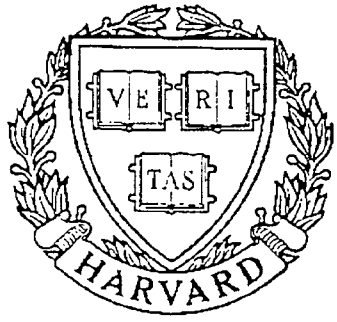
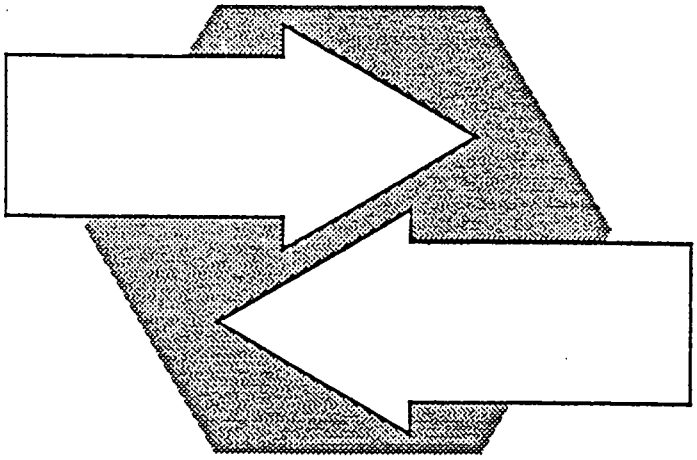
TECHNICAL RESEARCH REPORT

SYSTEMS RESEARCH CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), the University of Maryland, Harvard University, and Industry

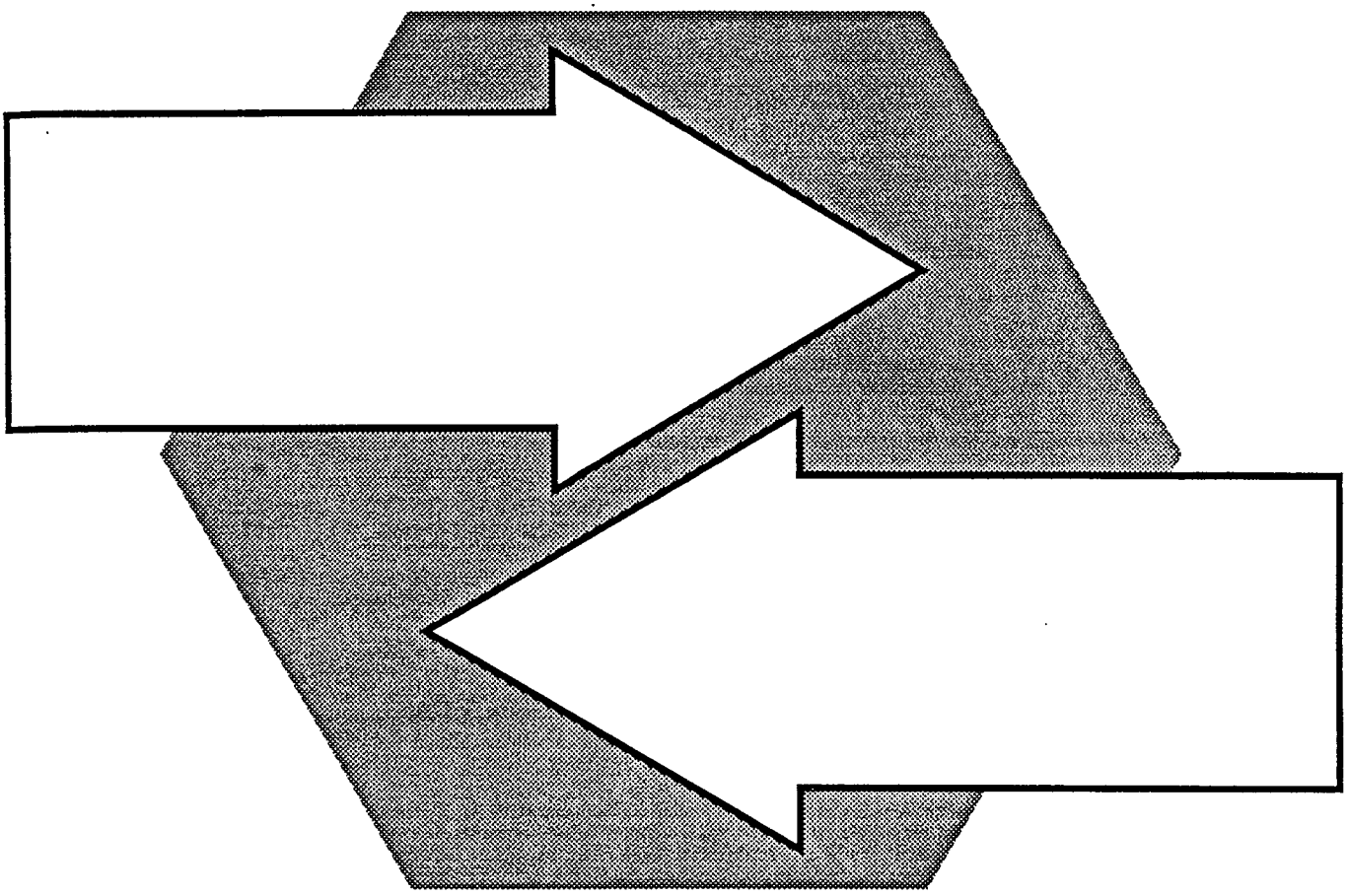


CONSOLE

User's Manual
(Version 1.1, Released 6/15/90)

by M.K.H. Fan, A.L. Tits, J. Zhou, L.-S. Wang and J. Koninckx

TR 87-212r2



CONSOLE

User's Manual (Version 1.1, Released 6/15/90)

Michael K.H. Fan, Andre L. Tits,
Jian Zhou, Li-Sheng Wang and Jan Koninckx

Systems Research Center, The University of Maryland
College Park, Maryland 20742

CONSOLE User's Manual (Version 1.1, Released 6/15/90)

PREFACE

The CONSOLE tandem is a tool for optimization-based design of a large class of systems. The essential requirements are that a simulator be available for evaluating the performance of instances of the system under consideration and that the parameters to be optimally adjusted are allowed to take on any real value in a given range. To date, CONSOLE has been used on applications as diverse as design of various circuits, design of controllers for a flexible arm, a high performance aircraft, a robotic manipulator, or determination of optimal flow rate and temperature profile for a copolymerization reactor.

CONSOLE has been developed on Unix systems. While it has already been ported to VMS, its use on Unix systems is strongly encouraged, as the Unix version is subjected to intensive testing at the University of Maryland, and as certain current or future features (e.g., graphics related) will not be available on other versions. In this manual, Unix specific features are often not explicitly mentioned as such and this may at times be confusing for the non Unix user.

This manual is essentially self-contained, although the serious CONSOLE user would be well advised to consult at least reference [1]. Some familiarity with the C language is assumed throughout and assistance of a local C guru may be helpful when dealing with the question of interfacing CONSOLE with simulators.

The manual is organized as follows. In Chapter 1, the ideas and principles upon which CONSOLE is constructed are outlined and the design methodology underlying CONSOLE is sketched. Chapter 2 introduces the novice user to CONSOLE by way of a simple tutorial example. This chapter is strongly recommended to new users as it leads them step by step through a CONSOLE session. Chapter 3 is entirely devoted to CONVERT. It includes a thorough description of the different data types, assignments and commands that form the CONVERT syntax. Chapter 4 discusses SOLVE. The essential features of the optimization algorithm are outlined and the operation of SOLVE is discussed. Special attention is given to the interactive capabilities of SOLVE, in particular the Pcomb display. In Chapter 5, the question of using an interface between SOLVE and simulators of the user's choice is discussed. A general structure for such an interace is given. Finally, Chapter 6 presents two design examples. Appendices A and B consist of reference manuals, for CONVERT and SOLVE respectively.

Development of the CONSOLE package and preparation of this User's Manual has been a team effort. Many of the ideas that led to the conception and implementation of CONSOLE came out of our daily experience with the Berkeley DELIGHT system and its offshoot DELIGHT.MaryLin, and our continuous interaction with Dr. W.T. Nye. We wish to express deep gratitude to Bill for this. Central to the operation of SOLVE (one of the components of the CONSOLE tandem) is the technique of dynamic loading. Chris Torek introduced us to this technique and assisted us many times with his expert advice. We address him many thanks. Several undergraduate and graduate students at the Chemical and Electrical Engineering Departments of the University of Maryland spent long hours trying out preliminary versions of CONSOLE and provided us with invaluable feedback. Of particular help here were Digendra Butala, Xin Chen and Tam Nguyen. All are gratefully acknowledged. Many thanks are due to Dr. E. Panier who thoroughly proofread an early version of this manual and pointed out many potential sources of confusion.

CONSOLE Version 1.1 invokes several pieces of codes written elsewhere. This includes the *xorral* routine written by C. Lemaréchal of INRIA (computation of the projection of the origin on a polytope), the QPSOL quadratic programming solver developed at Stanford University [2], as well as many short pieces of code (in particular for symbol table look-up) borrowed from the book "The C Programming Language" by B.W. Kernighan and D.M. Ritchie. Also, at several places, CONSOLE code is strongly inspired from routines due to others. This includes our dynamic loader for which we borrowed ideas from a program written by P. Powell at the University of Waterloo, as well as CONSOLE's graphics routines, patterned after routines from the DELIGHT graphics library written by W.T. Nye at the University of California, Berkeley. This manual was typeset using D. Knuth's TEX system and references were organized by J. Alexander's Tib bibliographic preprocessor to TEX.

Finally, the development of the CONSOLE package and the preparation of this manual would not have been possible without the support of the National Science Foundation (grants No. DMC-84-20740 and DMC-88-15996 and NSF's Engineering Research Center Program No. NSFD-CDR-88-03012) and that of the Westinghouse Corporation.

Main Enhancements in Version 1.1

(i) New, much more powerful optimization algorithm (see Section 4.2).

(ii) New commands in CONVERT (global) and in SOLVE (algo, freeze/unfreeze, goutput).

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Motivation and Scope

CONSOLE is a software tandem for interactive, optimization-based design. It has been developed (and is under further development) at the University of Maryland, College Park.

The most challenging task when designing a complex engineering system is that of coming up with an appropriate system *structure.* This task calls extensively upon the engineer's ingenuity, creativity, intuition and experience. After a structure has been (maybe temporarily) selected, it remains to determine the *best* value of a number of *design parameters.* The engineer's input is still essential here, as multiple tradeoffs are bound to appear. However, except in the simplest cases, achieving anything close to optimal would be impossible without the support of numerical optimization. Providing such support while emphasizing tradeoff exploration through man-machine interaction is the purpose of interactive optimization-based design packages such as CONSOLE.

As suggested above, design problems typically involve several competing objectives. Also, while some design specifications must be met imperatively, others are amenable to tradeoffs. Another typical aspect of optimization problems arising from design problems is that it is generally impossible or at least impractical for the designer to exactly characterize at the outset what (s)he means by an optimal design. Rather, a congenial environment would allow the designer to refine his characterization of optimality as suboptimal designs are obtained. A design methodology was recently developed, based on these considerations [1]. The CONSOLE package is based on this methodology.

A designer using an optimization-based design package typically goes through two phases while designing a system. In the first phase, the designer formulates the problem; some time is spent checking and debugging this problem formulation. During the second phase, solution of the optimization problem is attempted. These two phases have different characters. The first phase is comparable to writing a classic C or Fortran program and debugging it. During the second phase the user-machine interaction is more thorough and

qualitatively different, typically involving graphics. Accordingly, CONSOLE is composed of two main programs: CONVERT and SOLVE. CONVERT is a batch program. It takes as input the Problem Description File — a file containing a description of the optimization problem to be solved — and compiles it into two binary files. After reading in these two files, SOLVE solves the optimization problem in close interaction with the designer, allowing him/her to explore design tradeoffs.

The range of problems that can be solved efficiently using a CAD tool depends very much on the ability of this tool to be interfaced with arbitrary simulators. For instance, the design of a control system will, in general, rely upon the characteristics of the plant, and therefore at least an approximate model of the plant under study has to be made available to the CAD tool. The CONSOLE tandem allows for an easy interfacing of almost any simulator the user has available.

The following are among the main features of the CONSOLE tandem :

- ⋆ The problem description is closely related to the character of a design problem.
- ⋆ The problem description syntax is strict, but not hard to use. The C language is used to describe objectives, functional objectives, constraints and functional constraints. The Problem Description File is easy to read and understand.
- ⋆ CONVERT allows easy debugging of the Problem Description File.
- ⋆ SOLVE is interactive, with short and clearly defined commands providing efficient communication between the program and the user.
- ⋆ Interactive graphics provide the user with easy-to-interpret information on the current design.
- ⋆ Typically, one or more system simulators are invoked in the Problem Description File. Essentially, any simulator can be used by the designer without complicated procedures to interface it with SOLVE. Building in a simulator is not more difficult than linking a program.

Figure 1 illustrates the structure of CONSOLE. The Problem Description File is the input to CONVERT, which checks for all possible syntax errors and some logic errors and then generates two files. One file is a data file which contains, among other things, the names of design parameters and specifications, as well as *good* and *bad* values or curves to be used for tradeoff exploration (see below). The other file is an object file. It contains a

compiled version of the various specifications (objectives, functional objectives, constraints and functional constraints) . Both of these files are input to SOLVE, together with any object files for a simulator (or simulators) the user wishes to use. SOLVE then iterates together with the user to obtain a solution with maximum or sufficient satisfaction.

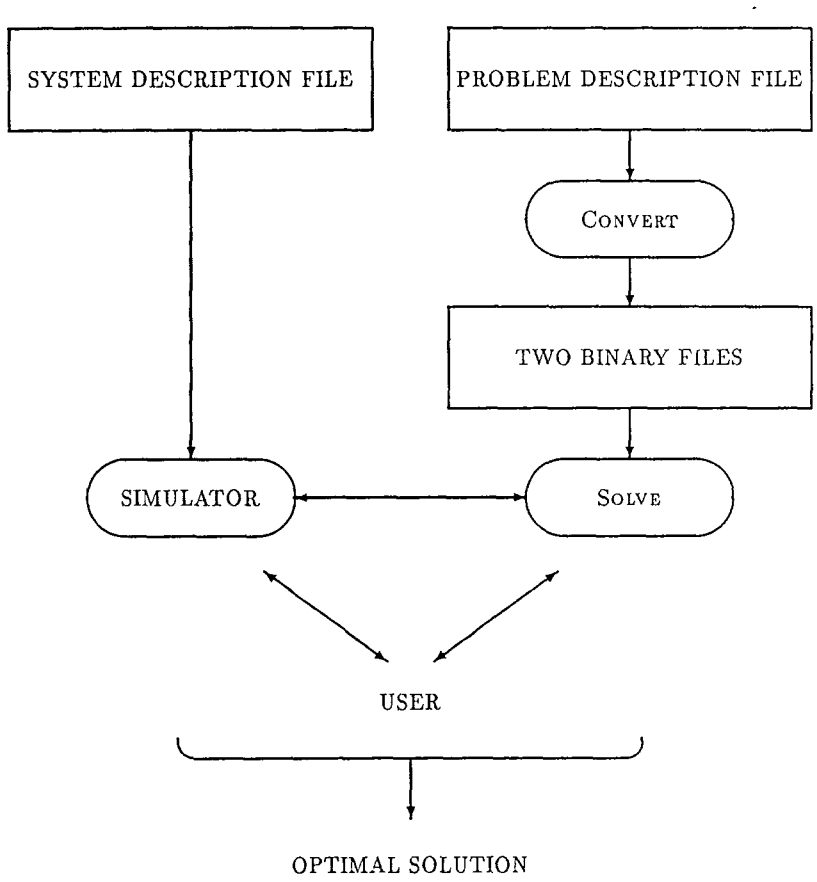


Figure 1: Structure of CONSOLE CAD tandem

1.2 Problem Description

In an attempt to better represent real world design problems, the methodology proposed in [1] allows for three qualitatively different types of design specifications. An

objective is a specification of a quantity that should be minimized or maximized. Typically multiple competing objectives are present. A *hard constraint* is a specification of a quantity that must achieve a specified threshold, or the corresponding design has no or little value. A *soft constraint* is a specification of a quantity that should achieve or at least approach a specified threshold, i.e., should be minimized or maximized as long as this threshold is not achieved. Soft constraints can be thought of as intermediate between objectives and hard constraints. A precise meaning is then given to the optimization problem by means of *good* and *bad* values assigned by the designer to each objective and soft constraint, according to the following uniform satisfaction/dissatisfaction rule: having any of the various objectives or soft constraints achieve its corresponding good value should provide the same level of *satisfaction* to the designer, while having any of them achieve its bad value should provide the same level of *dissatisfaction.* Also, the good value of a soft constraint must be the corresponding target value. Each objective and soft constraint will later be scaled by SOLVE using its good and bad values according to the formula

$$\text{scaled\_value} = \frac{\text{raw\_value} \ - \ \text{good\_value}}{\text{bad\_value} \ - \ \text{good\_value}},$$

so that achieving the good value will yield a scaled value of 0 while achieving the bad value will yield a scaled value of 1. For possible use in phase 1 (see below) where the maximum hard constraint violation is forced to decrease, hard constraints are also assigned good values (which are the threshold values) and bad values (which must be consistent across hard constraints). The only exception is that of hard bounds on the allowable values of individual design parameters. For these, no good and bad values need be specified, as there is little technical difficulty in keeping them satisfied throughout the optimization process. The resulting optimization problem is as follows:

$$(P) \qquad \begin{array}{lll} \underset{x}{\text{minimize}} & \text{obj}_k(x) & \forall k \\ \text{subject to} & \text{soft}_i(x) \le 0 & \forall i \\ & \text{hard}_j(x) \le 0 & \forall j \\ & \text{hard\_bound}_\ell(x) \le 0 & \forall \ell \end{array}$$

where obj_k, soft_i, hard_j are now *scaled values* of objectives, soft constraints, and hard constraints (except hard bounds), respectively and where hard_bound_ℓ represent the hard

bound type constraints. Problem (P) is then assigned three different meanings, corresponding to three different cases (or *phases*) according to feasibility or infeasibility of x with respect to hard and soft constraints.

Phase 1: not all hard constraints are satisfied, (P) takes the form

$$\begin{array}{lll} \text{minimize} & \max_j & \text{hard}_j(x) \\ x & & \\ \text{subject to} & & \text{hard\_bound}_\ell(x) \leq 0 \quad \forall \ell \end{array}$$

Phase 2: all hard constraints are satisfied. Not all (scaled) objectives and soft constraints are nonpositive, (P) takes the form

$$\begin{array}{lll} \text{minimize} & \max_{k,i} & \{\text{obj}_k(x), \text{soft}_i(x)\} \\ x & & \\ \text{subject to} & & \text{hard}_j(x) \leq 0 \quad \forall j \\ & & \text{hard\_bound}_\ell(x) \leq 0 \quad \forall \ell \end{array}$$

Phase 3: all hard constraints are satisfied and all (scaled) objectives and soft constraints are nonpositive; (P) takes the form

$$\begin{array}{lll} \text{minimize} & \max_k & \text{obj}_k(x) \\ x & & \\ \text{subject to} & & \text{soft}_i(x) \leq 0 \quad \forall i \\ & & \text{hard}_j(x) \leq 0 \quad \forall j \\ & & \text{hard\_bound}_\ell(x) \leq 0 \quad \forall \ell \end{array}$$

Typically most of the optimization run and user interaction (see below) will take place in phase 2 or 3.

To keep the exposition simple, we have so far left out the question of *functional* specifications, i.e., functional objectives and functional constraints. These are specifications according to which some quantity which depend on some free parameter (e.g., time or frequency) must be made small or large for all values of this parameter. Such are, e.g., specifications on some time or frequency response of a dynamical system. Similar to *ordinary* (non-functional) specifications, these can be objectives, hard constraints or soft constraints. They are normalized using user specified good and bad *curves* (i.e., functions of the free parameter), according to the formula (ω represents the free parameter)

$$\text{scaled\_valued}(\omega) = \frac{\text{raw\_value}(\omega) - \text{good\_value}(\omega)}{\text{bad\_value}(\omega) - \text{good\_value}(\omega)}$$

To complete this overview of the problem formulation, it remains to mention the concept of *nominal variation.* For each design parameter, the designer is requested to provide, besides an initial guess, a quantity indicating his degree of confidence in this guess, i.e., how close he believes it may be away from the optimal value. This is the purpose of the nominal variation, which should be selected according to the *uniform parameter influence rule*: a modification of any design parameter by an amount equal to its nominal variation should influence the most binding objectives and constraints to roughly the same degree. Admittedly, this rule is impractical. Thus it is suggested to choose as nominal variation the difference (in absolute value) between the initial guess and the next value the designer would try if he had to proceed 'by hand'. The nominal variations are used for the initial scaling of the parameter space according to the formula

$$\text{scaled\_value} = \frac{\text{raw\_value}}{\text{nominal\_variation}}$$

Thus the penalty for an improper choice of the nominal variation is slower initial progress of the optimization process. If no nominal variation is provided, the default value of 1 (no scaling) will be used.

1.3 Interactive Solution

When dealing with a nontrivial problem, it cannot be hoped that the set of good and bad values and curves provided by the designer will be such that the *optimal solution* obtained by the optimization process (be it in phase 2 or in phase 3) be the closest to the designer's aspirations. Thus typically the designer will want to interactively modify some of the good and bad values or curves, either to make them more realistic in view of the limitations just encountered, or to either tighten or relax one specification or another in order to alter the tradeoff solution. Such interaction is an essential component of the methodology of [1](see also [3]). To aid him/her in his/her tradeoff exploration, information is conveyed to the designer in numerical and graphical form, concerning the performance of the current design (Pcomb [1]).

Above, we have outlined the design methodology proposed in [1] to the extent needed for a basic understanding of CONSOLE; the reader is encouraged to consult [1] to get a better feel for the motivation and implications of this methodology.

A TUTORIAL EXAMPLE

2.1 Introduction

Let us begin with finding a solution for a simple nonlinear programming example. Our aim is to show how to describe the design problem in the Problem Description File, how to invoke CONVERT to translate the Problem Description File into two binary files, and how to invoke SOLVE to perform optimization interactively, and obtain a local optimal solution. The discussion will be kept simple and will not go into too many details so that a general picture of the design procedure using the CONSOLE tandem can be grasped. The user is strongly encouraged to run this example and to try out various modifications.

As pointed out in Chapter 1, the CONSOLE tandem can be used for design of a broad class of engineering systems, provided simulators are available. However in the present example, no simulator is necessary. In fact, due to the simplicity of this example, the simulator is actually contained in the Problem Description File. This point will become clear in a moment.

2.2 Example: A Simple Nonlinear Programming Problem

Consider the two-variable constraint optimization problem

$$\begin{aligned} \underset{x,\ y}{\text{minimize}} \quad & (x-1)^2 + (y-2)^2 \\ \text{subject to} \quad & x \geq 0 \\ & x + y \leq 1 \end{aligned}$$

and suppose that $x \geq 0$ is a hard constraint while $x + y \leq 1$ is soft with, as *good* value, the given target value of 1, and with a *bad* value of 2 (see Section 1.2). For the objective, assume a good value of 1 and a bad value of 4. Finally, suppose that our best *a priori* guess is $x = 5$ and $y = 10$, to be used as initial values for the optimization process.

Problem Description

A suitable Problem Description File for the given problem is as follows (the numbers

at the end of each line are used for the purpose of explanation only; they do not actually appear in the Problem Description File)

```
/* Solving a nonlinear programming   */      1
/* problem using the CONSOLE tandem. */      2
                                             3
design_parameter x init=5  min=0             4
design_parameter y init=10                   5
                                             6
objective "quadratic"                        7
   minimize {                                8
      import x, y;                           9
      return (x-1)*(x-1)+(y-2)*(y-2);       10
      }                                     11
   good_value=1                             12
   bad_value=4                              13
                                            14
constraint "linear" soft                    15
   {  import x, y;                          16
      return x+y;                           17
      }                                     18
   <= good_value=1                          19
      bad_value=2                           20
```

The above consists of four sections.

1. CONVERT follows the 'C' rule that any text between '/*' and '*/' is treated as a comment. Comments can span more than a line (the '*/' and '/*' at the end of line 1 and at the beginning of line 2 are used here for mere esthetic reasons).
2. Design parameters are declared on lines 4 and 5.* For instance, line 4 declares x as a design parameter, assigns 5 as its initial value and indicates that the value 0 is a hard lower bound.+ An expression such as 'min =[1,0]' instead of 'min=0' would have meant a *soft* lower bound with good and bad values of 1 and 0 respectively (good value comes first); 'max' could have been used similarly, alone or in conjunction with 'min'. For constraints that are not simple bounds on design parameter values, see 4 below.
3. Lines 7 through 13 show how a simple objective function is declared. First, a name (quoted string) is assigned to the objective. This name will be of great help later on, in the interactive phase (SOLVE). Then the objective function is specified, between curly

* Although, conceptually, such declarations may be unnecessary in some cases (except for the initial guess), they are mandatory in CONVERT, following the 'strong typing' philosophy.

+ As pointed out in Section 1.2, hard parameter bounds are forced to be satisfied throughout the optimization process, so that good and bad values are unnecessary.

brackets, following the key word 'minimize' (alternatively, 'maximize' could be used). The code between the curly brackets is 'C' code, except for the 'import' statement indicating that x and y are the identifiers declared earlier, outside the braces (in this case, design parameters). This *pseudo-C code* must include a return statement assigning a value to the objective. The assignments on lines 12 and 13 are self-explanatory. Code similar to that of lines 7 to 13 should be included for each of possibly many objectives.

4. Finally, the remaining constraint (not a simple bound on a design parameter) is specified on lines 15 to 20. Again, a name is assigned (quoted string on line 15), this time followed by the keyword 'soft' indicating a soft constraint ('hard' could have been used instead). Lines 16 to 20 follow a syntax similar to that of lines 8 to 13, except for the absence of the keyword 'minimize' and the presence of the operator '<=' (could have been '>=') on line 19.

Let us assume that the Problem Description File has name *qp*. The next step is to invoke CONVERT by typing

```
convert qp
```

A message similar to the following appears on the terminal

```
          Welcome to CONVERT, Version 1.1 (Released 6/15/90)

                    Copyright (c) 1988, 1990
                              by
            Michael K.H. Fan    Andre L. Tits    Jian Zhou
                    Li-Sheng Wang    Jan Koninckx
                         All Rights Reserved.

[processing qp]
[compiling qp.c]
[writing qp.d]
```

The bottom three lines (in brackets) correspond to three successive operations performed by CONVERT. Each of these lines appears on the screen at the start of the corresponding operation. First CONVERT generates a C routine, *qp.c*, containing the definitions of objective and constraint functions. The main body of this routine consists of the various pseudo-C code portions of *qp*. The *import* statements are appropriately compiled into formal arguments of the C routine and suitable heading and tailing parts are included.

After *qp.c* has been generated, CONVERT invokes the C compiler to generate the object file *qp.o*. If no error has been detected, the file *qp.c* is removed. Finally, CONVERT generates a binary data file, *qp.d*, containing information such as names, initial values and bounds of design parameters, as well as names and good and bad values of objectives and constraints. If any error is detected, CONVERT reports the corresponding file name and line number and aborts.

Problem Solution

Once a Problem Description File has been processed successfully by CONVERT, the next step then is to invoke SOLVE by typing

```
solve qp
```

to perform optimization interactively. A message similar to the following appears on the terminal.

```
            Welcome to SOLVE, Version 1.1 (Released 6/15/90)

                        Copyright (c) 1988, 1990
                                by
             Michael K.H. Fan    Andre L. Tits    Jian Zhou
                    Li-Sheng Wang    Jan Koninckx
                         All Rights Reserved.

[loading/reading qp.o]
[reading qp.d]
[calling simulator initialization (if any)]
[calling problem initialization (if any)]

type "help" for help
type "help info" for information
type "help news" for local news

<0>
```

Similar to CONVERT, the lines in brackets correspond to successive operations performed by SOLVE. First SOLVE invokes the Unix loader to load *qp.o* to a temporary file created in the directory '/tmp' and then reads the resulting file into memory if there is no error during the loading step. This process is called *dynamic loading* or *incremental loading.* SOLVE then reads in the binary data file *qp.d.* Following that, SOLVE would perform the simulator initialization if a routine named *simlat* had been given to SOLVE (see Chapter 5) and would perform the problem initialization if it had been declared in the Problem

Description File (see Chapter 3) (neither actually occurs in the present case). SOLVE then prints out the information on how to get the on-line help for the use of SOLVE; on how to obtain the CONSOLE package; and on how to read local news if there is any. Finally, SOLVE gives the prompt '<0>' which indicates that it is ready to receive commands from the user to perform optimization.

A typical interactive session could be as follows. First the user types

```
identify
```

and gets the response

```
PROBLEM: qp
  2  Design Parameter(s)
  1  Objective(s)
  1  Constraint(s)
<0>
```

which gives information on the problem being solved. The command

```
print
```

prompts the output

```
Name        Value        Variation  wrt 0   Prev   Iter=0
x           5.00000e+00  1.0e+00
y           1.00000e+01  1.0e+00
<0>
```

indicating the current (initial) value of the design parameters and the corresponding nominal variations. To get a detailed explanation of the command *print*, the user can invoke the on-line manual by typing

```
help print
```

while typing

```
help
```

gives general information of the on-line manual. The purpose of the session being to solve the optimization problem, the user may then type

```
run 2
```

requesting that 2 iterations of the optimization algorithm be executed. The new prompt

```
<2>
```

indicates that the first two iterations have been completed. Typing

```
print
```

displays the new value of the design parameters (at iteration 2), the percentage change of design parameters with respect to iteration 0 (initial values) and previous iteration (iteration 1 in this case), respectively

```
Name        Value         Variation  wrt 0   Prev   Iter=2
x           0.00000e+00   1.0e+00    -100%   -100%
y           1.64042e+00   1.0e+00     -83%    -77%
<2>
```

Here for example, design parameter y at iteration 2 has value `1.64042`, which is 83% below its initial value (10.0) and 77% below its value at the previous iteration (iteration 1). At this time one could issue the command

```
pcomb
```

which prompts the output

```
Pcomb (Iter=  2) (Phase 2) (MAX_COST_SOFT= .640424)

SPECIFICATION     PRESENT   GOOD                G          B              BAD
O1   quadratic    1.13e+00  1.00e+00 ==========*          |              4.00e+00
C1   linear       1.64e+00  1.00e+00 =================*   |              2.00e+00
```

The *pcomb* command displays some relevant information for the current iteration (it stands for *performance comb* and is borrowed from [1,4]). Among other things, it indicates the current phase number in the optimization algorithm (`Phase 2`); the maximal scaled value of objectives and soft constraints (`MAX_COST_SOFT= 0.640424`); the names (`quadratic` and `linear`), present values (`1.13+00` and `1.64+00`), and good (`1.00e+00` and `1.00e+00`) and bad (`4.00e+00` and `2.00e+00`) values of objectives and soft constraints at the current iteration. Finally, it graphically displays the scaled values of all specifications to help the user quickly assess the current design (see Section 4.4 and Appendix B).

Commands *print* and *pcomb* can be used as arguments of the command *run*. For instance, typing

```
run 2 print pcomb
```

requests that 2 more iterations be run, and that the *print* and *pcomb* commands be executed at the end of each of these iterations. The optimization algorithm in SOLVE (and/or the computation of the simulator if any) can be traced by the *trace* command. For example, typing

```
trace
```

turns on the trace for the optimization algorithm. A subsequent input

```
run 1
```

shows

```
[RUNNING FSQP ALGORITHM]

[computing specifications for iteration 2]
[computing search direction]
   [computing gradients of specifications by finite difference]
   [computing search direction]
[computing stepsize along search direction]
   [trying with stepsize = 1.000e+00 ...
   SPECIFICATION     PRESENT           SCALED
   O1   quadratic    1.591571787e+00   1.971905956e-01
   C1   linear       1.229562171e+00   2.295621712e-01
                                                    accepted]

[computing specifications for iteration 3]
<3>
```

indicating the various steps in the optimization algorithm when they are being performed. The user may then type

```
trace 0
```

to turn off the trace, and type

```
run 10
```

to request 10 more iterations. The following output would appear

```
          CONGRATULATIONS !!

          It seems that SOLVE has obtained a local optimal solution
          for your design problem. Type "help optimal" for the
          optimality condition.

          ENJOY !!

Optimal code 5

<7>
```

This message indicates that SOLVE has found a locally optimal solution at the end of iteration 7 and that the *optimal code* is 5. The meaning of this code is clarified by typing

```
help optimal
```

Typing

```
print
```

gives the local optimizer

```
Name        Value        Variation  wrt 0   Prev   Iter=7
x           1.02084e-01  1.0e+00     -97%    0%
y           1.10208e+00  1.0e+00     -88%    0%
<7>
```

and typing

```
pcomb
```

displays the corresponding values of the objective and constraint

```
Pcomb (Iter=  7) (Phase 2) (MAX_COST_SOFT= .204168)

SPECIFICATION    PRESENT   GOOD              G          B            BAD
O1   quadratic   1.61e+00  1.00e+00 ============*        |          4.00e+00
C1   linear      1.20e+00  1.00e+00 ============*        |          2.00e+00
<7>
```

There are some other features in SOLVE. For instance, plotting functional specifications; interactively changing the value of a design parameter; interactively modifying selected good/bad values or curves; storing values of design parameters into a file; interrupting the optimization algorithm; etc. All of these will be the topics of Chapter 4. Also, in Chapter 5, we will discuss the question of linking SOLVE to existing programs (simulators).

CHAPTER 3

CONVERT

3.1 Introduction

In this chapter we discuss the language in which a Problem Description File is written. This language is not meant for general purpose programming. Rather, its structure was only required to be versatile enough to allow one to describe a design problem. Thus its syntax is very simple and consists of only two types of statements, *assignment* and *command*. An *assignment* gives a value to a variable and a *command* performs a task such as declaration of a design parameter or of a specification. Assignments and commands are constructed from identifiers, expressions, operators, and sections of pseudo-C code.

3.2 Identifiers, Expressions, Operators, and Pseudo-C Code

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore '_' is considered a letter. Upper and lower case are distinguished. An identifier must have a type. An identifier type is either predeclared by CONVERT or declared by the user, and later redeclaration of an identifier is prohibited. The possible identifier types will be the topic of the next section.

An expression consists of identifiers, balanced parentheses and operators. Operators are unary operators (+,-), exponentiation operator (**), multiplicative operators (*,/) and additive operators (+,-). Table 1 summaries the rules for precedence and order of evaluation of all operators. * An expression has a *double precision* floating point value which is called *the value of the expression.* An expression is evaluated when CONVERT encounters it. Therefore, an expression must be evaluable in the sense that it can be finally translated to a form with only constants, parenthesis and operators and computed with no syntax errors. See Appendix A for the syntax rules of expressions. During the evaluation, besides

* These operators are strictly for CONVERT expressions. Expressions in the sections of pseudo-C code of a Problem Description File must abide by the rules of C.

syntax errors, errors such as division by zero, or attempt to evaluate the logarithm of a non-positive number are also detected and reported.

Operator		Order of Evaluation
+, -	unary affirmation, negation	left to right
**	exponentiation	right to left
*, /	multiplication, division	left to right
+, -	addition, subtraction	left to right

Table 1: Operators

As mentioned in Section 2.2, we call *pseudo-C code* certain sections of code, enclosed in curly brackets, consisting of valid C code augmented by the *import* statement. An *import* statement consists of the identifier `import` followed by a list of identifiers separated by commas. These identifiers must have been declared with identifier type *variable*, *define* or *design parameter* (see Section 3.4 below). The import statement makes them accessible from within the section of pseudo-C code. An *import* statement ends at a semicolon or a newline character, whichever comes first. In Section 3.5 below we discuss the question of how to call C, Fortran and Pascal routines from inside a section of pseudo-C code.

3.3 Identifier Types

The identifier types can be classified into six categories: *variable*, *define*, *design parameter*, *command*, *function* and *keyword*. All the identifiers with identifier types command, function or keyword are predeclared by CONVERT, i.e., the user cannot declare any identifier with one of those identifier types. The declaration of identifier type is done by context, as discussed below. An identifier may be declared only once.

Variable

An identifier with type *variable* is an identifier having a *double precision* floating point value. The declaration of a variable and the assignment of its value are done by an *assignment* (see below). A variable which appears in an expression is replaced by its value when the expression is evaluated by CONVERT.

Define

CONVERT

Command	Functionality
constraint	declare a constraint
define	declare a define
design_parameter	declare a design parameter
exit	force CONVERT to exit the current file
functional_constraint	declare a functional constraint
functional_objective	declare a functional objective
global	add plain C code
include	read input from a file
initialization	declare an initialization routine
objective	declare an objective
set	change the initial value of a design parameter
trace	trace the value of a variable

Table 2: Commands

An identifier with type *define* is an identifier having a definition, which is a string of characters. The declaration of a define and the assignment of its definition are done by the command *define* (see below). A define which appears in an expression will be replaced by its definition when the evaluation of the expression is being performed. Furthermore, the replacement string is rescanned by CONVERT.

Design parameter

An identifier with type *design parameter* is an identifier whose value will be adjusted by the optimization algorithm. A design parameter has 4 attributes: *initial value*, *nominal variation*, *minimum value* and *maximum value*. All of these have default values. Declaring a design parameter and assigning its attributes are done via the command *design_parameter* (see below).

Command

An identifier with type *command* is an identifier which performs a special task. A command may have arguments, some of which may be optional. Details of commands will be discussed in Section 3.4. All commands are predeclared by CONVERT. Their names and functionalities are given in Table 2.

Function

An identifier with type *function* is an identifier which takes argument(s) in balanced parentheses, performs a specific function and returns a double precision floating point value. Functions must appear in expressions. All functions are predeclared by CONVERT. Table 3 gives a summary of various functions.

Function	Description
abs(x)	returns the absolute value of x
acos(x)	returns the arc cosine of x in $[0, \pi]$
asin(x)	returns the arc sine of x in $[-\frac{\pi}{2}, \frac{\pi}{2}]$
atan(x)	returns the arc tangent of x in $[\frac{-\pi}{2}, \frac{\pi}{2}]$
atan2(x,y)	returns the arc tangent of x/y in $[-\pi, \pi]$
cos(x)	returns trigonometric cosine function of x (x in radians)
exp(x)	returns the exponential of x
log(x)	returns the natural logarithm of x
log10(x)	returns the base 10 logarithm of x
max2(x,y)	returns x or y, whichever is larger
min2(x,y)	returns x or y, whichever is smaller
sign(x)	returns 1 if $x \geq 0$, –1 otherwise
sin(x)	returns trigonometric sine function of x (x in radians)
tan(x)	returns trigonometric tangent function x (x in radians)

Table 3: Functions

Keyword

An identifier with type *keyword* is an identifier which is used in the argument of a command, to separate arguments and increase readability. All keywords are predeclared by CONVERT. Table 4 lists them, together with the names of the commands in which they may occur.

3.4 Assignment and Commands

Keyword	Occurs in Command
bad_curve	functional_objective, functional_constraint
bad_value	objective, constraint
by	functional_objective, functional_constraint
dec	functional_objective, functional_constraint
for	functional_objective, functional_constraint
from	functional_objective, functional_constraint
good_curve	functional_objective, functional_constraint
good_value	objective, constraint
hard	constraint, functional_constraint
init	design_parameter
max	design_parameter
maximize	objective, functional_objective
min	design_parameter
minimize	objective, functional_objective
soft	constraint, functional_constraint
times	functional_objective, functional_constraint
to	functional_objective, functional_constraint
variation	design_parameter

Table 4: Keywords

In this section we specify the syntax rules of an assignment and of various commands. We use the convention that anything printed in boldface must appear exactly as is, while anything surrounded by '⟨⟩' stands for a generic term and its meaning is given in Appendix A by a BNF-like rule. For example, we will write

`define` ⟨*identifier*⟩ ⟨*quoted string*⟩

The user is referred to Appendix A for more details.

Assignment

An assignment has the form

⟨*identifier*⟩ = ⟨*expression*⟩

If ⟨*identifier*⟩ has not been declared before, the assignment declares it to be a variable with the value of ⟨*expression*⟩. Otherwise, ⟨*identifier*⟩ must exist as a variable, and the assignment simply changes its value to the value of ⟨*expression*⟩. For example, the assignment

```
a = 1
```

declares a to be a variable if a has not been declared before and assigns to it the value 1. The subsequent assignment

```
a = a+1
```

changes that value to 2.

Define

The command *define* has the form

`define` ⟨*identifier*⟩ ⟨*quoted string*⟩

⟨*identifier*⟩ may not have been declared before. The command *define* then declares ⟨*identifier*⟩ to be a define with definition ⟨*quoted string*⟩ surrounded by parentheses. For example, the statement

```
define PI "3.1416"
```

declares the identifier `PI` to be a define with the definition `(3.1416)`. The reason that CONVERT adds balanced parentheses to the definition automatically is to avoid the possible ambiguity in the following case

```
define a "1+2"
```

Suppose that the definition of the identifier a were '1+2' rather than '(1+2)'. Then the expression '`a*a`' will be expanded to '`1+2*1+2`' which is incorrect.

A define cannot appear in its own definition directly or indirectly. For example, the following statements are incorrect (on two counts)

```
define a "a+1"
define b "c+1"
define c "b**2"
```

However, CONVERT does not check the recursion until the define is actually used in an expression.

Trace

The *trace* command is used to trace the value of a variable during the processing of the Problem Description File. It has the form

`trace` ⟨*identifier*⟩

⟨*identifier*⟩ here must be a variable. For example, if the first 4 lines of the file *test* are

```
a = 5
trace a
b = a**2 - a
trace b
```

then CONVERT will print

```
TRACE at line 2 in test, a = 5
```

after it processes the first *trace* and

```
TRACE at line 4 in test, b = 20
```

after the second.

Design_parameter

The *design_parameter* command declares a design parameter, i.e., a parameter which will be adjusted by the optimization algorithm. It has the form

```
design_parameter ⟨identifier⟩                \
   init=⟨expression⟩                         \
   variation=⟨expression⟩                    \
   min=⟨soft or hard expression⟩             \
   max=⟨soft or hard expression⟩
```

The command *design_parameter* terminates at the first newline character (the end of line character). If continuation lines are desired, as shown above, a backslash '\' must be used at the end of each of the continued lines. The command *design_parameter* is the only one with optional arguments. Specifically, any of

`init=`⟨*expression*⟩
`variation=`⟨*expression*⟩

`min=`⟨*soft or hard expression*⟩
`max=`⟨*soft or hard expression*⟩

can be omitted; the ones that are included can appear in any order. The specification

`init=`⟨*expression*⟩

if given, assigns as initial value (or 'initial guess') of the design parameter the value of ⟨*expression*⟩. The default value is 0. The line

`variation=`⟨*expression*⟩

if given, assigns to the nominal variation of the design parameter (see Section 1.2) the value of ⟨*expression*⟩. The default value is 1. The nominal variation must be strictly positive. In many cases, the absolute value of the initial guess is suitable for the nominal variation (unless it is zero). The line

`min=` ⟨*soft or hard expression*⟩

if given, indicates a lower bound constraint on the corresponding design parameter. ⟨*soft or hard expression*⟩ is either

`[` ⟨*expression*⟩`,` ⟨*expression*⟩ `]`

or

⟨*expression*⟩

The line

`min = [` ⟨*expression*⟩`,` ⟨*expression*⟩ `]`

indicates a soft lower bound constraint with good value being the value of the first ⟨*expression*⟩ and bad value being the value of the second. The good value in this case must be greater than the bad value since the design parameter is constrained from below. The line

`min =` ⟨*expression*⟩

indicates a hard lower bound constraint. The default is a hard constraint with value -10^{20} (i.e., no constraint). Similar rules apply to the line

`max =` ⟨*soft or hard expression*⟩

except that, in the case of a soft upper bound constraint, the good value must be smaller than the bad value, and the default is a hard bound with value 10^{20}.

Global

The *global* command allows the user to put a piece of plain C code into the C program generated by CONVERT. When *global* is encountered, CONVERT puts everything till the end of the line into the C program. CONVERT puts all such lines on the top of the C program in the same order as they appear in the Problem Description File.

The *global* command provides an easy way to declare, for instance, functions or variables that will be used in the pseudo-C code of the specifications. The following example shows a use of the command.

```
global double cost(x,y)
global double x, y;
global {
global    return (x-1)*(x-1)+(y-2)*(y-2);
global }

objective "quadratic"
   minimize {
      import x, y;
      double cost();     /* this line is not necessary */
      return cost(x,y);
      }
   good_value=1
   bad_value=4
```

Include

The *include* command has the form

include ⟨*quoted string*⟩

It causes the replacement of that line by the entire contents of the file with name ⟨*quoted string*⟩. If the named file does not begin with a slash '/' (i.e., it is given as a relative pathname), CONVERT searches in the directory in which the Problem Description File is located. For example, suppose the Problem Description File is

```
/glu/wangli/design/robot
```

The line

```
include "robot.additional"
```

in the Problem Description File is equivalent to

```
include "/glu/wangli/design/robot.additional"
```

which causes the named file to be processed. Include commands may be nested.

Exit

The *exit* command has the form

```
exit
```

It causes CONVERT to stop processing and leave the current file prematurely. That is, anything that follows in the current file will be skipped.

Objective

The *objective* command declares an objective. It has the form

`objective` ⟨*quoted string*⟩
⟨*optimize*⟩⟨*pseudo-C code*⟩
`good_value=` ⟨*expression*⟩
`bad_value=` ⟨*expression*⟩

Note that for this and the following commands, no backslash is necessary (but it is allowed) to indicate the continuation of lines (unlike the command *design_parameter*), as the arguments are all required. Above, the form of the command *objective* is written on multiple lines with indentation only for the sake of readability. It makes no difference if it is compacted into fewer lines or expanded to more lines with comments or empty lines in between. ⟨*quoted string*⟩ serves as an identification for the objective and could be a character string of any length. However only the first 10 characters will be considered by CONVERT. ⟨*optimize*⟩ is either '`minimize`' or '`maximize`'. The returned value of ⟨*pseudo-C code*⟩ is the value of the objective (which should be design parameter dependent) to be minimized or maximized. The last two lines then set the good value and bad value respectively. The two ⟨*expression*⟩ above should contain no blanks or tabs. The good value and bad value must be given in a consistent manner, i.e., for the objective to be minimized (resp. maximized), the good value must be smaller (resp. larger) than the bad value. The following two examples have identical objective functions.

Example 1.

```
objective "money spent" minimize {
   import x, y;  /* x and y are design parameters */
   double a;     /* local variable */
   a = x+y;
   return (a);
   } good_value=100  bad_value=200
```

Example 2.

```
objective "money spent"
                    /* comment here ... */
   minimize {
      import x, y;   /* x and y are design parameters */
      return x+y;    /* x+y is the money spent */
      }
   good_value = 100  /* comment here ... */
   bad_value  = 200  /* comment here ... */
```

Constraint

The *constraint* command declares a constraint. It has the form

`constraint` ⟨*quoted string* ⟩⟨*soft or hard*⟩
⟨ *pseudo-C code*⟩⟨*inequality*⟩
`good_value=`⟨*expression*⟩
`bad_value=`⟨*expression*⟩

⟨*soft or hard*⟩ is either '`soft`' or '`hard`' and ⟨*inequality*⟩ is either '`<=`' or '`>=`' The *constraint* command is very similar in form to the *objective* command, except for the indication of whether it is soft or hard. The definitions of soft and hard constraints have been given in Section 1.2. An example follows.

```
constraint "highest temperature" hard
   { import heat;
     return heat*3.2;   /* formula relating heat and temperature */
     }
   <= good_value = 55
      bad_value  = 60
```

Functional_objective

The *functional_objective* command declares a functional objective. It has the form

`functional_objective` ⟨*quoted string*⟩
`for` ⟨*identifier*⟩ `from` ⟨*expression* ⟩ `to` ⟨*expression*⟩ ⟨*mesh type* ⟩ ⟨*expression*⟩
⟨*optimize* ⟩⟨*pseudo-C code*⟩
`good_curve=` ⟨*pseudo-C code*⟩
`bad_curve=` ⟨*pseudo-C code*⟩

Again, ⟨*quoted string*⟩ serves as an identification of the functional objective. Recall that a functional objective is a specification according to which some quantity which depends on some free parameter must be made small or large for all values of this parameter in an interval (see Section 1.2). Practically, the interval is discretized to a finite set of points specified by

`for` ⟨*identifier*⟩ `from` ⟨*expression*⟩ `to` ⟨*expression*⟩⟨*mesh type*⟩⟨*expression*⟩

where ⟨*identifier*⟩ is the free parameter, the first two ⟨*expression*⟩ specify the interval and the discretization is given by ⟨*mesh type*⟩ and the last ⟨*expression*⟩. ⟨*mesh type*⟩ is either `by`, `times` or `dec`. For examples, the line

```
for t from a to b by c
```

requests that the corresponding functional objective be made small or large for t at all values of a, a+c, a+2c, ... until $t > b$. Similarly,

```
for t from a to b times c
```

indicates that t is to take values of $a, ac, ac^2, \ldots$ until $t > b$.

```
for t from a to b dec c
```

is the same as

```
for t from a to b times d
```

with $d = 10^{\frac{1}{c}}$ (c points per decade). The following is an example of a functional objective.

```
final = 5
functional_objective "upper bound" hard
   for t from 0 to final by final/20
   minimize {
      import x1, x2;
      return t*x1+x2;
      }
   good_curve = { return 2*t; }
   bad_curve  = { return 3*t+1; }
```

It should be noticed that the free parameter (t in this example) does not have any identifier type belonging to any of the categories given in Section 3.3. It is a C identifier declared (automatically) outside a *pseudo-C code* section with data type *double* and is known to any of the three *pseudo-C code* above without explicit declarations or *import* statements.

In fact, any further declaration of this parameter would create a local variable, and this usually results in errors. Following is an erroneous example.

```
final = 5
functional_objective "upper bound" hard
   for t from 0 to final by final/20
   minimize {
      import x1, x2, t;          /* no need to import t; detected by CONVERT */
      return t*x1+x2;
      }
   good_curve = { double t;      /* should not redeclare t */
                  return 2*t; }
   bad_curve  = { double t;
                  return 3*t+1; }
```

Another important point that must be remembered is that the good and bad curves should not be design parameter dependent. This is because their values are evaluated and stored at the very beginning of the optimization. No further evaluation will be performed during the optimization.

Functional_constraint

The *functional_constraint* command declares a functional constraint. It has the form

`functional_constraint` ⟨*quoted string*⟩
`for` ⟨*identifier* ⟩ `from` ⟨*expression* ⟩ `to` ⟨*expression*⟩ ⟨*mesh type*⟩ ⟨*expression*⟩
⟨*pseudo-C code* ⟩⟨*inequality*⟩
`good_curve=` ⟨*pseudo-C code*⟩
`bad_curve=` ⟨*pseudo-C code*⟩

The syntax of the command *functional_constraint* can be easily understood by comparing it to that of the *constraint* and *functional_objective* commands. Following is an example.

```
final = 10**3
functional_constraint "upper bound" hard
   for w from 0.1 to final dec 5  /* five mesh points per decade */
   {  import x1, x2;
      return w*(w-1)+(1-w)*(-3.0/4.0*x1+7.0/4.0)+w*(x1+x2);
      }
   <= good_curve = { return w*w; }
      bad_curve  = { return 0.5*w*w; }
```

Set

The *set* command has the form

`set` ⟨*identifier* ⟩ = ⟨*expression*⟩

Here ⟨*identifier*⟩ must be a design parameter. The *set* command sets the initial value of the design parameter to the value of ⟨*expression*⟩, irrespective of whether or not its initial value has been given by the command *design_parameter* or a previous *set* command. Files created by the *store* command (in SOLVE, see Appendix B) consist of instances of the *set* command. As an example, suppose that *gain* is a design parameter. Then

```
set gain = 1+2
```

produces the output

```
design parameter gain is set to 3.000
```

which indicates its initial value has been set to 3.000.

Initialization

The *initialization* command has the form

`initialization` ⟨*pseudo-C code*⟩

It declares a C routine that will be executed *only* one time before the optimization is performed. Usually, this C routine is used to initialize the design problem. This could include reading the system configuration under consideration (such as a circuit description if the simulator SPICE is used) or setting simulator-specific parameters to accommodate the design problem. Following is an example (assume the routine 'read_system_configuration' is provided by the user to perform one-time problem-specific initialization of the simulator).

```
initialization {                /* initialization */
   read_system_configuration();
   }
```

3.5 Calling Programs in a Section of Pseudo-C Code

In this section we discuss how to call programs in a section of pseudo-C code. Most of this quotes computer messages sent to us by Chris Torek, of the Computer Science Department; we gratefully acknowledge his help. Any program written in C, Fortran or Pascal can be called from pseudo-C code since their object codes are compatible. However, the following three rules must be respected: *consistent data type passing*, *identifier matching*

and *consistent data passing mechanism.*

Consistent Data Type Passing

It should be obvious that the data types in the calling program and called program must be consistent. For instance, when calling the following C program

```
foo (n, x, r)
int n;
double x[], r;
{
   ...
}
```

in the pseudo-C code, one may have

```
{ #define TOTAL 100
  double samples[TOTAL];
  import width;   /* width is a design parameter */
     ...
  foo (10, samples, width);
     ...
  }
```

Here we show an instance where an inconsistency occurs and is not easily noticed by the user. Suppose n is a variable declared by the statement

```
n = 10
```

Then the following usage of variable n in pseudo-C code causes inconsistency in data passing.

```
{ #define TOTAL 100
  double samples[TOTAL];
  import width;   /* width is a design parameter */
  import n;       /* n has value 10 */
     ...
  foo (n, samples, width);
     ...
  }
```

This is because the line

```
foo (n, ...)
```

will be translated to

```
foo (10.0, ...)
```

rather than

```
foo (10, ...)
```

since all CONVERT variables and expressions have double precision. The latter case causes the routine *foo* to receive an unexpected value of the first argument. To overcome this difficulty, one may, instead of declaring *n* as a variable, declare it to be a *define* by giving

```
define n '10'
```

If this is done, the translated line becomes

```
foo((10), ...);
```

which is valid (recall that the surrounding parentheses are added automatically). Alternatively, the user may introduce an extra local variable for data translation as follows.

```
{ #define TOTAL 100
  double samples[TOTAL];
  import width;    /* width is a design parameter */
  import n;        /* n has value 10 */
  int n_int;      /* temporary integer variable */
     ...
  n_int = n;      /* now n_int has integer value 10 */
  foo (n_int, samples, width);
     ...
  }
```

Identifier Matching

The second rule is to match nonlocal identifiers, i.e., the identifiers in the object files must match in order that the loader invoked by SOLVE may correctly resolve the references. Each compiler has its own methods for mapping from source to object. Within one language we may safely ignore this mapping; but when mixing tongues, it becomes important indeed. The Unix C compiler takes any global identifier and prepends an underscore character. Identifiers are not limited in length. Thus

```
int global_var;
char *
somefunc()
{
...
```

generates the identifier '_global_var' and '_somefunc'. The F77 compiler limits identifiers to six characters, then prepends and appends an underscore. For instance, given

```
subroutine sub
...
```

it generates '_sub_'. F77 does not allow underscores in source-level identifiers, e.g., 'subroutine sub_1' is illegal. The Berkeley Pascal compiler strings together the identifiers of all nested procedures to concoct unique global identifiers. Only variables defined in the 'program' part are global, and these identifiers are constructed in the same way as C's globals. The Pascal compiler does not permit source-level identifiers to contain an underscore, e.g., 'procedure proc_a' is illegal.

It should be clear at this point that C programs can call any F77 or Pascal subroutines (procedures) or functions.

Consistent Data Passing Mechanism

The last rule that needs to be kept in mind is that data passing mechanism between programs must be consistent. The F77 compiler uses call by reference; the Pascal compiler uses call by value or call by reference, depending on the declaration of the called routine. The C compiler invariably uses call by value, but the language is powerful enough to simulate other data passing mechanisms using only call by value. One thing that can be done in Pascal but not C is to pass arrays by value. (This can be simulated in C using structures.) For example, suppose the routine *foo* mentioned above is written in Fortran as

```
subroutine foo (n, x, r)
integer n
double precision x(1), r
...
return
end
```

Then to call it in a pseudo-C code, one may have

```
foo_ (&n, samples, &width);
```

Note that an underscored is appended. Also identifiers *n* and *width* are passed by their addresses (the identifier *samples* itself is a pointer to a double array). See *interface* in Section 6.2 for another example.

CHAPTER 4

SOLVE

4.1 Introduction

This chapter describes SOLVE in detail. SOLVE is used after CONVERT, as it takes CONVERT's output as its input. It performs the main task, namely solving the optimization problem defined by the designer.

4.2 Optimization Algorithm

This is the core of SOLVE. Yet, in line with the premise that CONSOLE users need not be optimization experts, its details are mostly hidden.

Probably the main enhancement in CONSOLE Version 1.1 is the replacement of the first order feasible direction algorithm used in Version 1.0 by a recently developed quasi-Newton type feasible direction algorithm [5], extended to handle the various types of specifications available in CONSOLE [6] (see Section 1.2 above). This algorithm enjoys a local superlinear rate of convergence and extensive experimentation on standard test problems has shown that it compares well with the most popular algorithms (not of the feasible direction type). Feasible direction algorithms, which generate sequences of iterates that all satisfy the hard constraints, are especially valuable in the present context. This is because (i) a design may have no practical value unless at least the hard constraints are satisfied, (ii) tradeoff exploration entails that hard constraints be satisfied, (iii) in some cases, it may not even be possible to evaluate the objectives and soft constraints when hard constraints are violated (e.g., unstable dynamical systems). Forcing feasible iterates also leads to simplifications in the algorithm (see [5,6]).

SOLVE is structured in such a way that a new optimization algorithm can be substituted if deemed appropriate. In this section, we briefly discuss the main features that are required (or recommended) from any optimization algorithm to be used in SOLVE. These features are directly related to the underlying design methodology alluded to in the introduction and exposed in detail in [1]. Except where otherwise indicated, they are present in the

algorithm currently used by SOLVE.

We introduced earlier the three possible phases in which the optimization process can happen to be: phase 1 where some hard constraints are violated, phase 2 where all hard constraints are satisfied but some good values are not achieved by objectives or soft constraints, and phase 3 where hard constraint are satisfied and all good values of objectives and soft constraints are achieved. Natural evolution of the optimization process would be, after a possible start in phase 1, to remain in phase 2 until, possibly, phase 3 is reached. This means in particular that, once all hard constraints are satisfied, they should remain satisfied,* and (essentially) similarly for soft constraints. A second and related desirable feature of the optimization algorithm would be to improve the design at each iteration. This is important if the designer is to get a feel for how the optimization progresses, which is essential is an interaction-oriented environment. To this end the current overall objective (maximum of the 'objectives' in the current minimax problem) should decrease at each iteration. It should be noted that the two desirable properties just mentioned generally cannot be achieved in the presence of *functional* objectives or constraints, as only a finite subset of values of the independent parameter are taken into account at each iteration. This however is generally a minor problem, as long as there are no functional hard constraints, which is typical.

While the above are desirable features, it is obviously required that the algorithm be able to handle constrained minimax problems with functional objectives and constraints, and highly recommended that a specifically tailored scheme be used to handle bound type constraints (i.e. lower and upper bounds on design parameters.)

Suitable scaling of the design parameters is essential. Initial scaling should be based on the nominal variation provided by the user (see Section 1.2). Automatic scaling as provided by quasi-Newton methods (such as that used in this Version 1.1) is highly desirable and will significantly affect the behavior of the algorithm at the latest after a number of iterations of the order of the number of design parameters. Still, in almost every real world design problem, manual initial scaling via the nominal variations will be useful when introduced correctly. If no other clue is available, a good (first) estimate for the nominal variation is the absolute initial value of the design parameter (unless it is zero).

---

* As indicated in Section 1.2, hard bounds on individual design parameters must be satisfied throughout.

4.3 Invoking SOLVE

We have indicated in Section 2.2 how to invoke SOLVE. Here we give more details. SOLVE takes an arbitrary number of arguments, but at least one. The first argument must be the name of a Problem Description File that has been successfully processed by CONVERT. The remaining arguments can be anything that is a valid argument for the Unix loader *ld*. They can be classified into four categories :

1. A C, Fortran or Pascal object file (ending in `.o`).
2. A collection of C, Fortran and/or Pascal object files obtained as output of the Unix command *ar*, i.e., a library archive file (ending in `.a`).
3. The output file of the Unix loader *ld*. In this case, the `-r` option of *ld* must be given to preserve the relocation bits such that it may become the input for a further *ld* run (for more detail, see Unix manual entry for *ld*). Such file should not contain any part of the standard C, Fortran and Pascal libraries.
4. A standard library archive file (without the '`.a`' but preceded by '`-l`').

The third type is suggested for the final version since it results in minimum dynamic loading time with SOLVE. The user is encouraged to consult the *ld* entry of the Unix manual for details on the various types of files. In certain cases, the order of the arguments is important.

Before SOLVE is ready to receive commands from the user, it performs four successive operations. First SOLVE invokes the Unix loader *ld* to load the compiled version of the Problem Description File together with the simulators and interface routines (they could be of any of the 4 types mentioned above), if any, into a temporary file created in the directory '/tmp' and reads the resulting file into memory if there is no error during the loading phase. SOLVE then reads in the binary data file associated with the Problem Description File. Following that, SOLVE calls the routine named *simlat* with certain arguments to perform an initialization of the simulator. If this routine is not provided by the user, SOLVE calls a dummy one. In Chapter 5, we will thoroughly discuss the routine *simlat*. Finally, SOLVE calls the routine declared by the command *initialization*, if given, in the Problem Description File (see Chapter 3). The prompt

```
<0>
```

then indicates that the four operations have been completed without error, and SOLVE is

ready to receive commands from the user to perform the optimization task. Otherwise, SOLVE simply reports the error and aborts.

4.4 Interaction

SOLVE is a command-driven package. It currently includes 18 commands. Table 5 describes the functionalities of various commands. The *pcomb* command, central to the design methodology used by CONSOLE, is discussed here. For a detailed description of each command, the user is referred to Appendix B and the on-line manual.

The complex information that needs to be conveyed to an optimization user to assist him/her in making decisions requires graphical feedback showing design performance. The graphical display Pcomb (*performance comb*) allows the user to quickly grasp the performance tradeoffs of a design (see [1] for more details). Currently, only a nongraphical Pcomb is available in CONSOLE. An example is as follows.

```
Pcomb (Iter=  0) (Phase 2) (MAX_COST_SOFT= 4.15)

SPECIFICATION     PRESENT   GOOD                G          B              BAD
O1    cost        1.00e+00  0.00e+00 ================*    |              2.00e+00
O2    stability   9.00e+00  1.00e+00 <================================  0.00e+00
FO1   error       2.00e+00  1.00e+00 ==============*      |              4.00e+00
C1    volume      8.00e+00  1.20e+01            |         |   *=======  9.00e+00
C2    snsitivity  5.00e+00  1.00e+00 ================================>  2.00e+00
FC1   tmperature  6.00e+00  1.00e+01 ------*    |         |              2.00e+01
L1    gain1       3.00e+00  3.00e+00            *======================  2.00e+00
U2    gain2       7.00e+00  4.00e+00 ======================*            7.00e+00
```

This example is constructed artificially to illustrate the essential features of the Pcomb display.

On the Pcomb display, the first row gives the current iteration number '`(Iter= 0)`', phase number '`(Phase 2)`', and the value of the worst (largest) scaled specification '`(MAX_COST_SOFT= 4.15)`'. In the given example, the latter has name `MAX_COST_SOFT` because, in Phase 2, objectives (costs) and soft constraints are all 'objectives' of the current minimax problem (see Section 1.2). Similarly, `MAX_HARD` would be given in Phase 1 and `MAX_COST` in Phase 3. Under the headings, one row is associated with each specification which is identified by both a symbol and its name. For instance, '`O1`' identifies the first objective with the name '`cost`' given in the Problem Description File. The value under '`PRESENT`' is its current (unscaled) value. Its good and bad values are indicated under '`GOOD`' and '`BAD`' respectively. Similarly,

SOLVE

Command	Functionality
algo	display/change optimization algorithm
erase	erase graphic screen
freeze	freeze design parameter(s)
goutput	redirect graphic output
help	display on-line manual
identify	identify the problem being solved
iter	display/change iteration number
pcomb	display performance
plot	plot functional objective/constraint
print	display design parameters
quit	exit SOLVE
reset	reset SOLVE
run	perform optimization
scale	change nominal variation of design parameter
set	change value of design parameter
setgb	change good and bad values or curves of specification
sim	call simlat(x,`INTA`) (see Section 5.2)
store	store design parameters into file
time	display CPU time
trace	trace optimization algorithm and/or simulator (on/off)
unfreeze	unfreeze design parameter(s)

Table 5: Commands

'`FO1`' identifies the first (and only) functional objective '`error`', '`C1`' and '`C2`' identify the constraints '`volume`' and '`snsitivity`' (SOLVE only allows up to 10 characters for the names of design parameters and specifications) and '`FC1`' identifies the functional constraint '`temperature`'. Finally, '`L1`' identifies a (soft) lower bound on the design parameter '`gain1`' and '`U2`' identifies a (soft) upper bound on the design parameter '`gain2`'. The current, good and bad values of a functional specification are given at the value of the free parameter at which

the specification has the maximal *scaled* value.

The scaled value of each specification is graphically displayed in relation with the column marked 'GOOD' and the column marked 'BAD'. The position of the tip of each line represents the scaled value of the specification. If the scaled value is between -1 and 2, the corresponding tip position is shown in the display window and marked by a '*' (such as O1, FO1, C1, FC1, L1 and U2). A tip position lying under the heading 'G' (on the vertical dashed 'good' line) represents a scaled value close to 0 (such as L1), and a tip position under the heading 'B' (on the vertical dashed 'bad' line) indicates a scaled value close to 1 (such as U2). If the scaled value is smaller than -1, it is displayed by an arrow towards the left (such is the case for O2). If the scaled value is larger than 2, it is then displayed by an arrow towards the right (such is the case for C2). Lines drawn from the left (O1, FO1, C2, FC1, U2) corresponds to specifications which are to be minimized or upper bound constraints and lines drawn from the right (O2, C1, L1) corresponds to specifications which are to be maximized or lower bound constraints. Thus a 'short' line always corresponds to a 'small' numerical value of the corresponding quantity. Finally, lines drawn with '=' represent either objectives or soft constraints, and lines drawn with '-' represent hard constraints.

Pcomb may be output automatically during each optimization iteration or manually, say, after adjusting the good or bad values for a particular objective or constraint. As discussed in Section 4.2, within a given phase, the maximum scaled value of the current 'objectives' (may include soft constraints and even hard constraints, depending on the phase) is decreased at each iteration. Correspondingly, the rightmost tip among these 'objectives' will move to the left at each iteration. Tradeoffs between competing objectives or constraints can be explored by adjusting good and bad values using the *setgb* command after best or near best performance of the system being designed has been achieved following several iterations of optimization.

4.5 Error Checking

Although CONVERT attempts to check for all possible inconsistencies in the Problem Description File, some inconsistencies may still find their way through to SOLVE. These errors will very often be related to the formulation of functional objectives or constraints. CONVERT cannot check the consistency of the good and bad curves of functional specifications. For a functional objective that has to be minimized (maximized), or for a functional

constraints that is '<=' ('>='), the good curve has to be below (above) the bad curve over the entire range of the free parameter. When CONVERT scans the Problem Description File and processes it to its output (the two binary files indicated in Figure 1 (Section 1.1)), it does not have an executable version of the good and bad curves of the functional specifications available. SOLVE, however, has an executable version of the curves available and will check them and report any inconsistency.

4.6 Interrupt

When the user types, say

```
run 5
```

in SOLVE, the optimization algorithm will try to perform five more iterations and give the control to the user. It may stop earlier if a local optimal solution has been obtained. However, it could be also forced to stop prematurely. This is done by issuing an interrupt signal, i.e., by depressing 'control-C', 'break' or 'delete' (depending on the current terminal setup, see the Unix manual entry *stty* for details). When SOLVE detects an interrupt signal while running the optimization algorithm, it prints out the message

```
An interrupt has been detected by SOLVE. SOLVE will
stop  running the optimization algorithm as soon as
it finishes the current iteration.


CONTINUING EXECUTION ...
```

and calls '`simlat(x,SIM_INTR)`' (see Chapter 5). As indicated in the message above, SOLVE will stop and give control to the user as soon as the current iteration is completed (soft interrupt). If the user issues a second interrupt signal before this happens, a 'hard' interrupt is generated: SOLVE stops immediately and gives control to the user. The user is cautioned against using hard interrupts as the optimization process will likely be put into an inconsistent state.

CHAPTER 5

SOLVE AND SIMULATORS

5.1 Introduction

While solving the optimization problem, SOLVE repeatedly makes use of the values of the various objectives and constraints (and possibly of their gradients) for the current design parameter vector. Computation of these values is performed by a simulator. In the simplest cases, the simulator is part of the Problem Description File. For instance, in the tutorial example of Chapter 2, the statement

```
return (x-1)*(x-1)+(y-2)*(y-2);
```

in the objective

```
objective "quadratic"
   minimize {
      import x, y;
      return (x-1)*(x-1)+(y-2)*(y-2);
      }
   good_value=1
   bad_value=4
```

can be thought of as a simulator. For practical design problems, however, the definition of various specifications requires a fairly large amount of code and hence the simulator usually exists as a stand alone program. The simplest such case is when the simulator is a Fortran (or C, or Pascal) subroutine with design parameters as input arguments and quantities involved in the specifications as output arguments. For example, we could have the following C routine to perform the evaluation of the objective in the tutorial example.

```
double cost(x,y)
double x, y;
{
   return (x-1)*(x-1)+(y-2)*(y-2);
}
```

The objective would then be expressed as

```
objective "quadratic"
   minimize {
      import x, y;
      double cost();
      return cost(x,y);
      }
   good_value=1
   bad_value=4
```

When invoking SOLVE, one would then type

```
solve qp cost.o
```

where the file *cost.o* is the object code of the function *cost* above. In many instances, however, communication between SOLVE and a simulator is more complex. If an off-the-shelf simulator is used, the details of this communication will be handled by an interface. We now discuss these.

5.2 Components of an Interface

Typically, most of the CPU time is spent in simulations. It is thus important to avoid multiple simulations for the same values of the design parameters. To this effect, calls for simulation in the Problem Description File could be made through an interface routine, say *output*, that would store values of design parameters and corresponding values of simulator outputs. Pushing this idea further one could have the interface routine save intermediate results (LU factorization, Schur decomposition, ...) that could be of help for later calls to the simulator.

In many cases, the interface between CONSOLE and the simulator must be yet more sophisticated. This is so, among other instances, when the simulator itself is interactive, or when the simulator needs to be initialized or reset at appropriate times. Also, it is often convenient that values of design parameters be passed *automatically* to the simulator whenever they are modified. Most of these tasks are performed by a routine called *simlat*. As SOLVE is unaware of what simulator is currently being used, it calls *simlat* whenever there is a possibility that the simulator require that some action be taken. If no *simlat* routine is provided with the simulator, the default dummy routine will be called and no action will be taken. *simlat* has arguments as follows:

```
simlat(x,flag)
double x[];
int flag;
```

where x is a design parameter vector and where *flag* indicates what type of actions may have to be taken. At this writing, 9 different flags are used by SOLVE. In the interfaces we have already implemented, these flags are represented by defines as follows:

```
#define   SIM_INIT    101
#define   SIM_INTA    102
#define   SIM_INTR    103
#define   SIM_ITER    104
#define   SIM_NOTR    105
#define   SIM_PUPD    106
#define   SIM_QUIT    107
#define   SIM_RSET    108
#define   SIM_TRAC    109
```

The meaning and occurrences of the various flags are as follows.

SIM_INIT: sent by SOLVE right after the data file created by CONVERT has been read. x is the initial design parameter vector. This should be used to perform one-time (problem independent) initialization of the simulator if necessary. For problem dependent initialization, see Section 3.4.

SIM_INTR: sent by SOLVE when an interrupt signal has been issued by the user. x is the current design parameter vector. If appropriate, *simlat* should notify the simulator that an interrupt has occurred, so that appropriate action can be taken.

SIM_ITER: sent by SOLVE whenever an iteration has been completed. x is the new design parameter vector.

SIM_ITRA: sent by SOLVE when the *sim* command is issued. x is the current design parameter vector. If the simulator can be used interactively, *simlat* should then initiate an interactive session.

SIM_NOTR: sent by SOLVE when the command *trace* is issued with the argument 0. x is the current design parameter vector. If the simulator has the capability to trace its computation, *simlat* should disable it.

SIM_PUPD: sent by SOLVE whenever the value of a design parameter is modified. x is the new design parameter vector. If appropriate, *simlat* should then modify accordingly the design parameter vector that may have been stored by the simulator.

SIM_QUIT: sent by SOLVE when the *quit* command is issued. x is the current design parameter vector. If appropriate, *simlat* should then request a simulator cleanup, e.g., that

temporary files created by the simulator be deleted or that the simulator environment be saved to a file.

SIM_RSET: sent by SOLVE when the *reset* command is issued. x is the current design parameter vector. If appropriate, *simlat* should reset the interface and/or the simulator.

SIM_TRAC: sent by SOLVE when the command *trace* is issued with the argument 2 or 3. x is the current design parameter vector. If the simulator has the capability to trace its computation, *simlat* should request it.

Information concerning the design parameters (their names, nominal variations, etc.), the name of the Problem Description File and the finite difference approximation parameter *pdelta* (see below) can also be accessed by the simulator (or *simlat*). This is done by including the following piece of C code.

```
extern struct _despar {
      char *name;
      double variation;        /* variation */
      int  min_soft_or_hard;   /* =0 no lower bound, =1 if soft lower bound, */
                               /* =2 if hard lower bound */
      int  max_soft_or_hard;   /* similar to above, for upper bound */
      double min_good;         /* if min_soft_or_hard = */
                               /*   0:  min_good = -pow(10.0, 20.0) */
                               /*   1:  min_good = good value of lower bound */
                               /*   2:  min_good = lower bound */
      double min_bad;          /* if max_soft_or_hard = */
                               /*   0:  not used */
                               /*   1:  min_bad = bad value of lower bound */
                               /*   2:  not used */
      double max_good;         /* similar to min_good above, for upper bound */
      double max_bad;          /* similar to min_bad above, for upper bound */
      };

extern int numdes;                  /* number of design parameters */
extern struct _despar **despars;    /* pointer to design parameters */
extern char *pdf;                   /* Problem Description File name */
extern double pdelta;               /* delta used in computing gradients */
                                    /* by finite difference */
```

Parameter *pdelta* is used by SOLVE for computing gradients by finite differences. Specifically, SOLVE makes use of the approximations

$$(\nabla f(x))_j \cong \frac{f(x + \delta_j * variation_j * e_j) - f(x)}{\delta_j}$$

where f is a scaled specification, x is the raw (unscaled) design parameter vector, $variation_j$ the nominal variation corresponding to the j-th component of x, and e_j the j-th standard

basis vector, and δ_j is obtained from *pdelta* as follows. When the default algorithm (FSQP) is used, δ_j is given by

$$\delta_j = \max\{\delta_m,\ \delta_{p_j},\ \delta_u\}$$

with

$$\delta_m = \sqrt{machine\ precision},$$

$$\delta_{p_j} = \delta_m \max\{1,\ \frac{|x_j|}{variation_j}\},$$

and δ_u is *pdelta*. When the first order algorithm is used (see description of *algo* command), all δ_j's are equal to *pdelta*. The default value of *pdelta* is the square root of the machine precision when FSQP is used, and it is 10^{-5} when the first order algorithm is used. (The latter should be suitable if computations are performed in double precision; if computations are performed in single precision, a somewhat larger value, say 10^{-3}, may be more appropriate.)

For an example of a relatively sophisticated interface, the reader is referred to [7].

DESIGN EXAMPLES

6.1 Introduction

In this chapter we provide two design examples: control of a copolymerization reactor and control of a flexible mechanical arm. * For each example, we divide the discussion into three parts. First we give some details on the system for which a design is to be performed. Then the problem description is given. Finally, running the design problem with SOLVE is discussed. For detailed discussion of the design examples, the reader is referred to [8] and [9], respectively.

6.2 Control of a Copolymerization Reactor

The System

Polymerization is a group of chemical chain reactions that links a large number (more than a thousand) of 'monomer' molecules together. The product, called polymer, can consist of long strings, entangled in each other, but can also be branched or even have a three dimensional structure. Copolymerization is polymerization using two different monomers at the same time. The group of materials popularly identified as *plastics* mainly consists of polymers and copolymers with a carbon skeleton. Plastics show a vast variety of physical properties (strength, color, elasticity, etc.). The giant polymer molecule structure and the location of the chemically active groups on this molecule determine the properties of the product. However, the relation between these properties and the molecular structure is not fully understood yet, and therefore the former cannot be predicted.

Polymers and copolymers are produced in different types of reactors. Some are produced in continuous processes, others are produced in batch processes. We consider the copolymerization of styrene and acrylonitrile. This is done in a well-mixed *semi-batch* reactor. This means that the reactor, in this case a vessel with a stirring mechanism and a

* These designs were actually carried out as part of two Systems Research Center research projects directed by Drs. Choi and Krishnaprasad respectively.

cooling water jacket, is initially filled with a certain volume of feed. During the production, phase feed is added, but no production stream leaves the reactor. The product is removed at the end of the batch time. The composition of the feed to the semi-batch copolymerization reactor is fixed. It consists of both monomers, a solvent and the initiator. The initiator is a substance that initiates the polymerization reaction.

The producer of polymer or copolymer wants to produce as much product as possible during a fixed amount of time. It is important however for the product to have the correct properties. Products that are even only slightly *off-spec* will be refused by the clients and are absolutely worthless. Therefore designing a control strategy to obtain the product with the appropriate properties is necessary. These properties are not measurable on-line. The requirements will be adjusted every few batch runs as lab results come in, so that changes in equipment and feedstock can be slowly adjusted for. The objective of obtaining a certain molecular copolymer structure is simplified to obtaining a certain molecular weight and a given ratio of both monomers in the product. The molecular weight is a measure of the average number of monomer molecules chained together in each copolymer molecule. The ratio of both monomers in the product, is often referred to as the copolymer composition. All control problems are definitely not solved yet. Neither the molecular weight, nor the copolymer composition is measurable on-line. Therefore the control is selected using a model, and afterwards the model parameters are corrected based upon the prediction error. The selection of this control is done with CONSOLE. The manipulated variables are the feed flowrate and the reactor temperature which are all functions of time.

The simulator, *copoly*, is a Fortran program that existed before CONSOLE was developed. It simulates an existing copolymerization reactor which is part of the experimental equipment of the Department of Chemical and Nuclear Engineering of the University of Maryland at College Park.

Problem Description

We give here (piece by piece) the entire contents of the Problem Description File. First, one must specify the time span of interest and the distance between two consecutive time points at which the simulation outputs should be recorded, e.g.,

```
final_time = 5.0      /* in hours */
mesh_size = 0.25      /* in hours */
```

Then, the design parameters must be declared. Since CONSOLE cannot solve problems with infinite dimensional parameter space, an approximate problem is solved instead. It is thus assumed that the temperature and feed flowrate profiles are polynomials in time. This will result in a suboptimal solution. However, if the order of the polynomials is sufficient, the performance obtained will be very close to the true optimal. Thus, temperature and feed flowrate are expressed as

$$\text{temperature} = a_1 + a_2 t + a_3 t^2 + a_4 t^3$$

and

$$\text{feed flowrate} = b_1 + b_2 t + b_3 t^2 + b_4 t^3.$$

The design parameters are then a_i's and b_i's for $i = 1, 2, 3, 4$.

```
design_parameter a1  init=333   variation=5.0   min=323.0  max=368.0
design_parameter a2  init=10    variation=1.0   min=-10.0  max=10.0
design_parameter a3  init=0.1   variation=0.01  min=-1.0   max=1.0
design_parameter a4  init=0.1   variation=0.01  min=-0.1   max=0.1
design_parameter b1  init=20    variation=1.0   min=5.0    max=50.0
design_parameter b2  init=-1    variation=0.1   min=-2.0   max=2.0
design_parameter b3  init=-0.1  variation=0.01  min=-0.2   max=0.2
design_parameter b4  init=-0.01 variation=0.001 min=-0.05  max=0.05
```

The initial value, variation, minimum and maximum values of each design parameter are obtained either from a previous design or from physical considerations (for temperature expressed in degree K and feed flowrate expressed in liters/minute).

Every call to the simulator *copoly* will be made through an interface routine 'interface'. This routine has many input arguments (among which the design parameters) and many output arguments (simulation outputs). As the calling sequence is practically identical for many specifications, the following define will be used.

```
define call_interface                                                    "\
   import a1,a2,a3,a4,b1,b2,b3,b4;                                        \
   import final_time, mesh_size;                                          \
                                                                          \
   double molecular_weight_t,  /* molecular weight at time 'time' */      \
          composition_t,       /* composition at time 'time' */           \
          final_volume;        /* final volume */                         \
                                                                          \
   interface (a1,a2,a3,a4,b1,b2,b3,b4,     /* input arguments */          \
              final_time,time,mesh_size,                                  \
              &molecular_weight_t,         /* output arguments */         \
              &composition_t,                                             \
              &final_volume)"
```

There will be two design objectives, both functional: the maximum (over time) square deviation from the desired molecular weight and the maximum (over time) square deviation from the desired composition should both be made as small as possible. The former is required only after transients have died out. The formulation is as follows.

```
desired_molecular_weight = 3e4
bad_Mn_sqerr = 5e3**2
functional_objective "(Mn-Mns)^2"   /* square of error in molecular weight */
   for time from 3 to final_time by mesh_size
   minimize {
      double diff;
      import desired_molecular_weight;
      import call_interface;   /* so that the define can be used here */

      call_interface;
      diff = molecular_weight_t - desired_molecular_weight;
      return (diff*diff);
      }
   good_curve = {return 0.0;}
   bad_curve  = {import bad_Mn_sqerr;
                 return bad_Mn_sqerr;}
```

```
desired_composition = 1  /* proportion of monomers 1 and 2 */
bad_CC_sqerr = 0.225**2
functional_objective "(CC-CCs)^2" /* square of error in copolymer composition */
   for time from 0 to final_time by mesh_size
   minimize {
      double diff;
      import desired_composition;
      import call_interface;

      call_interface;
      diff = composition_t - desired_composition;
      return (diff*diff);
      }
   good_curve = {return 0.0;}
   bad_curve  = {import bad_CC_sqerr;
                 return bad_CC_sqerr;}
```

The volume of product at the end of the batch is limited by the size of the tank. This is expressed by the following design constraint.

```
constraint "final volume" hard
   {  import call_interface;
      double time;          /* here time is dummy; but it must be declared */
                            /* because it appears in the call_interface define*/

      call_interface;
      return final_volume;
      }
   <= good_value = 4.0  /* liters */
      bad_value  = 4.1  /* liters */
```

Constraints on operating variables are necessary for safety and reasonable operation of the reactor. Based on the requirements for reaction rate, heat transfer limitations and reactor safety, upper and lower bounds on reactor temperature are defined as constraints. Similarly, in order to avoid negative flowrate and to limit the maximum flowrate which can be handled by the reactor system, upper and lower bounds on feed flowrate are defined as constraints. Since the constraints are time dependent, they are given as functional constraints as follows.

```
functional_constraint "upper temperature" hard  /* upper bound on temperature */
   for time from 0 to final_time by mesh_size
   {  import a1,a2,a3,a4;
      return a1+a2*time+a3*pow(time,2.0)+a4*pow(time,3.0);
      }
   <= good_curve = {return 363.0;}  /* degrees K */
      bad_curve  = {return 364.0;}  /* degrees K */
```

```
functional_constraint "lower temperature" hard  /* lower bound on temperature */
   for time from 0 to final_time by mesh_size
   {  import a1,a2,a3,a4;
      return a1+a2*time+a3*pow(time,2.0)+a4*pow(time,3.0);
      }
   >= good_curve = {return 328.0;}  /* degrees K */
      bad_curve  = {return 323.0;}  /* degrees K */
```

```
functional_constraint "upper flowrate" hard   /* upper bound on feed flowrate */
   for time from 0 to final_time by mesh_size
   {  import b1,b2,b3,b4;
      return b1+b2*time+b3*pow(time,2.0)+b4*pow(time,3.0);
      }
   <= good_curve = {return 0.07;}   /* liters per minute */
      bad_curve  = {return 0.075;}  /* liters per minute */
```

```
functional_constraint "lower flowrate" hard   /* lower bound on feed flowrate */
   for time from 3 to final_time by mesh_size
   {  import b1,b2,b3,b4;
      return b1+b2*time+b3*pow(time,2.0)+b4*pow(time,3.0);
      }
   >= good_curve = {return 0.0;}    /* liters per minute */
      bad_curve  = {return -0.005;} /* liters per minute */
```

This completes the Problem Description File.

We now suggest a possible interface routine for this example. It should be clear that interface routines should not be problem dependent, but merely simulator dependent. Thus, in the present case, the same interface could be used with virtually any Problem Description File, provided the same simulator is used.

For our purpose, the simulator *copoly* can be viewed as 'unsophisticated'. Thus the interface used here acts as a mere buffer, avoiding redundant calls to *copoly* (see Chapter 5 for more on interfaces). The entire listing of the interface (the file *interface.c*, consisting of C-code) is given now, with a discussion of the various sections.

The first part contains the declarations. The static type is used to ensure that the value of the memorized variables will remain unchanged between calls.

```
/* interface between SOLVE and Copolymerization simulator (copoly) */

#define UNDEFINED -10e20
#define MNMP       100      /* maximal number of mesh points */
                            /* thus ftime can be as large as 100*meshsz */

static double sa1=UNDEFINED,    sa2=UNDEFINED,   /* saved coefficients ai's */
              sa3=UNDEFINED,    sa4=UNDEFINED,
              sb1=UNDEFINED,    sb2=UNDEFINED,   /* saved coefficients bi's */
              sb3=UNDEFINED,    sb4=UNDEFINED,
              sftime=UNDEFINED,                  /* saved final time */
              smeshsz=UNDEFINED,                 /* saved mesh size */

              smn[MNMP],                         /* saved molecular weight */
              scomp[MNMP],                       /* saved composition */
              sfvol;                             /* saved final volume */

interface (a1,a2,a3,a4,b1,b2,b3,b4,ftime,time,meshsz,mnt,compt,fvol)
double a1,a2,a3,a4,b1,b2,b3,b4,ftime,time,mesh,*mnt,*compt,*fvol;
{
   int k;        /* counter for mesh points */
```

First it is checked whether or not the design parameters changed between the current call and the previous call. If they did not, then no new simulation is necessary. The values corresponding to the meshpoint closest to the requested time are returned.

```
   if (a1 == sa1 && a2 == sa2 && a3 == sa3 && a4 == sa4 &&
       b1 == sb1 && b2 == sb2 && b3 == sb3 && b4 == sb4 &&
       ftime == sftime &&          /* if inputs are the same as at the */
       meshsz == smeshsz) {        /* previous call, simply return the */
                                   /* stored values */
      k = (int) time/meshsz;  /* meshpoint closest to 'time' */
      *mnt = smn[k];
      *compt = scomp[k];
      *fvol = sfvol;
      }
```

However, if at least one of the design variables changed, then a new simulation run is needed.

```
   else {

      /* store all value of inputs */
      sa1 = a1; sa2 = a2; sa3 = a3; sa4 = a4;
      sb1 = b1; sb2 = b2; sb3 = b3; sb4 = b4;
      sftime = ftime; smeshsz = meshsz;

      /* and call the simulator (routine copoly) */
      copoly_(&a1,&a2,&a3,&a4,&b1,&b2,&b3,&b4,&ftime,&meshsz,smn,scomp,&sfvol);
```

After the simulation, all the time profiles are stored, and the values corresponding to the meshpoint closest to the requested time are returned.

```
        /* then return the stored values */
        k = (int) time/mesh;
        *mnt = smn[k];
        *compt = scomp[k];
        *fvol = sfvol;
        }
    }
```

Running the Problem

Let 'copoly-design' be the name of the Problem Description File listed above. One then first 'compiles' this file by typing

```
convert copoly-design
```

which prompts the output

```
          Welcome to CONVERT, Version 1.1 (Released 6/15/90)

                    Copyright (c) 1988, 1990
                                by
            Michael K.H. Fan    Andre L. Tits    Jian Zhou
                    Li-Sheng Wang    Jan Koninckx
                         All Rights Reserved.

[processing copoly-design]
[compiling copoly-design.c]
[writing copoly-design.d]
```

Then SOLVE is invoked

```
solve copoly-design copoly.o interface.o
```

yielding

```
            Welcome to SOLVE, Version 1.1 (Released 6/15/90)

                        Copyright (c) 1988, 1990
                                by
              Michael K.H. Fan    Andre L. Tits    Jian Zhou
                      Li-Sheng Wang    Jan Koninckx
                           All Rights Reserved.

[loading/reading copoly-design.o and ...]
[reading copoly-design.d]
[calling simulator initialization (if any)]
[calling problem initialization (if any)]

type "help" for help
type "help info" for information
type "help news" for local news

<0>
```

Issuing the command

```
run 0 print pcomb
```

displays the initial values and the corresponding Pcomb

```
Name        Value         Variation  wrt 0   Prev   Iter=0
a1          3.33000e+02   5.0e+00
a2          1.00000e+01   1.0e+00
a3          1.00000e-01   1.0e-02
a4          1.00000e-01   1.0e-02
b1          2.00000e+01   1.0e+00
b2         -1.00000e+00   1.0e-01
b3         -1.00000e-01   1.0e-02
b4         -1.00000e-02   1.0e-03
```

```
Pcomb (Iter=  0) (Phase 1) (MAX_HARD= 35)

SPECIFICATION    PRESENT   GOOD                G          B             BAD
FO1  (MN-MNs)^2  1.57e+08  0.00e+00 ==================================>  2.50e+07
FO2  (CC-CCs)^2  4.75e-02  0.00e+00 ====================*|               5.06e-02
C1   final vol   5.91e+00  4.00e+00 ---------------------------------->  4.10e+00
FC1  upper temp  3.98e+02  3.63e+02 ---------------------------------->  3.64e+02
FC2  lower temp  3.33e+02  3.28e+02 *----------------------------------  3.23e+02
FC3  upper flow  2.00e-02  7.00e-02 <--        |          |             7.50e-02
FC4  lower flow  1.12e-02  0.00e+00 <---------------------------------- -5.00e-03
```

The Pcomb shows that, for the given initial values of the design parameters, the molecular weight (FO1), copolymer composition (FO2), final volume of the product (C1) and temperature (FC1) are all unsatisfactory. Plots of the functional specifications can be also seen by the command *plot* (see Section 4.4 and Appendix B). Typing

```
run 6
```

causes SOLVE to run 6 iterations of the optimization algorithm (this may take some time ...), until finally the prompt

```
<6>
```

appears on the screen. The command

```
pcomb
```

yields

```
Pcomb (Iter=  6) (Phase 2) (MAX_COST_SOFT= 0.0766327)

SPECIFICATION    PRESENT   GOOD              G          B            BAD
FO1  (MN-MNs)^2  1.92e+06  0.00e+00 ==========*          |           2.50e+07
FO2  (CC-CCs)^2  3.88e-03  0.00e+00 ==========*          |           5.06e-02
C1   final vol   3.47e+00  4.00e+00 <--        |        |           4.10e+00
FC1  upper temp  3.53e+02  3.63e+02 <--        |        |           3.64e+02
FC2  lower temp  3.45e+02  3.28e+02 <---------------------------------  3.23e+02
FC3  upper flow  9.70e-03  7.00e-02 <--        |        |           7.50e-02
FC4  lower flow  6.00e-03  0.00e+00 <-------------------------------- -5.00e-03
```

All hard constraints are now satisfied. Objectives `FO1` and `FO2` have almost reached their good values and seem to be competing against each other. If, in the designer's opinion, the current molecular weight is acceptable while a composition closer to the desired is highly desirable, a suitable action would now be to type

```
setgb FO1 = 2e6, 2.5e7
```

After a few more iterations, e.g,

```
run 5 pcomb
```

the designer may be satisfied. He would then type

```
print
```

to see the 'optimal' design parameters

```
Name        Value         Variation   wrt 0     Prev   Iter=27
a1          3.53188e+02   5.0e+00       6%       0%
a2         -3.19240e+00   1.0e+00     -68%       0%
a3          9.13515e-02   1.0e-02       9%       0%
a4          5.31340e-02   1.0e-02      47%       0%
b1          9.69793e+00   1.0e+00      52%       0%
b2         -3.74885e-01   1.0e-01      63%       0%
b3         -4.07314e-02   1.0e-02      59%       0%
b4         -6.40500e-03   1.0e-03      36%       0%
```

then

```
store "copoly_optimal_design_parameters"
```

and finally

```
quit
```

6.3 Control of a Flexible Arm

The System

Here we want to design a controller for a one-link flexible robot arm (see [9]). The system includes a one-meter flexible beam, a DC brush motor, three sensors (position encoder, tachometer, tip accelerometer), A/D and D/A converters, and an IBM PC/AT. We use a linear time-invariant model of state-dimension 6, completely observable from the three sensors [10]. A Luenberger observer sensor is implemented in the IBM PC/AT and, based on experimental observations, it is assumed that the estimated states is close enough to the exact states that their differences can be neglected. The input to the system is the armature voltage of the DC motor and it is intended to control the system using full state constant linear feedback, to be implemented in the PC/AT. Thus we have

$$\dot{x}(t) = Ax(t) + bu(t)$$

$$y(t) = Cx(t)$$

$$u(t) = kx(t) + v(t)$$

where v is the external input. Here we consider as output y all the quantities susceptible of being involved in design specifications, namely the full state and the tip acceleration, so that C is a 7×6 matrix. * The first state variable represents the tip position, and the last

* Clearly the last output is redundant; it is included for sake of simplifying the formulation of the design problem.

two, respectively the hub velocity and position.

We will use the linear simulator *MaryLin*. A corresponding System Description File has to be set up. It consists of the specification of the matrices $A + bk$ (renamed A), b, and c, and of the chosen input v (here a unit step). It is as follows

```
PI = 3.1415926
G = 0.125
R1 = 9*2*PI
R2 = 20.6*2*PI
R3 = 9.5*2*PI
R4 = G/(0.383 * 2.124)
A1 = 0.04*R2+0.22*R1
A2 = R1*R1+0.0088*R2*R1+R2*R2
A3 = 0.22*R2*R2*R1+0.04*R2*R1*R1
A4 = R2*R2*R1*R1
B2 = R1*R2*R1*R2/(R3*R3)
B3 = 2*B2*R3
B4 = B2*R3*R3
B6 = 2*PI*1.16*2.124
C1 = (A1*A1-A2)*0.383*R4
C2 = -A1*0.383*R4
C3 = 0.383*R4
C5 = B2*0.383*R4
```

```
system_size Ninputs=1 Nstates=6 Noutputs=8
```

```
readmatrix A
-A1          1          0          0          0          0
-A2          0          1          0          B2         0
-A3          0          0          1          B3         0
-A4          0          0          0          B4         0
0            0          0          0          0          1
-B6*k1     -B6*k2     -B6*k3     -B6*k4     -B6*k5     -7.29-B6*k6
```

```
readmatrix  B
0
0
0
0
0
B6
```

```
readmatrix C
1          0          0          0          0          0
0          1          0          0          0          0
0          0          1          0          0          0
0          0          0          1          0          0
0          0          0          0          1          0
0          0          0          0          0          1
C1         C2         C3         0          C5         0
```

```
readmatrix Ut
1
```

Note that values of the feedback gains $k1, \cdots, k6$ are not specified (and thus are initially 0), as it is intended to use these gains as design parameters. Other values (than 0) could have been given here but they would be overridden (with the initial value specified in the Problem Description File) at the start of the optimization.

Problem Description

Again, we list here the contents of the Problem Description File. As suggested above, we select as design parameters the feedback gains $k1$ through $k6$.

```
design_parameter k1 init=8.234939958e+00  variation=1.43e-1
design_parameter k2 init=-1.58345004e-01  variation=2e-1
design_parameter k3 init=-1.579990251e-03 variation=1.4e-3
design_parameter k4 init=2.38706e-5       variation=1e-5
design_parameter k5 init=4.089939087e1    variation=1e1
design_parameter k6 init=5.853e-1         variation=1e-1
```

In the specifications below, all system variables are divided by the final value of the tip position (approximated by its value at $t = 20$), to take into account the effect of a magnification of the step input to achieve a final value of 1 for the tip position. The design objective is that, given a step input, the response at the tip position be close to a step function. Good and bad curves are given on Figure 2.

This is declared as follows as two functional objectives.

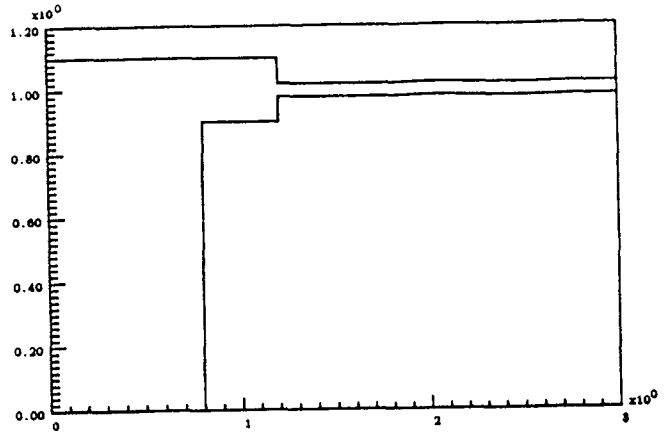


Figure: 2 Desired Position Profile

```
functional_objective "max-positive"
   for t from 0 to 3 by .03
   minimize {
      double Ytr();
      return Ytr("1",t)/Ytr("1",20.0);
      }
   good_curve = {
       if( t <= 1.2 ) return 1.1;
       else           return 1.02;
       }
   bad_curve = {
       if( t <= 1.2 ) return 1.2;
       else           return 1.1;
       }
```

```
functional_objective "min-positive"
   for t from 0.8 to 3 by .03
   maximize {
      double Ytr();
      return Ytr("1",t)/Ytr("1",20.0);
      }
   good_curve = {
       if( t <= 1.2 ) return 0.9;
       else           return 0.98;
       }
   bad_curve = {
       if( t <= 1.2 ) return 0.85;
       else           return 0.95;
       }
```

`Ytr` is an interface routine. It recognizes the string "1" as indicating that the value of the first output (as defined in the system description file) at the given time t must be returned (see Chapter 5 for more on interface routines). The design parameters are not passed explicitly as arguments to `Ytr`. Instead *simlat* automatically sends the values of the design

parameters to *MaryLin* whenever they are modified (see Chapter 5: flag PUPD). Next, one could also include constraints on the tip acceleration, to ensure a smooth motion.

```
functional_constraint "max-accel" hard
   for t from 0 to 3 by .03
   {
      double Ytr();
      return Ytr("7",t)/Ytr("1",20.0);
   }
   <=
   good_curve = { return 10; }
   bad_curve = { return 11; }
```

```
functional_constraint "min-accel" hard
   for t from 0 to 3 by .03
   {
      double Ytr();
      return Ytr("7",t)/Ytr("1",20.0);
   }
   >=
   good_curve = { return -10; }
   bad_curve = { return -11; }
```

Finally, the armature voltage may not exceed the specification of the motor.

```
functional_constraint "control" hard
   for t from 0 to 3 by .03
   {
      import k1 k2 k3 k4 k5 k6
      double Ytr();
      return fabs( 1-(k1*Ytr("1",t)+k2*Ytr("2",t)+k3*Ytr("3",t)+k4*Ytr("4",t)
             +k5*Ytr("5",t)+k6*Ytr("6",t) ))/Ytr("1",20.0);
   } <=
   good_curve = {
      return 4;
       }
   bad_curve = {
      return 5;
      }
```

Running the Problem

The initial guess of the design parameters is chosen as described in [10]. After SOLVE has run for several iterations, the final values of the design parameters are

```
Name        Value         Variation  wrt 0   Prev   Iter=12
k1           8.27618e+00  1.4e-01      0%     1%
k2          -2.14794e-01  2.0e-01     35%    -2%
k3          -1.02094e-03  1.4e-03    -35%     0%
k4           5.67104e-06  1.0e-05    -76%     0%
k5           1.04953e+01  1.0e+01    -74%    -2%
k6           6.03690e-01  1.0e-01      3%     1%
```

with the Pcomb output

```
Pcomb (Iter=  4) (Phase 3) (MAX_COST= -0.000817857)

SPECIFICATION    PRESENT   GOOD               G          B             BAD
FO1  max-positi  1.00e+00  1.02e+00 ========* |          |             1.10e+00
FO2  min-positi  9.02e-01  9.00e-01          *========================  8.50e-01
FC1  max-accel   2.96e+00  1.00e+01 <--       |          |             1.10e+01
FC2  min-accel  -4.75e-01 -1.00e+01 <---------------------------------- -1.10e+01
FC3  control     1.07e+00  4.00e+00 <--       |          |             5.00e+00
```

showing that the design is satisfactory. Typing

```
plot FO1
plot FC3
```

one obtains Figures 3 and 4.

Finally, it should be clear that specifications involving frequency responses could have been included, alone or in conjunction with time-domain specifications.

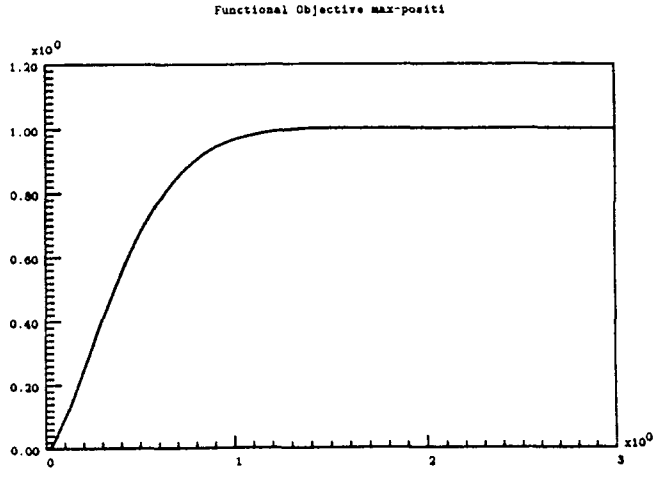


Figure 3: Plot of Tip Position

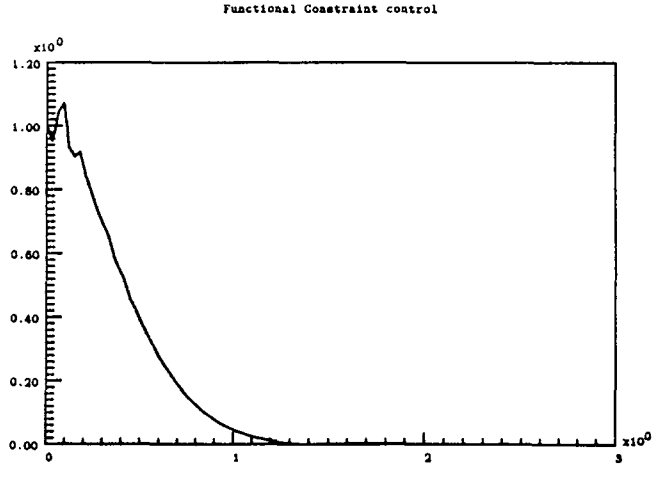


Figure 4: Plot of Control Signal

REFERENCES

[1] W.T. Nye & A.L. Tits, "An Application-Oriented, Optimization-Based Methodology for Interactive Design of Engineering Systems," *Internat. J. Control* 43 (1986), 1693–1721.

[2] P.E. Gill, W. Murray, M.A. Saunders & M.H. Wright, "User's Guide for SOL/QPSOL: A FORTRAN Package for Quadratic Programming," Stanford Univ., Technical Report SOL 83-7, 1983.

[3] A.L. Tits & Z. Ma, "Interaction, Specification Refinement, and Tradeoff Exploration in Optimization-Based Design of Engineering Systems," in *Proceedings of the 1985 IFAC Workshop on Control Applications of Nonlinear Programming and Optimization*, G. DiPillo, ed., Pergamon Press, 1986, 189–194.

[4] W.T. Nye, *DELIGHT: An Interactive System for Optimization-Based Engineering Design*, Ph.D. Thesis, Department EECS, University of California, Berkeley, California, 1983.

[5] E.R. Panier & A.L. Tits, "On Feasibility, Descent and Superlinear Convergence in Inequality Constrained Optimization," Systems Research Center, University of Maryland, Technical Report SRC-TR-89-27, College Park, MD 20742, 1989.

[6] A.L. Tits & J. Zhou, "Fast Feasible Direction Methods, With Engineering Applications," in *Proceedings of the 1991 European Control Conference*, Hermès, 1991.

[7] M.K.H. Fan, "Optimization-Based Design of Nonlinear Systems Using CONSOLE and SIMNON," Systems Research Center, University of Maryland, TR 87-204, College park, Maryland 20742, 1987.

[8] D. Butala, K.-Y. Choi & M.K.H. Fan, "Multiobjective Dynamic Optimization of Semibatch Free Radical Copolymerization Process with Interactive CAD Tools," *Computers in Chemical Engineering* 12 (1988), 1115–1127.

[9] L.S. Wang, "Control System Design for a Flexible Arm," University of Maryland, M.S. Thesis TR-87-164, College Park, MD 20742, 1987.

[10] G.H. Frank, "Design and Real-time Control of a Flexible Arm," University of Maryland, M.S. Thesis, College Park, MD 20742, 1986.

CONVERT Reference Manual

This appendix describes the assignment and the various commands,which are used to write a Problem Description File as presented in Chapter 3, by a BNF (Backus Normal Form)–like rule. Identifiers enclosed in triangular brackets are *non-terminal symbols*; the symbols '→', '|', and '\' are so called *meta-symbols*; their meanings is given below. All others are *terminal symbols* and they may appear in the Problem Description File. The production rule is as follows: a '→' indicates that the item to its left may generate the items to its right. A '|' indicates that either the item to its left or right may be generated by the same non-terminal symbols. A '\' indicates the continuation of a line and it may be actually used in the Problem Description File. Besides the rules, some explanations or examples may be also added. By the productions of these rules, the user may write a Problem Description File without any syntax error. Part A below gives BNF rules for the assignment and the various commands described in Chapter 3. Part B contains the rules for the necessary elements.

Part A.

⟨*assignment*⟩ →

⟨*identifier*⟩ = ⟨*expression*⟩

⟨*constraint*⟩ →

`constraint` ⟨*quoted string*⟩ ⟨*soft or hard*⟩
⟨*pseudo-C code*⟩ ⟨*inequality type*⟩
`good_value=`⟨*expression*⟩
`bad_value=`⟨*expression*⟩

⟨*define*⟩ →

`define` ⟨*identifier*⟩ ⟨*quoted string*⟩

⟨*design parameter*⟩ →

`design_parameter` ⟨*identifier*⟩ \
`init=`⟨*expression*⟩ \

`variation=`⟨*expression*⟩ \
`min=`⟨*soft or hard expression*⟩ \
`max =` ⟨*soft or hard expression*⟩

Here, all the lines, except the first one, are optional. Also, the order is not important.

⟨*exit*⟩ →

`exit`

⟨*functional constraint*⟩ →

`functional_constraint` ⟨*quoted string* ⟩ ⟨*soft or hard*⟩
`for`⟨*identifier*⟩ `from` ⟨*expression*⟩ to ⟨*expression*⟩ ⟨*mesh type*⟩ ⟨*expression*⟩
⟨*pseudo-C code*⟩ ⟨*inequality type*⟩
`good_curve=`⟨*pseudo-C code*⟩
`bad_curve=`⟨*pseudo-C code*⟩

⟨*functional objective*⟩ →

`functional_objective` ⟨*quoted string*⟩
`for` ⟨*identifier*⟩ `from` ⟨*expression*⟩ `to` ⟨*expression*⟩ ⟨*mesh type*⟩ ⟨*expression*⟩
⟨*optimize*⟩ ⟨ *pseudo-C code*⟩
`good_curve=`⟨*pseudo-C code*⟩
`bad_curve=`⟨*pseudo-C code*⟩

⟨*global*⟩ →

`global` ⟨*newline-ended string*⟩

⟨*include*⟩ →

`include` ⟨*quoted string*⟩

⟨*initialization*⟩ →

`initialization` ⟨*pseudo-C code*⟩

⟨*objective*⟩ →

`objective` ⟨*quoted string*⟩
⟨*optimize*⟩ ⟨*pseudo-C code*⟩
`good_value=`⟨*expression*⟩
`bad_value=`⟨*expression*⟩

⟨*set*⟩ →

`set` ⟨*identifier*⟩ = ⟨*expression*⟩

⟨*trace*⟩ →

`trace` ⟨*identifier*⟩

Part B.

⟨*binary operator*⟩ →

`**` | `*` | `/` | `+` | `-`

⟨*constant*⟩ →

⟨*constant*⟩ is defined as any Fortran constant.
Examples: `1 -5.2 3.6e2 5d2`

⟨*digit*⟩ →

`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

⟨*expression*⟩ →

⟨*identifier*⟩ |
⟨*constant*⟩ |
⟨unary operator⟩ ⟨*expression*⟩ |
⟨*expression*⟩⟨binary operator⟩⟨*expression*⟩ |
(⟨*expression*⟩) |
⟨*two argument function*⟩ (⟨*expression*⟩ , ⟨*expression*⟩) |
⟨*one argument function*⟩ (⟨*expression*⟩)

⟨*optimize*⟩ →

`minimize` | `maximize`

⟨*identifier*⟩ →

⟨*letter*⟩ | ⟨*identifier*⟩⟨ *letter*⟩ | ⟨*identifier*⟩⟨*digit*⟩

⟨*inequality type*⟩ →

`<=` | `>=`

⟨*letter*⟩ →

`A` | ... | `Z` | `a` | ... | `z` | `_`

⟨*mesh type*⟩

`by` | `dec` | `times`

⟨*newline-ended string*⟩ →

⟨*newline-ended string*⟩ is defined as any character string ended at a newline character. The newline character is part of the string.

⟨*one argument function*⟩ →

```
abs  | acos  | asin  | atan  | cos  | sin  | sign
cosh | sinh  | exp   | log   | log10 | tan
```

⟨*pseudo-C code*⟩ →

⟨*pseudo-C code*⟩ is defined as a valid C code augmented by the *import* statement enclosed in curly brackets. An import statement consists of the identifier 'import' followed by a list of identifiers separated by commas. These identifiers must have been declared with identifier type 'variable', 'define' or 'design parameter'.

An import statement ends at a semicolon or a newline character.

⟨*quoted string*⟩ →

⟨*quoted string*⟩ is defined as any character string quoted in balanced double quotes or single quotes., so that (double) quoted strings may contain (single) quotes and conversely. Quotes are not part of the string.

Examples:

```
"This is a string quoted by double quotes"
'This is a string quoted by single quotes'
"I don't know your phone number"
```

⟨*soft or hard*⟩ →

```
soft | hard
```

⟨*soft or hard expression*⟩ →

[⟨*expression*⟩, ⟨*expression*⟩] | ⟨*expression*⟩

⟨*two argument function*⟩ →

```
atan2 | max2 | min2
```

⟨*unary operator*⟩ →

```
+ | -
```

APPENDIX B

SOLVE Reference Manual

Outline

```
NAME
     algo - display/change optimization algorithm

SYNOPSIS
     algo
     algo FSQP
     algo FIRST_ORDER

DESCRIPTION
     Used without argument, 'algo' displays the optimization
     algorithm currently being used. Otherwise, 'algo' switches
     to corresponding algorithm. If there is a change of algorithm,
     'algo' resets iteration number to 0, uses the current design
     parameter vector as the initial one, changes the prompt
     (<> for FSQP, [] for FIRST_ORDER). The default algorithm
     is FSQP.
```

```
NAME
     erase - erase graphic screen

SYNOPSIS
     erase

DESCRIPTION
     'erase' erases the graphic screen.

SEE ALSO
     plot
```

```
NAME
     freeze   - freeze design parameter(s)
     unfreeze - unfreeze design parameter(s)

SYNOPSIS
     freeze <name1> <name2> ...
     unfreeze <name1> <name2> ...

DESCRIPTION
     'freeze' freezes the value of design parameters <name1>,
     <name2>, ...,  during subsequent optimization until they
     are 'unfreeze'd.

EXAMPLE
     freeze x
     freeze x y
     unfreeze x z
```

```
NAME
     goutput - redirect graphical output

SYNOPSIS
     goutput
     goutput <file>

DESCRIPTION
     Used without argument, 'goutput' displays the current
     redirection of graphical output. The default is the
     standard output. Otherwise, it redirects future graphical
     outputs to the file <file>.
```

```
NAME
     help - display manual entry

SYNOPSIS
     help
     help <command>
     help <concept>

DESCRIPTION
     Used without argument or with 'help' as its first argument,
     'help' prints this message. Otherwise, 'help' prints the
     manual entry for the command <command> or the concept
     <concept>. Following are available commands:

     algo       - display/change optimization algorithm
     erase      - erase graphic screen
     freeze     - freeze design parameter(s)
     goutput    - redirect graphic output
     help       - print this message
     identify   - identify design problem being solved
     iter       - display/change iteration number
     pcomb      - display performance
     plot       - plot functional objective/constraint
     print      - display design parameters
     quit       - exit SOLVE
     reset      - reset SOLVE
     run        - perform optimization
     scale      - change nominal variation of design parameter
     set        - set values of design parameters
     setgb      - change good/bad values or curves of specification
     sim        - call simlat(x,SIM_INTA)
     store      - store design parameters into file
     time       - display CPU time
     trace      - trace optimization algorithm and/or simulator (on/off)
     unfreeze   - unfreeze design parameter(s)

     and information for which help is available :

     info       - how to obtain the CONSOLE package
     interrupt  - stop optimization prematurely
     matlab     - how to use CONSOLE and MATLAB jointly
     news       - local news
     optimal    - explanation of optimal code (optimality)
     simlat     - interface between CONSOLE and simulator
     version    - changes in the version 1.1 of CONSOLE
```

```
NAME
     identify - identify design problem being solved

SYNOPSIS
     identify

DESCRIPTION
     'identify' identifies the design problem being solved.
     A typical output would be

        PROBLEM: robot_joint
           4  Design Parameter(s)
           1  Objective(s)
           2  Functional Objective(s)
           3  Constraint(s)
           1  Functional Constraint(s)
```

```
For assistance with or information on CONSOLE, contact:

   Professor Michael Fan
   School of Electrical Engineering
   Georgia Institute of Technology
   Atalanta, GA 30332
   (404) 853-9828
   Email: eefacmf@prism.gatech.edu

or

   Professor Andre Tits
   Systems Research Center
   University of Maryland
   College Park, MD 20742
   (301) 405-3669
   Email: andre@src.umd.edu

CONSOLE is available for VAX(UNIX/VMS), MicroVAX(UNIX/VMS), SUN computers.
```

SUBJECT

interrupt - stop optimization algorithm prematurely

DESCRIPTION

When a command such as 'run 5' is issued, the optimization algorithm will usually perform 5 more iterations and stop, or it could stop earlier if a local optimal solution is obtained (see 'optimal' for details). However, it could be forced to stop prematurely by the user. This is done by sending interrupt signals. An interrupt signal could be generated by depressing either 'control-C', 'break' or 'delete' key which depends on the current terminal setup (see Unix manual entry 'stty' for details). When an interrupt is given, SOLVE calls the C routine simlat(x,SIM_INTR), where x is the design parameter vector at the current iteration. If this is the first interrupt, SOLVE also prints out the message

```
An interrupt has been detected by SOLVE. SOLVE will
stop  running the optimization algorithm as soon as
it finishes the current iteration.


CONTINUING EXECUTION ...
```

and continues the execution of the optimization until the current iteration is finished. This is known as 'soft interrupt'. If a second interrupt is given at the same iteration, SOLVE prints out the message

```
WARNING: A second interrupt has been received before
SOLVE has processed the first ... SOLVE is possibly
hung in a routine.


INTERRUPTED ...
```

and it stops momentarily. This is known as a 'hard interrupt'.

SEE ALSO

optimal, simlat, trace

```
NAME
     iter - display/change iteration number

SYNOPSIS
     iter
     iter <number>

DESCRIPTION
     Used without argument, 'iter' displays the current
     iteration and the number of the last available iteration.
     The last available iteration is defined as the iteration
     of highest number for which design parameters have been
     computed. If a nonnegative integer <number> is given
     as the argument, 'iter' sets the current iteration to
     <number> provided that <number> is no more than the last
     available iteration.

EXAMPLE
     iter
     iter 0
```

```
SUBJECT
     matlab

DESCRIPTION
     The SOLVE-MATLAB interface is included in the standard
     distribution of CONSOLE. Assuming that the user is familiar
     with the use of CONSOLE and MATLAB as individual programs,
     here we describe how to use them together. Consider the
     tutorial example in the manual, for which the corresponding
     PDF (Problem Description File) looks like

        design_parameter x init=5  min=0
        design_parameter y init=10

        objective "quadratic"
           minimize {
              import x, y;
              return (x-1)*(x-1)+(y-2)*(y-2);
              }
           good_value=1
           bad_value=4

        constraint "linear" soft
           {  import x, y;
              return x+y;
              }
           <= good_value=1
              bad_value=2

     Now suppose that we wish to use MATLAB to evaluate the
     objective (i.e., (x-1)*(x-1)+(y-2)*(y-2)) and the constraint
     (i.e., x+y) for given values of x and y when the optimization
     is performed. Here is a list of things we need to do :

        1. create a M-file named "init.m" which does all operations
           that do not depend upon the design parameters (x and y).
           This file is optional, and in our case, is not necessary.
           However, we still write one to show how it looks like :

             %init - simulator one time initialization
               one = 1;
              two = 2;

        2. create a M-file named "simu.m" which does all operations
           that depend upon the values of the design parameters and
           then save the results in the file "simu.mat" :

             %simu - perform simulation
               obj = (x-one)^2 + (y-two)^2;
              const = x+y;
```

```
      save simu obj const
```

3. modify the PDF as follows :

```
      design_parameter x init=5  min=0
      design_parameter y init=10

      objective "quadratic"
         minimize {
            double getout();
            return getout("obj", 1);
            }
         good_value=1
         bad_value=4

      constraint "linear" soft
         {  double getout();
            return getout("const", 1);
            }
         <= good_value=1
            bad_value=2
```

4. then, at csh command prompt, type (assuming the file pdf is PDF)

```
      % convert pdf
       % solve -matlab pdf
```

   and proceed as before. The optimization should give you identical results compare to that if you use CONSOLE alone.

There are a couple of useful remarks :

1. As mentioned before, the file "init.m" is optional. However, the file "simu.m" must be present and properly defined.

2. The working directory must be writable by the user (since MATLAB will create the file "simu.mat" for storing simulation results).

3. In the file "simu.m", the names follow "save simu" could be either variables or arrays.

4. The routine getout() reads the file "simu.mat" and returns the simulation result in double according to its arguments. The first argument is the name of variable or array, and the second is the index. For variables, the index is always one, and for arrays, the index gives the corresponding element in the array.

5. You are probably wondering why the variables x and y are never defined in either "init.m" or "simu.m". Here is the basic concept of the operation between SOLVE and MATLAB : first SOLVE asks MATLAB to evaluate the M-file "init.m". Then during optimization, each time when SOLVE changes the values of the design parameters, it sends out strings like

   ```
   x=5.000000000;
   y=10.00000000;
   ```

   to MATLAB, and then asks MATLAB to evaluate the M-file "simu.m". Finally, the routine getout() gets back the simulation results to SOLVE. Therefore, any assignment to a design parameter in the file "init.m" will be overwritten by the assignment given by SOLVE, and any assignment to a design parameter in the file "simu.m" will overwrite the assignment given by SOLVE. You may still want to put initialization of design parameters in the file "init.m" such that "init.m" and "simu.m" can run separately with MATLAB. However, you never put assignments of design parameters in the file "simu.m".

6. In the prompt of SOLVE, when you type

   ```
   sim
   ```

   and you will see

   ```
   Enter MATLAB, type 'back' to leave
   >>
   ```

   Now, you are effectively talking to the copy of MATLAB which SOLVE was talking to. You may examine or even modify the value of any variable (including variables affecting the optimization). In principle, you may talk to MATLAB just like you are using MATLAB alone, with the exceptions that (1) commands that give more than a screenful of output (e.g., "help" and "demo") should be avoided (SOLVE may hang up), and (2) interrupt is disabled.

SUBJECT

optimal - explanation of optimal code (optimality)

DESCRIPTION

SOLVE stops optimization process under one of the following circumstances and prints out a message with an 'optimal' code associated with each case (codes 1-3 are for algorithm FIRST_ORDER, and codes 5-8 are for algorithm FSQP) :

Optimal code	Descriptions
1	A local optimal solution has been found. (The Euclidean norm of the search direction is small and the threshold for determining the most active specifications is smaller than 1e-5.)
2	A local optimal solution has been found. (The Euclidean norm of the gradient of the most active specification is smaller than 1e-5.)
3	Further decrease of the most active specifications cannot be achieved. (A suitable move along the search direction cannot be made after 100 trials. This is sometimes due to inaccurate evaluations of gradients).
4	Not used.
5	A local optimal solution has been found. (The Euclidean norm of the search direction is smaller than the square root of machine precision.)
6	A feasible point has been found that satisfies all constraints and there is no objective specification.
7	A proper search direction cannot be found due to errors reported by QPSOL (quadratic programming sub-problems).
8	Further decrease of the most active specifications cannot be achieved. (A suitable move along the search direction cannot be made after 50 trials, or the Euclidean norm of the difference of two successive iterates is smaller than machine precision.)

```
NAME
     pcomb - display performance

SYNOPSIS
     pcomb

DESCRIPTION
     'pcomb' displays the performance of the optimization
     algorithm and the design problem at the current
     iteration. Please refer to 'CONSOLE Users's Manual'
     (Section 4.4) for details. The command 'pcomb' can be
     issued for each iteration automatically (see 'run'
     for details).

EXAMPLE
     pcomb

SEE ALSO
     run
```

```
NAME
     plot - plot functional objective/constraint

SYNOPSIS
     plot FO <number>
     plot FC <number>
     plot FO <number> log
     plot FC <number> log

DESCRIPTION
     'plot' erases the graphic screen and plots <number>th
     functional objective (FO) or functional constraint (FC).
     the x-axis of the plot is of linear scale if the
     mesh type of the specification is 'by'. Otherwise
     ('dec' or 'times') the x-axis is of logarithmic scale.
     Y-axis is of logarithmic scale if the argument 'log'
     is present. Otherwise, linear scale is used.

SEE ALSO
     erase
```

```
NAME
     print - display design parameters

SYNOPSIS
     print
     print <number>

DESCRIPTION
     Used without argument, 'print' displays values, variations
     and changes with respect to iteration 0 and previous
     iteration of design parameters at the current iteration. If
     an integer number <number> is given, 'print' displays the
     similar information for iteration <number>. <number> should
     not exceed the number of the available iteration.
     A typical output looks like

        Name         Value         Variation  wrt 0   Prev   Iter=10
        x            1.35200e+00   1.0e+00      1%     2%
0        y            9.99091e+01   1.0e+00     -2%     4%
3
     The columns marked "wrt 0" and "Prev" stand for change with
     respect to iteration 0 and previous iteration respectively.
     If the magnitude of a percentage change is more than 9999,
     '****' is printed. The command 'print' can be issued for
     each iteration automatically (see 'run' for details).

EXAMPLE
     print
     print 3

SEE ALSO
     iter, store, run
```

```
NAME
     quit - exit SOLVE

SYNOPSIS
     quit

DESCRIPTION
     'quit' calls the C routine simlat(x,SIM_QUIT) if it has
     been given to SOLVE by the user and exits SOLVE, where x
     is the design parameter vector at the current iteration.

SEE ALSO
     simlat
```

```
NAME
     reset - reset SOLVE

SYNOPSIS
     reset

DESCRIPTION
     'reset' erases values of specifications which have been
     stored for the current iteration. It then calls the C
     routine simlat(x,SIM_RSET) if it has been given to SOLVE
     by the user, where x is the design parameter vector
     at the current iteration.

SEE ALSO
     simlat
```

```
NAME
     run - perform optimization

SYNOPSIS
     run <number>
     run <number> <commands>

DESCRIPTION
     'run' performs <number> iterations of the optimization
     algorithm. <number> must be nonnegative integer. 'run 0'
     means to evaluate various specifications for the current
     iteration. Iterations may prematurely end due to an
     interrupt sent by the user (see 'interrupt' for details)
     or due to a an optimality condition is satisfied in the
     optimization algorithm (see 'optimal' for details). If
     a list of commands <commands> is also given, 'run'
     executes them during or after each iteration. Possible
     commands are

       active - display gradients of active specifications
                and the search direction in the optimization
                algorithm for the current iteration.

       print  - display design parameters for the current
                iteration.

       pcomb  - display performance for the current iteration.

       time   - display CPU time.

       pause  - pause after each of the above commands until
                the user types 'return'.

     The order of commands given is irrelevant. The command
     'active', if given, is executed after the optimization
     algorithm finds a search direction. The commands 'print',
     'pcomb' and 'time', if given, are executed after each
     iteration is completed. The order of execution is
     'print', 'pcomb' then 'time'.

EXAMPLE
     run 0
     run 0 pcomb
     run 3 active print pause
     run 2 pcomb print active pause time

SEE ALSO
     print, pcomb, time
```

```
NAME
     scale - change nominal variation of design parameter

SYNOPSIS
     scale <name> = <number>

DESCRIPTION
     'scale' changes the nominal variation of a design
     parameter. <name> is the design parameter and <number>
     its new nominal variation. <number> is either a positive
     integer or a positive floating point constant.

EXAMPLE
     scale x = 10
```

```
NAME
     set - change value of design parameter

SYNOPSIS
     set <name> = <number>

DESCRIPTION
     'set' changes the value of a design parameter at the
     current iteration. <name> is the design parameter and
     <number> its new value. <number> is either an integer
     or a floating point constant.

EXAMPLE
     set x = 0.5
```

```
NAME
     setgb - change good/bad values or curves of specification

SYNOPSIS
     setgb O  <number> = <good>, <bad>
     setgb C  <number> = <good>, <bad>
     setgb FO <number> = <good>, <bad>
     setgb FC <number> = <good>, <bad>

DESCRIPTION
     'setgb' changes the good and bad values or curves of
     the <number>th objective (O), constraint (C), functional
     objective (FO) or functional constraint (FC). The new good
     and bad values or curves are <good> and <bad> respectively.
     For functional specification, the good and bad curves
     become constant curves after changed by 'setgb'.

EXAMPLE
     setgb O 1 = 1, 2
     setgb FC 2 = 10, 2.5
```

```
NAME
     sim - call simlat(x,SIM_INTA)

SYNOPSIS
     sim

DESCRIPTION
     'sim' calls the C routine simlat(x,SIM_INTA) if it has
     been given to SOLVE by the user, where x is the design
     parameter vector at the current iteration.

SEE ALSO
     simlat
```

```
SUBJECT

    simlat - interface between CONSOLE and simulator

DESCRIPTION
    The C routine 'simlat' is the main part for a sophistical
    interface between CONSOLE and a simulator. While SOLVE
    is unaware of what simulator is currently being used,
    it calls the routine 'simlat' with specific arguments
    whenever there is a possibility that the simulator requires
    that some action be taken. If no routine 'simlat' is
    provided with the simulator, the default dummy routine
    will be called and no action will be taken. Please refer
    to 'CONSOLE Users manual' (Section 5.2) for details.
```

```
NAME
     store - store design parameters into file

SYNOPSIS
     store <file>

DESCRIPTION
     'store' stores the design parameters at the current
     iteration into file <file> in a form that can be added
     directly into the Problem Description File, where <file>
     is any quoted string.

EXAMPLE
     store "dp_save"
```

```
SUBJECT
     terminal

DESCRIPTION
     The version 1.1 of SOLVE only supports tektronic 4014 terminal
     type. Consequently, no terminal setup is necessary.

SEE ALSO
     goutput
```

```
NAME
     time - display CPU time

SYNOPSIS
     time

DESCRIPTION
     'time' displays a summary of CPU time used by SOLVE.
     A typical output is

        Cpu time:  Total 5   Delta 2 (seconds)

     which shows that the total CPU time used by SOLVE is
     five seconds and that two seconds have elapsed since
     last 'time' command was issued. The command 'time' can
     be issued for each iteration automatically (see 'run'
     for details).

SEE ALSO
     run
```

```
NAME
     trace - trace optimization algorithm and/or simulator
             (on/off)

SYNOPSIS
     trace
     trace <number>

DESCRIPTION
     Used without argument, 'trace' is the same as 'trace 1'.
     'trace' traces the optimization algorithm and/or the
     computation of the simulator based on the value of
     <number>, which should be 0, 1, 2 or 3. The corresponding
     action is as follows.

        <number>  Action

        0         'trace' turns off the trace for the
                  optimization algorithm and calls the C
                  routine simlat(x,SIM_NOTR), where x is
                  the design parameter vector at the
                  current iteration.

        1         'trace' turns on the trace for the
                  optimization algorithm.

        2         'trace' calls the C routine
                  simlat(x,SIM_TRAC), where x is the design
                  parameter vector at the current iteration.

        3         'trace' turns on the trace for the
                  optimization algorithm and calls the C
                  routine simlat(x,SIM_TRAC), where x is
                  the design parameter vector at the
                  current iteration.

     simlat(x,SIM_NOTR) and simlat(x,SIM_TRAC) may be
     implemented in such a way that 'trace 0' turns off the
     trace for both the optimization algorithm and the
     simulator; 'trace 1' turns on the trace only for the
     optimization algorithm; 'trace 2' turns on the trace
     only for the simulator; and 'trace 3' turns on the
     trace for both the optimization algorithm and the
     simulator.

EXAMPLE
     trace
     trace 2

SEE ALSO
```

```
simlat
```

```
SUBJECT

    Changes in Version 1.1 of CONSOLE

DESCRIPTION
    Things changed from version 1.0 to version 1.1:

    1. A superlinearly convergent algorithm (FSQP) is added.

    2. New commands algo, freeze, unfreeze, goutput are added.

    3. Hard interrupt is implemented.

    4. Only support tektronic 4100 terminal for graphics.
       Consequently, the command terminal is removed.

SEE ALSO
    algo, freeze, goutput, interrupt
```

INDEX

INDEX