An introduction to the Standard Data Input Format (SIF) for nonlinear mathematical programming problems

by A.R. Conn[1], Nick Gould[2] and Ph.L. Toint[3]

Report 91/8 March 7, 1995

[1] **IBM T.J. Watson Research Centre**
Yorktown Heights, USA

[2] **Rutherford Appleton Laboratory Oxfordshire, England**

[3] **Department of Mathematics,**
Facultés Universitaires ND de la Paix, Namur, Belgium

Contents

About this report

If you read this line, this is probably because you are interested in solving some optimization problem(s), and that you hope that the LANCELOT package might help you. The purpose of this manual is to help this hope materialize: starting from simple examples, we intend to tell you

- how to specify the problem you are interested in,
- how to use LANCELOT to solve it.

We will do both at once, because this process is actually quite simple. Of course, specifying a problem and solving it are formally two different tasks, each requiring a different set of tools. This will force us to distinguish these tasks in our examples as well, but we will always try to remember that specifying the problem is only a step towards its practical solution.

The general approach that we intend to use throughout this chapter is to learn some basic problem specification techniques first and to see how to solve the resulting simple problems, gradually building on these to tackle more and more complex cases. This manual is therefore a learning tool, and should not be considered as a reference guide for expert users. Such a reference text is presented in [5].

A bit of history: LANCELOTand the SIF

Around 1986 and after much debate and some experience, the authors of this manual agreed on the statement that it was indeed very difficult to solve large nonlinear optimization problems. Both a proven methodology and suitable software were indeed badly needed. The subsequent discussion and collaboration led first to the design of a robust trust region method for nonlinear minimization problems with simple bounds [3] and subsequently to a more general method capable of handling the general nonlinear programming problem [6]. The implementation that went along with these theoretical developments always had as a priority the capacity to solve large scale problems, using the structure of partially separable [10], and later that of group partially separable functions [4]. LANCELOT(an acronym for Large And Nonlinear Constrained Extended Lagrangian Optimization Techniques) is the result of that implementation. We cannot resist the pleasure of showing you the LANCELOTbanner (Fig. 1).

Once a first version of the much discussed large-scale nonlinear programming software (the version A– of LANCELOT) was available, it soon became apparent that the very statement of these problems (they were clearly needed for testing the code!) was in itself a rather formidable challenge: the amount of information present in the structure of a large scale optimization problem, although crucial for the acceptable performance of the algorithms, was also very difficult to

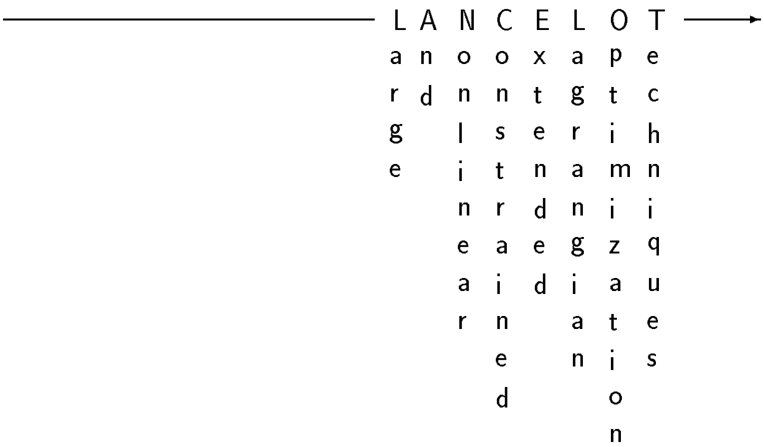

Figure 1: The LANCELOTbanner

specify in a complete and understandable format. Further discussion with other researchers in the area soon convinced the authors that this challenge could be tackled by using a "standard input", that is a simple formal language in which these structural concepts could be expressed unambiguously. This particular approach had indeed been rather fruitful in the more restricted domain of linear programming, where the MPS format [7], despite many known shortcomings, has and still is playing a considerable role. It was therefore decided that a Standard Data Input Format (SIF) for large nonlinear problems was the next step. This of course required more discussions and tests, but a proposal eventually materialized (see [5]). The proposal was also soon turned into software to "compile" the problems specified in this format into structures[1] that LANCELOTcould actually use: the SIF interface to LANCELOT.

Today, more than 500 problems have been specified in SIF and most of them have subsequently been solved by LANCELOT. This collection features a wide variety of problem types: large and small, academic and practical, simple and difficult. It has been our experience that, once understood, the SIF approach is indeed quite efficient for problem specification. We started to be really enthusiastic when somebody came in to one of our offices with a highly structured constrained nonlinear programming problem arising from his research area (energy scenario simulation) where the number of variables exceeded 1000: in just one hour, the problem was specified, validated (and even solved!) using SIF and LANCELOT...Although this favourable scenario will (alas) probably remain exceptional, it indicates the potential effectiveness of the approach.

Of course, the SIF is not restricted to its use in tandem with LANCELOT(see [5] for a discussion), but this is the context we consider in this report. We hope that we will be able to convince you that it provides a practical and effective way to state and solve the nonlinear programming problems that you have in mind...

The heritage of MPS

As we have already stated, the very idea of defining an SIF was forced on the authors by the sheer difficulty they experienced in specifying large problems. It was also motivated by the successful record of MPS, a standard input format for linear programming (LP). As the purpose of the SIF is to cover the general mathematical programming problem, of which linear programs are

[1]Fortran subprograms, as a matter of fact.

specialized examples, it is natural to require that the MPS syntax constitute a subset of the SIF syntax. This option has the further advantage that the sizeable collection of LP test problems already available in MPS format can also be used within the SIF context *without modification*.

However, preserving MPS compatibility has disadvantages. Maybe the worst is the requirement of using a fixed format (that is of aligning keywords and data in well defined columns). MPS also has a (somewhat outdated) “style” for the keywords themselves that we had to adopt in the SIF syntax definition. The authors are well aware that a completely new design would have avoided those old fashioned “features”, but the price to pay was to give up MPS compatibility. That seemed, to us, to be an even worse choice.

One must also stress that a Standard Data Input Format by no means replaces a true modelling language. Such languages have already been developed in the context of linear programs (OMNI [11], for instance) and also in a more general context (GAMS [1] or AMPL [8]). However, none of these actually provides a way of specifying the nonlinear structure as does the SIF. Moreover, they do not belong to the public domain, which puts additional restrictions on their widespread use by the scientific community.

Although the previous paragraph sounds a little bit like a disclaimer (all right, it is one), we strongly believe that using the SIF is now a viable and efficient technique, and we hope that you will share this belief when you know more about its innards.

1 Getting started with the SIF: linear programs

1.1 The fields in a SDIF statement

Before we actually begin to specify our first mathematical programming problem, we have to briefly discuss some prerequisites about the way words and numbers are understood within the SDIF syntax.

When you specify a problem in SIF, you do so by writing one or more *chapter(s)*. We start by considering the first chapter, the Standard *Data* Input Format (or SDIF for short). We will return to the subsequent chapters after gaining some knowledge of the first.

The SDIF contains a number of ordered *sections* (just as in MPS) using five kinds of objects: keywords, codes, numbers, names and Fortran names.

- The *keywords* are the headers (titles) of the different sections that together constitute the problem SDIF. We will meet these sections (and their titles) in due course in this introduction.

- *Codes* consist of one or two upper case letters. We will also learn about them as we go along. Their purpose is to specify, within the sections, various kinds of information on the problem at hand.

- *Numbers* consists of at most 12 characters which may include a decimal point and an optional sign (a positive number is assumed unless a - is given). The value may be followed by a decimal exponent, written as an `E` or `D`, followed by a signed or unsigned one or two digit integer. Blanks are not allowed within numbers. Hence “3.14”, “-0.001”, “10” and “+4.23D-06” are numbers but “2.0D 5” is not. Observe that the decimal point is optional, and that it is therefore possible to specify integer numbers as “10” or “-3”.

- *Names* (that you will want to attach to various parts of the problem specification, as variables, constraints...) are made up of *valid characters*. The valid characters are all characters whose ASCII decimal code lies in the range 32 to 126 (binary 0100000 to 11111110, hex 20 to 7E). These include lower and upper case roman alphabetic characters, the digits 0 to 9, the blank character and other mathematical and grammatical symbols. The allowed length for a name varies between 6 and 10, as we will discover later. The only excluded names are "'SCALE'", "'DEFAULT'", "'MARKER'" and the name consisting entirely of blanks.

For instance "NAME", "lower name" , "3.141592" and "@&#+-*/_!!" are valid names. Of course, you would usually choose names that mean something to you, because that helps in understanding the relationships between the parts of the problems, so you might want to use "COST", "km/h", "PI" or "Heavens!" instead of the above.

- The SIF/LANCELOT interface produces Fortran subroutines that are subsequently used by the LANCELOT optimizers. These routines contain variables that are named within the SIF problem specification. We call these names *Fortran name*. They are restricted to contain at most 6 characters (upper-case alphabetic letters and digits only) and must not start with a digit.

In addition to these three categories of objects, you may also want to insert comments into your problem description. This is obtained by starting a line with *, and anything that follows on that line is your comment, as in

```
* This useful comment was written by me today (version 1).
```

As alluded to above, codes, names and numbers must occur in well defined *fields* within the line. The definition for the 6 fields that can occur[2] on a line is as follows

```
F1 <---F2---><---F3---><----F4---->   <---F5---><----F6---->
↑  ↑         ↑         ↑              ↑         ↑          ↑
2  5         15        25             40        50         61
```

where the numbers below indicate the columns in which the respective fields start. Note that the fields (4 and 6) are 12 characters wide: field 4 ends in column 36 while field 6 ends in column 61. The fields contain different types of information:

- field 1 contains a code,
- fields 2, 3 and 5 contain names,
- fields 4 and 6 contain numbers.

Schematically, a typical SDIF line is thus organized as

```
F1 <---F2---><---F3---><----F4---->   <---F5---><----F6---->
cd <--name--><--name--><--number-->   <--name--><--number-->
```

Of course, such an organization is very rigid and has a number of drawbacks, but it also has the advantage of simplicity and readability. It is also absolutely necessary because blank characters are significant in MPS names, and can therefore not be used as separators. Section 6.9 will indicate how the rigidity of this format may be alleviated, but we will use it until then.

A word of caution might be appropriate here. Notice that the blank characters are significant within names, which implies that "`A `" and "` A`" are different names! We therefore recommend that you flush all names to the left of the field where they appear, in order to avoid mistakes that are quite hard to detect...

Finally, we said that the SDIF sections must appear in a certain order, as is the case in MPS. This order is implicitly given in all the examples we will see (watch out!), but is also formalized in Section 6.6 if you have any doubt.

[2]We will also see later that, in two particular cases, the fields are defined somewhat differently (see Sections 4.1 and 3.1).

1.2 The simplest example

We are now ready to specify our first problem. Obviously, we will start by a very simple example. In fact, it is the simplest example that we could think of. The problem is to find the smallest real nonnegative number, or, in its mathematical form, the following linear program:

$$\min_{x \in \mathbf{R}} x \tag{1}$$

subject to the constraint

$$x \geq 0. \tag{2}$$

As you can see, it is very hard to find a simpler one. Let us now see how we can specify it in the SDIF.

The first thing we have to do is to define a name for this problem, "SIMPLEST" for instance[3]. We then give it that name by writing:

```
NAME          SIMPLEST
```

In this line, the keyword `NAME` starts in column 1 and the name itself is in field 3.

We then have to specify the variable of the problem, x. The task of specifying the problem's variables and naming them is performed in the `VARIABLES` section. As indicated above, the SDIF is divided in a number of such "sections", just as in the MPS format. Each one has its well defined purpose, that we will discover little by little below. After specifying the `VARIABLES` section, our problem description then becomes

```
NAME          SIMPLEST

*   The simplest linear program

VARIABLES
     x
```

Notice that we have also added a comment to describe the problem. This comment is surrounded by two blank lines: blank lines are totally ignored in SDIF: they can thus be used to "space out" the content of the specification and make it more readable. Then we have started the `VARIABLES` "section" by using the associated keyword, starting in column 1. All keywords (marking the beginning of sections) start in column 1, just as `VARIABLES` and `NAME`. Finally, we have declared that `x` is the (only) variable for the problem.

Once the problem is named and its variable(s) known, we must of course define the objective function. Ours is quite simple and is just equal to $1.0 \times x$. This is what we mean by completing our specification:

```
NAME          SIMPLEST

*   The simplest linear program

VARIABLES
    x
GROUPS
 XN Object    x          1.0
ENDATA
```

[3]This is one of the few cases where there is an extra restriction on the length of a name. As many users may wish to use the name of the problem as the name of the file containing the problem description, it was necessary to restrict the length of the problem's name to what can be accepted as a filename within most popular operating systems. It turns out that some unfriendly systems still limit the length of filenames to 8 characters, so this is the limit in SDIF as well.

As above, we first started a section: this one is called `GROUPS` and its purpose is to state the objective function (and, later, the constraints) of the problem as well as the linear contribution of the variables to these functions. The objective is declared by using the code `XN`. Observe that `XN` is a code and thus starts in column 2. We then say that variable `x` appears in the objective function (that we call `Object`) linearly with a coefficient equal to 1.0. Notice that `1.0` appears in field 4, one of the two "numerical" fields (the other being field 6). As a matter of fact, it can be written anywhere within this field.

Then we have added the keyword `ENDATA`, which says that this is the end of the SDIF (and, in this simple case, of the SIF) specification! Quite a short description, isn't it? We now want to use LANCELOT to solve this problem. Assuming we just saved our problem description in a file called `SIMPLEST.DAT`[4], we simply issue the basic command to run LANCELOT (double precision version) on a SIF problem description, i.e.

```
sdlan SIMPLEST
```

and watch what happens. Your faithful computer then answers:

```
PROBLEM NAME: SIMPLEST

Double precision version will be formed.

The objective function uses      1 linear group


There is       1 variable  bounded only from below


objective function value =   0.00000000000000D+00

 x                 0.00000000000000D+00
```

Well, this is quite obviously a brief summary of our problem, followed by an equally brief statement of its solution!

But you will immediately object: "How come LANCELOT understands that the variable x must be nonnegative, although we did not specify it anywhere?" In fact, we did not forget it, but specified it implicitly. We used the convention that SDIF (as MPS) assumes that *all variables are restricted to be nonnegative, unless stated otherwise.* This type of information appears in another section surprisingly called `BOUNDS`, so that our problem can equally be specified by

```
NAME          SIMPLEST

*   The simplest linear program (version 2)

VARIABLES
    x
GROUPS
 XN Object    x         1.0
BOUNDS
 XL SIMPLEST  x         0.0
ENDATA
```

We added a `BOUNDS` section, in which we specified that variable `x` has a lower bound of 0, using the code `XL`. (Rerunning LANCELOT with this more explicit SIF description of course still produces the same answer.) The repetition of the problem's name after the `XL` code is a consequence of

[4] At least on a DOS, OS/2, VMS or UNIX system. On a CMS system the file might be called SIMPLEST DAT.

our decision to be MPS compatible: MPS indeed require that bounds "sets" (that is the set of bound specifications corresponding to one instance of the problem) be explicitly named.

Specifying upper bounds is just as easy: if we wish to impose the additional constraint

$$x \leq 5, \tag{3}$$

we simply modify our problem statement to

```
NAME          SIMPLEST

*   The simplest linear program (version 3)

VARIABLES
    x
GROUPS
 XN Object    x         1.0
BOUNDS
 XU SIMPLEST  x         5.0
ENDATA
```

remembering now that the line

```
 XL           x         0.0
```

is implicitly present. LANCELOT now says:

```
PROBLEM NAME: SIMPLEST

Double precision version will be formed.

The objective function uses      1 linear group


There is      1 variable  bounded from below and above


objective function value =   0.00000000000000D+00

 x                 0.00000000000000D+00
```

The only difference from what we saw above is in the summary line that now states that there is 1 variable (the only one here) that is bounded from below (by 0) and from above (by 5). Other type of bounds can of course be imposed on a variable; some of the usual relevant codes and their meanings are given in Table 1. They may appear in any order, the last one used taking precedence.

Note that a variable can be fixed (its lower and upper bounds are equal), which in effect means that it is not allowed to vary! It therefore acts as parameter in the problem definition, sometimes a very useful trick.

There is another part of the problem that we avoided entirely above: the specification of a starting point for the minimization. Of course, this is not so important for linear programs (as `SIMPLEST`), but may be vital for nonlinear problems. *If nothing is explicitly said, the SDIF assumes that the starting point is the origin.* So in fact we started solving `SIMPLEST` from its solution! Is LANCELOT capable of solving the problem when started from somewhere else? Fortunately yes, as you can quickly check by introducing a different starting value for `x` (using the code `XV`, where `V` stands for Variable, in a section obviously called `START POINT`).

Codes	Meaning
XL	Bounded below by the value... (Lower)
XU	Bounded above by the value... (Upper)
XP	Unbounded above (Plus infinity)
XM	Unbounded below (Minus infinity)
XR	Unbounded above and below (fRee)
XX	Bounded above and below by the common value... (fiXed)

Table 1: Codes for bounding variables (in the BOUNDS section).

```
NAME          SIMPLEST

*   The simplest linear program (version 4)

VARIABLES
    x
GROUPS
 XN Object    x         1.0
START POINT
 XV SIMPLEST  x         3.141592
ENDATA
```

(The starting points are also given the problem's name, just as the bound sets.) Running LANCELOT again (just in case) now dissipates your last doubts.

1.3 Complicating our first example

Equipped with our fresh knowledge, we now enter the realm of problems with more complex constraints. As a cautious first step, we (barely) complicate our initial problem (1)–(2) and consider

$$\min_{x,y\in\mathbf{R}} x+1 \tag{4}$$

subject to the constraints

$$x \geq 0 \tag{5}$$

and

$$x+2y=2, \tag{6}$$

where we have added one variable (y) and the constraint (6). This is easily translated into

```
NAME          EXTRASIM

*   An extremely simple linear program

VARIABLES
    x
    y
GROUPS
 XN Object    x         1.0
 XE Cautious  x         1.0            y         2.0
CONSTANTS
    EXTRASIM  Object    -1.0
    EXTRASIM  Cautious  2.0
```

```
BOUNDS
 XR          y
ENDATA
```

The introduction of `y` and its declaration as a free variable doesn't call on anything new. What is new is the code `XE` employed in the `GROUPS` section to specify Equality constraints. It is followed by the name of the constraint (`Cautious` will do nicely) and a list of the variables occurring linearly in the constraint (`x` and `y`) together with their respective nonzero coefficients (1.0 and 2.0 here). We also specified the right-hand-side (constant) part of the new constraint in the `CONSTANTS` section in a rather obvious way, specifying that the constant value associated with the constraint `Cautious`. Just notice that the name of the relevant constraint now appear in field 3. In fact, *the difference between field 2 and fields 3 and 5 is that the former is used to indicate names of the object being* defined *by the current statement, while the latters contain names of already defined objects*: the statement

```
 XE Cautious  x         1.0            y         2.0
```

defines the constraint `Cautious`, while

```
    EXTRASIM  Cautious  2.0
```

defines the set of constants named `EXTRASIM` and merely assigns a constant value to the already defined constraint. Another novelty in `EXTRASIM` is that we have added the constant 1 to the objective function. This constant must therefore be specified in the `CONSTANTS` section together with the right-hand-side of the constraints, which we have done by writing

```
    EXTRASIM Object    -1.0
```

Notice the we have written "`-1.0`" instead of the expected "`1.0`", because the objective has no proper right-hand-side, and thus the constant should be put in the left-hand-side, which implies it changes sign!

After saving the specification in `EXTRASIM.DAT`, this yields

```
sdlan EXTRASIM

 PROBLEM NAME: EXTRASIM

 Double precision version will be formed.

 The objective function uses      1 linear group

 There  is      1 linear equality constraint

 There  is      1 free variable
 There  is      1 variable  bounded only from below


 objective function value =   1.00000000000000D+00

            x                 0.00000000000000D+00
            y                 1.00000000000000D+00
```

which is indeed what we expect: the problem summary has been updated and the new solution calculated by LANCELOT is correct.

As the first step was not too difficult, let us be a bit more daring and add two more constraints to our problem. We will still consider (4)–(6), but subject to the additional requirements that

$$2x + y \geq 2 \tag{7}$$

and

$$y - x \leq 1 \tag{8}$$

Our problem specification is soon completed and becomes

```
NAME          SUPERSIM

*   A super-simple linear program

VARIABLES
    x
    y
GROUPS
 XN Object    x         1.0
 XE Cautious  x         1.0            y         2.0
 XG Daring    x         2.0            y         1.0
 XL Foolish   x         -1.0           y         1.0
CONSTANTS
    SUPERSIM  Object    -1.0
    SUPERSIM  Cautious  2.0
    SUPERSIM  Daring    2.0
    SUPERSIM  Foolish   1.0
BOUNDS
 XR           y
ENDATA
```

The codes `XL` is used to specify constraints where a `L`ess than or equal signs separates the variables from the constant. Similarly, `XG` specifies "`G`reater than or equal" constraints. This is summarized in Table 2.

Codes	Meaning		
XN	Objective function − constant		
XL	Constraint	$\leq$	constant
XE	Constraint	$=$	constant
XG	Constraint	$\geq$	constant

Table 2: Codes for declaring constraint types (in the `GROUPS` section).

We can now return to LANCELOT with the resulting problem file `SUPERSIM.DAT`, and type

```
sdlan SUPERSIM

 PROBLEM NAME: SUPERSIM

 Double precision version will be formed.

 The objective function uses      1 linear group

 There  is      1 linear equality constraint
 There are      2 linear inequality constraints
```

```
There  is      1 free variable
There  is      1 variable  bounded only from below
There are      2 slack variables



objective function value =   1.66666666666667D+00

           x                 6.66666666666667D-01
           y                 6.66666666666666D-01
           Daring            0.00000000000000D+00
           Foolish           1.00000000000000D+00
```

A quick glance at the result shows the solution is determined by both the `Daring` and `Cautious`[5] constraints, but that the `Foolish` one was unnecessary[6] and does not play any role in the final outcome[7].

1.4 Summary

In our first contact with LANCELOT, we learned some basic concepts about the objects that are used in an SDIF problem description: *keywords* to delimit sections, *codes* to indicate all kinds of informations that are relevant to a given section, *names* which can be given to variables, the objective function and constraints, and finally *numbers*.

We also introduced a number of sections.

`NAME` : defines the name of the problem.

`VARIABLES` : starts the section where the problem variables are given a name.

`GROUPS` : starts the section where the objective function (code `XN`) and constraints (codes `XL`, `XE` or `XG`) are given a name, and where the linear contribution of each variables to these is specified. The meaning of these codes is summarized in Table 2.

`CONSTANTS` : starts the section where the constant terms of the groups are named and defined. These are the constant term in the objective function (if any) and the right-hand-sides of the constraints (a zero right-hand-side is assumed unless otherwise stated).

`BOUNDS` : starts the section where the bounds on the variables are named and defined (a lower bound of 0 is assumed on the variables unless stated otherwise). The codes for specifying bounds are summarized in Table 1.

`START POINT` : starts the section where the proposed starting point for the problem is named and specified using the code `XV` (the value 0 is assigned to all variables unless stated otherwise).

`ENDATA` : declares the end of the SDIF file for the current problem.

Finally note that empty `CONSTANTS`, `BOUNDS` or `START POINT` sections, i.e. sections where the default specification is appropriate for all objects covered in the section (constants, bounds or starting point components respectively), may safely be omitted, as is the case in our `SIMPLEST` example, where all three sections are missing. The same sections require a name to be given to

[5] Equality constraints are always active at the solution!

[6] As is often the case for such constraints in real life.

[7] At variance with what usually happens in real life.

the specification that they contain. It is of course necessary to choose the same name for all these sections: in the above examples, we have taken the convention to use the problem's name in all cases.

2 SDIF acquires a new skill: using the named parameters

2.1 What are the named parameters?

Let us return for a moment to our `SUPERSIM` problem, as described in page 1.3 and remove the unnecessary `Foolish` constraint from its definition. We are now interested in testing how the problem solution varies when we change the value of the constant appearing in both the `Daring` and `Cautious` constraints (a typical "sensitivity" study). One way of exploring this issue is to change the two corresponding lines in the `CONSTANTS` section (setting a common value for both constants that is different from 2.0), and then to run LANCELOT again. We might then change those two lines once more in order to try a third value, etc. However, the SDIF provides a way to change only one line instead of two for each value we want to try. The idea is to introduce a *parameter* to which we assign the common constant used in both our constraints and then to use this parameter in the definition of the constraints. Such a parameter will have a name as do all objects within the SDIF. If `Level` is the name we wish to give to this parameter, the description of problem `SUPERSIM` becomes:

```
NAME SUPERSIM

*   A super-simple linear program (version 2)

VARIABLES
    x
    y
GROUPS
 XN Object    x         1.0
 XE Cautious  x         1.0            y         2.0
 XG Daring    x         2.0            y         1.0
CONSTANTS
 RE Level               2.0
 Z  SUPERSIM  Cautious                 Level
 Z  SUPERSIM  Daring                   Level
BOUNDS
 XR SUPERSIM  y
ENDATA
```

We have introduced here two new codes, namely `RE` and `Z`. The first (`RE`) is used to assign the real value appearing in field 4 to the real parameter `Level` (the mnemonic for the code comes from the interpretation that `Level` Equals 2.0). The second (`Z`) merely says that the value to assign to the constants in the `Cautious` and `Daring` constraints is no longer to be found in the numerical field 4, but that the value of the parameter whose name appears in field 5 (`Level`) is to be used instead.

If we now wish to create a variant of the `SUPERSIM` problem, it is easy to modify only the value given in field 4 of the definition of `Level`, for instance by rewriting this line as

```
 RE Level               3.141565
```

Notice that we have assigned a value to the parameter `Level` just before it is needed for the first time, but we could also introduce it anywhere in the specification *before its value is actually needed.* For instance, the specification

```
 NAME         SUPERSIM

*   A super-simple linear program (version 3)

 RE Level             2.0
VARIABLES
    x
    y
GROUPS
 XN Object    x         1.0
 XE Cautious  x         1.0              y         2.0
 XG Daring    x         2.0              y         1.0
CONSTANTS
 Z  SUPERSIM  Cautious                   Level
 Z  SUPERSIM  Daring                     Level
BOUNDS
 XR SUPERSIM  y
ENDATA
```

is equally valid: we have placed the definition of `Level` in the 0-th section of the SDIF. This is often done for problem dependent parameters, because it makes them easy to find for later modifications.

2.2 The `X` vs. `Z` codes

Our last example is probably an excellent occasion to introduce a difference in codes that will appear quite often in the SDIF syntax: the difference between codes starting with an `X` and codes starting with a `Z`. Observe that in the line

```
 Z  SUPERSIM  Cautious                   Level
```

one has to decide somehow that the value of the constant for the `Cautious` constraint is not given by a number in the numerical field 4, but is instead given by the actual value that the parameter `Level` currently happens to have. We will use the `Z` code to indicate that we expect the value to be defined by that of an already defined named parameter.

In general, *The SDIF will use codes starting with* `X` *whenever the value associated with the action specified by the code is to be found in one (or both) of the "numerical fields" 4 and 6. Codes starting with* `Z` *will be used whenever this value is that of the parameter whose name appears in the "name field" 5.* To complicate things a bit further, the code consisting of the single character `X` can sometimes[8] be omitted, as above.

To illustrate this point, just observe that

```
    SUPERSIM  Cautious  2.0
    SUPERSIM  Daring    2.0
```

is equivalent to

```
 X  SUPERSIM  Cautious  2.0
 X  SUPERSIM  Daring    2.0
```

which, in turn, is equivalent to

```
 RE Level             2.0
 Z  SUPERSIM  Cautious                   Level
 Z  SUPERSIM  Daring                     Level
```

[8]But not always: see Section 5.1.

This equivalence holds in the `CONSTANTS` section, as in our example, but also anywhere else in the SDIF problem specification! In fact, a number of the codes we have seen so far also have a `Z` form. For instance, the `GROUPS` section of the `SUPERSIM` problem could be written as

```
RE one                1.0
RE two                2.0
ZN Object    x                       one
ZE Cautious  x                       one
ZE Cautious  y                       two
ZG Daring    x                       two
ZG Daring    y                       one
```

although this is somewhat verbose in this case. Note that we have continued the description of the linear variables in groups `Cautious` and `Daring` on more than one line because we now have to use one line per variable appearing in the group. One can even write

```
RE 1.0                1.0
RE 2.0                2.0
ZN Object    x                       1.0
ZE Cautious  x                       1.0
ZE Cautious  y                       2.0
ZG Daring    x                       2.0
ZG Daring    y                       1.0
```

where we have defined two parameters called "`1.0`" and "`2.0`" (both are valid names), to which we have assigned the numerical values 1.0 and 2.0. We then have used the parameters *by their names*, hence the `Z`-type codes! This trick sometimes helps, but be careful not to define a variable called "`1.0`" whose numerical value is 0.0 or any other value different from 1.0: this can generate appreciable confusion and be very hard to read or correct at a later stage...

2.3 Arithmetic with real named parameters

It does not take much playing with the parameters we just introduced before one wishes to perform simple arithmetic operations involving their values. Fortunately, these parameter operations are possible: assignment, the four basic arithmetic operations and a few basic numerical function are indeed available in the SDIF.

We already saw the most elementary case above in the line

```
RE Level              2.0
```

This is the simple assignment of a real numerical value to a named parameter. One may wish to add a numerical value to that of an already defined parameter: this is done by

```
RA Higher    Level    1.4
```

where the constant 1.4 is added to the value of `Level` to obtain the new real parameter `Higher`, i.e.

$$\texttt{Higher} := 1.4 + \texttt{Level}. \tag{9}$$

The code is `RA` because it is a `R`eal `A`ddition. A `R`eal `S`ubtraction is also available in the form

```
RS Rest      Higher   7.0
```

where we have specified the operation

$$\texttt{Rest} := 7.0 - \texttt{Higher}, \tag{10}$$

and not the reverse[9]! Similarly, the `R`eal `M`ultiplication is specified as

[9]This somewhat unexpected definition turns out to be the right one in many cases. Also, it would be extremely easy to add -7.0 with the `RA` operation is the reverse operation is wanted.

```
RM Plenty    Rest      27.3
```

which just means

$$\texttt{Plenty} := 27.3 \times \texttt{Rest} \tag{11}$$

while the Real Division is written as

```
RD Pi        Plenty    308,75572599
```

to compute

$$\texttt{Left} := 308,75572599/\texttt{Plenty} \tag{12}$$

Note that the first argument divides the second, and not the opposite. Also note that the value of the parameter **Plenty** in field 3 must be different from 0![10] One can finally take the square root of a number and assign the result to a parameter, as in

```
RF Pi        SQRT      9.869604
```

where the code RF stands for Real Function, and where the name of the considered function is indicated by its name in field 3. The square root's name is SQRT, but other functions are also available: they are given in Table 3. For instance, one can write

```
RF Zero      SIN       3.14159265
```

or

```
RF Zero      ARCCOS    1.0
```

Name	Meaning		Restriction
SQRT	Square root	$f(x) = \sqrt{x}$	$x \geq 0$
ABS	Absolute value	$f(x) = \lvert x \rvert$	
EXP	Exponential	$f(x) = e^x$	
LOG	Natural logarithm	$f(x) = \log(x)$	$x > 0$
LOG10	Logarithm in base 10	$f(x) = \log_{10}(x)$	$x > 0$
SIN	Sine	$f(x) = \sin(x)$	
COS	Cosine	$f(x) = \cos(x)$	
TAN	Tangent	$f(x) = \tan(x)$	
ARCSIN	Inverse sine	$f(x) = \sin^{-1}(x)$	$\lvert x \rvert \leq 1$
ARCCOS	Inverse cosine	$f(x) = \cos^{-1}(x)$	$\lvert x \rvert \leq 1$
ARCTAN	Inverse tangent	$f(x) = \tan^{-1}(x)$	
HYPSIN	Hyperbolic sine	$f(x) = \sinh(x)$	
HYPCOS	Hyperbolic cosine	$f(x) = \cosh(x)$	
HYPTAN	Hyperbolic tangent	$f(x) = \tanh(x)$	

Table 3: Available real functions in SDIF

The next step is, of course, to perform arithmetic operations involving parameters only, and no longer explicitly given numerical values. This is achieved in a way very similar to what we just saw. In fact, the now familiar sequence of operations

[10] This is *not* 0 factorial!

```
RE Level              2.0
RA Higher    Level    1.4
RS Rest      Higher   7.0
RM Plenty    Rest     27.3
RD Pi        Plenty   308,75572599
RF Pi        SQRT     9.869604
```

can equally be rewritten as

```
RE Level              2.0
RE A                  1.4
RE B                  7.0
RE C                  27.3
RE D                  308,75572599
RE E                  9.869604

R+ Higher    Level                   A
R- Rest      B                       Higher
R* Plenty    Rest                    C
R/ Pi        D                       Plenty
R( Pi        SQRT                    E
```

where we have used the parameters `A`, `B`, `C`, `D` and `E` to hold the numerical values explicitly given before. The codes `R+`, `R-`, `R*`, `R/`[11] and `R(` have replaced `RA`, `RS`, `RM`, `RD` and `RF` respectively. Also note that the order of the operands for the subtraction (`R-`) and division (`R/`) operations has been reversed[12]...

2.4 Integer parameters

In Section 1.1, we have indicated that parameters come in two kinds: reals and integers. The arithmetic operations we have defined up to now only handle real parameters; it is thus natural to introduce their integer counterparts. This is done most easily: the codes used for handling integer parameters differ from those used for real one only in their first letter, where the `R` of `Real` is simply replaced by the `I` of `Integer`! The following lines are thus perfectly valid:

```
IE Two                2
IE One                1
IA Three     Two      1
IS Four      Three    7
ID Five      Four     20
IM MinusOne  One      -1
I+ Nine      Four                    Five
I- Seven     Nine                    Two
I* MinusTwo  Two                     MinusOne
I/ OneAgain  Three                   Two
```

There is no integer equivalent of `RF` and `R(`, of course, because the functions of Table 3 take a real argument and return a real value. Observe also that `ID` and `I/` stand for integer division, i.e. the real result of the division is truncated to an integer.

Finally, it is also possible to truncate a real parameter to obtain an integer one, and also to promote an integer parameter to real, using the codes `IR` and `RI` respectively. This is illustrated in the following lines:

[11] As for `RD`, the value of the parameter whose name appears in field 5 must be different from zero.

[12] Yes, we think this is confusing too.

```
RE Pi                  3.1415926536
IR Nearpi     Pi
RI Pi-approx  Nearpi
```

which specifies that the integer parameter `Nearpi` has the value 3 and that the real parameter `Pi-approx` has the value 3.0.

2.5 Summary

In this section, we have introduced the real and integer named parameters, as well as the operations that the SDIF is capable of performing with them. The type of a parameter (real or integer) is specified by its first definition in the problem specification[13].

Real parameters are defined and operated upon in lines with codes starting with the letter `R`, while integer parameters are defined and modified in lines with codes starting with the letter `I`.

Code first letter	Type of the defined parameter
`R`	Real parameters and numbers
`I`	Integer parameters and numbers

Table 4: The meaning of the first letter in parameter related codes

The second letter of the code indicates what operation is to be performed with the content of fields 3 to 5 in order to define the value of the parameter named in field 2. Table 5 summarizes these operations. In this table the symbols f_i stands for the content of the i-th field. The table also indicates whether or not the order in which the operand appear is important.

Code second letter	Operations	Order
`E`	$f_2 = f_4$	
`A`	$f_2 = f_4 + f_3$	
`S`	$f_2 = f_4 - f_3$	•
`M`	$f_2 = f_4 \times f_3$	
`D`	$f_2 = f_4 / f_3$	•
`=`	$f_2 = f_3$	
`+`	$f_2 = f_3 + f_5$	
`-`	$f_2 = f_3 - f_5$	•
`*`	$f_2 = f_3 \times f_5$	
`/`	$f_2 = f_3 / f_5$	•

Table 5: The meaning of the second letter in codes for parameter arithmetic

We have also seen that simple real functions can be applied to real parameters to define new real parameters, using the syntax given in Table 6 where f_i again stands for the content of the i-th field and where $F(\cdot)$ is a real function whose name is chosen in Table 3 and appears in field 3.

Finally, conversion to `Real` from `Integer` and to `Integer` from `Real` are supplied by the codes `RI` and `IR` respectively.

[13]It should also be its first occurrence in the file!

Code	Operation
`RF`	$f_2 = F(f_4)$
`R(`	$f_2 = F(f_5)$

Table 6: Syntax for applying simple functions to real parameters

3 Groups, linear least-squares and unary operators

After exploring the civilized domain of simple linear programs with LANCELOT and his faithful S(D)IF, we now venture in a wilder territory: that of nonlinear functions. In this section, we will however be cautious and only consider how to specify the tamest of all nonlinear problems, the linear least-squares problem.

3.1 Group functions and unary operators

Let us first consider the very simple nonlinear problem of minimizing the square of the difference between two nonnegative variables x and y subject to the constraint that x and y sum to 1.0. Formally, the problem can be expressed as

$$\min_{x,y\in\mathbf{R}} (x-y)^2 \tag{13}$$

subject to the constraints that

$$x + y = 1.0 \tag{14}$$

and also that

$$x \geq 0 \ \text{ and } \ y \geq 0. \tag{15}$$

Assume, for an instant, that the square is not present in the objective definition. Using our knowledge of SDIF for linear programs, we could write the corresponding specification as follows.

```
 NAME          WRONG

*   Not quite a simple constrained linear least-squares problem

VARIABLES
    x
    y
GROUPS
 XN Object    x         1.0            y         -1.0
 XE Constr    x         1.0            y         1.0
CONSTANTS
 X  WRONG     Constr    1.0
ENDATA
```

This would be quite simple indeed, but the square in (14) can unfortunately not be forgotten. The idea is now to specify a nonlinear transformation of the objective group (in our case, raising it to the power two). Such transformations are called *group transformation* in SIF. Each such transformation is said to be of a specific *group type*, indextype of a group and all group types are introduced in a special purpose section cunningly called `GROUP TYPE`. For our problem, we need a `GROUP TYPE` section of the form

```
GROUP TYPE
 GV SQUARE    ALPHA
```

where we first used the keyword `GROUP TYPE` to indicate the beginning of the new section, and then the code `GV` to specify that the new group type, named `SQUARE`, involves a *G*roup *V*ariable, called `ALPHA`, that is the variable needed to specify the group transformation (here, the transformation is α^2).

Of course, we still have to say that the objective group is of type `SQUARE` (that is involves the raising to the square) while the linear constraint `Constr` is of the usual type, called the trivial type. In fact, the trivial type uses the identity (or trivial) transformation of its implicit group variable, hence its name. Assigning types to groups is one of the purposes of yet another new section, called `GROUP USES`, which can be written as follows.

```
GROUP USES
 XT Object    SQUARE
```

Again we used the appropriate keyword to start the section, and specified that the objective group `Object` is of Type `SQUARE`, using the code `XT`. Observe that we did not assign any type to the group `Constr`, *in which case the type* trivial *is assumed.*

One may then think that we have specified everything, but this not quite the case: we still have to define what we mean when we say that the group type `SQUARE` of group variable `ALPHA` is equal (in our simple example) to the square of that variable. We also have to specify the derivatives of this associated function in a form that can be converted to a useful (Fortran) subroutine at a later stage. Specifying the actual functional form and derivatives of the nonlinear elements is done in another chapter of our problem file: the SGIF (Standard *Group* Input Format). In fact, the SDIF and SGIF parts of our problem specification are conceptually quite different and it is even possible to keep them in two different files! We will however avoid this complication here, and simply append the SGIF for our problem to its SDIF definition. The SGIF starts with the keyword `GROUPS` followed by the problem name (in field 3), i.e.

```
GROUPS        TAME
```

We next specify the individual nontrivial groups of our problem (just one, in our example) in a section of the SGIF called `INDIVIDUALS` as follows:

```
INDIVIDUALS
 T  SQUARE
 F                      ALPHA**2
 G                      2.0 * ALPHA
 H                      2.0
ENDATA
```

The code `T` indicates the type of group we are about to describe: this is the `SQUARE` type in our case. The functional expression for the group (as a function of its group variable(s)) is then specified in a line starting with the code `F` (as in `F`unction). Its gradient with respect to its (unique) variable is then specified in the line starting with the code `G` (as in `G`radient). Its second derivative is next specified in the line starting with the code `H` (as in `H`essian). Finally, and as for the SDIF, the SGIF is ended with the `ENDATA` keyword.

Gathering all this new material, we obtain the following specification file.

```
NAME          TAME

*   A simple constrained linear least-squares problem

VARIABLES
    x
    y
GROUPS
```

```
 XN Object    x          1.0            y          -1.0
 XE Constr    x          1.0            y          1.0
CONSTANTS
 X  TAME      Constr     1.0
GROUP TYPE
 GV SQUARE    ALPHA
GROUP USES
 XT Object    SQUARE
ENDATA

GROUPS        TAME
INDIVIDUALS
 T  SQUARE
 F                       ALPHA**2
 G                       2.0 * ALPHA
 H                       2.0
ENDATA
```

After saving these lines in the file TAME.DAT, we can once more verify that LANCELOT can solve the problem for us by simply issuing the command

```
sdlan TAME
```

and verifying that our faithfull knight has once more done his job.

The observant reader surely noticed that we have broken one more of our conventions of the first section: in the line

```
 G                       2.0 * ALPHA
```

for instance, we see that field 4 does not contain a number, but instead seems to contain an arithmetic expression involving the group variable ALPHA! In fact, the line format for the functions' specification within the INDIVIDUALS section of the SGIF is different from that in the SDIF, and is now of the form

```
F1 <---F2---><---F3---><------------------F7------------------>
cd <--name--><--name--><--------- Fortran expression ---------->
```

where the new field 7 is 41 characters long, starts in column 25 (as field 4) and ends in column 65. Field 7 may contain any Fortran expression for defining function, gradient or Hessian values. Observe that this restriction also imposes that the (arbitrary) name given to the group variables (ALPHA, for instance) must be valid Fortran names (they cannot start by a digit, for instance).

These restrictions do not apply however to the SGIF lines specifying the type of element being defined: the element type names can be as long as 10 characters and are not restricted by the more stringent rules applying to valid Fortran names: we could choose "Alpha**2" instead of "SQUARE" without a problem.

At this point, it is interesting to note that the introduction of group functions does, in fact, provides the necessary tool for specifying *unary* problems. Unary problems have their objective and constraint functions sharing a specific structure: each of this function is a one-to-one twice differentiable real transformation of a linear expression involving the problem variables. In our example, we used the trivial and square transformations, but the mechanism is ready for handling more complex ones, as in the problem

$$\min_{x,y\in\mathbf{R}} e^{(x-2y)} \tag{16}$$

subject to the constraint

$$\sin(y-x-1)=0 \tag{17}$$

and the bounds

$$-2 \leq x \leq 2 \text{ and } -1.5 \leq y \leq 1.5. \tag{18}$$

Using the group technique, this can now be simply specified as

```
NAME          ALSOTAME

*   Another simple constrained  problem

VARIABLES
    x
    y
GROUPS
 XN Object    x         1.0            y         -2.0
 XE Constr    x         -1.0           y         1.0
CONSTANTS
 X  ALSOTAME  Constr    1.0
BOUNDS
 XL ALSOTAME  x         -2.0
 XU ALSOTAME  x         2.0
 XL ALSOTAME  y         -1.5
 XU ALSOTAME  y         1.5
GROUP TYPE
 GV EXPN      ALPHA
 GV SINE      ALPHA
GROUP USES
 XT Object    EXPN
 XT Constr    SINE
ENDATA

GROUPS        ALSOTAME
TEMPORARIES
 R  EXPA
 R  SINA
 M  EXP
 M  SIN
 M  COS
INDIVIDUALS
 T  EXPN
 A  EXPA                EXP( ALPHA )
 F                      EXPA
 G                      EXPA
 H                      EXPA

 T  SINE
 A  SINA                SIN( ALPHA )
 F                      SINA
 G                      COS( ALPHA )
 H                      - SINA
ENDATA
```

Notice the additional section we have introduced in the SGIF, called `TEMPORARIES`. This section is used to declare the type of temporary variables and/or functions that are used within the Fortran expressions of the `INDIVIDUALS` section. In our example, we use the variables `EXPA` and `SINA` to avoid additional calls to the exponential or sine intrinsic functions. These variables are declared to be `R`eal by using the code `R`. Similarly, the exponential and sine are supplied by the `M`achine, hence the code `M`. The last new feature introduced in our example is the code `A` in the

INDIVIDUALS section, which is used to Assign a value to a temporary variable (EXPA and SINA in our case). All assignment lines (i.e. lines starting with code A) appear before the function line (with code F). It is also required that all statements associated with a single group type are kept together under the relevant type declaration, as in the example above. The INDIVIDUALS section

```
INDIVIDUALS
*   An example of incorrect mixing of group types
 T  EXPN
 A  SINA                 SIN( ALPHA )
 A  EXPA                 EXP( ALPHA )
 F                       EXPA
 G                       EXPA
 H                       EXPA

 T  SINE
 F                       SINA
 G                       COS( ALPHA )
 H                       - SINA
```

is therefore *incorrect*, because the variable SINA is assigned a value in the part of the section corresponding to the group type EXPN instead of the part corresponding to the SINE group type.

3.2 Using more than a single group in the objective function

Let us consider now a somewhat less (?) trivial linear least-squares problem. The problem is to fit a straight line of the form $ax + b$ to a set of five measurements at x_1, x_2, x_3, x_4 and x_5 with respectives values y_1, y_2, y_3, y_4 and y_5. We also want the slope of the straight line to be nonnegative. Furthermore, we require that the affine function $ax + b$ corresponding to this line has a value lower than a given constant c at $x = 1$. Mathematically, this can expressed as

$$\min_{a,b\in\mathbf{R}} \frac{1}{2}\sum_{i=1}^{5}(ax_i + b - y_i)^2 \tag{19}$$

subject to the bound

$$a \geq 0 \tag{20}$$

and the additional constraint

$$a + b \leq c. \tag{21}$$

The technique is now to *use several groups to describe the objective function* We will choose the groups to correspond to each one of the five terms in our objective (19), which gives us the following specification

```
NAME          LSQFIT

*   An elementary constrained linear least-squares fit

*   Data points
 RE X1                   0.1
 RE X2                   0.3
 RE X3                   0.5
 RE X4                   0.7
 RE X5                   0.9
*   Observed values
 RE Y1                   0.25
 RE Y2                   0.3
```

```
 RE Y3                  0.625
 RE Y4                  0.701
 RE Y5                  1.0
*   Upper bound at 1.0
 RE C                   0.85
VARIABLES
    a
    b
GROUPS
*   Objective function groups
 ZN Obj1      a                        X1
 XN Obj1      b         1.0
 XN Obj1      'SCALE'   2.0
 ZN Obj2      a                        X2
 XN Obj2      b         1.0
 XN Obj2      'SCALE'   2.0
 ZN Obj3      a                        X3
 XN Obj3      b         1.0
 XN Obj3      'SCALE'   2.0
 ZN Obj4      a                        X4
 XN Obj4      b         1.0
 XN Obj4      'SCALE'   2.0
 ZN Obj5      a                        X5
 XN Obj5      b         1.0
 XN Obj5      'SCALE'   2.0
*   Constraint group
 XL Cons      a         1.0            b         1.0
CONSTANTS
 Z  LSQFIT    Obj1                     Y1
 Z  LSQFIT    Obj2                     Y2
 Z  LSQFIT    Obj3                     Y3
 Z  LSQFIT    Obj4                     Y4
 Z  LSQFIT    Obj5                     Y5
 Z  LSQFIT    Cons                     C
BOUNDS
 XR LSQFIT    b
GROUP TYPE
 GV SQUARE    ALPHA
GROUP USES
 XT Obj1      SQUARE
 XT Obj2      SQUARE
 XT Obj3      SQUARE
 XT Obj4      SQUARE
 XT Obj5      SQUARE
ENDATA

GROUPS        LSQFIT
INDIVIDUALS
 T  SQUARE
 F                      ALPHA * ALPHA
 G                      ALPHA + ALPHA
 H                      2.0
ENDATA
```

We added one further new construct in this specification: the scaling factors for the groups. You have noticed the line

```
XN Obj1       'SCALE'   2.0
```

and other similar ones in the GROUPS section. This line indicates that the group Obj1 is to be *divided* ("SCALEd") by the factor 2.0 (that is multiplied by 0.5!) before it is incorporated in the objective function. The string 'SCALE' is a reserved name[14] to indicate this use of the value that is specified further in the line. Such group scaling factors can also be specified via a named parameter, as would be the case if we had written

```
RE two                  2.0
ZN Obj1       'SCALE'                   two
```

using an Z-type code instead of an Z-type one. Such a scaling may of course be specified for any group, including constraint groups, if any. We could of course incorporate the factor $\frac{1}{2}$ (common to all groups) in the definition of the group function itself, but we would then miss the opportunity to illustrate an interesting construct...

Also note that we just defined more than one objective group (as announced), using the codes ZN and XN for each of the five groups Obj1...Obj5. We have interspersed X and Z codes, just to refresh your mind about their differences (X for numerical values, Z for values via parameters). Also observe that, although there are five nonlinear groups, we only had to define the SQUARE group type once, a very handy feature when the number of observed values increases. We did not declare any type for the constraint group in the GROUP USES section because this group is of trivial type, and therefore needs no explicit typing. Finally, we have defined all the problem dependent data (data points and observed values) at the beginning of the file, which makes later modifications easy.

3.3 Robust regression and conditional statements in the SGIF

The SQUARE group type is probably the most common in practical applications, just because least-square calculations are so frequently occurring in many fields of science. It is interesting to note that robust regression techniques (see [13] for more details) can also be accommodated quite easily in this framework. For instance, consider the LSQFIT problem with the Huber loss function

$$h_k(\alpha) = \begin{cases} \frac{1}{2}\alpha^2 & \text{for } |\alpha| \leq k \\ k|\alpha| - \frac{1}{2}k^2 & \text{for } |\alpha| > k \end{cases} \tag{22}$$

instead of the ℓ_2 measure of (19). The problem is now expressed as

$$\min_{a,b\in\mathbf{R}} \sum_{i=1}^{5} h_k(ax_i + b - y_i) \tag{23}$$

(where h is defined in (22)) subject to the bound (20).

In fact, we just have to introduce (22) as a group type. This raises the apparent difficulty that the Huber function involves a conditional statement, which we have not yet mentioned in the SGIF syntax. You will be relieved to learn that such a statement nevertheless exists. The idea is to define a few additional temporary variables in the TEMPORARIES section, of which one (OUT, say) is a *logical variable*, and then assign "true" or "false" to it (using a Fortran statement) in the INDIVIDUALS section, depending whether or not the condition

$$|\alpha| > k \tag{24}$$

holds. We then write

[14] Defined in MPS.

```
A  HUBERK              1.5
A  ABSA                ABS( ALPHA )
A  OUT                 ABSA .GT. HUBERK
```

We may then compute the value of the function and derivatives depending on the value of the logical variable OUT. This is what is specified on the following lines

```
I  OUT       FF        HUBERK * ABSA - 0.5 * HUBERK * HUBERK
E  OUT       FF        0.5 * ABSA * ABSA
```

In other words, If OUT is true, then FF is set to the value of the Huber function when (24) holds, Else (i.e. if OUT is false), FF is set to the alternate value. The principle is identical for the first derivative, except that we have the additional problem that the sign of α may now be important when (24) holds. We therefore write

```
A  NEGOUT              OUT .AND. ( ALPHA .LT. 0.0 )
A  POSOUT              OUT .AND. ( ALPHA .GE. 0.0 )
I  POSOUT    GG        HUBERK
I  NEGOUT    GG        - HUBERK
E  OUT       GG        ALPHA
```

which puts the first derivative value in the temporary variable GG. Finally, the second derivative is obtained in HH much in the same way

```
I  OUT       HH        0.0
E  OUT       HH        1.0
```

Of course, we must declare all these temporary variables in the TEMPORARIES section:

```
TEMPORARIES
 L  OUT
 L  NEGOUT
 L  POSOUT
 R  ABSA
 R  HUBERK
 R  FF
 R  GG
 R  HH
 M  ABS
```

where the code L is used to declare that OUT, NEGOUT and POSOUT are logical variables. Putting the complete problem together, we finally obtain

```
NAME          HUBFIT

*   An elementary fit using the Huber loss function
...
(problem data, VARIABLES, GROUPS, CONSTANTS and BOUNDS sections as above)
...
GROUP TYPE
 GV HUBER     ALPHA
GROUP USES
 XT Obj1      HUBER
 XT Obj2      HUBER
 XT Obj3      HUBER
 XT Obj4      HUBER
 XT Obj5      HUBER
ENDATA
```

```
GROUPS        HUBFIT
TEMPORARIES
 L  OUT
 L  NEGOUT
 L  POSOUT
 R  ABSA
 R  HUBERK
 R  FF
 R  GG
 R  HH
 M  ABS
INDIVIDUALS
 T  HUBER
 A  HUBERK              1.5
 A  ABSA                ABS( ALPHA )
 A  OUT                 ABSA .GT. HUBERK
*  Case 1: alpha is outside [-k, k]
 I  OUT       FF        HUBERK * ABSA - 0.5 * HUBERK * HUBERK
 A  NEGOUT              OUT .AND. ( ALPHA .LT. 0.0 )
 A  POSOUT              OUT .AND. ( ALPHA .GE. 0.0 )
 I  POSOUT    GG        HUBERK
 I  NEGOUT    GG        - HUBERK
 I  OUT       HH        0.0
*  Case 2: alpha is inside [-k, k]
 E  OUT       FF        0.5 * ABSA * ABSA
 E  OUT       GG        ALPHA
 E  OUT       HH        1.0
*  Assign the final values
 F                      FF
 G                      GG
 H                      HH
ENDATA
```

We have simply reordered the statements to group the cases and added a few comments.

Of course, the group function has discontinuous second derivatives at $-k$ and k, but fortunately LANCELOT can still cope with this problem as trying out quickly reveals...

Two final remarks on the SGIF part of the specification file:

- The SGIF in fact contains one more (optional) section, called `GLOBALS`. This section is intended to contain the assignment of *global variables* whose value does not depend on any particular type of group. We could use this section in our example, because there is only one group type, and thus the parameter `HUBERK` does not depend on the group type and could be assigned a value in the `GLOBALS` section. If we we make this (unnecessarily verbose) choice, the new section is simply

  ```
  GLOBALS
   A  HUBERK              1.5
  ```

  We can then remove the statement assigning 1.5 to `HUBERK` from the `INDIVIDUALS` section.

  Note that the `GLOBALS` section uses the same format as the `INDIVIDUALS` one and that any variable appearing in field 2 must have been declared in the `TEMPORARIES` section. Interestingly, the conditional assignment lines with codes `I` and `E` can also be used in the `GLOBALS` section.

The practical use of this technique is of doubtful interest in our very simple case, but can bring advantages when there are several group types within the same problem that involve complicated common subexpressions.

- Although it did not happen in either our simple example, it may well be the case that the Fortran expression for the group value or for one of its derivatives is too long to fit in field 7 on a single line. The solution is obviously to write more than one line, all "continuation lines" starting with the codes `I+`, `E+`, `A+`, `F+`, `G+`, or `H+`, depending on the type of expression they provide continuation for. For instance, a peculiar but correct way to write

```
I  OUT       FF        HUBERK * ABSA - 0.5 * HUBERK * HUBERK
```

could be

```
I  OUT       FF        HUBERK * ABSA
I+                                   - 0.5 * HUBERK * HUBERK
```

There may be up to 19 such continuation lines[15].

3.4 Summary

In this section, we have introduced the groups and their associated group functions. We saw that the SIF is able to represent the problem of minimizing a nonlinear objective function containing one or more groups subject bounds on the variables and to nonlinear constraints, each of which corresponds to separate group. Mathematically speaking the problem is of the form

$$\min_{x\in\mathbf{R}^n} \sum_{i=1}^{p} \frac{1}{s_i} g_i(a_i^T x - b_i) \tag{25}$$

subject to the constraints

$$\frac{1}{s_i} g_i(a_i^T x - b_i) \left\{ \begin{array}{c} \leq \\ = \\ \geq \end{array} \right\} 0 \quad (i = p+1, \ldots, m) \tag{26}$$

and to the bounds

$$l_j \leq x_j \leq u_j \quad (j = 1, \ldots, n). \tag{27}$$

The specification of the group functions $g_i(\cdot)$ is made in two steps: group function types are first declared in the SDIF chapter of the file and each group assigned its type; the group functions associated to each type are subsequently described (i.e. expressions for their value and derivatives are specified) in the SGIF chapter of the problem file. This latter part provides special sections

- to declare temporary variables and user or machine supplied functions (the `TEMPORARIES` section),
- to assign values to global temporary variables (the `GLOBALS` section),
- to assign the temporary variables, group function value, first and second derivative using Fortran assignment statements (the `INDIVIDUALS` section).

Furthermore, the `GLOBALS` and `INDIVIDUALS` sections also provide a mechanism allowing for the conditional assignment of variables, which in turn allows the specification of piecewise group functions. These conditional assignments use logical variables and "if-then-else" codes.

[15] This limit is due to Fortran.

4 Introducing more complex nonlinearities: the element functions

At last, we are ready to introduce the not-so-gentle (perhaps, even vicious) problems, that is problems involving more serious nonlinearities. In general, the use of the group functions alone will not be sufficient to cover the more complex cases we will examine in this section. Instead, or often besides, we will need additional ways of specifying nonlinear components of our objective and/or constraint functions.

4.1 A classic: the Rosenbrock "banana" unconstrained problem

As before, we will introduce the necessary concepts with the help of an example. In this case, we will choose a classic[16]: the famous Rosenbrock unconstrained minimization problem in two variables. Mathematically, the problem is the following:

$$\min_{x,y \in \mathbf{R}} 100(y - x^2)^2 + (x-1)^2. \tag{28}$$

It is sometimes called the "banana valley" problem because drawing the level curves of (28) reveals a narrow curved valley (induced by the first term in the expression) whose bottom has itself a comparatively gently curvature (the effect of the second term), the resulting level curves taking the form of nested "bananas".

Like a large percentage of nonlinear problems, this example has the least-squares structure, but the terms inside the squares are no longer all linear: the first one indeed involves the square of the variable x. Therefore, we will want to use the least-squares group function, but we will also need additional constructs to take the more complicated structure into account. Proceeding cautiously, we may first want to specify as much of the problem as we can with the techniques that we have learned already. This will give a file of the form

```
NAME          ROSENBR

*   A first attempt to specify the famous Rosenbrock problem

VARIABLES
    x
    y
GROUPS
 XN Obj1      y          1.0
 XN Obj1      'SCALE'    0.01
 XN Obj2      x          1.0
CONSTANTS
 X  ROSENBR   Obj2       1.0
BOUNDS
 XR ROSENBR   x
 XR ROSENBR   y
GROUP TYPE
 T  SQUARE    ALPHA
GROUP USES
 T  Obj1      SQUARE
 T  Obj2      SQUARE
ENDATA

GROUPS        ROSENBR
```

[16]To nonlinear programmers, of course.

```
INDIVIDUALS
 T  SQUARE
 F                     ALPHA * ALPHA
 G                     ALPHA + ALPHA
 H                     2.0
ENDATA
```

We now would like to indicate that group `Obj1` is not as simple and that it contains a nonlinear *element function*, as we call such troublesome terms. You will not be surprised to hear that the task is achieved by specifying *element types* in a section called `ELEMENT TYPE` , and then assigning these types to the actual element functions that appear in the different groups[17]. More precisely, we write the following

```
ELEMENT TYPE
 EV SQ         V
```

, for the `ELEMENT TYPE` section, thereby specifying that we consider a (single) element type called `SQ`[18] depending on a single *elemental variable* `V`. At this stage, this elemental variable `V` is floating around freely, and we now must define the particular problem variable (`x` or `y`) that will be assigned to `V`. This is the task of the `ELEMENT USES` section, which, in our case, reads as follows

```
ELEMENT USES
 T  XSQ        SQ
 ZV XSQ        V                        x
```

The line starting with the code `T` assigns the element type `SQ` to the element named `XSQ`, while the line starting with the code `ZV` assigns the problem variable `x` to the the elemental variable `V` *for the particular element* `XSQ`.

The next task is then to say that group `Obj1` involves the element `XSQ`, which is done in the `GROUP USES` section. This latter section thus becomes

```
GROUP USES
 T  Obj1       SQUARE
 XE Obj1       XSQ       -1.0
 T  Obj2       SQUARE
ENDATA
```

where the code `XE` indicates that we assign Element functions to a group (in our case `XSQ` to the group `Obj1`), after multiplication by the *weighting factor* -1.0.

Finally, we still have to specify the actual nonlinear behaviour that we have hidden behind the definition of the element types. As was the case for the group functions, this is done in another chapter of the problem specification file, called the Standard Element Input Format (SEIF), whose syntax is very similar to that of the SGIF. For the Rosenbrock function, this SEIF is as follows

```
ELEMENTS       ROSENBR
INDIVIDUALS
 T  SQ
 F                       V**2
 G  V                    2.0 * V
 H  V          V         2.0
ENDATA
```

[17] The parallel with the groups is hopefully obvious.

[18] We used `SQ` instead of `SQUARE` just to avoid the confusion between the group and element types, but using the same name is perfectly legal.

The only difference (beyond the different names associated with the type and involved variable) between the SEIF and the the least-square SGIF is the fact that an element type might involve several variables (in more complicated cases, see below), and therefore that it is necessary to indicate with respect to which variable differentiation is performed. Here, it is of course with respect to `V` for the gradient, and with respect to `V` and `V` for the Hessian. At variance with the group function specification, the gradient and Hessian specification are optional in the SEIF. However, it is our experience that it is very often helpful to provide expressions for gradient and Hessian entries: it usually pays off in faster and more robust convergence of LANCELOT[19]. *We therefore recommend that expressions for gradient and Hessian components be supplied whenever possible.*

Assembling the pieces of the puzzle, we thus obtain the complete SIF specification for the Rosenbrock problem which is given below.

```
NAME          ROSENBR

*   The famous Rosenbrock problem

VARIABLES
    x
    y
GROUPS
 XN Obj1      y         1.0
 XN Obj1      'SCALE'   0.01
 XN Obj2      x         1.0
CONSTANTS
 X  ROSENBR   Obj2      1.0
BOUNDS
 XR ROSENBR   x
 XR ROSENBR   y
START POINT
 XV ROSENBR   x         -1.2
 XV ROSENBR   y         1.0
ELEMENT TYPE
 EV SQ        V
ELEMENT USES
 T  XSQ       SQ
 ZV XSQ       V                       x
GROUP TYPE
 T  SQUARE    ALPHA
GROUP USES
 T  Obj1      SQUARE
 XE Obj1      XSQ       -1.0
 T  Obj2      SQUARE
ENDATA

ELEMENTS      ROSENBR
INDIVIDUALS
 T  SQ
 F                      V**2
 G  V                   2.0 * V
 H  V         V         2.0
ENDATA

GROUPS        ROSENBR
```

[19]And also of all algorithms that can exploit user-supplied exact derivative information.

```
INDIVIDUALS
 T  SQUARE
 F                       ALPHA**2
 G                       2.0 * ALPHA
 H                       2.0
ENDATA
```

(We have also specified the classical starting point for this problem). Note that the ELEMENT TYPE and ELEMENT USES section immediately precede the GROUP TYPE and GROUP USES sections, and also that, by convention, the SEIF is written between the SDIF and the SGIF.

Clearly, the complexity of this specification may appear formidable for such a small example: comparing equation (28) and the above file for conciseness is not to the advantage of the latter. However, this simple case has been enough for us to become acquainted with a very general way of analyzing and describing a problem that will pay off many times when dealing with larger and more complex optimization problems.

4.2 The internal dimension and internal variables

There is one more structure to analyze before we move to the techniques allowing the specification of really large problems, namely the structure of the *internal variables* and *internal dimension.* As always, this structure is best introduced by an example.

Let us consider the (again very simple) problem of finding the pair of symmetric 2 by 2 matrices whose Frobenius distance is minimum and such that the first is positive semidefinite while the second is negative semidefinite. Once, more we express the problem in its mathematical form, which is, in this case,

$$\min_{x_{i,j},y_{i,j}\in\mathbf{R}} (x_{11}-y_{11})^2+2(x_{12}-y_{12})^2+(x_{22}-y_{22})^2 \tag{29}$$

subject to the constraints

$$x_{11}x_{22}-x_{12}^2\geq 0, \tag{30}$$

$$y_{11}y_{22}-y_{12}^2\leq 0 \tag{31}$$

and to the bounds

$$x_{11}\geq 0, \quad x_{22}\geq 0 \tag{32}$$

and

$$y_{11}\leq 0, \quad y_{22}\leq 0. \tag{33}$$

Note that the bounds on the diagonal entries of X (resp. Y) together with the determinant constraint (30) (resp. (31)) are sufficient to ensure the positive (resp. negative) semidefiniteness of this matrix[20]. Although the mathematical solution of this problem is immediate, its formulation in the above form is not so trivial (it indeed involves a nonlinear objective function and two nonlinear constraints), and will illustrate quite well how the different SIF construct can contribute in defining a non too simple nonlinear minimization problem.

We first perform some now familiar tasks: we give a name to the problem, declare the problem variables, specify the nature of the three groups, state the bounds and define a starting point:

```
NAME          MATRIX2

*   A small nonlinear matrix problem

VARIABLES
```

[20]The old Sylvester theorem.

```
      X11
      X12
      X22
      Y11
      Y12
      Y22
GROUPS
 XN Frobdist
 XG Xposdef
 XL Ynegdef
BOUNDS
 XR MATRIX2   X12
 XM MATRIX2   Y11
 XR MATRIX2   Y12
 XM MATRIX2   Y22
START POINT
 XV MATRIX2   X11       1.0
 XV MATRIX2   X12       1.0
 XV MATRIX2   X22       1.0
 XV MATRIX2   Y11       1.0
 XV MATRIX2   Y12       1.0
 XV MATRIX2   Y22       1.0
```

We then examine the objective function (29), which constitutes its own group of trivial type. This implies that the group value is composed of three nonlinear element functions, each one of the form $(v-w)^2$. Examining this last formula, we quickly see that it consists of a nonlinear function (the square) of a linear combination of the variables appearing in the element[21]. In fact, we can write the simple transformation

$$(v-w)^2 = u^2 \tag{34}$$

where we define the *internal variable* u by

$$u = v - w. \tag{35}$$

This of course means that the derivatives of the element function $(v-w)^2$ have some additional structure that we want to exploit. More precisely, it is sufficient to specify the derivatives of u^2 with respect to the internal variable u and (35) instead of specifying the derivatives of $(v-w)^2$ with respect to v and w. The number of internal variables of an element type is called its *internal dimension*, as opposed to the number of elemental variables, which is its *elemental dimension*.

In the `ELEMENT TYPE` section, we therefore declare that our element function type has two `E`lemental `V`ariables, named `V` and `W`, and also that it has one `I`nternal `V`ariable named `U`. This section therefore consists of

```
ELEMENT TYPE
 EV ISQ       V                        W
 IV ISQ       U
 EV SQ        X
 EV 2PR       X                        Y
```

where we have added the declaration of an element type called `SQ` for the `SQ`uare appearing in the second terms of (30) and (31), and an element type called `2PR` for the products appearing as first terms of the same constraint equations.

[21]The similarity with the groups containing linear combinations of the variables must be striking to the attentive reader. The only difference is that groups also contain nonlinear elements besides their linear part.

Each of the elements (three for the objective function and two for each constraint) must then be named explicitly and their elemental variables assigned the correct problem variables. This is done in the ELEMENT USES section.

```
ELEMENT USES
*   Elements of the objective
 XT XY11SQ    ISQ
 ZV XY11SQ    V                        X11
 ZV XY11SQ    W                        Y11
 XT XY12SQ    ISQ
 ZV XY12SQ    V                        X12
 ZV XY12SQ    W                        Y12
 XT XY22SQ    ISQ
 ZV XY22SQ    V                        X22
 ZV XY22SQ    W                        Y22
*   Constraint that det(X) >= 0
 XT X1122     2PR
 ZV X1122     X                        X11
 ZV X1122     Y                        X22
 XT X12SQ     SQ
 ZV X12SQ     X                        X12
*   Constraint that det(Y) <= 0
 XT Y1122     2PR
 ZV Y1122     X                        Y11
 ZV Y1122     Y                        Y22
 XT Y12SQ     SQ
 ZV Y12SQ     X                        Y12
```

Notice that we only needed three element types for seven different element functions.

We then assign all these elements to their groups:

```
GROUP USES
*   Objective function
 XE Frobdist  XY11SQ                   XY22SQ
 XE Frobdist  XY12SQ    2.0
*   det(X) >= 0
 XE Xposdef   X1122                    X12SQ     -1.0
*   det(Y) <= 0
 XE Ynegdef   Y1122                    Y12SQ     -1.0
```

Observe that we omitted the weighting factors whose value is equal to 1.0. This is allowed because *all unspecified element weighting factors are assumed to be 1.0.*

Finally, we specify the (very simple) nonlinear nature of the elements and groups, and obtain

```
NAME          MATRIX2
VARIABLES
    X11
    X12
    X22
    Y11
    Y12
    Y22
GROUPS
 XN Frobdist
 XG Xposdef
 XL Ynegdef
BOUNDS
```

```
 XR MATRIX2   X12
 XM MATRIX2   Y11
 XR MATRIX2   Y12
 XM MATRIX2   Y22
START POINT
 XV MATRIX2   X11       1.0
 XV MATRIX2   X12       1.0
 XV MATRIX2   X22       1.0
 XV MATRIX2   Y11       1.0
 XV MATRIX2   Y12       1.0
 XV MATRIX2   Y22       1.0
ELEMENT TYPE
 EV ISQ       V                        W
 IV ISQ       U
 EV SQ        X
 EV 2PR       X                        Y
ELEMENT USES
*   Elements from the objective
 XT XY11SQ    ISQ
 ZV XY11SQ    V                        X11
 ZV XY11SQ    W                        Y11
 XT XY12SQ    ISQ
 ZV XY12SQ    V                        X12
 ZV XY12SQ    W                        Y12
 XT XY22SQ    ISQ
 ZV XY22SQ    V                        X22
 ZV XY22SQ    W                        Y22
*   det(X) >= 0
 XT X1122     2PR
 ZV X1122     X                        X11
 ZV X1122     Y                        X22
 XT X12SQ     SQ
 ZV X12SQ     X                        X12
*   det(Y) <= 0
 XT Y1122     2PR
 ZV Y1122     X                        Y11
 ZV Y1122     Y                        Y22
 XT Y12SQ     SQ
 ZV Y12SQ     X                        Y12
GROUP USES
*   Objective function
 XE Frobdist  XY11SQ                   XY22SQ
 XE Frobdist  XY12SQ    2.0
*   det(X) >= 0
 XE Xposdef   X1122                    X12SQ     -1.0
*   det(Y) <= 0
 XE Ynegdef   Y1122                    Y12SQ     -1.0
ENDATA

ELEMENTS      MATRIX2
INDIVIDUALS
*   Square of an internal variable equal to the difference between
*   two elemental variables
 T  ISQ
 R  U         V         1.0            W         -1.0
 F                      U * U
```

```
 G  U                   U + U
 H  U         U         2.0
*   Square of an elemental variable
 T  SQ
 F                      X * X
 G  X                   X + X
 H  X         X         2.0
*   Product of 2 elemental variables
 T  2PR
 F                      X * Y
 G  X                   Y
 G  Y                   X
 H  X         Y         1.0
ENDATA
```

The only new feature appears in the INDIVIDUALS section of the SEIF, where we find the line

```
 R  U         V         1.0          W         -1.0
```

within the specification of the element type ISQ. As you would expect from the names, this lines just specifies the relation (35), that is the particular linear combination of the elemental variables V and W that gives the internal variable U. The code R is because this transformation is often called a Range transformation (see [10]).

Notice finally that we have not specified the four entries of the 2 by 2 Hessian matrix of the element type 2PR. The rules that govern the specification of derivatives in the SEIF are as follows.

1. The Hessian or gradient of the function associated element type is assumed to be unavailable unless at least one of its components is specified. The explicit specification of a single entry in a gradient or Hessian declares this gradient or Hessian to be available.
2. Either the i, j-th or the j, i-th entry of a Hessian matrix may be specified.
3. All entries of an available gradient or Hessian that are not explicitly specified are assumed to be identically zero.

Note that the last rule allows a description of the sparsity pattern of derivatives. However, we recommend that this technique is not used to specify complete problem structure instead of the more powerful analysis of the problem in terms of groups and elements.

4.3 Summary

We have introduced the nonlinear elements and their related concepts and constructs. This extends the class of problems that SIF/LANCELOT can handle to

$$\min_{x \in \mathbf{R}^n} \sum_{i=1}^{p} \frac{1}{s_i} g_i \left(a_i^T x - b_i + \sum_{j \in J_i} w_j f_j(x) \right) \tag{36}$$

subject to the constraints

$$\frac{1}{s_i} g_i \left(a_i^T x - b_i + \sum_{j \in J_i} w_j f_j(x) \right) \left\{ \begin{array}{c} \leq \\ = \\ \geq \end{array} \right\} 0 \quad (i = p+1, \ldots, m) \tag{37}$$

and to the bounds

$$l_j \leq x_j \leq u_j \quad (j = 1, \ldots, n). \tag{38}$$

The $f_j(x)$ are the nonlinear element functions with their associated weighting factors w_j.

In order to specify the nonlinear element functions, six steps should be followed:

1. First specify suitable element types with their elemental and internal variables (in the `ELEMENT TYPE` section),
2. Assign a type to each element function appearing in the problem (in the `ELEMENT USES` section),
3. Assign the relevant problem variables to the elemental variables of the corresponding element type (in the `ELEMENT USES` section),
4. Assign the elements to their group(s)[22], together with their associated weighting factors (in the `GROUP USES` section),
5. State the particular linear combinations of the elemental variables that define the internal variables associated with the element types, if any (in the `INDIVIDUALS` section of the SEIF),
6. Specify expressions for calculating the value (and derivatives) of the nonlinear functions associated with the element types (in the `INDIVIDUALS` section of the SEIF).

5 Tools of the trade: loops, indexing and default values

Although specifying small problems using the constructs detailed above is acceptably concise, the sheer size of the specification soon becomes unbearable with larger ones. The specification of problem variables (one per line) is, for instance, unacceptably painful if there are many variables. The purpose of this section is to present the tools available within the SDIF that allow a much more flexible statement of problems involving many variables and/or many constraints. These tools essentially consist of facilities to index SDIF names (for variables, constraints,bounds, constants, elements, groups,...), constructs to specify loops on these indices, and codes that allow the setting of default values for many SDIF objects at once.

To be consistent with our past approach, we motivate the introduction of these features via an example: we consider here a classical problem in nonlinear variational calculations: the minimum surface problem. This problem is one of finding the surface of minimum area that interpolates a given continuous function on the boundary of the unit square. The unit square itself is discretized uniformly into p^2 smaller squares, as shown in Figure 2 with its associated labelling.

The variables of the problem will be chosen as the "height" of the unknown surface above the $(p+1)^2$ vertices of the p^2 smaller squares. The area of the surface above the (i,j)-th discretized square is then approximated by the formula

$$s_{i,j}(x_{i,j}, x_{i+1,j}, x_{i,j+1}, x_{i+1,j+1}) = \frac{1}{p^2}\sqrt{1+\frac{p^2}{2}[(x_{i,j}-x_{i+1,j+1})^2+(x_{i,j+1}-x_{i+1,j})^2]}, \tag{39}$$

where the squares and variables are indexed as shown in Figure 2. The variables in the sets

$$\{x_{1,j}\}_{j=1}^{p+1}, \quad \{x_{i,p+1}\}_{i=1}^{p+1}, \quad \{x_{p+1,j}\}_{j=1}^{p+1}, \text{ and } \{x_{i,1}\}_{i=1}^{p+1}, \tag{40}$$

correspond to the boundary conditions and are assigned given values, while the others are left free. The complete problem is then to minimize the objective function

$$f(x) = \sum_{i,j=1}^{p} s_{i,j}(x_{i,j}, x_{i+1,j}, x_{i,j+1}, x_{i+1,j+1}) \tag{41}$$

[22] Yes, an element may be assigned to more than one group!

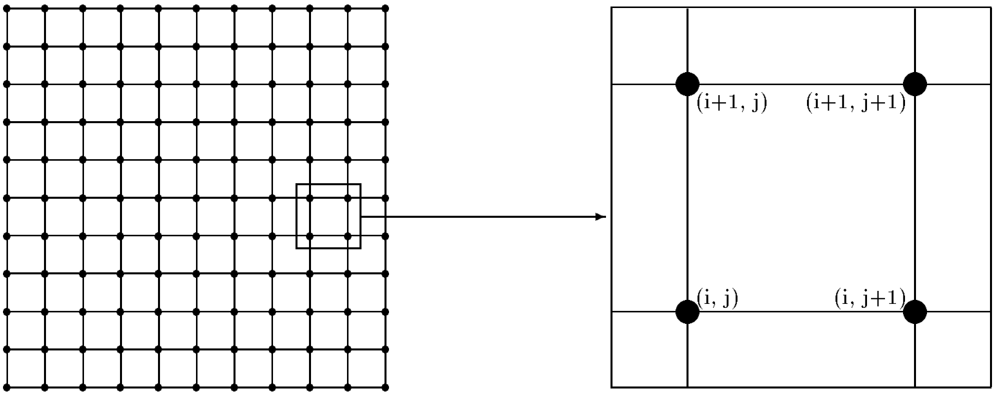

Figure 2: Discretisation of the unit square

over all the free variables[23].

If we now examine the function $s_{i,j}$ in more detail, we easily see that we can decompose it in a nonlinear group function of the form

$$g_{i,j}(y) = \frac{1}{p^2}\sqrt{y}, \tag{42}$$

where y (the group variable) is given by

$$y = 1 + \frac{p^2}{2}(u_{i,j}^2 + v_{i,j}^2) \tag{43}$$

with

$$u_{i,j} \stackrel{\mathrm{def}}{=} x_{i,j} - x_{i+1,j+1} \ \text{ and } \ v_{i,j} = x_{i,j+1} - x_{i+1,j}. \tag{44}$$

Each group thus contains two nonlinear elements of a unique type.

To illustrate the concepts of this section, we consider the simple case where the boundary condition (that is the height of the surface above the frontier of the unit square) is identically equal to 1. Of course, it does not make much sense to subdivide the unit square into four or nine discretized squares. The smallest reasonable value of p might be seven, which means that the problem has 64 variables appearing in 98 nonlinear elements, themselves merged in 49 groups.

5.1 Indexed variables and loops

When attempting to specify the minimum surface problem, it immediately appears that one would like to use variables with (double) indices, $x_{i,j}$ say. This is the case in fact for most of the large nonlinear problems arising from the discretization of an infinite dimensional variational calculation. Given the importance of this area, indexed variables are therefore a must, and have consequently been allowed within the SDIF syntax. Since it wasn't much more difficult, the SDIF also allows the possibility of indexing not only variable names, but most names that appear in the SDIF syntax.

In our minimum surface example, we would like to use a two dimensional array of variables of the form `X(i,j)`, which would nicely correspond to our mathematical statement. Furthermore,

[23] Various obstacle problems can also be obtained by specifying suitable bounds on the variables.

we also need to handle this array inside loops that would save us from writing down every single case. This is exactly what we do when we write the first lines of our problem specification as follows:

```
NAME          MINSURF
*   Discretization parameter
 IE P                    7
*   Other parameters
 IE 1                    1
 IA P+1       P          1
VARIABLES
 DO i         1                            P+1
 DO j         1                            P+1
 X  X(i,j)
 OD j
 OD i
```

Although the syntax above might seem reasonably self explanatory[24], it nevertheless requires some explanation.

The first point is that we just introduced two *loops*, one on the integer parameter `i` and the other on the integer parameter `j`. Such loops may appear anywhere within `VARIABLES`, `GROUPS`, `CONSTANTS`, `BOUNDS`, `START POINT`, `ELEMENT USES` and `GROUP USES` sections. Starting a loop is easy. The line starting with the `DO` code, i.e.

```
 DO i         1                            P+1
```

initiates a loop on the integer parameter `i`, ranging from the value of the integer parameter whose name appears in field 3 to that of the integer parameter whose name appears in field 5. In this case, the parameter `i` will therefore take the values 1, 2, 3, 4, 5, 6, 7 and 8 successively. The loop is terminated further down by the line starting with the code `OD`, namely

```
 OD i
```

which simply indicates the end of the loop using the integer parameter `i`. All statements between theses two lines constitute the "body" of the loop.

As is clear from the example, loops can be *nested*[25] up to a depth of at most three. It is even possible to specify a loop increment different from 1, by using the code `DI` as in

```
 DO iloop     istart                       iend
 DI i         incr
 (body of the loop)
 OD i
```

However, loops must not pass across section boundaries. The loop index (`iloop`) will now take the values

$$\texttt{iloop} = \texttt{istart} + j \times \texttt{incr} \tag{45}$$

for all positive j for which `iloop` lies between (and including) `istart` and `iend`. The body of the loop is skipped if no such j exists. Note that a negative value of the *loop increment* `incr` is allowed.

Finally, we declared our two dimensional array of variables in the line

```
 X  X(i,j)
```

[24] At least to Fortranolics!

[25] It is of course forbidden to overlap loops in any other way.

Remember that we merely *give names* to our variables in this section. So how are the *names* of the indexed variables built? The rule is very simple: if the integer parameters `i` and `j` have the values 2 and 3, say, then the string `X(i,j)` is equivalent to the string `X2,3`. *The last string* (sometimes called the *realization* of the first one for the given values of the parameters) *is obtained by replacing the name of the involved parameter(s) by the string containing their value as integers and by dropping the parenthesis* `(` *and* `)`. The loop

```
DO i        1                        P
X  Z(i,j)
OD i
```

is thus entirely equivalent (when `P` has the value 7 and `j` the value 11) to

```
X  Z1,11
X  Z2,11
X  Z3,11
X  Z4,11
X  Z5,11
X  Z6,11
X  Z7,11
```

which, in the `VARIABLES` section, would define a hypothetical set of variables `Z1,11`, `Z2,11`, `Z3,11`, `Z4,11`, `Z5,11`, `Z6,11` and `Z7,11`. As the length of the names is limited to ten characters at most and since `X100,100,100` is already over this limit, the SDIF imposes a limit of three possible levels of indexing, corresponding to the three levels of loop nesting.

One additional point on indexed variables, which we also call *arrays*. You certainly noticed the use of the code `X` in the line

```
X  X(i,j)
```

although we did not use any code to declare variables in our previous examples. Of course, you wondered at the reason of this sudden introduction. The reason is that the SDIF interpreter needs a special code to realize that `X(i,j)` is to be expanded to `X12,17`, say, and is not just a fancy name with parenthesis and a comma (which is allowed in the MPS syntax). Hence, the following important rule: *every SDIF line containing an array name must start with a code whose first letter is either* `X` *or* `Z`, depending on where the associated data is to be read (see Section 2.2). The fact that we can use `X`- and `Z`-codes for ordinary (non array) names is because they are considered to be a special case of indexed names, a very handy convention. Observe that the `ELEMENT TYPE` and `GROUP TYPE` sections do not contain lines with such codes: array names are therefore not admissible in these sections[26].

Of course, we can also use indexed names for the groups (there is one group per small square, which we choose to index by the coordinates of its bottom-left corner), constants and bounds of our problem:

```
GROUPS
 RI RP        P
 R* RPSQ      RP                       RP
 DO i         1                        P
 DO j         1                        P
 XN S(i,j)
 ZN S(i,j)                             RPSQ
 OD j
 OD i
CONSTANTS
```

[26] They also exclude the use of loops.

```
 DO i         1                        P
 DO j         1                        P
 X  MINSURF   S(i,j)    -1.0
 OD j
 OD i
BOUNDS
*   Free variables inside the unit square
 IE 2                    2
 DO i         2                        P
 DO j         2                        P
 XR MINSURF   X(i,j)
 OD j
 OD i
*   Boundaries
 DO i         1                        P+1
 XX MINSURF   X(1,i)    1.0
 XX MINSURF   X(P+1,i)  1.0
 XX MINSURF   X(i,1)    1.0
 XX MINSURF   X(i,P+1)  1.0
 OD i
```

Note that the above lines stay valid if we change the value of `P` in the beginning of the file, a crucial advantage! Also note that we had to assign the value 2 to the integer parameter named `2` before using it as starting index for a loop.

Loops cannot be used (and are of debatable interest) in the `ELEMENT TYPE` section, which is then easily written as

```
ELEMENT TYPE
 EV ISQ       V                        W
 IV ISQ       U
```

where we specified the *unique* type of element present in our problem, that is the square of the difference between two variables.

Loops are however of great interest in the `ELEMENT USES` section:

```
ELEMENT USES
 DO i         1                        P
 IA i+1       i         1
 DO j         1                        P
 IA j+1       j         1
*   Diagonal
 XT A(i,j)    ISQ
 ZV A(i,j)    V                        X(i,j)
 ZV A(i,j)    W                        X(i+1,j+1)
*   Antidiagonal
 XT B(i,j)    ISQ
 ZV B(i,j)    V                        X(i,j+1)
 ZV B(i,j)    W                        X(i+1,j)
 OD j
 OD i
```

We have chosen to define `A`-elements for those involving the variables at the bottom-left and top-right corners of the discretized squares (the diagonal), and `B`-elements for those involving the other two corners (the antidiagonal). Notice the technique of defining `i+1` by incrementing `i` within the loop where it is defined.

The groups of our problem are again of a unique type (square root):

```
GROUP TYPE
 GV SQROOT    ALPHA
```

and we can use the loop mechanism again to declare the type of our groups and assign the nonlinear elements to them:

```
GROUP USES
 RM WEIGHT    RPSQ      0.5
 DO i         1                        P
 DO j         1                        P
 XT S(i,j)    SQROOT
 ZE S(i,j)    A(i,j)                   WEIGHT
 ZE S(i,j)    B(i,j)                   WEIGHT
 OD j
 OD i
```

We started by computing the weighting factor ($p^2/2$) before looping over the groups.

The only thing left is to write the SEIF and SGIF parts of the specification file, and append them to what we have written already. The complete file is thus as follows:

```
NAME          MINSURF

*   A version of the minimum surface problem
*   on the unit square with simple boundary conditions.

*   Discretization parameter
 IE P                     7
*   Other parameters
 IE 1                     1
 IA P+1       P           1
 RI RP        P
 R* RPSQ      RP                       RP
VARIABLES
 DO i         1                        P+1
 DO j         1                        P+1
 X  X(i,j)
 OD j
 OD i
GROUPS
 DO i         1                        P
 DO j         1                        P
 XN S(i,j)
 ZN S(i,j)    'SCALE'                  RPSQ
 OD j
 OD i
CONSTANTS
 DO i         1                        P
 DO j         1                        P
 X  MINSURF   S(i,j)    -1.0
 OD j
 OD i
BOUNDS
*   Free variables inside the unit square
 IE 2                     2
 DO i         2                        P
 DO j         2                        P
 XR MINSURF   X(i,j)
```

```
 OD j
 OD i
*   Boundaries
 DO i         1                        P+1
 XX MINSURF   X(1,i)     1.0
 XX MINSURF   X(P+1,i)   1.0
 XX MINSURF   X(i,1)     1.0
 XX MINSURF   X(i,P+1)   1.0
 OD i
ELEMENT TYPE
 EV ISQ       V                        W
 IV ISQ       U
ELEMENT USES
 DO i         1                        P
 IA i+1       i          1
 DO j         1                        P
 IA j+1       j          1
*   Diagonal
 XT A(i,j)    ISQ
 ZV A(i,j)    V                        X(i,j)
 ZV A(i,j)    W                        X(i+1,j+1)
*   Antidiagonal
 XT B(i,j)    ISQ
 ZV B(i,j)    V                        X(i,j+1)
 ZV B(i,j)    W                        X(i+1,j)
 OD j
 OD i
GROUP TYPE
 GV SQROOT    ALPHA
GROUP USES
 RM WEIGHT    RPSQ       0.5
 DO i         1                        P
 DO j         1                        P
 XT S(i,j)    SQROOT
 ZE S(i,j)    A(i,j)                   WEIGHT
 ZE S(i,j)    B(i,j)                   WEIGHT
 OD j
 OD i
ENDATA

ELEMENTS      MINSURF
INDIVIDUALS
 T  ISQ
 R  U         V          1.0           W          -1.0
 F                       U * U
 G  U                    U + U
 H  U         U          2.0
ENDATA

GROUPS        MINSURF
TEMPORARIES
 R  SQRAL
 M  SQRT
INDIVIDUALS
 T  SQROOT
 A  SQRAL                SQRT( ALPHA )
```

```
 F                     SQRAL
 G                     0.5 / SQRAL
 H                     -0.25 / ( ALPHA * SQRAL )
ENDATA
```

In this file, we have omitted the START POINT section, implying that the value 0 should be used for all problem variables: it is up to the minimization algorithm to first set the fixed variables to their assigned values before the minimization proceeds. LANCELOT does this, of course.

At this point, it is probably a good idea to reread the last problem specification and make sure you understand every single line of it, before we move on to further complications. It is, in particular, important to note that changing the discretization mesh is a trivial exercise with this specification (how?), and that it is simple to modify the problem to incorporate more complicated boundary conditions and/or obstacles (that is constraints on the height of the surface at certain locations within the unit square).

5.2 Setting default values

Using loops to define values or names for array components is reasonably efficient, but the SDIF offer a useful alternative. This is the possibility of defining *defaults*. This notion is again best understood via examples. For instance, the BOUNDS section of the preceding example could be written as

```
BOUNDS
*   Free variables
 FR MINSURF   'DEFAULT'
*   Boundaries
 DO i         1                        P+1
 XX MINSURF   X(1,i)     1.0
 XX MINSURF   X(P+1,i)   1.0
 XX MINSURF   X(i,1)     1.0
 XX MINSURF   X(i,P+1)   1.0
 OD i
```

which is both shorter and more elegant. The idea is that we say, in the line

```
 FR MINSURF   'DEFAULT'
```

that all variables are free. We may then modify the status of some variables (fix the boundary ones), thereby overriding the default. (Of course, the default must be set *before* any other bound specification.) Similarly, the CONSTANTS section can be written as

```
CONSTANTS
    MINSURF   'DEFAULT' -1.0
```

instead of explicitly specifying all constants individually.

Defaults can be set for a number of things within the SDIF problem specification file. Possible uses of this technique are given is Table 7.

The syntax for default setting is quite simple. The idea is that *the line defining a default value for a class of objects is identical to the line that would be used to assign a value to a single member of this class, except that the name of the class member is replaced by the string* "'DEFAULT'". In the line

```
 FR MINSURF   'DEFAULT'
```

the string 'DEFAULT' replaces the name of the variable, X say, that would be declared free by the line

Section	Uses	Field
`CONSTANTS`	Set default value for group constants	3
	Set default value for group constants	3
`BOUNDS`	Set all variables to be bounded below	3
	Set all variables to be bounded above	3
	Set all variables to fixed to a value	3
	Set all variables to be free	3
	Set all variables to be unbounded above	3
	Set all variables to be unbounded below	3
`START POINT`	Set default starting value for all variables	3
`ELEMENT USES`	Set default type for elements	2
`GROUP USES`	Set default type for groups	2

Table 7: Possible default settings

```
FR MINSURF   X
```

Similarly, one may declare that all groups are by default of the least-squares type (`SQUARE`) by the line

```
T  'DEFAULT' SQUARE
```

because declaring the group `AGROUP` to be of the same type would be written as

```
T  AGROUP    SQUARE
```

The lines

```
XL PROBLEM   'DEFAULT' 2.0
ZU PROBLEM   'DEFAULT'                UBOUND
```

are valid default definitions within the `BOUNDS` section, while

```
X  PROBLEM   'DEFAULT'  5.0
```

or

```
Z  PROBLEM   'DEFAULT'                DEFVALUE
```

are appropriate in the `CONSTANTS` section.

The syntax thus implies that the string "`'DEFAULT'`" may appear in fields 2 or 3, depending on the section. The exact field where it must appear is detailed, section by section, in the third column of Table 7.

5.3 Summary

This section has introduced the use of indexed variables, loops on indices and default setting.

Indexed variables permit the problem description to follow its mathematical expression much more closely, especially if the problem at hand arises from the discretization of some infinite dimensional computation. Variational calculations, processes governed by partial or ordinary differential equations, regular network and boundary value determination are just a few of the application areas where this facility is most useful. Indexed variables contain up to three (integer) indices within `()`, and these indices are separated by commas, as in `X(i,j,k)`. Any SDIF line referring to an indexed variable must start with a code whose first letter is either `X` or `Z`.

The appeal of indexed variables is clearly dependent on the facility that allows loops on indices to be written within a SDIF file. The syntax for these loops is similar to that used in Fortran: they start with a DO line and end with an OD line. It is also possible to nest loops up to three levels, and also to define the increment by which the loop indexed is increased each time the loop is executed.

The third important novelty in this section is the setting of default values for possibly large sets of objects, as constants, bounds, starting point components or element/group types. The syntax used for this purpose is identical that that used for defining a single object, except that the object name is replaced by the string 'DEFAULT': All relevant objects then take the value unless explicitly declared to have a different value at a later stage in the specification.

The tools introduced in this section are crucial for the specification of many large scale nonlinear programming problems.

6 More advanced techniques

You now possess the basic skills required to specify a large number of problems, from small simple ones to large structured cases. However, there is still more to learn. The additional concepts and constructs that we will introduce in this section may not be absolutely necessary for the statement of your favourite problem, but they may also be quite helpful.

6.1 Elemental and group parameters

The first additional construct that we will examine is the use of *parameters* in element and group types. Consider the simple problem of fitting, in the least squares sense, an exponential to a set of points on the bisector of the positive orthant of the plane. Mathematically, the problem is expressed as

$$\min_{\alpha,\beta} \sum_{i=0}^{p} \left(\alpha e^{ih\beta} - ih\right)^2, \tag{46}$$

given some positive integer p and some stepsize $h > 0$. This could be stated using the techniques developed in the previous sections, but would then require the definition of $p+1$ distinct element types of the form

$$ve^{ihw} \quad (i = 0, \ldots, p). \tag{47}$$

This is most definitely a waste of energy, especially if p exceeds two or three. An alternative is to consider i as a *parameter* associated with an element of the form

$$ve^{ihw}, \tag{48}$$

and then to assign to this parameter the required values when the problem variables α and β are assigned to the elemental variable v and w respectively. Such parameters are called *elemental parameters* . Practically speaking, we first have to declare that our (unique) element type has a parameter (called RI because it is the real value of the otherwise integer i). This is done using an EP line, as in

```
ELEMENT TYPE
 EV EXPIH     V                        W
 EP EXPIH     RI
```

where the name of the considered element type is repeated in field 2[27]. The next stage is to assign the correct values to this parameter in the ELEMENT USES section. This is written as

[27]just as for the declaration of internal variables on the IV- lines.

```
ELEMENT USES
 DO i         1                        P
 RI Reali     i
 XT E(i)      EXPIH
 ZV E(i)      V                        ALPHA
 ZV E(i)      W                        BETA
 ZP E(i)      RI                       Reali
 OD i
```

where the ZP code indicates that the value of the elemental Parameter is to be found in field 5 (hence the Z). We would use a similar XP code if this value was to be read in the numerical field 4.

```
NAME          EXPFIT

*   A simple exponential fit in 2 variables

*   Number of points
 IE P                    10
*   Stepsize
 RE H                    0.25
*   Other parameters
 IE 1                    1
VARIABLES
    ALPHA
    BETA
GROUPS
 DO i         1                        P
 XN R(i)
 OD i
CONSTANTS
 DO i         1                        P
 RI Reali     i
 R* Reali*H   Reali                    H
 Z  EXPFIT    R(i)                     Reali*H
 OD i
BOUNDS
*   Free variables inside the unit square
 FR EXPFIT    'DEFAULT'
ELEMENT TYPE
 EV EXPIH     V                        W
 EP EXPIH     RI
ELEMENT USES
 DO i         1                        P
 RI Reali     i
 XT E(i)      EXPIH
 ZV E(i)      V                        ALPHA
 ZV E(i)      W                        BETA
 ZP E(i)      RI                       Reali
 OD i
GROUP TYPE
 GV L2        GVAR
GROUP USES
 XT 'DEFAULT' L2
 DO i         1                        P
 XE R(i)      E(i)
 OD i
ENDATA
```

```
ELEMENTS      EXPFIT
TEMPORARIES
 R  IH
 R  EXPWIH
 M  EXP
INDIVIDUALS
 T  EXPIH
 A  IH                  0.25 * RI
 A  EXPWIH              EXP( W * IH )
 F                      V * EXPWIH
 G  V                   EXPWIH
 G  W                   V * IH * EXPWIH
 H  V         W         IH * EXPWIH
 H  W         W         V * IH * IH * EXPWIH
ENDATA

GROUPS        EXPFIT
INDIVIDUALS
 T  L2
 F                      GVAR**2
 G                      2.0 * GVAR
 H                      2.0
ENDATA
```

A similar situation could occur, when parameters are wanted within the group functions. An example might be the robust regression problem already considered in Section 3.3, where the parameter k associated with the Huber function $h_k(\cdot)$ could be dependent on the particular observation[28] and therefore vary from group to group.

Such parameters, called *group parameters*, are introduced in a way entirely similar to that of the elemental parameters, the codes `EP` (in the `ELEMENT TYPE` section), `P`, `XP` and `ZP` (in the `ELEMENT USES` section) being replaced by `GP` (in the `GROUP TYPE` section), `P`, `XP` and `ZP` (in the `GROUP USES` section) respectively. For instance, the `GROUP TYPE` section appropriate for declaring k to be a parameter for our robust regression problem is

```
GROUP TYPE
 GV HUBER     ALPHA
 GP HUBER     HUBERK
```

6.2 Scaling the problem variables

We already discussed in Section 3.2 how to scale the constraints and the objective function of a problem so that their respective size are comparable[29] by using the `'SCALE'` reserved name. MPS, and therefore SDIF, also provide a way to scale the *problem variables* much in the same way. If the variables of a problem are expected to take values that differ by orders of magnitude in the region of interest (that is "between" the starting point and the solution), it is indeed often useful to *scale* them. This is to say that each variable is *divided* by a suitable fixed scalar. If we consider, for instance, the problem of minimizing the simple two dimensional quadratic function

$$q(x_1, x_2) = 10^{-10}x_1^2 + 10^{20}x_2^2 \tag{49}$$

[28]This parameter indeed represents a level above which "outliers" are less penalized in the fit. It is therefore linked to the quality of the observations, which may itself vary from observation to observation.

[29]Nearly always a good idea, but not always easy to do.

the actual value of x_1 will not, in typical double precision say, contribute to that of the objective function, except when x_2 is itself extremely tiny. The simple change of variables

$$y_1 = \frac{x_1}{10^{10}}, \quad y_2 = \frac{x_2}{10^{20}} \tag{50}$$

produces the function

$$q(x_1, x_2) = y_1^2 + y_2^2 \tag{51}$$

which may be substantially easier to minimize on a computer. The required values of x_1 and x_2 may then be safely recovered from the relations (50). A typical case where a scaling of this type is often necessary is in calculations for electronic circuits. Indeed, if resistances are measured in ohms and capacitances in farads, typical values for resistance and capacitance values may be 10^5 and 10^{-12} respectively! It is[30] much safer from the numerical point of view to use scaled units such as megohms or picofarads...

The change of variable (50) is easy to specify in MPS and SDIF. The reserved word `'SCALE'` is again used for this purpose, but now appears in the `VARIABLES` section. The `VARIABLES` section for the simple problem of minimizing (49) may the look like

```
VARIABLES
 X  X1        'SCALE'   1.0D-10
 X  X2        'SCALE'   1.0D20
```

An efficient optimization algorithm such as LANCELOT will of course use the transformed variables in its inner calculations and return the optimal values to the user after the necessary back transformation: the change of variable is entirely implicit and hidden from the user's sight (and mind) within the numerical algorithm itself.

6.3 Two-sided inequality constraints

It sometimes happens that inequality constraints are specified both with an upper and a lower limiting value. For instance, the water level in a reservoir must always be nonnegative, but it must also not exceed the height of the dam that closes the valley (unless the safety of the valley inhabitants is of no consequence). Mathematically, if $h(r, f)$ represents the water height in the reservoir as a function of rainfall and outflow at the dam, we have a constraint of the form

$$0 \leq h(r, f) \leq h_{\text{disaster}} \tag{52}$$

where h_{disaster} is the height beyond which water starts flooding the valley downstream. Of course, one could state (52) as *two* constraints of the form

$$h(r, f) \geq 0 \ \text{ and } \ h(r, f) \leq h_{\text{disaster}}, \tag{53}$$

but this is slightly inelegant, and also inefficient because one has to introduce $h(r, f)$ twice in the formulation. It is much nicer to specify one of the constraints (53), the first, say, as an ordinary inequality constraint, and then to add the constraint that the village stays dry after the fact, by specifying *another value for the "other side" of the constraint.* In practice, the first step would require the `GROUPS` and `CONSTANTS` sections to contain lines of the form

```
GROUPS
 XG H2OLEVEL
CONSTANTS
 X  RESERVOIR H2OLEVEL   0.0
```

[30] Quite obviously!

(where the complicated nonlinear element function giving $h(r,f)$ is later assigned to the group H2OLEVEL)[31]. It is now possible to specify the upper bound on the water height by introducing yet another new section, the RANGES section, where "other sides" of inequality constraints may be given. The syntax of the RANGES section is entirely similar to that of the CONSTANTS, so that the RANGES section for our example reads

```
RANGES
 Z  RESERVOIR H2OLEVEL                  HDISASTER
```

The value HDISASTER is called the *range value* associated with the H2OLEVEL constraint. As the constraint H2OLEVEL has already been declared in the GROUPS section to be a "greater-than-or-equal-to" constraint (with associated value HMUDDY given in the CONSTANTS section), the value given in the RANGES section to the *same constraint group* is automatically associated with a "less-than-or-equal-to" constraint involving the same group. *The value specified in the* RANGES *section is always associated with an inequality constraint of the type* opposed *to that specified for the relevant constraint within the* GROUPS *section.*

We have said that the syntax of the RANGES section is identical to that of the CONSTANTS section. The codes X and Z are thus available (depending on the field in which the actual value is to be found), but defaults can also be set for the ranges values, and loops can be used. As for constants, a name is associated with a vector of range value (RESERVOIR in our example). The attentive reader may also guess that the initial default values have to be chosen as to have no impact at all on the original inequality constraints. Hence, the default range value is set to $+\infty$ for "greater than" groups, and to $-\infty$ for "less than" groups[32].

This may seem simple, but this appearance is quite deceptive. In fact, the RANGES section only allows the specification of constraints of the type

$$0 \left\{ \begin{array}{c} \leq \\ \geq \end{array} \right\} \frac{1}{s_i} g_i \left(\sum_{j \in J_i} w_{ij} f_j(x) + a_i^T x - b_i \right) \left\{ \begin{array}{c} \leq \\ \geq \end{array} \right\} r_i \; (i \in I_I), \tag{54}$$

for possibly infinite values of r_i. Our previous example was chosen so that the lower bound of zero coincides with the left-hand-side of (54), making the specification easier. We now illustrate the potential complications by assuming that the water level must not only be nonnegative, but also above the level of the mud at the bottom of the reservoir. It may then be impossible to specify the new constraint

$$h_{\text{muddy}} \leq h(r,f) \leq h_{\text{disaster}} \tag{55}$$

as a single constraint with two sides, except if the group type associated with this constraint is linear! This is because the only freedom we have to specify this new lower bound is via the constant b_i which appears *inside* the group value. More precisely, assume that the group is trivial; it is then enough to replace (55) by

$$0 \leq h(r,f) - h_{\text{muddy}} \leq h_{\text{disaster}} - h_{\text{muddy}}, \tag{56}$$

and we therefore find the form of (54) again. We then may write the following lines

```
GROUPS
 XG H2OLEVEL
CONSTANTS
 Z  RESERVOIR H2OLEVEL                  -HMUDDY
RANGES
 R- HUSEFUL   HDISASTER                 HMUDDY
 Z  RESERVOIR H2OLEVEL                  HUSEFUL
```

[31] The CONSTANTS section is also optional in our example, because it merely specifies a zero value, which is identical to the default for this section.

[32] If you wish to modify defaults for range values, be sure that you understand what you are doing...

On the other hand, if the constraint group type is nonlinear, it may not be possible to incorporate the constant $-h_{\mathrm{muddy}}$ in the group value... and one therefore has to specify two separate constraints in this unlucky (and, fortunately, infrequent) case.

6.4 Bounds on the objective function value

There is one feature of the SDIF syntax that is not used by the current version of LANCELOT. Its purpose is to allow the knowledgeable user to specify a priori known limits on the possible values of the objective function. For instance, the objective function of a least squares fitting problem, being a sum of squared terms, will never be negative. This is *not a constraint* that one *imposes* on the objective function, but merely a way to specify a priori *knowledge* of the problem. Because the S(D)IF is not only intended to work only in tandem with LANCELOT, and because some algorithms could well exploit this knowledge, a syntax is provided to state these "objective function bounds". They appear in a specialized section, called the `OBJECT BOUND` section. A typical section (for a least-squares problem) is given by

```
OBJECT BOUND
 LO PROBNAME          0.0
```

where `PROBNAME` is the name of a *set* of objective bounds, just as they are sets of bounds, constants or ranges. Again, we use the convention of specifying the problem name as set name.

The code `LO` allows to specify a `LO`wer bound on the objective function, but other codes can be used to specify lower and upper bounds, just as in the `BOUNDS` section. The possible codes and their meaning are given in Table 8. The codes `XL` and `XU` are provided for the (somewhat farfetched) case where the objective function name is an array name; they are otherwise equivalent to `LO` and `UP` respectively.

Code	Meaning
LO XL	specifies a numerical lower bound in field 4
UP XU	specifies a numerical upper bound in field 4
ZL	specifies a parametric lower bound in field 5
ZU	specifies a parametric upper bound in field 5

Table 8: Codes for specifying bounds on the objective function

6.5 Column oriented formulation

Let us now consider a type of problem that frequently arises in large scale nonlinear optimization: a nonlinear network problem[33]. Consider the network shown at Figure 3 and assume that we are interested in computing the best (in a measure described below) flows on each of its arcs as to ensure flow conservation at all internal nodes and so as to satisfy a demand of 10 units at node 1 and an supply of 10 units at node 4. In addition, we also impose some bounds on the flows.

More precisely, we consider the mathematical programming problem given by

$$\min \quad x_2 e^{x_1+x_3} + (x_3x_4)^2 + (x_3 - x_5)^2 \tag{57}$$

[33]We will however only consider a small one to avoid unnecessary tree cutting in the rain forests.

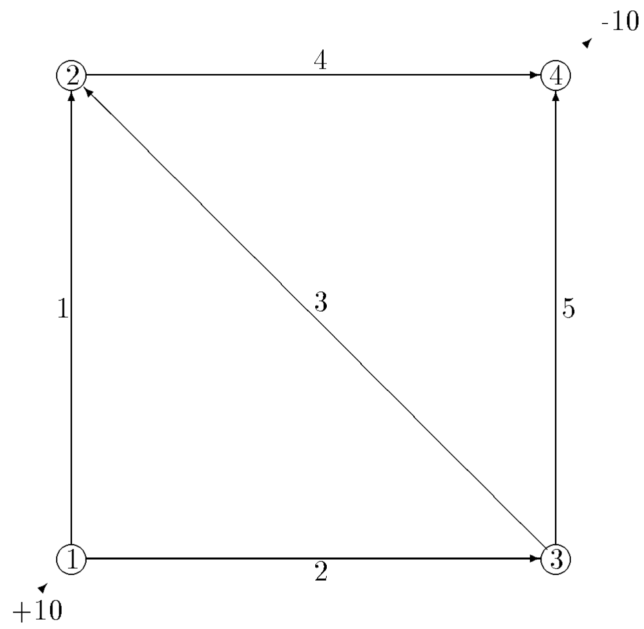

Figure 3: A small network with associated supply and demand

subject to the constraints

$$\begin{array}{rrrrrcr} x_1 & +x_2 & & & & = & 10, \\ -x_1 & & -x_3 & +x_4 & & = & 0, \\ & -x_2 & +x_3 & & +x_5 & = & 0, \\ & & & -x_4 & -x_5 & = & -10 \end{array} \tag{58}$$

and the bounds

$$2 \leq x_1 \leq 4, \quad 6 \leq x_2 \leq 8 \text{ and } 0 \leq x_3 \leq 5. \tag{59}$$

We have used the "node egoistic" convention that a flow leaving a node is given a negative value while a flow arriving at a node is given a positive value.

In (58), each equation corresponds to the conservation of flow at a node, which dictates that the total flow arriving at a node is equal to the flow that leaves it. In the form (57)–(59), the problem is well suited to an input of the type we have used up to now. However, it is often more natural to forget about the equations (58) and to consider Figure 3 instead. In this figure, the arcs are more pre-eminent, and we may be tempted to describe the network arc by arc, especially since there is a problem variable associated with each one of them. It is then natural to specify the arcs (the variables) by indicating their origin and destinations nodes. Assume that we wish to call the flows `X1` to `X5` and the nodes `N1` to `N4`. we may then say, in a new form of the `VARIABLES` section,

```
VARIABLES
    X1        N1        -1.0          N2        1.0
    X2        N1        -1.0          N3        1.0
    X3        N3        -1.0          N2        1.0
    X4        N2        -1.0          N4        1.0
    X5        N3        -1.0          N4        1.0
```

In this form, there is one line per arc in the network, and its origin and destination are easily identifiable. A brief comparison with (58) quickly reveals that each line of this new section correspond to a *column* of this set of linear equalities. This formulation is therefore referred to as the *column oriented formulation* of the problem, as opposed to the *row oriented formulation* that we have always used until now.

Because we have already described the relation between nodes and arcs in the VARIABLES section, there is no need to repeat it in the GROUPS section, which is then simply

```
GROUPS
 DO I         1                         NNODES
 XE  N(I)
 OD I
```

where we have assumed that the integer parameters 1 and NN have already been assigned the values of the constant 1 and of the number of nodes in the network respectively. We use XE codes because each of the flow conservation laws is an equality. This seems simple enough, but there is one additional catch: we have used the name of the constraints in the VARIABLES section, which requires that these names are known by then! This explains why the GROUPS section must precede the VARIABLES section in a column oriented formulation, the result thus being

```
NAME          SMALLNET
 IE 1                   1
 IE NNODES              4
GROUPS
 DO I         1                         NNODES
 XE  N(I)
 OD I
VARIABLES
    X1        N1        -1.0            N2        1.0
    X2        N1        -1.0            N3        1.0
    X3        N3        -1.0            N2        1.0
    X4        N2        -1.0            N4        1.0
    X5        N3        -1.0            N4        1.0
CONSTANTS
    SMALLNET  N1        10.0
    SMALLNET  N4        -10.0
```

where we have added the CONSTANTS section, which indicates the demand and supply at nodes 1 and 4.

Of course, column oriented formulation is not restricted to network problems: it can be used in general as a substitute for the row oriented description. They cannot be mixed, however, if only because one has to choose an order for the GROUPS and VARIABLES sections. Just as in the row oriented formulation, it is only possible to specify the linear part of the constraints in the fashion described above, the nonlinear elements always being assigned to the relevant groups in the GROUP USES section. In fact, except for the GROUPS and VARIABLES sections, the syntax of a column oriented problem description is identical to that of the corresponding row oriented one.

Which of the two formulation is better depends very much on the problem itself.

- A column oriented description might be more appropriate for problems with a large number of linear constraints[34] Network problems are a special case of this class.
- A row orientation is probably more appropriate for more nonlinear problems, where constraints are usually expressed row by row, after one knows what the problem variables are.

Obviously, the SIF and LANCELOT do not care which orientation is actually chosen.

[34] Only the column oriented formulation is allowed by MPS, the standard for linear programming input.

6.6 The order of the SIF sections

The introduction of the column oriented formulation and of the resulting permutation between the `VARIABLES` and `GROUPS` section raises the long eluded question of the order in which the SDIF sections are deemed to appear in the problem description file.

This order and the order of the associated keywords are not left to your fantasy but comes in only two types: one for row oriented input and the other for column oriented. They are both given in complete detail in Table 9, where optional sections are identified with keywords within brackets[35].

<table>
<tr><th>chapter</th><th>row oriented</th><th>column oriented</th></tr>
<tr><td rowspan="13">hline SDIF</td><td colspan="2">NAME</td></tr>
<tr><td>VARIABLES
GROUPS</td><td>GROUPS
VARIABLES</td></tr>
<tr><td colspan="2">[CONSTANTS]</td></tr>
<tr><td colspan="2">[RANGES]</td></tr>
<tr><td colspan="2">[BOUNDS]</td></tr>
<tr><td colspan="2">[ELEMENT TYPE]</td></tr>
<tr><td colspan="2">[ELEMENT USES]</td></tr>
<tr><td colspan="2">[GROUP TYPE]</td></tr>
<tr><td colspan="2">[GROUP USES]</td></tr>
<tr><td colspan="2">[OBJECT BOUND]</td></tr>
<tr><td colspan="2">ENDATA</td></tr>
<tr><td>[SEIF]</td><td colspan="2">ELEMENTS
[TEMPORARIES]
[GLOBALS]
INDIVIDUALS
ENDATA</td></tr>
<tr><td>[SGIF]</td><td colspan="2">GROUPS
[TEMPORARIES]
[GLOBALS]
INDIVIDUALS
ENDATA</td></tr>
</table>

Table 9: Possible orders for the SDIF/SEIF/SGIF keywords and sections

In this Table, `[SEIF]` means that the complete SEIF file is optional, but the fact that the `ELEMENTS` keyword is not within brackets indicates that it is mandatory *if the SEIF file is present at all.* The same comment applies to the SGIF keywords and sections.

6.7 Ending all loops at once

We metioned in Section 5.1 that nested loops on integer parameter were allowed up to a maximum of three levels. Of course, nested loops must use different loop parameters, and we might have a `VARIABLES` section like

```
VARIABLES
 DO I         1                          NX
 DO J         1                          NY
```

[35] Which means, of course, that the sections having keywords without brackets are mandatory.

```
 DO K         1                         NZ
 X  X(I,J,K)
 OD K
 OD J
 OD I
```

to define variables corresponding to points on a three dimensional grid, say. The SDIF syntax provides one additional facility for such cases: it allows one to *terminate all currently open loops at once*, in a single statement. This statement only contains the code ND, so that the above specification of variables is entirely equivalent to

```
VARIABLES
 DO I         1                         NX
 DO J         1                         NY
 DO K         1                         NZ
 X  X(I,J,K)
 ND
```

The use of the ND statement of course results in more compact problem specification, but is also less explicit than individual loop termination. The choice between both techniques therefore mostly depends on the readability of a particular problem file: the ND loop terminator is quite acceptable when the context is clear...

6.8 Specifying starting values for Lagrange multipliers

We now present a feature for mathematical programming experts. They indeed know that most algorithms for solving constrained problems make some use of the so-called *Lagrange multipliers*[36], which are the components of the vector λ in the expression of the Lagrangian

$$L(x, \lambda) = f(x) + \lambda^T c(x) \tag{60}$$

associated with the problem

$$\min f(x) \tag{61}$$

such that

$$c(x) \left\{ \begin{array}{c} \leq \\ = \\ \geq \end{array} \right\} 0 \tag{62}$$

It is not our intention to start here a discussion on the use, importance and meaning of these multipliers[37]. We just emphasize that it is very often helpful (from the algorithmic point of view) to know an approximation of their value at the problem's solution, especially if the starting point is close to to this solution. It then make sense to specify these "starting values" for the Lagrange multipliers as well as the starting values for the problem variables. This can be done in the START POINT section, using the codes XM or ZM, where the M stands for Multiplier and the X or Z as usual depends on the actual field in which the associated value is to be found. (The codes X and M are also available for the case where none of the names used is an array name.) Of course, multiplier starting values can be defaulted, just as for problem variables. A rather complete START POINT section for a constrained problem may then look like

```
START POINT
 ZV STARTP    'DEFAULT'                 X0-VALUE
 XM STARTP    'DEFAULT' 0.0
 XV STARTP    X1        1.0
 ZM STARTP    CONSTR1                   LAMBDA1
```

[36]Sometimes also called *shadow prices* in the econometric jargon associated with linear programming.
[37]Are you a mathematical programming expert?

where the real parameters `LAMBDA1` and `XO-VALUE` contain suitable starting values for the Lagrange multiplier associated with the constraint `CONSTR1` and default values for the problem variables respectively.

If you are really short on file space, there is even the possibility of mixing problem variable and Lagrange multiplier starting values on the same line, using a code consisting only of blank characters. One may then have a `START POINT` section containing a line of the form

```
          CONPROB    X1         3.14            LAMBDA1   2.71
```

but this kind of mix is prohibited on lines that indicate default values.

It is important to note at this point that the initial default value for starting Lagrange multipliers is zero, just as for the problem variables themselves.

6.9 Using free format in SIF files

So far, we have been quite specific on the format that we allow for writing the lines of a problem description in SIF. We used fields spanning well-defined ranges of columns within the line, each of these fields being used for specific purposes. The rigidity of this *fixed format* can sometimes be considered an intolerable burden[38]. We discuss in this paragraph a second form of input, which we will refer to as the *free format*, although the freedom here is not quite complete[39].

The idea of the free format is that the user should be able to arrange the problem data in the way she prefers, writing several SIF statements on a single line or spacing the fields in an entirely different fashion. The SIF free format however imposes a crucial restriction: *every line of a free format problem specification must be such that it can be translated unambiguously to one (or more) fixed format line.*

The first difficulty, if we are to allow data to appear anywhere on a line, is to define separators between fields. In fixed format, these are unnecessary, because fields are identified by the range of columns in which they lie, but free format requires *explicit separators*, whose purpose is to separate *strings* on a given line. Aline may contain at most 160 characters. The natural separator in a problem description file is probably the blank character, but, unfortunately, the MPS standard allows significant blanks within valid names! In fact, we have made all ASCII characters valid in SDIF names, which leaves none for the separators...The only way out is the exclude special characters from the free format files, except when they are meant as explicit separators.

" " (blank) indicates that a string is finished and that the next string is about to begin. More than one consecutive blanks are interpreted as a single blank, allowing the user to space the strings as she pleases on the line.

";" indicates that a string is finished (and another is to follow), but also indicates that the next string should start a new line in the fixed format translation of the line. The semi-colon is thus used to separate "fixed format statements", much in the same way that its separates C statements, for instance.

"_" indicates an empty string (with its preceding and following separators). For instance, ___ indicates three empty fields.

"$" indicates that what is left of the current line is to be interpreted as comment (i.e. neglected).

How does one decide in which format a particular problem description is written? The rule is that every SIF file is in fixed format in the beginning. This fixed format can be converted to free format anywhere in the file by inserting the line

[38] Despite our experience that fixed format often helps in keeping the problem files reasonably organized and readable...

[39] But what in life is?

```
FREE FORMAT
```

Any line following this declaration is assumed to be in free format, until one encounters the line

```
FIXED FORMAT
```

(after which fixed format is again assumed) or the end of the current SIF file.

We now illustrate our new technique by re-writing a free format variant of the specification for the `LSQFIT` problem of page 24 as follows:

```
NAME         LSQFIT
FREE FORMAT
$   An elementary constrained linear least-squares fit
RE X1_0.1;RE X2_0.3;RE X3_0.5;RE X4_0.7;RE X5_0.9      $ Data points
RE Y1_0.25;RE Y2_0.3;RE Y3_0.625;RE Y4_0.701;RE Y5_1.0 $ Observed values
IE 1_1;IE 5_5;RE C_0.85                                $ Constants, upper bound
VARIABLES;_a;_b
GROUPS;XL Cons a 1.0 b 1.0                                       $ Constraint
DO I 1_5;ZN Obj(I) a_X(I);XN Obj(I) b 1.0;XN Obj(I) 'SCALE' 2.0;ND$ Objective
CONSTANTS;DO I 1 5;Z LSQFIT Obj(I)_Y(I);ND;Z LSQFIT Cons_C
BOUNDS;XR LSQFIT b
GROUP TYPE;GV SQUARE ALPHA
GROUP USES;DO I 1 5;XT Obj(I) SQUARE;ND
ENDATA

GROUPS       LSQFIT
FREE FORMAT
INDIVIDUALS;T SQUARE;F__ALPHA * ALPHA;G__ALPHA + ALPHA;H__2.0
ENDATA
```

This is of course shorter (even discounting the use of loops in this version), but also shows that free format, if not carefully laid out, can make a problem file very much harder to read...

Again, it is hard in general to recommend one of the free or the fixed format in favour of the other. Both have their advantages, the free format being mostly useful for its conciseness. Much may depend on how constrained you feel in the columns oriented fixed format. Also notice that, unfortunately, free format does not allow for extra features like very long names or numbers with 24 digits, which is probably its most limiting feature.

6.10 MPS features kept for compatibility

The last paragraph of this section is devoted to the description of certain features of the SDIF that we do not particularly recommend, but that have been kept in the standard for compatibility with MPS.

The first of these features is the existence of alternative equivalent definitions for certain keywords. They are listed in Table 10

For instance, `ROWS` can be used anywhere `GROUPS` appears: they are entirely equivalent.

The second feature is a further facility present in the MPS standard: the ability to define new constraints by linear combination of other constraints. We admit to not liking this technique very much, but it is nevertheless available in SDIF for the sake of compatibility. The idea is rather simple. Assume we have already written the beginning of the `GROUPS` section as

```
GROUPS
 XL CONS1
 XG CONS2     X          1.0             Y            2.0
 XE CONS3
```

Keyword used above	Equivalent keywords
GROUPS	CONSTRAINTS ROWS
VARIABLES	COLUMNS
CONSTANTS	RHS RHS'

Table 10: Equivalent keywords

where we have two purely nonlinear groups, the first, `CONS1`, being of the "less than" type and the second, `CONS3`, being of the "equals" type. We also have another "greater than" group, named `CONS2`, in which the variables `X` and `Y` appear linearly, and which may also contain nonlinear elements. We then *create a new group by taking a linear combination of two already existing groups*, `CONS1` and `CONS2` in our case. This is done by writing

```
 DL MESS       CONS1      5.0                CONS2      -1.0
```

in which case the new "less than" constraint `MESS` is defined by

$$\mathtt{MESS} = 5 \times \mathtt{CONS1} - \mathtt{CONS2}$$

The constants associated with this new constraint is then defined in the `CONSTANTS` section, as for any other group.

The codes `DN`, `DL`, `DG` and `DE` are available to create objective groups, "less than", "greater than" and equality constraints respectively. Of course, all groups whose names appear in fields 3 and 5 of these `D` lines must have been already defined in the `GROUPS` section.

6.11 Summary

In this section, we have introduced a number of advanced (?) features present in the SIF. As the features are relatively uncorrelated, it is rather difficult to summarize the whole section. We have been considering

- the definition and use of parameters in group and element function,
- a remedy for the possible bad scaling of the problem's variables,
- the introduction of two-sided inequality constraints,
- the specification of a priori known bounds on the objective function values,
- the column oriented problem formulation, as opposed to the row oriented formulation,
- a special code (`ND`) for ending all open loops at once,
- ways to initialize the Lagrange multipliers associated with constraints of the problem,
- the free format that allows greater freedom in the layout of the problem file, and, finally,
- some MPS features that have been kept in the SDIF to ensure compatibility between the two standards.

7 Some typical SIF examples

Although it has been necessary to introduce an exact specification of the allowed input formats, we feel that the best way to learn how to input problems is by studying specific examples. Consequently, we present here three further examples which use many of the salient features of the standard. We hope that you will be encouraged to trying converting your favourite test examples into the standard form.

7.1 A simple constrained problem

Firstly, we give an SIF file for the 65th constrained example collected by Hock and Schittkowski [12, p. 87]. The objective function is

$$(x_1 - x_2)^2 + (x_1 + x_2 - 10)^2/9 + (x_3 - 5)^2.$$

There is a single inequality constraint

$$x_1^2 + x_2^2 + x_3^3 \leq 48$$

and additional simple bounds

$$-4.5 \leq x_1 \leq 4.5, \quad -4.5 \leq x_2 \leq 4.5, \quad -5 \leq x_3 \leq 5$$

on the variables. The values $x_1 = -5$, $x_2 = 5$, $x_3 = 0$ give the starting point for the minimization. Noting the use of range transformations, an appropriate SIF file might be:

```
NAME          HS65

*  Number of variables

 IE N                   3

*  Useful parameters

 IE 1                   1
 IE 3                   3
 RE NINE                9.0
 RD 1/9       NINE      1.0

VARIABLES

    X1
    X2
    X3

GROUPS

*  Objective function

 N  OBJ

*   Constraints functions

 L  CON

CONSTANTS

    HS65      CON       48.0

BOUNDS

 LO HS65      X1        -4.5
 UP HS65      X1        4.5
```

```
 LO HS65      X2        -4.5
 UP HS65      X2        4.5

 LO HS65      X3        -5.0
 UP HS65      X3        5.0

START POINT

    HS65      X1        -5.0           X2        5.0
    HS65      X3        0.0

ELEMENT TYPE

 EV DIFSQR    V1                       V2
 IV DIFSQR    U

 EV SUMSQR    V1                       V2
 IV SUMSQR    U

 EV SQ-5      V
 EV SQ        V

ELEMENT USES

*   Nonlinear elements for the objective function

 T  O1        DIFSQR
 V  O1        V1                       X1
 V  O1        V2                       X2

 T  O2        SUMSQR
 V  O2        V1                       X1
 V  O2        V2                       X2

 T  O3        SQ-5
 V  O3        V                        X3

*   Nonlinear elements for the constraint function

 DO I         1                        3

 XT C(I)      SQ
 ZV C(I)      V                        X(I)

 ND

GROUP USES

 E  OBJ       O1                       O3
 ZE OBJ       O2                       1/9
 E  CON       C1                       C2
 E  CON       C3

ENDATA

*   Specify the form of the different element types

ELEMENTS      HS65

TEMPORARIES

 R  DIF

INDIVIDUALS

*  square of V, where V = U1 - U2

 T  DIFSQR
```

```
 R  U         V1        1.0             V2        -1.0
 F                      U * U
 G  U                   U + U
 H  U         U         2.0

*  square of ( U - 10 ), where U = V1 + V2

 T  SUMSQR

 R  U         V1        1.0             V2        1.0
 A  DIF                 U - 10
 F                      DIF * DIF
 G  U                   DIF + DIF
 H  U         U         2.0

*  square of ( V - 5 )

 T  SQ-5

 A  DIF                 V - 5
 F                      DIF * DIF
 G  V                   DIF + DIF
 H  V         V         2.0

*  square of V

 T  SQ

 F                      V * V
 G  V                   V + V
 H  V         V         2.0

ENDATA

*  Specify the form of the different group types

GROUPS        HS65

*  All groups are trivial

ENDATA
```

7.2 A system of nonlinear equations with single-indexed variables

Here we give an SIF file for the discrete boundary value problem in n variables given by Moré, Garbow and Hillstrom [15, p.27, problem 28]. We set the problem up as a system of nonlinear equations (constraints)

$$2x_i - x_{i-1} - x_{i+1} + \frac{h^2}{2}(x_i + ih + 1)^3 = 0, \quad (i = 1, \ldots, n),$$

where $h = 1/(n+1)$ and $x_0 = x_{n+1} = 0$. We specify $n = 8$ and use the starting point $x_i = ih(ih-1)$ for $i = 1, \ldots, n$. An appropriate SIF file might be:

```
NAME          BDVLE

*   N is the number of internal discretization points

 IE N                   8

*   Define useful parameters

 IE 0                   0
 IE 1                   1
 IA N+1       N         1

 RI RN+1      N+1
```

```
 RD H          RN+1      1.0
 R* H2         H                        H
 RM HALFH2     H2        0.5

VARIABLES

 DO I          0                        N+1

 X  X(I)

 ND

GROUPS

 DO I          1                        N
 IA I-1        I         -1
 IA I+1        I          1

 XE G(I)       X(I-1)    -1.0           X(I)      2.0
 XE G(I)       X(I+1)    -1.0

 ND

BOUNDS

 FR BDVLE      'DEFAULT'
 XX BDVLE      X(0)      0.0
 XX BDVLE      X(N+1)    0.0

START POINT

 X  BDVLE      X(0)      0.0            X(N+1)    0.0

 DO I          1                        N

 RI RI         I
 R* IH         RI                       H
 RA IH-1       IH        -1.0
 R* TI         IH                       IH-1

 Z  BDVLE      X(I)                     TI

 ND

ELEMENT TYPE

 EV WCUBE      V
 EP WCUBE      B

ELEMENT USES

 DO I          1                        N

 RI REALI      I
 R* IH         REALI                    H
 RA IH+1       IH        1.0

 XT E(I)       WCUBE
 ZV E(I)       V                        X(I)
 ZP E(I)       B                        IH+1

 ND

GROUP USES

 DO I          1                        N

 ZE G(I)       E(I)                     HALFH2

 ND
```

```
ENDATA

*   Specify the form of the different element types

ELEMENTS      BDVLE

*   the cube of (V + B)

TEMPORARIES

 R  VPLUSB

INDIVIDUALS

 T  WCUBE
 A  VPLUSB                V + B
 F                        VPLUSB**3
 G  V                     3.0 * VPLUSB**2
 H  V         V           6.0 * VPLUSB

ENDATA

*  Specify the form of the different group types

GROUPS        BDVLE

*  all groups are trivial

ENDATA
```

7.3 A constrained problem with triple indexed variables

Finally, we give an SIF file for the problem of determining the maximum growth possible when performing Gaussian elimination with complete pivoting (see [9]). Starting with an n by n real matrix $X^{(1)}$, we let $X^{(k)}$ be the matrix which remains to be eliminated after $k-1$ steps of Gaussian elimination without pivoting. Let $x_{i,j,k}$ be the (i,j)-th entry of $X^{(k)}$. We thus wish to maximize $x_{n,n,n}$ subject to the restrictions that the matrices $X^{(k)}$ and $X^{(k+1)}$ are related to each other by elimination restrictions, that the largest element in the bottom $n-k-1$ block of $X^{(k)}$ occurs in position (k,k,k) and that the initial matrix $X^{(1)}$ is scaled so that the largest entry in magnitude is 1. This leads to the problem

$$\text{minimize} - x_{n,n,n}$$

subject to the elimination constraints

$$x_{i,j,k+1} - x_{i,j,k} + \frac{x_{i,k,k}x_{k,j,k}}{x_{k,k,k}} = 0 \text{ for } k \le i \le n, k \le j \le n \text{ and } k = 1, \ldots, n-1,$$

constraints which make the signs of the pivots unique,

$$x_{k,k,k} \ge 0 \text{ for } k = 1, \ldots, n,$$

a normalizing constraint,

$$x_{1,1,1} = 1,$$

and complete pivoting constraints

$$-1 \le x_{i,j,1} \le 1 \text{ for } 1 \le i \le n, 1 \le j \le n$$

and

$$-x_{k,k,k} \le x_{i,j,k} \le x_{k,k,k} \text{ for } k \le i \le n, k \le j \le n \text{ and } k = 2, \ldots, n-1.$$

These details, along with a suitable starting point, are given for the case $n = 6$ in the following SIF file:

```
NAME          GAUSS

*   size of the matrices = n

 IE N                   6

*   other parameter definitions

 IE 1                   1
 IE 2                   2
 IA N-1       N         -1

VARIABLES

 DO K         1                        N
 DO J         K                        N
 DO I         K                        N

 X  X(I,J,K)

 ND

GROUPS

*  objective function

 DO K         N                        N

 XN OBJ       X(K,K,K)  -1.0

 ND

*   elimination constraints

 DO K         1                        N-1

 IA K+        K         1

 DO I         K+                       N
 DO J         K+                       N

 XE E(I,J,K)  X(I,J,K+) 1.0            X(I,J,K)  -1.0

 ND

*   complete pivoting constraints (submatrices 2 to n-1)

 DO K         2                        N-1
 DO I         K                        N
 DO J         K                        N

 XL M(I,J,K)  X(I,J,K)  1.0            X(K,K,K)  -1.0
 XG P(I,J,K)  X(I,J,K)  1.0            X(K,K,K)  1.0

 ND

BOUNDS

*   default = free variables

 FR GAUSS     'DEFAULT'

*   complete pivoting constraints (submatrix 1)

 DO I         1                        N
 DO J         1                        N

 XL GAUSS     X(I,J,1)  -1.0
 XU GAUSS     X(I,J,1)  1.0
```

```
 ND

*   ensure pivotal elements are nonnegative

 DO K         1                        N

 XL GAUSS     X(K,K,K)  0.0

 ND

*   normalize first pivot

 FX GAUSS     X1,1,1    1.0

START POINT

*   default value for starting point component

 V  GAUSS     'DEFAULT' 0.01

*   Set initial matrices to perturbed identities

 DO K         1                        N
 DO I         K                        N

 X  GAUSS     X(I,I,K)  1.0

 ND

ELEMENT TYPE

 EV ELIM      V1                       V2
 EV ELIM      V3

ELEMENT USES

 DO K         1                        N-1

 IA K+        K         1

 DO I         K+                       N
 DO J         K+                       N

 XT A(I,J,K)  ELIM
 ZV A(I,J,K)  V1                       X(I,K,K)
 ZV A(I,J,K)  V2                       X(K,J,K)
 ZV A(I,J,K)  V3                       X(K,K,K)

 ND

GROUP USES

 DO K         1                        N-1

 IA K+        K         1

 DO I         K+                       N
 DO J         K+                       N

 XE E(I,J,K)  A(I,J,K)

 ND

ENDATA

*  Specify the form of the different element types

ELEMENTS      GAUSS

TEMPORARIES
```

```
 R  VALUE
 R  V3SQ

INDIVIDUALS

 T  ELIM

 A  VALUE                V1 * V2 / V3
 A  V3SQ                 V3 * V3

 F                       VALUE

 G  V1                   V2 / V3
 G  V2                   V1 / V3
 G  V3                   - VALUE / V3

 H  V1         V2        1.0 / V3
 H  V1         V3        - V2 / V3SQ
 H  V2         V3        - V1 / V3SQ
 H  V3         V3        2.0 * VALUE / V3SQ * V3

ENDATA

*  Specify the form of the different group types

GROUPS         GAUSS

* all groups are trivial

ENDATA
```

7.4 A test problem collection

A collection of SIF files for many sets of standard test problems is currently being compiled. To date, we have translated, amongst others, the problems given by Moré, Garbow and Hilstrom [15], Toint [17], Moré [14] and Buckley [2]. Work is proceeding on the problems given in the books by Hock and Schittkowski [12] and Schittkowski [16]. It is the author's intention to put the resulting files in the public domain.

8 A complete template for SIF syntax

In this last section of the introduction to the SIF, we finally provide a complete template containing all possible types of valid SIF lines (in fixed format). This template is intended as an aid to allow you to verify that the lines that you are undoubtly starting to write[40] are syntaxically correct. This template is included in machine readable form with the LANCELOT distribution tape.

In this template, we use the following conventions defined in Table 11.

[40] We can hear you!

Name	Object named	Max length (in chars)	Fortran restr.
`prb-name`	the problem	8	
`b-name`	a vector of bounds	10	
`c-name`	a vector of constants	10	
`el-name`	an element	10	
`ep-nam`	an elemental parameter	6	y
`et-name`	an element type	10	
`ev-nam`	an elemental variable	6	y
`f-name`	a function (see Table 3)	6	y
`gt-name`	a group type	10	
`gp-nam`	a group parameter	6	y
`gv-nam`	the group variable	6	y
`gr-name`	a group	10	
`ip-name`	an integer parameter	10	
`it-nam`	a temporary integer parameter	6	y
`iv-nam`	an internal variable	6	y
`mf-nam`	a machine supplied Fortran function	6	y
`numerical-vl`	a number	12	
`lt-name`	a temporary logical parameter	6	y
`o-name`	a vector of bounds on the objective	10	
`pv-name`	a problem variable	10	
`r-name`	a vector of ranges	10	
`rt-name`	a temporary real parameter	6	y
`rpa-name`	a real array of parameters	10	
`rp-name`	a real parameter	10	
`s-name`	a vector of starting point	10	
`uf-nam`	a user supplied Fortran function	6	

Table 11: Naming conventions for the SIF template

```
NAME          prb-name
 IE ip-name             numerical-vl
 IR ip-name   rp-name
 IA ip-name   ip-name   numerical-vl
 IS ip-name   ip-name   numerical-vl
 IM ip-name   ip-name   numerical-vl
 ID ip-name   ip-name   numerical-vl
 I= ip-name   ip-name
 I+ ip-name                           ip-name
 I- ip-name                           ip-name
 I* ip-name                           ip-name
 I/ ip-name                           ip-name
 RE rp-name             numerical-vl
 RI rp-name   ip-name
 RA rp-name   rp-name   numerical-vl
 RS rp-name   rp-name   numerical-vl
 RM rp-name   rp-name   numerical-vl
 RD rp-name   rp-name   numerical-vl
 RF rp-name   f-name    numerical-vl
 R= rp-name   rp-name
 R+ rp-name                           rp-name
 R- rp-name                           rp-name
 R* rp-name                           rp-name
 R/ rp-name                           rp-name
 R( rp-name   f-name                  rp-name
 AE rpa-name            numerical-vl
 AI rpa-name  ip-name
 AA rpa-name  rpa-name  numerical-vl
 AS rpa-name  rpa-name  numerical-vl
 AM rpa-name  rpa-name  numerical-vl
 AD rpa-name  rpa-name  numerical-vl
 AF rpa-name  f-name    numerical-vl
 A= rpa-name  rpa-name
 A+ rpa-name                          rpa-name
 A- rpa-name                          rpa-name
 A* rpa-name                          rpa-name
 A/ rpa-name                          rpa-name
 A( rpa-name  f-name                  rpa-name
 DO ip-name   ip-name                 ip-name
 DI ip-name   ip-name
 OD ip-name
 ND
GROUPS
ROWS
CONSTRAINTS
 N  gr-name
 N  gr-name   pv-name   numerical-vl
 N  gr-name   pv-name   numerical-vl  pv-name    numerical-vl
 N  gr-name   'SCALE'   numerical-vl
 G  gr-name
 G  gr-name   pv-name   numerical-vl
 G  gr-name   pv-name   numerical-vl  pv-name    numerical-vl
 G  gr-name   'SCALE'   numerical-vl
 L  gr-name
 L  gr-name   pv-name   numerical-vl
 L  gr-name   pv-name   numerical-vl  pv-name    numerical-vl
 L  gr-name   'SCALE'   numerical-vl
 E  gr-name
 E  gr-name   pv-name   numerical-vl
 E  gr-name   pv-name   numerical-vl  pv-name    numerical-vl
 E  gr-name   'SCALE'   numerical-vl
 XN gr-name
 XN gr-name   pv-name   numerical-vl
 XN gr-name   pv-name   numerical-vl  pv-name    numerical-vl
 XN gr-name   'SCALE'   numerical-vl
 XG gr-name
 XG gr-name   pv-name   numerical-vl
 XG gr-name   pv-name   numerical-vl  pv-name    numerical-vl
 XG gr-name   'SCALE'   numerical-vl
 XL gr-name
```

```
 XL gr-name   pv-name   numerical-vl
 XL gr-name   pv-name   numerical-vl   pv-name    numerical-vl
 XL gr-name   'SCALE'   numerical-vl
 XE gr-name
 XE gr-name   pv-name   numerical-vl
 XE gr-name   pv-name   numerical-vl   pv-name    numerical-vl
 XE gr-name   'SCALE'   numerical-vl
 ZN gr-name
 ZN gr-name   pv-name                  rpa-name
 ZN gr-name   'SCALE'                  rpa-name
 ZG gr-name
 ZG gr-name   pv-name                  rpa-name
 ZG gr-name   'SCALE'                  rpa-name
 ZL gr-name
 ZL gr-name   pv-name                  rpa-name
 ZL gr-name   'SCALE'                  rpa-name
 ZE gr-name
 ZE gr-name   pv-name                  rpa-name
 ZE gr-name   'SCALE'                  rpa-name
 DN gr-name   gr-name   numerical-vl
 DN gr-name   gr-name   numerical-vl   gr-name    numerical-vl
 DG gr-name   gr-name   numerical-vl
 DG gr-name   gr-name   numerical-vl   gr-name    numerical-vl
 DL gr-name   gr-name   numerical-vl
 DL gr-name   gr-name   numerical-vl   gr-name    numerical-vl
 DE gr-name   gr-name   numerical-vl
 DE gr-name   gr-name   numerical-vl   gr-name    numerical-vl
VARIABLES
COLUMNS
    pv-name
    pv-name   gr-name   numerical-vl
    pv-name   gr-name   numerical-vl   gr-name    numerical-vl
    pv-name   'SCALE'   numerical-vl
 X  pv-name
 X  pv-name   gr-name   numerical-vl
 X  pv-name   gr-name   numerical-vl   gr-name    numerical-vl
 X  pv-name   'SCALE'   numerical-vl
 Z  pv-name   gr-name                  rpa-name
 Z  pv-name   'SCALE'                  rpa-name
CONSTANTS
RHS
RHS'
    c-name    'DEFAULT' numerical-vl
 X  c-name    'DEFAULT' numerical-vl
 Z  c-name    'DEFAULT'                rpa-name
    c-name    gr-name   numerical-vl
    c-name    gr-name   numerical-vl   gr-name    numerical-vl
 X  c-name    gr-name   numerical-vl
 X  c-name    gr-name   numerical-vl   gr-name    numerical-vl
 Z  c-name    gr-name                  rpa-name
RANGES
    r-name    'DEFAULT' numerical-vl
 X  r-name    'DEFAULT' numerical-vl
 Z  r-name    'DEFAULT'                rpa-name
    r-name    gr-name   numerical-vl
    r-name    gr-name   numerical-vl   gr-name    numerical-vl
 X  r-name    gr-name   numerical-vl
 X  r-name    gr-name   numerical-vl   gr-name    numerical-vl
 Z  r-name    gr-name                  rpa-name
BOUNDS
 LO b-name    'DEFAULT' numerical-vl
 UP b-name    'DEFAULT' numerical-vl
 FX b-name    'DEFAULT' numerical-vl
 FR b-name    'DEFAULT'
 MI b-name    'DEFAULT'
 PL b-name    'DEFAULT'
 XL b-name    'DEFAULT' numerical-vl
 XU b-name    'DEFAULT' numerical-vl
 XX b-name    'DEFAULT' numerical-vl
 XM b-name    'DEFAULT'
```

```
 XP b-name    'DEFAULT'
 ZL b-name    'DEFAULT'                rpa-name
 ZU b-name    'DEFAULT'                rpa-name
 ZX b-name    'DEFAULT'                rpa-name
 LO b-name    pv-name   numerical-vl
 UP b-name    pv-name   numerical-vl
 FX b-name    pv-name   numerical-vl
 FR b-name    pv-name
 MI b-name    pv-name
 PL b-name    pv-name
 XL b-name    pv-name   numerical-vl
 XU b-name    pv-name   numerical-vl
 XX b-name    pv-name   numerical-vl
 XR b-name    pv-name
 XM b-name    pv-name
 XP b-name    pv-name
 ZL b-name    pv-name                  rpa-name
 ZU b-name    pv-name                  rpa-name
 ZX b-name    pv-name                  rpa-name
START POINT
 V  s-name    'DEFAULT' numerical-vl
 XV s-name    'DEFAULT' numerical-vl
 ZV s-name    'DEFAULT'                rpa-name
 M  s-name    'DEFAULT' numerical-vl
 XM s-name    'DEFAULT' numerical-vl
 ZM s-name    'DEFAULT'                rpa-name
    s-name    'DEFAULT' numerical-vl
 X  s-name    'DEFAULT' numerical-vl
 Z  s-name    'DEFAULT'                rpa-name
 V  s-name    pv-name   numerical-vl
 V  s-name    pv-name   numerical-vl   pv-name   numerical-vl
 XV s-name    pv-name   numerical-vl
 XV s-name    pv-name   numerical-vl   pv-name   numerical-vl
 ZV s-name    pv-name                  rpa-name
 M  s-name    pv-name   numerical-vl
 M  s-name    pv-name   numerical-vl   pv-name   numerical-vl
 XM s-name    pv-name   numerical-vl
 XM s-name    pv-name   numerical-vl   pv-name   numerical-vl
 ZM s-name    pv-name                  rpa-name
    s-name    pv-name   numerical-vl
    s-name    pv-name   numerical-vl   pv-name   numerical-vl
    s-name    gr-name   numerical-vl
    s-name    gr-name   numerical-vl   gr-name   numerical-vl
    s-name    pv-name   numerical-vl   gr-name   numerical-vl
    s-name    gr-name   numerical-vl   pv-name   numerical-vl
 X  s-name    pv-name   numerical-vl
 X  s-name    pv-name   numerical-vl   pv-name   numerical-vl
 X  s-name    gr-name   numerical-vl
 X  s-name    gr-name   numerical-vl   gr-name   numerical-vl
 X  s-name    pv-name   numerical-vl   gr-name   numerical-vl
 X  s-name    gr-name   numerical-vl   pv-name   numerical-vl
 Z  s-name    pv-name                  rpa-name
 Z  s-name    gr-name                  rpa-name
ELEMENT TYPE
 EV et-name   ev-nam
 EV et-name   ev-nam                   ev-nam
 IV et-name   iv-nam
 IV et-name   iv-nam                   iv-nam
 EP et-name   ep-nam
 EP et-name   ep-nam                   ep-nam
ELEMENT USES
 T  'DEFAULT' et-name
 XT 'DEFAULT' et-name
 T  el-name   et-name
 XT el-name   et-name
 V  el-name   ev-nam                   pv-name
 ZV el-name   ev-nam                   pv-name
 P  el-name   ep-nam    numerical-vl
 P  el-name   ep-nam    numerical-vl   ep-name   numerical-vl
 XP el-name   ep-nam    numerical-vl
```

```
 XP el-name   ep-nam    numerical-vl   ep-name   numerical-vl
 ZP el-name   ep-nam                   rpa-name
GROUP TYPE
 GV gt-name   gv-nam
 GP gt-name   gp-nam
 GP gt-name   gp-nam                   gp-nam
GROUP USES
 T  'DEFAULT' gt-name
 XT 'DEFAULT' gt-name
 T  gr-name   gt-name
 XT gr-name   gt-name
 E  gr-name   el-name
 E  gr-name   el-name   numerical-vl
 E  gr-name   el-name                  el-name
 E  gr-name   el-name   numerical-vl   el-name
 E  gr-name   el-name   numerical-vl   el-name   numerical-vl
 XE gr-name   el-name                  el-name   numerical-vl
 XE gr-name   el-name
 XE gr-name   el-name   numerical-vl
 XE gr-name   el-name                  el-name
 XE gr-name   el-name   numerical-vl   el-name
 XE gr-name   el-name   numerical-vl   el-name   numerical-vl
 XE gr-name   el-name                  el-name   numerical-vl
 ZE gr-name   el-name                  rpa-name
 P  gr-name   gp-nam    numerical-vl
 P  gr-name   gp-nam    numerical-vl   gp-nam    numerical-vl
 XP gr-name   gp-nam    numerical-vl
 XP gr-name   gp-nam    numerical-vl   gp-nam    numerical-vl
 ZP gr-name   gp-name                  rpa-name
OBJECT BOUND
 LO o-name              numerical-vl
 UP o-name              numerical-vl
 XL o-name              numerical-vl
 XU o-name              numerical-vl
 ZL o-name                             rpa-name
 ZU o-name                             rpa-name
ENDATA

ELEMENTS      prb-name
TEMPORARIES
 I  it-nam
 R  rt-nam
 L  lt-nam
 M  mf-nam
 F  uf-nam
GLOBALS
 A  it-nam              fortran-expression
 A  rt-nam              fortran-expression
 A  lt-nam              fortran-expression
 A+                     fortran-expression
 I  lt-nam    it-nam    fortran-expression
 I  lt-nam    rt-nam    fortran-expression
 I  lt-nam    lt-nam    fortran-expression
 I+                     fortran-expression
 E  lt-nam    it-nam    fortran-expression
 E  lt-nam    rt-nam    fortran-expression
 E  lt-nam    lt-nam    fortran-expression
 E+                     fortran-expression
INDIVIDUALS
 T  et-name
 R  iv-nam    ev-nam    numerical-vl   ev-nam    numerical-vl
 R  iv-nam    ev-nam    numerical-vl
 A  it-nam              fortran-expression
 A  rt-nam              fortran-expression
 A  lt-nam              fortran-expression
 A+                     fortran-expression
 I  lt-nam    it-nam    fortran-expression
 I  lt-nam    rt-nam    fortran-expression
 I  lt-nam    lt-nam    fortran-expression
 I+                     fortran-expression
```

```
 E  lt-nam    it-nam    fortran-expression
 E  lt-nam    rt-nam    fortran-expression
 E  lt-nam    lt-nam    fortran-expression
 E+                     fortran-expression
 F                      fortran-expression
 F+                     fortran-expression
 G  iv-nam              fortran-expression
 G  ev-nam              fortran-expression
 G+                     fortran-expression
 H  iv-nam    iv-nam    fortran-expression
 H  ev-nam    ev-nam    fortran-expression
 H+                     fortran-expression
ENDATA

GROUPS        prb-name
TEMPORARIES
 I  it-nam
 R  rt-nam
 L  lt-nam
 M  mf-nam
 F  uf-nam
GLOBALS
 A  it-nam              fortran-expression
 A  rt-nam              fortran-expression
 A  lt-nam              fortran-expression
 A+                     fortran-expression
 I  lt-nam    it-nam    fortran-expression
 I  lt-nam    rt-nam    fortran-expression
 I  lt-nam    lt-nam    fortran-expression
 I+                     fortran-expression
 E  lt-nam    it-nam    fortran-expression
 E  lt-nam    rt-nam    fortran-expression
 E  lt-nam    lt-nam    fortran-expression
 E+                     fortran-expression
INDIVIDUALS
 T  et-name
 R  iv-nam    ev-nam    numerical-vl   ev-nam    numerical-vl
 R  iv-nam    ev-nam    numerical-vl
 A  it-nam              fortran-expression
 A  rt-nam              fortran-expression
 A  lt-nam              fortran-expression
 A+                     fortran-expression
 I  lt-nam    it-nam    fortran-expression
 I  lt-nam    rt-nam    fortran-expression
 I  lt-nam    lt-nam    fortran-expression
 I+                     fortran-expression
 E  lt-nam    it-nam    fortran-expression
 E  lt-nam    rt-nam    fortran-expression
 E  lt-nam    lt-nam    fortran-expression
 E+                     fortran-expression
 F                      fortran-expression
 F+                     fortran-expression
 G                      fortran-expression
 G+                     fortran-expression
 H                      fortran-expression
 H+                     fortran-expression
ENDATA
```

References

[1] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS: a User's Guide.* The Scientific Press, Redwood City, USA., 1988.

[2] A. G. Buckley. Test functions for unconstrained minimization. Technical Report CS-3, Computing Science Division, Dalhousie University, Dalhousie, Canada, 1989.

[3] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. Global convergence of a class of trust region algorithms for optimization with simple bounds. *SIAM Journal on Numerical Analysis*, 25:433–460, 1988. See also same journal 26:64–767, 1989.

[4] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. An introduction to the structure of large scale nonlinear optimization problems and the lancelot[18 project. In R. Glowinski and A. Lichnewsky, editors, *Computing Methods in Applied Sciences and Engineering*, pages 42–54, Philadelphia, USA, 1990. SIAM.

[5] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. A proposal for a standard data input format for large-scale nonlinear programming problems. Report CS-89-61, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, 1990.

[6] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. A globally convergent augmented Lagrangian algorithm for optimization with general constraints and simple bounds. *SIAM Journal on Numerical Analysis*, 28(2):545–572, 1991.

[7] International Business Machine Corporation. Mathematical programming system/360 version 2, linear and separable programming-user's manual. Technical Report H20-0476-2, IBM Corporation, 1969.

[8] R. Fourer, D. M. Gay, and B. W. Kernighan. AMPL: A mathematical programming language. Computer science technical report, AT&T Bell Laboratories, Murray Hill, USA, 1987.

[9] N. I. M. Gould. On growth in Gaussian elimination with complete pivoting. *SIAM Journal on Matrix Analysis*, 12(2):354–361, 1991.

[10] A. Griewank and Ph. L. Toint. On the unconstained optimization of partially separable functions. In M. J. D. Powell, editor, *Nonlinear Optimization 1981*, pages 301–312, London and New York, 1982. Academic Press.

[11] Haverley Systems, Inc. *OMNI linear programming system: User and operating manual.* Haverley Systems, Denvile, USA, 1976.

[12] W. Hock and K. Schittkowski. *Test Examples for Nonlinear Programming Codes.* Springer Verlag, Berlin, 1981. Lectures Notes in Economics and Mathematical Systems 187.

[13] P. J. Huber. *Robust Statistics.* J. Wiley and Sons, New York, 1981.

[14] J. J. Moré. A collection of nonlinear model problems. Technical Report ANL/MCS-P60-0289, Argonne National Laboratory, Argonne, USA, 1989.

[15] J. J. Moré, B. S. Garbow, and K. E. Hillstrom. Testing unconstrained optimization software. *ACM Transactions on Mathematical Software*, 7(1):17–41, 1981.

[16] K. Schittkowski. *More Test Examples for Nonlinear Programming Codes.* Springer Verlag, Berlin, 1987. Lecture notes in economics and mathematical systems, volume 282.

[17] Ph. L. Toint. Test problems for partially separable optimization and results for the routine pspmin. Technical Report 83/4, Department of Mathematics, FUNDP, Namur, Belgium, 1983.

Index