design	4.4
`dif`	`list dif` prints lines not in previous program	8.9
`discrete`	declare discrete variables	3.4
`dv`	compute *D*-value of design	5.3, 12
`echeck`	check if design satisfies current conditions, print *E*-value	8.2
`echo`	display file names	5.4
`echo`	`gosset` option to echo input lines	3.1
`ev`	compute *E*-value of design	5.3, 12

`examine`	examine design being optimized	6.4
`extra`	number of runs above minimal number	4.4
`fcheck`	check if design satisfies current conditions, print *F*-value	8.2
`fieldmin`	minimum field width in `interp` output	5.2
`genlib.sh`	produce `gosset` program to explain design in `codelib.a`	10.3
`get`	fetch file from archive	8.7
`gosset`	call the main program	3.1
`icheck`	check if design satisfies current conditions, print *IV*-value	8.2
`interp`	interpret design	5.2
`invert`	`interp` option to take input from file	5.2
`iv`	compute average prediction variance *IV* of design	5.3, 12
`jcheck`	check if design satisfies current conditions, print *J*-value	8.2
`jv`	compute *J*-value of design	5.3, 12
`kill`	kill `moments` or `design`	8.8
`list` or `l`	give a listing of something	8.9
`m4`	define macro	8.22
`misc`	define constants	3.5
`model`	specify model	3.7
`moments` or `m`	compute moment matrix	4.3
`old`	load an old program	8.10
`pcheck`	check if design satisfies current conditions, print *P*-value	8.2
`precision`	number of digits in `interp` output	5.2
`processors`	number of processors to be used	4.4
`program`	specify optimization strategy for search	4.4
`prompt`	set prompt string	8.12
`pv`	compute *P*-value of design	5.3, 12
`quit` or `q`	exit from `gosset`	8.13
`radius`	radius of sphere	3.2
`range`	declare cubical variables	3.3
`renum`	renumber lines in program	8.15
`runs`	number of runs in design	4.4
`search`	specify search libraries	8.16
`set`	name a set of variables	3.7
`skipconstr`	`gosset` option to bypass check for redundant constraints	4.2
`skipdet`	`gosset` option to bypass check for singular model	4.2
`skiplinear`	`gosset` option, bypasses linear form checking	4.2
`skipmix`	`gosset` option, bypasses special mixture-moments	4.12
`skipsym`	`gosset` option to bypass search for symmetries	4.2
`skipxmom`	`gosset` option to bypass search for exact moments	4.2
`sphere`	declare spherical variables	3.2
`start`	specifies starting points for search	3.9
`start`	`interp` option producing `start` lines	5.2
`status` or `s`	status of job	6.2
`steps`	limit to number of steps in search	4.4
`tag`	four-letter label for program	8.18
`tiny`	minimal step size in search	4.4
`trace` or `t`	detailed report on progress	6.5
`unwait`	cancel a `wait` command	8.21
`use`	points to be included in design	3.8
`use`	`interp` option producing `use` lines	5.2

`version`	report which version of `gosset` you have	8.19
`wait`	wait for `moments` or `design` to finish	8.20
`watch`	watch progress of job	6.3
`write`	write program	8.9
`x`	variable in measurement region	3.11
`x'`	variable in modeling region (when different from measurement region)	3.11
`#values`	number of values taken by discrete variable	3.4
`<`	take input from file	8.14
`<#`	take input from file, add line numbers	8.14
`>`	divert output to file	8.11
`|`	divert output to shell command	8.11
`!`	shell escape	8.17
`*`	matches any string in a file name	5.4
`?`	matches any single character in a file name	5.4
`_`	the previous program	4.5

15. References

[AD92] A. C. Atkinson and A. N. Donev, *Optimum Experimental Designs*, Oxford Univ. Press, 1992.

[BCW88] R. A. Becker, J. M. Chambers and A. R. Wilks, *The New S Language*, Wadsworth and Brooks, Pacific Grove, CA, 1988.

[BPW79] C. S. Beightler, D. T. Phillips and D. J. Wilde, *Foundations of Optimization*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1979.

[BD59] G. E. P. Box and N. R. Draper, "A basis for the selection of a response surface design," *J. American Statistical Association*, vol. 54 (1959), pp. 622-654.

[BD63] G. E. P. Box and N. R. Draper, "The choice of a rotatable second-order design," *Biometika*, vol. 50 (1963), pp. 335-352.

[BD87] G. E. P. Box and N. R. Draper, *Empirical Model-Building and Response Surfaces*, Wiley, NY, 1987.

[CH91] J. M. Chambers and T. J. Hastie, editors, *Statistical Models in S*, Wadsworth and Brooks, Pacific Grove, CA, 1991.

[Co87] G. M. Constantine, *Combinatorial Theory and Statistical Design*, Wiley, N. Y. 1987.

[SPLAG] J. H. Conway and N. J. A. Sloane, *Sphere Packing, Lattices and Groups*, Springer-Verlag, NY, 2nd. edition, 1992 to appear.

[Co73] H. S. M. Coxeter, *Regular Polytopes*, Dover, NY, 3rd. edition, 1973.

[Do70] D. H. Doehlert, "Uniform shell designs," *J. Roy. Stat. Soc. Ser. C*, vol. 19 (1970), pp. 231-239.

[DK72] D. H. Doehlert and V. L. Klee, "Experimental designs through level reduction of the *d*-dimensional cuboctahedron," *Discrete Math.*, vol. 2 (1972), pp. 309-334.

[Ga85] Z. Galil, "Computing *D*-Optimum Weighing Designs: Where Statistics, Combinatorics, and Computation Meet," in *Proceedings Berkeley Conference in Honor of Jerzy Neyman and Jack Kiefer*, L. M. Le Cam and R. A. Olshen, editors., Wadsworth, Monterey, Calif., Vol. II, pp. 635-650.

[HS1] R. H. Hardin and N. J. A. Sloane, "Computer-generated minimal (and larger) response surface designs: (I) the sphere," manuscript, 1991.

[HS2] R. H. Hardin and N. J. A. Sloane, "Computer-generated minimal (and larger) response surface designs: (II) the cube," manuscript, 1991.

[HS3] R. H. Hardin and N. J. A. Sloane, "New spherical 4-designs," *Discrete Math.*, vol. 106/107 (1992), pp. 255-264.

[HS4] R. H. Hardin and N. J. A. Sloane, "A new approach to the construction of optimal designs," *J. Statistical Planning Inference,* 1993, in press.

[HS5] R. H. Hardin and N. J. A. Sloane, "Tables of Response Surface Designs," manuscript in preparation.

[HS6] R. H. Hardin and N. J. A. Sloane, "Transcripts of a series of Gosset demonstrations," manuscript, 50 pp., 1992.

[HS7] R. H. Hardin and N. J. A. Sloane, "Quadratic response surface designs in the ball and the cube," manuscript in preparation.

[HS8] R. H. Hardin and N. J. A. Sloane, "I-optimal mixture designs for linear and quadratic models," manuscript in preparation.

[HS9] R. H. Hardin and N. J. A. Sloane, "Optimal linear designs that protect against the loss of a run," manuscript in preparation.

[HS10] R. H. Hardin and N. J. A. Sloane, "Designs with pair-wise correlated measurements," manuscript in preparation.

[HS11] R. H. Hardin and N. J. A. Sloane, "New weighing designs," manuscript in preparation.

[HSS93] R. H. Hardin, N. J. A. Sloane and Warren D. Smith, *Spherical Codes* (book), in preparation.

[HJ61] R. Hooke and T. A. Jeeves, "'Direct search' solution of numerical and statistical problems", *J. Assoc. Comp. Mach.,* vol. 8 (1961), pp. 212-229.

[No82] W. Notz, "Minimal point second order designs," *J. Stat. Plann. Inf.,* vol. 6 (1982), pp. 47-58.

[PK70] E. S. Pearson and M. G. Kendall, editors, *Studies in the History of Statistics and Probability,* Hafner, Darien, CT, 1970.

[Ra71] D. Raghavarao, *Constructions and Combinatorial Problems in Design of Experiments,* Wiley, New York, 1971.

[Ro76] K. G. Roquemore, "Hybrid designs for quadratic response surfaces," *Technometrics,* vol. 17 (1976), pp. 419-423.

[STW91] A. C. Shoemaker, K. Tsui and C. F. T. Wu, "Economical experimentation methods for robust design", *Technometrics,* vol. 33 (1991), pp. 415-427.

[Ta87] G. Taguchi, *System of Experimental Design,* UNIPUB/Krause, White Plains, 1987.

16. Index

The numbers refer to Sections.