NUOPT for S-PLUS

User's Guide

Version 1.4

Mathematical Systems, Inc.
Tokyo, Japan

Insightful Corporation
Seattle, Washington, USA

March 2002

How to Contact Insightful Corporation

Telephone	+1 (206) 283 8802
Fax	+1 (206) 283 8691
WWW	`www.insightful.com`
Mail	Insightful Corporation 1700 Westlake Avenue N, Suite 500 Seattle WA 98109 USA
Email	`info@insightful.com`
Technical Support	`support@insightful.com`
Bug reports	`bugs@insightful.com`

Proprietary Notice

The correct bibliographical reference for this document is as follows:

> *NUOPT for S-PLUS User's Guide Version 1.4*, Mathematical Systems, Tokyo, Japan and Insightful Corp., Seattle, Washington, USA (2002).

Contents

Table of Contents

1 Introduction

NUOPT for S-PLUS is a combination of the S-PLUS programming environment, the NUOPT optimizer and the SIMPLE modeling language. NUOPT stands for NUmerical OPTimization and
SIMPLE stands for System for Interactive Modeling in a Programming Language Environment.
NUOPT is a collection of powerful optimization methods, including:

- primal-dual interior point method with higher order correction for Linear Programming (LP) models. [higher]
- simplex method for Linear Programming (LP) and mixed integer programming (MILP) models. [simplex]
- primal-dual interior point method based on line search for general Convex Programming (CP) models including convex Quadratic Programming (CQP) models. [line]
- primal-dual interior point method based on trust region method for general Non-Linear Programming (NLP) models. [trust]
- primal-dual interior point method based on quasi-Newton method for general Non-Linear Programming (NLP) models. [bfgs]
- active set method for convex Quadratic Programming (CQP) models and mixed integer Quadratic Programming(MIQP) models. [asqp]

The various methods within NUOPT have different characteristics and address different problem classes. The algorithms can handle both unconstrained and constrained problems. The simplex interior-point methods (except bfgs), and active set method in NUOPT are designed to handle large-scale problems. Note that the interior point method based on trust region [trust] and quasi-Newton method [bfgs] can deal with general Non-Linear problem, but the obtained solution cannot guaranteed to be a global optimal solution but a local solution. That is why only the convergence to one of the local solutions is proved in theory. There is no established method to obtain the global optimal for general non-linear solution, and problem-specific technique is indispensable. This is out of scope of this package. But one may be able to accomplish this by writing some S-PLUS script that run NUOPT many times by changing starting value or constraints.

By default, the choice of the method is transparent to users because NUOPT automatically selects an appropriate method based on information from SIMPLE (see Table 1). Users can also specify the optimization method explicitly if they wish to do so.

	L P	MILP	CP	CQP	NLP	MIQP
simplex	+	⊕				
higher	⊕					
line	+		+	+		
trust	+		⊕	⊕	⊕	
bfgs	+		+	+	+	
asqp	+			+		⊕

Table 1: Optimizations methods in NUOPT. The presence of a symbol (either + or ⊕) indicates that a method (row) is applicable to a model (column). The symbol ⊕ indicates NUOPT's default used via modeling language SIMPLE[1].

Optimization problems are communicated to NUOPT through the modeling language SIMPLE, which is designed to describe mathematical systems. Such systems are characterized by mathematical relations between various quantities of interest. In the case of optimization problems, these relations are equality constraints and inequality constraints involving independent (unknown) variables, together with an objective function to be optimized.

The syntax of SIMPLE is intended to be close that of familiar mathematical expressions. In SIMPLE, mathematical relations that contain indexed quantities are written literally, allowing users to describe large-scale systems in a compact way. Because the models are described as sets and elements whose values are specified separately, users can solve many different systems using the same abstract models with different data.

SIMPLE computes information about function values, as well as first and second derivatives, which is made available to NUOPT in a form that can be processed efficiently by the various solvers. To compute derivatives, SIMPLE uses recently developed automatic differentiation techniques, executed by constructing computational graphs of the system and using the operator-overloading functionality of the S-PLUS language. SIMPLE statements create pointers to underlying C++ code, so that, unlike S-PLUS objects, SIMPLE objects do not persist after an S-PLUS session is terminated.

In addition to solving models defined by SIMPLE, the NUOPT optimizer can also be directly applied to linear (LP) and quadratic programming (QP) models with the S-PLUS function solveQP (see section 2).

[1] When used via solveQP interface, the default method becomes slightly different. That is, for QP problems without integer variables becomes “line” (line search method for Convex programming).

1.1 Starting NUOPT

To start NUOPT for S-PLUS in UNIX or Windows:

1. Start S-PLUS.
2. Include the SIMPLE system.

```
>module(nuopt,first=T)
```

2 solveQP for Mixed Integer Linear and Quadratic Programs

The S-PLUS function solveQP is designed for solving (mixed integer) linear and quadratic programming models with NUOPT without using the SIMPLE interface.

The general form of a quadratic program is

$$\min imize_x \quad \tfrac{1}{2} x^T Q x + q^T x$$

subject to

(1)

$$\begin{aligned} c_{LO} \;&\le\; Ax \;&\le\; c_{UP} \\ b_{LO} \;&\le\; x \;&\le\; b_{UP} \\ & & x_i : Integer\,,\ i \in I \\ & & x_i : Continuous\ ,\ i \notin I \end{aligned}$$

where Q is an $n \times n$ symmetric matrix specifying the Hessian matrix (quadratic part) of the objective function, and q is an n-vector specifying the linear part of the objective function. When Q vanishes, the system (1) reduces to the general form of a linear program. The factor $\frac{1}{2}$ appears in the standard form of the model in order to avoid a factor of 2 in the derivatives (gradient vector and Hessian matrix). An important application of quadratic programming is the solution of linearly-constrained linear least-squares problems, including portfolio optimization problems (see section 6.4). Some of the variables (with indices that belongs to index set I) may be restricted to be integer. In such a case this becomes mixed integer Linear/Quadratic problem.

The S-PLUS function solveQP for solving linear and quadratic programs has the following argument list:

```
> args(solveQP)
function(objQ, objL, A, cLO, cUP, bLO, bUP, x0, isint,
         type = minimize, trace = T)
NULL
```

where objQ corresponds to the Hessian matrix Q of (1) and objL corresponds to the vector q specifying the linear part of the objective in (1). The argument x0 is a vector specifying initial

values for the variables `x`, whose default value is the zero vector of length n. The argument isint is a logical vector (T or F) specifying which component of variables are integer variables.

For example, suppose we want to find the minimum Euclidean norm solution to a system of underdetermined linear equations. This problem can be posed as the following quadratic program:

$$\text{minimize}_{\beta} \quad \beta^T \beta \qquad (2)$$

$$\text{subject to} \quad X\beta = y,$$

where X is a matrix with more columns than rows, and y is a vector. This can be solved with `solveQP` as follows:

```
> objQ <- diag(ncol(X))   # no need for factor of 1/2 since no linear term
> sol <- solveQP(objQ=objQ, A=X, cLO=y, cUP=y, type = "minimize")
```

This problem can also be solved using the singular-value decomposition (e. g. [5]). However, in the large, sparse case, the quadratic programming formulation may be more efficient. Matrices `objQ` and `A` can be specified to `solveQP` in a sparse format. For example, if X is given by:

```
> X
      [,1] [,2] [,3] [,4]
 [1,]  1.1  0.0  -1.3  0.0
 [2,]  0.0 -2.2   0.0  2.4
```

and y is any vector of length 2, the QP (2) would be solved in sparse format. The following is the representation of the same problem by sparse format.

```
> p <- ncol(X)
> sparseQ <- list(1:p,1:p,rep(1,p))
> sparseX <- list(c(1,1,2,2), c(1,3,2,4), c(1.1,-1.3,-2.2,2.4))
> solveQP(objQ=sparseQ, A=sparseX, cLO=y, cUP=y, type="minimize")
```

The matrix is given by a list that consists of the three vectors as below:

Row number of nonzero elements

Column number of nonzero elements

Value of nonzero elements

All these vectors have the same length that is identical to the number of non-zero elements in the

matrix. For many applications, this may be more compact representation of Hessian or Constraint matrix.

You can specify some of the variables are integer variables by specifying the logical vector `isint` whose length is identical to the number of variables. The 'T' components indicates the corresponding component of variable is integer variable.

The knapsack problem provides an example of the use of integer variables in `solveQP`. A mathematical statement of the problem is:

$$maximize_x \quad v^T x$$ same as (5) in 3.3.1

$$\text{subject to} \quad x \in \{0,1\}$$
$$s^T x \le C$$
$$x \in \{0,1\}$$

To solve this problem with

$$v = (39,13,68,15,10,20,31,15,41,16)^{\mathrm{T}}$$
$$s = (39,13,68,15,10,20,31,15,41,16)^{\mathrm{T}}$$
$$C = 121$$

you should define the corresponding S-PLUS vectors `v, s,` and scalar `C`, and

```
> n <- length(v)
> solveQP(objL=v,A=matrix(s,1,n),cLO=0,cUP=C,bLO=rep(0,n)
                ,bUP=rep(1,n),type=maximize,isint=c(rep(T,n))
```

The underlined arguments specify that some (this time all) of the variables are to be integer.

3 Defining Optimization Models with SIMPLE

There are two approaches to defining an optimization model with SIMPLE. They can be defined dynamically by typing the necessary definitions at the S-PLUS prompt level, or else they can be defined within functions. Note that for the S-PLUS environment for windows, "prompt level" means the S-PLUS "commands window," **not** the script file window. The definitions typed in the script window outside a function definition are ignored.

As an example, we use a statistical model that is a linear least-squares regression with positivity constraints on the regression parameters. Given a matrix X and and a vector y whose length is equal to the number of rows in X, the idea is to minimize the sum of squared residuals, where $r = y - X\beta$ is the vector of residuals, subject to the constraints that the coefficients β be non-negative. A mathematical statement of this problem would be:

$$\min imize_{\beta} \quad r^T r \text{, where } r \equiv y - X\beta \qquad (3)$$

$$\text{subject to } \beta > 0$$

As an example let X and y be the S-PLUS datasets stack.x and stack.loss, respectively (a description is available by typing help(stack) at the S-PLUS prompt). First, we show how to set up the problem with the default model approach. The command

```
> new.model()
```

clears the default model defined by SIMPLE statements typed at the prompt. Now we type a sequence of SIMPLE statements that specify the model. We start by setting up pointers to the necessary sets and indexes (values will later be assigned to these automatically):

```
> Res <- Set()  # set of indexes for residuals
> Var <- Set()  # set of indexes for variables
> i <- Element(set=Res)
> j <- Element(set=Var)
```

Now we define the vector y and matrix X as parameters in SIMPLE:

```
> nres <- length(stack.loss) # number of residuals
> y  <- Parameter(list(1:nres, stack.loss), index = i)
> X  <- Parameter(stack.x, index = dprod(i,j))
```

Sets Res and Var now have values:

```
> Res
{ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 }

> Var
{ Air Flow Water Temp Acid Conc. }
```

Assignment of values to sets in SIMPLE is discussed in further detail in section 3.1. Next, we define the variables together with their bounds that are the problem constraints:

```
> beta <- Variable(index=j)  # pointer to variables
> beta[j] >=0              # variable bound constraints
```

Finally, we define the objective function:

```
> r <- Expression(index=i)          # pointer to vector of residuals
> r[i] ~ y[i]-Sum(X[i,j]*beta[j],j)   # define vector of residuals
> obj <- Objective(type=minimize)    # pointer to objective function
> obj ~ Sum(r[i]*r[i],i)            # define objective function
```

The model typed so far can be displayed as follows:

```
> show.model()
   1.  beta[j]>=0
   2.  r[i]~y[i]-(Sum(X[i,j]*beta[j],j))
   3.  obj~Sum(r[i]*r[i],i)
```

In order to solve the model, we use the System function to expand (compile) the model into a form (machine code) that can be directly processed by the solver:

```
> sys.stack <- System()
1-1 :  beta[Acid Conc.] >= 0
1-2 :  beta[Air Flow] >= 0
1-3 :  beta[Water Temp] >= 0
```

```
obj<objective>: (42+(-1)*(80*(beta[Air Flow])+27*(beta[Water Temp])+ ...
```

Finally, the system `sys.stack` can be solved by calling solve.

```
> sol.stack.pos <- solve(sys.stack)
```

Special solve methods have been created for use with `SIMPLE` system objects in `S-PLUS`. Once defined, variables can be selectively fixed and constraints can be added or removed from the model in an analysis (see section 4).

Another way to define models is to write a function containing the relevant SIMPLE commands. For example, the model (3) corresponds to the function `LregPos` defined as follows:

```
> LregPos
function(X, y)
{
        Res <- Set()
        Var <- Set()
        i <- Element(set = Res)
        j <- Element(set = Var)
        nres <- length(y)
        y <- Parameter(list(1:nres, y), index = i)
        X <- Parameter(X, index = dprod(i, j))
        beta <- Variable(index = j)
        beta[j] >= 0
        r <- Expression(index = i)
        r[i] ~ y[i] - Sum(X[i, j] * beta[j], j)
        obj <- Objective(type = "minimize")
        obj ~ Sum(r[i] * r[i], i)
}
```

Note that `LregPos` does not return a value like a typical `S-PLUS` function; instead it specifies a model for use with the `NUOPT` solver. The model function `LregPos` can be expanded with the stack data as follows:

```
> sys.LregPos.stack <- System(model=LregPos, stack.x, stack.loss)
```

giving the system we originally defined by typing at the prompt. The above processes can also be packaged into a single function:

```
SolveLregPos <- function(X,y)
{
       solve(System(model = LregPos, X, y))
}
```

You may have noticed that an alternate formulation optimization problem having the same solution as (3) is

$$\begin{aligned} \min imize_{\beta,r} \quad & r^T r \\ \text{subject to} \quad & r = y - X\beta \\ & \beta > 0 \end{aligned} \tag{4}$$

Here auxiliary variables r representing the residuals have been introduced, and their definition in terms of X and y added as linear constraints to the problem. This formulation of the problem can expressed in SIMPLE by changing statements

```
> r <- Expression(index=i)
> r[i] ~ y[i]-Sum(X[i,j]*beta[j],j)
```

to

```
> r <- Variable(index=i)
> r[i] == y[i]-Sum(X[i,j]*beta[j],j)   # constraint definition
```

An analysis of alternative formulations for a similar model is given in section 6.4.

Note also that (3) and (4) are quadratic programs, so that they could be solved with `solveQP`. In the case of (3), the Hessian matrix would be $X^T X$ (multiplied by a factor of 1 2 when calling `solveQP`), and the vector specifying the linear part would be $-y^T X$. For (4), the quadratic part of the objective would be the identity matrix with dimension equal to the length of the residual vector r (in `S-PLUS`, `diag(length(r))`), and there would be no linear part of the objective.

The classes of objects used in `SIMPLE` for defining optimization models are as follows:

Set, *Element*, *Parameter*, *Variable*, *IntegerVariable*, *Expression*, *Objective*, *Constraint*, *Graph*, *System*, and *Condition*.

Except for *Condition*, all have a generator function of the same name. Models in SIMPLE are lists of statements involving these objects. Their formal definitions are given in the appendix, and their use is described in more detail in the sections that follow.

In addition, a total of 16 functions are provided to facilitate designing, testing and solving systems (models) as shown in Table 3. These functions are also described in the sections that follow.

Table 3: Functions for system controlling

Function	Arguments	Brief Description
show.model	sys	list model definitions
System	model,<args>,trace	make a new system by expanding constraints <args>: arguments for function model
print.System	sys, type, obj, num	list contents of a system
delete.con	sys,obj,mem	remove a constraint from a system
restore.con	sys,obj,mem	restore a deleted constraint from a system
add.con	sys,obj	add a constraint to a system
solve.System	sys,obj,trace	solve a system
current	sys,obj	show the current value of obj of a system
init	sys,obj	show the init value of obj of a system
lowerB	sys,obj	show the lower bound of Variable or Constraint of a system
upperB	sys,obj	show the upper bound of Variable or Constraint of a system
dual	sys,obj	show the dual bound of Variable or Constraint of a system
as.list	obj	transform to a list
fix.Variable	sys,x	fix the value of a variable in a system
unfix.Variable	sys,x	unfix the value of a variable in a system.
new.model		renew the default model

3.1 *Set* and *Element* Objects; Indexing and Auto-assignment

Set and *Element* objects are used for indexing in SIMPLE. *Sets* in SIMPLE are analogous to vectors

of character strings in S-PLUS used as names for vectors and dimnames for arrays. For example, to specify a parameter vector StackLoss with the values of the S-PLUS dataset stack.loss:

```
> l <- length(stack.loss)
> I <- Set(1:l)
> i <- Element(set=I)
> StackLoss <- Parameter(index=i)   # sets up pointer
> StackLoss ~ list(1:l,stack.loss)  # assigns value
> StackLoss
   1  2   3  4   5  6  7   8  9 10 11 12 13 14 15 16 17 18 19 20 21
  42 37 37 28 18 18 19 20 15 14 14 13 11 12  8  7   8  8  9  15 15
attr(, "indexes"):
[1] "i"
```

In SIMPLE, an Element is used as a subscript for objects defined to represent elements of a base set. In the above example, i is defined to represent elements of set I, or, equivalently, I is the base set of the index i. The contents of a *Set* object in SIMPLE are independent of any associated *Element* objects. The latter, once defined, must always be used as subscripts for the base set specified in their definition, even if the contents of the base set are changed.

Another way of defining an index set in SIMPLE is through auto-assignment, for example:

```
> I <- Set()
> i <- Element(set=I)
> StackLoss <- Parameter(index=i)
> I
{ }
> StackLoss ~ list(1:l,stack.loss)
> I
{ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 }
```

Assignment to StackLoss had the side effect of changing the index set I in this example. The parameter StackLoss does not have to be indexed by numbers; since it has length 21, it could, for example, be indexed by letters of the alphabet:

```
> I <- Set()
> i <- Element(set=I)
```

```
> StackLoss <- Parameter(index=i)
> StackLoss ~ list(LETTERS[1:l],stack.loss)
> I
{ A B C D E F G H I J K L M N O P Q R S T U }
```

Unlike S-PLUS objects, subscripts of indexed SIMPLE objects must usually be explicitly expressed, except when the object is displayed, or when values are assigned to it (as shown above for StackLoss). For example, adding 1 to the SIMPLE parameter StackLoss requires different syntax than adding 1 to the S-PLUS vector stack.loss:

```
> StackLoss[i] + 1
  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U
 43 38 38 29 19 19 20 21 16 15 15 14 12 13  9  8  9  9 10 16 16
attr(, "indexes"):
[1] "i"
> stack.loss + 1
 [1] 43 38 38 29 19 19 20 21 16 15 15 14 12 13  9  8  9 10 16 16
```

Objects of class *Set* may have more than one dimension, depending on the dimensionality of the objects it is intended to define. For example, to define a SIMPLE parameter StackX whose value is that same as that of the S-PLUS matrix stack.x,

```
> Observations <- Set()
> Measurements <- Set()
> j <- Element(set=Observations)
> k <- Element(set=Measurements)
> StackX <- Parameter(index=dprod(Observations,Measurements))
> StackX ~ stack.x
> StackX
   Air Flow Water Temp Acid Conc.
1       80         27         89
2       80         27         88
          .          .          .
          .          .          .
          .          .          .
20      56         20         82
```

```
21      70        20        91
attr(, "indexes"):
 [1] "*" "*"
```

The parameter is indexed by the set `dprod(Observations, Measurements)`, which is the direct product of the set `Observations` and the set `Measurements`. Either

```
> StackX <- Parameter(index=dprod(j,k))
```

or

```
> StackX <- Parameter(index=Observations Measurements)
```

could also have been used in the above definition.

SIMPLE provides the following operations between sets:

Union: `A | B` Intersection: `A & B` Difference: `A - B` Direct Product: `A * B`

Multidimensional sets may also be defined explicitly, for example if

```
> N <- Set(1:2)
> L <- Set(c("a", "b"))
```

then the set

```
> NL <- Set(c(1, "a", 1, "b", 2, "a", 2, "b"), dim = 2)
```

is the direct product of sets N and L, also denoted by N*L.

3.2 Parameter, Variable, and Expression Objects

Objects of class *Parameter* are those values that are to be treated as constants by the NUOPT solver. Objects of class *Variable* are those whose values are to be determined by solving the model. Objects of class *Expression* are mathematical expressions containing at least one variable. In the constrained linear regression example (3), the matrix X and vector y are parameters, β is a vector of variables and r denotes an expression for the residuals.

Parameters, variables, and expressions can be evaluated in a system that has been defined

in `SIMPLE` through the function `current`. If we solve the `SIMPLE` model defined by function `LregPos` representing (3) with the `stack` data:

```
> sys.LregPos.stack <- System(model=LregPos, stack.x, stack.loss)
> sol.LregPos.stack <- solve(sys.LregPos.stack)
```

then the current value of the residual r can be found by:

```
> current(sys.LregPos.stack,r)
       1         2        3         4          5         6          7         8
17.59946 12.59947 14.14511 8.886675 -0.9813663 -1.047346 -0.1133327 0.8866673
         9        10        11        12        13        14        15
 -2.916397 -3.586491 -3.586503 -4.520523 -6.586494 -5.652488 -7.324605
       16        17        18        19        20        21
-8.324601 -7.390562 -7.390571 -6.456552 -2.152978 -6.111311
attr(, "indexes"):
 [1] "i"
```

Similarly, for current value of $X\beta$,

```
> current(sys.LregPos.stack,Sum(X[i,j]*beta[j],j))
       1        2        3        4        5        6        7        8
24.40054 24.40053 22.85489 19.11332 18.98137 19.04735 19.11333 19.11333
      9       10       11       12       13       14       15       16       17
17.9164 17.58649 17.5865 17.52052 17.58649 17.65249 15.32461 15.3246 15.39056
      18       19       20       21
15.39057 15.45655 17.15298 21.11131
attr(, "indexes"):
 [1] "i"
```

Initial values of parameters, variables, and expressions can similarly be recovered through the function `init`:

```
> init(sys.LregPos.stack,r)
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
42 37 37 28 18 18 19 20 15 14 14 13 11 12  8  7  8  8  9 15 15
```

```
attr(, "indexes"):
 [1] "i"
> init(sys.LregPos.stack,beta)
Air Flow Water Temp Acid Conc.
       0         0         0
attr(, "indexes"):
 [1] "j"
```

The default initial value of both parameters and variables is zero.

3.2.1 Arithmetic Operations in SIMPLE

S-PLUS math functions are supported in SIMPLE with the following exceptions:

cummax, cumprod, gamma, lgamma, round, signif, all, any, sum, prod, range

Functions sum and prod are replaced by Sum and Prod for which the usage is shown in Table 3.2.1.

Mathematical Expression	SIMPLE Expression
$\sum_{i \in I} f(i)$	Sum(F[i],i,i<I)
$\sum_{i=n}^{m} f(i)$	Sum(F[i],i,i>=n,i<=m)
$\sum_{i \in I, j \in J} g(i,j)$	Sum(G[i,j],i,j,i<I,j<J)
$\sum_{i \in I} \sum_{j \in J} g(i,j)$	Su m(Sum(G[i,j],j,j<I),i,i<I)

SIMPLE definitions:

```
> I <- Set(); J <- Set(); i <- Element(set=I); j <- Element(set=J)
> F <- Expression(index=I); G <- Expression(index=I*J)
```

Table 2: Sum notation in SIMPLE; Prod is analogous. In the SIMPLE definitions, the form of the expressions for F and G remains to be specified.

3.3 Integer Variables

For mixed integer programming problems, users can specify integer variables in their model definitions. An integer variable is defined in `SIMPLE` by the function `IntegerVarible`. The class of objects *IntegerVariable* created by this function is a subclass of class *Variable*.

The following types of integer variables may be specified:

binary: takes on only either 0 or 1 as values
semicontinuous: either the integer 0 or any real number ≥ 1
partial: takes on nonnegative integer values in a prespecified range that includes 0, and real values outside that range.

The following functions are provided for defining models with integer variables in `SIMPLE`:

`priority`: branch priority of an integer variable
`upPC`, `downPC`: round up/down pseudo cost of an integer variable
`direction`: branch direction of an integer variable

3.3.1 The Knapsack Problem

The knapsack problem provides an example of the use of integer variables in `SIMPLE`. In this problem we are given a `knapsack' of fixed capacity C, and a set of objects x_i each with a fixed size s_i and value v_i. A mathematical statement of the problem is:

$$\begin{aligned} &\min imize_x \quad v^T x \\ &\text{subject to} \quad x \in \{0,1\} \\ &\qquad\qquad s^T x \leq C \end{aligned} \tag{5}$$

The following function contains `SIMPLE` statements defining the knapsack model:

```
> Knapsack
function(value, size, Capacity)
{
       I <- Set()
       i <- Element(set = I)
       x <- IntegerVariable(index = i, type = "binary")
```

```
        v <- Parameter(value, index = i)
        s <- Parameter(size, index = i)
        obj <- Objective(type = maximize)
        obj ~ Sum(v[i] * x[i], i)      ## Objective
        Sum(s[i] * x[i], i) <= Capacity ## Constraint
}
```

We use a wrapper function

```
> KnapsackInit
function()
{
        value <- list(1:10, c(42, 12, 45, 5, 2, 61, 89, 32, 47, 18))
        size <- list(1:10, c(39, 13, 68, 15, 10, 20, 31, 15, 41, 16))
        Capacity <- 121
        return(Knapsack(value, size, Capacity))
}
```

to create a sample instance of the problem and solve it with NUOPT. Because NUOPT detect the existence of integer variables NUOPT choose the default method to be “simplex” capable of handling Mixed Integer Linear Programming (MILP) models. MILP is very difficult class of problems, and note that it may take hours to compute optimal solution of MILP with several hundred of integer variables.

```
> sys.KnapsackInit <- System(KnapsackInit)
> sol.KnapsackInit <- solve(sys.KnapsackInit,trace=F)

> sol.KnapsackInit
$variables:
$variables$x:
  1 2 3 4 5 6 7 8 9 10
  1 0 0 0 0 1 1 1 0  1
attr($variables$x, "indexes"):
 [1] "i"

$objective:
 [1] 242
```

```
$counter:
iters fevals vars
    0     0   10

$termination:
tolerance residual
    1e-08       0

$elapsed.time:
 [1] 0.02000046
```

3.4 Constraints and Conditions

3.4.1 Constraint Objects

Constraints are mathematical relations that a solution of an optimzation problem are required to obey. Examples are the bound constraints $\beta > 0$ in the linear regression model (3), expressed in `SIMPLE` as

```
beta[j] >= 0
```

and the capacity constraint $s^T x \leq C$ in the knapsack problem (5), expressed as

```
Sum(s[i] * x[i], i) <= Capacity
```

in `SIMPLE`. Constraints can also be defined with the `Constraint` function. For the above examples alternative specifications using `Constraint` would be:

```
bounds <- Constraint(index = j)
bounds[j] ~ beta[j] >= 0
```

and

```
cap <- Constraint()
cap ~ Sum(s[i] * x[i], i) <= Capacity
```

This feature is useful for analysis of model solutions, and also when modifying constraints in a model that has already been compiled for NUOPT through the function System (see section 4).

3.4.2 Condition Objects

While constraints in SIMPLE are defined by expressions and thus include variables, conditions apply to set elements. A condition is a logical expression for deciding whether or not to execute a model statement. In a *Condition* object, < is used to denote set inclusion ($\in$) in conditions of the form Element < Set.

There are two sorts of conditions: global and local. A global condition affects all subsequent model statements until another related global condition occurs. As an example consider the a linear regression model similar to (3):

```
> new.model()
> I <- Set(1:5)
> i <- Element(set=I)
> x <- Variable(index=i)
> i > 3      # global condition
> x[i] <= 1
> x[i] >= 0
> i < 3      # global condition
> x[i] >= -1
> x[i] <= 0
> i == 3     # global condition
> x[i] >= -1
> x[i] <= 1
> (i <= 0) %&&% (i >= 0)  # global condition removing restrictions on i
> f <- Objective(type = "maximize")
> f ~ Sum(x[i],i)
> sys <- System()
Evaluating
   1.  i>3
   2.  x[i]<=1
   3.  x[i]>=0
   4.  i<3
```

```
    5.   x[i]>=-1
    6.   x[i]<=0
    7.   i==3
    8.   x[i]>=-1
    9.   x[i]<=1
   10.   (i<=0)&&(i>=0)
   11.   f~Sum(x[i],i)
Expanding
(1/7).....(2/7).....(3/7).....(4/7).....(5/7).....(6/7).....(7/7)ok!
> sys
1-1 :  x[4] <= 1
1-2 :  x[5] <= 1

2-1 :  x[4] >= 0
2-2 :  x[5] >= 0

3-1 :  x[1] >= -1
3-2 :  x[2] >= -1

4-1 :  x[1] <= 0
4-2 :  x[2] <= 0

5-1 :  x[3] >= -1

6-1 :  x[3] <= 1

f<objective>: x[1]+x[2]+x[3]+x[4]+x[5] (maximize)
```

Local conditions occur in conjunction with subscripting:

```
> new.model()
> I <- Set()
> i <- Element(set=I)
> x <- Parameter(list(1:7,-3:3),index=I)
> x[i, i > 4]
 5 6 7
```

```
 1 2 3
attr(, "indexes"):
 [1] "i"
```

as well as within the Sum and Prod functions:

```
> new.model()
> I <- Set()
> i <- Element(set=I)
> x <- Parameter(list(1:7,-3:3),index=I)
> x

   1  2   3 4 5 6 7
  -3 -2 -1 0 1 2 3
attr(, "indexes"):
 [1] "*"
> Sum(x[i], i, i > 4)
 [1] 6
> Prod(x[i], i, i!=4)
 [1] -36
```

3.4.3 Conditional Expressions

In some models we may want expressions to vary according to the values of their variables. The function `ife(cons, e1, e2)` defines an expression that is either `e1` or `e2` depending on the value of a constraint (`cons`). If `cons` holds for the current values of the variables, then the function `ife(cons, e1, e2)` returns `e1`, otherwise `e2`. For example,

```
> new.model()
> I <- Set()
> i <- Element(set=I)
> Abs <- Expression(index=i)
> x <- Variable(index=i)
> x[i] ~ list(letters[1:19],-9:9)
> Abs[i] ~ ife(x[i]>=0, x[i], -x[i])
> Abs
```

```
 a b c d e f g h i j k l m n o p q r s
 9 8 7 6 5 4 3 2 1 0 1 2 3 4 5 6 7 8 9
attr(, "indexes"):
 [1] "i"
```

You can write down some difficult models with discontinuous function by using ife(). But this do not always mean NUOPT can solve the model. This is why the optimization methods implemented in NUOPT are all based on algorithms that assume the continuity of constraints and objective function and its derivatives around the local optimal solution. If your model with ife() violated this condition, NUOPT may not give the desired result.

There known many workaround to express such a conditional expression by using continuous variables (see for example [14] for details). It is strongly recommended to examine if you can write your model before writing down the model using ife().

3.5 System and Solution Objects

`SIMPLE` models must be expanded into *System* objects through the function `System` in order to be solved by `NUOPT`. As a simple example, consider the problem of maximizing the sum of two nonnegative variables that lie inside the unit circle:

$$\max imize_{x,y} \quad x + y \tag{6}$$

$$\text{subject to} \quad x^2 + y^2 \le 1$$
$$x \ge 0; \quad y \ge 0$$

In `SIMPLE`, this model can be expressed as:

```
> new.model()
> I <- Set(1:2)
> i <- Element(set=I)
> x <- Variable(index = I)
> unitcirc <- Constraint()
> unitcirc ~ Sum(x[i]*x[i], i) <= 1
> x[i] >= 0
> f <- Objective("maximize")
> f ~ Sum(x[i],i)
> x[i] ~ 1            # initial values
```

Now we expand the model into a system for solution by NUOPT:

```
> sys <- System()
Evaluating
   1.  unitcirc~Sum(x[i]*x[i],i)<=1
   2.  x[i]>=0
   3.  f~Sum(x[i],i)
   4.  x[i]~1
Expanding  (1/3)(2/3)(3/3)ok!
```

and display the expanded model:

```
> sys
1-1 (unitcirc): x[1]*x[1]+x[2]*x[2] <= 1

2-1 :  x[1] >= 0
2-2 :  x[2] >= 0

f<objective>: x[1]+x[2] (maximize)
```

Next, we solve the expanded model:

```
> sol <- solve(sys)
NUOPT 4.6.0, Copyright (C) 1991-2001 Mathematical Systems Inc.
NUMBER_OF_VARIABLES                                  2
NUMBER_OF_FUNCTIONS                                  2
PROBLEM_TYPE                                MAXIMIZATION
METHOD                                     TRUST_REGION
<preprocess begin>.........<preprocess end>
<iteration begin>
   res=1.9e+001 .... 2.5e-008
<iteration end>
STATUS                                         OPTIMAL
VALUE_OF_OBJECTIVE                          1.414213548
ITERATION_COUNT                                      5
FUNC_EVAL_COUNT                                      8
```

```
FACTORIZATION_COUNT                                  8
RESIDUAL                              2.459224485e-008
ELAPSED_TIME(sec.)                                0.68
```

The solution object is a list with several components:

```
> sol
$variables:
$x:
        1        2
 0.7071068 0.7071068
attr(, "indexes"):
[1] "*"


$objective:
[1] 1.414214

$counter:
 iters fevals vars
     5      8    2

$termination:
 tolerance     residual
  1.7e-006 2.459224e-008

$elapsed.time:
[1] 0.6809999

$errorCode:
[1] 0
```

`errorCode` is the error number of returned from NUOPT(if no error occurred it becomes zero). This is useful when calling the solve() method in some user defined program, checking the result of the optimization and modify the process.

There are more components to the solution object corresponding to lower bounds, upper

bounds, and dual variables (all components can be viewed by applying the function `unclass`). Separate functions `lowerB`, `upperB`, and dual are provided to extract these quantities:

```
> lowerB(sys,x)
 1 2
 0 0
attr(, "indexes"):
[1] "*"

> upperB(sys,x)
   1   2
 Inf Inf
attr(, "indexes"):
[1] "*"

> dual(sys,x)
           1           2
 4.333893e-007 4.333893e-007
attr(, "indexes"):
[1] "*"

> dual(sys,unitcirc)
[1] -0.7071066
```

Current and inital values of variables and expressions are also available through functions `current and init.`

```
> init(sys,x)
 1 2
 1 1
attr(, "indexes"):
[1] "*"
> current(sys,Prod(x[i],i))
```

[1] 0.5000004

4 Modifying Model Systems

NUOPT for S-PLUS provides several ways of modify *System* objects for their analysis.

4.1 Adding and Deleting Constraints

It is possible to add and delete constraints in an existing system through functions `add.con`, `delete.con`, and `restore.con`. We give an example using the following formulation of the linear regression problem with positivity constraints (4).

```
> LregPosAlt
function(X, y)
{
       Res <- Set()
       Var <- Set()
       i <- Element(set = Res)
       j <- Element(set = Var)
       nres <- length(y)
       nvar <- ncol(X)
       y <- Parameter(list(1:nres, y), index = i)
       X <- Parameter(X, index = dprod(i, j))
       beta <- Variable(index = j)
       B <- Constraint(index = j)
       B[j] ~ beta[j] >= 0
       r <- Variable(index = i)
       R <- Constraint(index = i)
       R[i] ~ r[i] == y[i] - Sum(X[i, j] * beta[j], j)
       obj <- Objective(type = "minimize")
       obj ~ Sum(r[i] * r[i], i)
}
```

First we form the system using the `stack` data:

```
> sys.LregPosAlt.stack <- System(LregPosAlt, stack.x, stack.loss)
```

```
1-1 (B[Acid Conc.]): beta[Acid Conc.] >= 0
1-2 (B[Air Flow]): beta[Air Flow] >= 0
1-3 (B[Water Temp]): beta[Water Temp] >= 0

2-1 (R[1]): -80 beta[Air Flow]-27 beta[Water Temp]- ...
2-2 (R[2]): -80 beta[Air Flow]-27 beta[Water Temp]- ...
    .
    .
    .
2-20 (R[20]): -56 beta[Air Flow]-20 beta[Water Temp]- ...
2-21 (R[21]): -70 beta[Air Flow]-20 beta[Water Temp]- ...

obj<objective>: (r[1])*(r[1])+(r[2])*(r[2])+(r[3])*(r[3])+...
```

Now the bound constraints:

```
> delete.con(sys.LregPosAlt.stack,1,1:3)
> sys.LregPosAlt.stack

1-1 (B[Acid Conc.]): beta[Acid Conc.] >= 0  <<<deleted>>>
1-2 (B[Air Flow]): beta[Air Flow] >= 0  <<<deleted>>>
1-3 (B[Water Temp]): beta[Water Temp] >= 0  <<<deleted>>>

2-1 (R[1]): -80*beta[Air Flow]-27*beta[Water Temp]- ...
2-2 (R[2]): -80*beta[Air Flow]-27*beta[Water Temp]- ...
    .
    .
    .
2-20 (R[20]): -56*beta[Air Flow]-20*beta[Water Temp]- ...
2-21 (R[21]): -70*beta[Air Flow]-20*beta[Water Temp]- ...

obj<objective>: (r[1])*(r[1])+(r[2])*(r[2])+(r[3])*(r[3])+...

> sol.LregPosAlt.stack <- solve(sys.LregPosAlt.stack)
```

Now restore the bound constraints and delete all but the first 9 residuals and solve:

```
> restore.con(sys.LregPosAlt.stack,1,1:3)
> delete.con(sys.LregPosAlt.stack,2,10:21)
> sys.LregPosAlt.stack
1-1 (B[Acid Conc.]): beta[Acid Conc.] >= 0
1-2 (B[Air Flow]): beta[Air Flow] >= 0
1-3 (B[Water Temp]): beta[Water Temp] >= 0

2-1 (R[1]): -80*beta[Air Flow]-27*beta[Water Temp]- ...
2-2 (R[2]): -80*beta[Air Flow]-27*beta[Water Temp]- ...
    .
    .
    .
2-20 (R[20]): -56*beta[Air Flow]-20*beta[Water Temp]- ... <<<deleted>>>
2-21 (R[21]): -70*beta[Air Flow]-20*beta[Water Temp]- ... <<<deleted>>>


obj<objective>: (r[1])*(r[1])+(r[2])*(r[2])+(r[3])*(r[3])+...
```

Now add a new constraint:

```
> add.con(sys.LregPosAlt.stack, Sum(beta[j],j) == 1)
1-1 (B[Acid Conc.]): beta[Acid Conc.] >= 0
1-2 (B[Air Flow]): beta[Air Flow] >= 0
1-3 (B[Water Temp]): beta[Water Temp] >= 0

2-1 (R[1]): -80*beta[Air Flow]-27*beta[Water Temp]- ...
2-2 (R[2]): -80*beta[Air Flow]-27*beta[Water Temp]- ...
    .
    .
    .
2-20 (R[20]): -56*beta[Air Flow]-20*beta[Water Temp]- ... <<<deleted>>>
2-21 (R[21]): -70*beta[Air Flow]-20*beta[Water Temp]- ... <<<deleted>>>

3-1 :  beta[Air Flow]+beta[Water Temp]+beta[Acid Conc.] == 1

obj<objective>: (r[1])*(r[1])+(r[2])*(r[2])+(r[3])*(r[3])+...
```

```
> solve(sys.LregPosAlt.stack)
```

4.2 Fixing Variables and Changing Parameters

Current values of Variables of the system can be changed, and the system can be resolved with these new values. We can also change some Parameters of a system may be modified by specifying them as "changeable". Another useful tool is to fix variables at specific values. For example,

```
 > initVal                    ## a sample model
 function()
 {
       a <- Set(1:10)
       i <- Element(set = a)
       p <- Parameter(list(1:10, 1:10), index = i, changeable = T)
       x <- Variable(list(1:10, rep(30, 10)), index = i)
       x[i] >= p[i]
       f <- Objective(minimize)
       f ~ Sum(-1/(x[i] - 50), i)
 }

> s1 <- System(initVal)        ## making a System
> sol1 <- solve(s1,f)    ## solve the System
> current(s1,x)               ## result of variable x

      1       2       3       4       5       6       7       8
1.000116 2.000001 3.000034 4.000105 5.000166 6.000199 7.000207 8.000199
      9       10
9.000184 10.00017

attr(, "indexes"):
 [1] "i"
> current(s1,p)"list(1:10,21:30)   ## assign to the changeable Parameter
> sol2 <- solve(s1,f)             ## solve the system once more
> current(s1,x)                   ## the result of x is changed
      1       2       3       4       5       6       7       8
```

```
21.00005 22.00005 23.00004 24.00004 25.00004 26.00003 27.00003 28.00002
       9       10
 29.00002 30.00001
attr(, "indexes"):
[1] "i"

> fix.Variable(s1,x[i,(i<5) %&&% (i>1)])    ## fix a part of x
> current(s1,p)"list(1:10,7:16)           ## changeable Param.
> current(s1,p)
1 2 3  4   5  6  7   8  9  10
7 8 9 10 11 12 13 14 15 16
attr(, "indexes"):
 [1] "i"
> sol3 <- solve(s1,f)                    ## solve one more time
> current(s1,x)                          ## x[2,3,4] are not changed
       1         2         3        4        5         6   7        8
7.000328 22.00005 23.00004 24.00004 11.00008 12.00003 13 14.00001
       9        10
15.00006 16.00018
attr(, "indexes"):
 [1] "i"

> unfix.Variable(s1,x[i,i==2])            ## unfix x[2]
> sol4 <- solve(s1,f)
  > current(s1,x)                         ## x[2] is changed
         1        2        3        4        5         6        7        8
   7.000063 8.000007 23.00004 24.00004 11.00005 12.00005 13.00005 14.00005
         9       10
   15.00004 16.00004
  attr(, "indexes"):
  [1] "i"
  > unfix.Variable(s1,x)                   ## unfix all x
  > sol5 <- solve(s1,f)
  > current(s1,x)                         ## x[3,4] are changed
         1        2        3        4      5       6        7        8
   7.000122 8.000117 9.000026 10.00001 11.0001 12.0001 13.00009 14.00009
```

```
         9        10
  15.00009 16.00008
 attr(, "indexes"):
 [1] "i"
```

4.3 Graph Objects

A graph is used for describing special models like network problems. A graph consists of two sets, namely node (dim=1) and arc (dim=2).

SIMPLE provides a number of functions for manipulating graphs (see Table 4.3) . We can check the behavior of these functions as follows.

Table 4.3: Graph functions

Function	Arguments	Brief Description
Arcs	graph	returns the set of arcs of graph
nodes	graph	returns the set of nodes of graph
input	graph,node	returns all arcs (of graph) incoming to node
output	graph,node	returns all arcs (of graph) outgoing from node
Orig	element	returns the first component of a arc
Dest	element	returns the second component of a arc

```
> new.model()
> ar <- Set(c(1,2,1,3,1,4,2,1,2,5),dim=2)
> g <- Graph(arcs=ar)
> g
      1 2 3 4 5
orig  1 1 1 2 2
dest  2 3 4 1 5
> nodes(g)
{ 2 3 4 1 5 }
> nodes(g)~1:3          ## arcs (1,4) and (2,5) are removed,
> arcs(g)               ## from the def. of graph.
    1 2 3
orig 1 1 2
dest 2 3 1
> ar~c(1,2,1,3,2,3,3,1)
```

```
> g
      1 2 3 4
orig 1 1 2 3
dest 2 3 3 1
> i <- Element(set=nodes(g))
> input(g,i)            ## incoming arcs:
[2] = { 1 2 }          ##   arc  to node "2" is  (1,2)
[3] = { 2 3 1 3 }            ##   arcs to node "3" are  (2,3) and (1,3)
[1] = { 3 1 }               ##   arc  to node "1" is  (3,1)
indexes= i
> output(g,i)              ## outgoing arcs:
[2] = { 2 3 }              ##    arc  from node "2" is  (2,3)
[3] = { 3 1 }              ##    arc  from node "3" is  (3,1)
[1] = { 1 3 1 2 }           ##    arcs from node "1" are (1,3) and (1,2)
indexes= i
> e <- Element(set=arcs(g))
> p <- Parameter(index=i)
> p[i]~i
> p[orig(e)]             ## first component of e, NA: not defined.
   2  3  1
1  1  1 NA
2 NA  2 NA
3 NA NA  3
attr(, "indexes"):
[1] "e"
```

Next, we give an example of a model defined on a graph.

4.3.1 The Minimum-Cost Flow Problem

The minimum cost flow problem finds a set of arc flows of a graph minimizing a linear cost function subject to the constraints that the flows produce a given divergence vector and lie within certain bounds. That is,

minimize: $\sum_{(i,j)\in A} a_{ij} x_{ij}$

constraint: $$\sum_{\{j|(i,j)\in A\}} x_{ij} - \sum_{\{j|(j,i)\in A\}} x_{ji} = s_i, \quad \forall_i \in N$$

$$b_{ij} \le x_{ij} \le c_{ij}, \qquad \forall (i,j) \in A$$

where a_{ij} , b_{ij} and s_i are given scalars.

For simplicity, we assume the bounds to be $b_{ij} = 0$ and $cij = \infty$ for each i, j . The minimum
cost flow problem is described as follows:

```
 > MinCostFlow
 function(cost, snodes)
{
    g <- Graph()
    i <- Element(set = nodes(g))
    e <- Element(set = arcs(g))
    eout <- Element(set = output(g, i))
    ein <- Element(set = input(g, i))
    x <- Variable(index = arcs(g))
    a <- Parameter(cost, index = arcs(g))
    s <- Parameter(snodes, index = nodes(g))
    Sum(x[eout], eout) - Sum(x[ein], ein) == s[i]
    x[e] >= 0
    f <- Objective(minimize)
    f ~ Sum(a[e] * x[e], e)
 }
```

To obtain its solution in SIMPLE, do the following:

```
> a <- list(c(1,2,1,3,2,3),c(2,4,3,4,3,2),c(1,3,3,1,1,1)) ## Cost
> sup <- list(c(1,4),c(1,-1))    ## The supply of nodes
> s5 <- System(MinCostFlow,a,sup)
Evaluating MinCostFlow(a,sup) ... ok!
Expanding  (1/3)(2/3)(3/3)ok!
```

```
> s5
1-1 :  x[2,3]+x[2,4]-x[3,2]-x[1,2] == 0
1-2 :  -x[2,4]-x[3,4] == -1
1-3 :  -x[2,3]+x[3,2]+x[3,4]-x[1,3] == 0
1-4 :  x[1,2]+x[1,3] == 1

2-1 :  x[1,2] >= 0
2-2 :  x[2,4] >= 0
2-3 :  x[1,3] >= 0
2-4 :  x[3,4] >= 0
2-5 :  x[2,3] >= 0
2-6 :  x[3,2] >= 0

f<objective>: x[2,3]+3*x[2,4]+x[3,2]+x[1,2]+x[3,4]+3*x[1,3] (minimize)
> sol5 <- solve(s5,f)
> sol5 <- solve(s5,f)
NUOPT 4.2.2, Copyright (C) 1991-2000 Mathematical Systems Inc.
NUMBER_OF_VARIABLES                                    6
NUMBER_OF_FUNCTIONS                                    5
PROBLEM_TYPE                                  MINIMIZATION
METHOD                                       HIGHER_ORDER
<preprocess begin>.........<preprocess end>
<iteration begin>
   res=1.9e+000 .... 2.7e-007  1.4e-010
<iteration end>
STATUS                                           OPTIMAL
VALUE_OF_OBJECTIVE                            3.000000002
ITERATION_COUNT                                        6
FUNC_EVAL_COUNT                                        8
FACTORIZATION_COUNT                                    7
RESIDUAL                                  1.37376295e-010
ELAPSED_TIME(sec.)                                   0.18
> sol5
$variables:
$variables$x:
           2           3           4
```

```
1 1.000000e+000 6.131224e-010          NA
2          NA 1.000000e+000 6.131224e-010
3 3.070398e-010          NA 1.000000e+000
attr($variables$x, "indexes"):
[1] "*"

$objective:
[1] 3

$counter:
 iters fevals vars
     6      8    6

$termination:
 tolerance      residual
    1e-008 1.373763e-010

$elapsed.time:
[1] 0.18
```

The time for solving MinCostFlow with NUOPT is shown in Table 4.3.1. (Sparc station 20 (64Mbyte memory); SunOS4.1.4, S-PLUS version 3.3)

Table 4.3.1: Time foe Solving MinCostFlow

Data Size	Number of Variables	Number of Eqs.	Time(sec.)
10x10	360	101	8.7E-01
20x20	1520	401	5.4E+00
30x30	3480	901	1.7E+01
50x50	9800	2501	9.5E+01

4.4 Interface Between SIMPLE Objects and S-PLUS Objects

S-PLUS objects (list or array) can be assigned to SIMPLE objects (Set, Variable, Parameter and Expression). Also, as.array and as.list functions can be applied to almost all SIMPLE objects to convert them to S-PLUS objects.

There are basic rules governing the values of the SIMPLE objects. The default initial value of Set is empty which corresponds to NULL in S. The default initial value of Variable and Parameter is zero. All zero-valued components are omitted when transforming to S-PLUS objects.

```
1. Interface between Set and vector and list

> new.model()

> a1 <- Set(1:5)     ## a1 is initialized to be {1,2,3,4,5 }

> a1
{ 1 2 3 4 5 }

> as.list(a1)
[1] 1 2 3 4 5

> i <- Element(set=a1)
> j <- Element(set=a1)

> a2<-Set(list(2:3,list(10:15,c("a",1))),index=i) ## set with one index
> a2

[2] = { 10 11 12 13 14 15 }
[3] = { a 1 }
indexes= I

> as.list(a2)
$indexes:
$indexes$i:
[1] 2 3

$values:
$values[[1]]:
[1] 10 11 12 13 14 15

$values[[2]]:
```

```
[1] "a" "1"

> a3 <- Set(index=dprod(i,j)) ## set with two indexes
> a3 ~ list(c(1,1,2,3,),c(1,2,3,1),list(c("a","b"),31:33,41:42,51:55))
> a3
[1,1] = { a b }
[3,1] = { 51 52 53 54 55 }
[1,2] = { 31 32 33 }
[2,3] = { 41 42 }
indexes= i j

> as.list(a3)       ## the same as unclass(a3)
$indexes:
$indexes$i:
[1] 1 3 1 2

$indexes$j:
[1] 1 1 2 3

$values:
$values[[1]]:
[1] "a" "b"

$values[[2]]:
[1] 51 52 53 54 55

$values[[3]]:
[1] 31 32 33

$values[[4]]:
[1] 41 42
```

2. Interface between `Parameter`, `Variable`, `Expression` and `vector`, `list`, `array`

```
> new.model()
```

```
> a <- Set(1:5)
> i <- Element(set=a)
> j <- Element(set=a)

> p <- Parameter(3.33)    ## p is initialized to 3.33
> p
[1] 3.33

> p1<-Parameter(list(2:3,c(2.2,3.3)),index=i)## parameter with one index
> p1                    ## the same as  as.array(p1) and unclass(p1)
  1   2   3 4 5
  0 2.2 3.3 0 0
attr(, "indexes"):
 [1] "i"

> as.list(p1)       ## p1 is transformed to a list corresponding to:
$indexes:           ## p1[2] = 2.2; p1[3] = 3.3;
$indexes$i:         ## where p1[1]=p1[4]=p1[5]=0 are removed
 [1] 2 3

$values:
 [1] 2.2 3.3
> p1[i]+1
1   2   3 4 5
1 3.2 4.3 1 1
attr(, "indexes"):    ## here, indexes is to specify  what
 [1] "i"              ## the values depend on.

> p2 <- Parameter(index=dprod(i,j))    ## parameter with two indexes
> p2 " list(c(1,2,3,1),c(3,2,1,1),c(1.5,2.5,3.5,4.5))
> p2
    1    2    3 4 5
1 4.5 0.0 1.5 0 0
2 0.0 2.5 0.0 0 0
3 3.5 0.0 0.0 0 0
4 0.0 0.0 0.0 0 0
```

```
5 0.0 0.0 0.0 0 0
attr(, "indexes"):
 [1] "i" "j"

> p3 <- Parameter(matrix(1:6,2,3),index=dprod(a,a))
> p3[j,i]
  1 2 3 4 5
1 1 3 5 0 0
2 2 4 6 0 0
3 0 0 0 0 0
4 0 0 0 0 0
5 0 0 0 0 0
attr(, "indexes"):
 [1] "j" "i"

> dt <- array(2:7,c(2,3),dimnames=list(c("a","b"),c(1,2,3)))
> p4 <- Parameter(dt,index=dprod(a,a))
> p4[i,j]
  1 2 3 4 5 a b
1 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0
a 2 4 6 0 0 0 0
b 3 5 7 0 0 0 0
attr(, "indexes"):
[1] "i" "j"
> as.list(p4[i,j])
$indexes:
$indexes$i:
[1] "a" "b" "a" "b" "a" "b"

$indexes$j:
[1] 1 1 2 2 3 3
```

```
$values:
[1] 2 3 4 5 6 7
```

`Variable` and `Expression` have the same correspondence to `S-PLUS` objects as `Parameter`.

3. Interface between `Parameter`, `Variable`, `Expression` and data frames.

Data frames are interpreted as arrays in SIMPLE. In order to assign data frame `d` to a `SIMPLE` object with only one index, it must first be transformed to a vector in `S-PLUS` (`unlist(as.vector(d))`).

5 The NUOPT Solver

The SIMPLE modeling language is combined with an optimizer called NUOPT, which is invoked by the solve command. When solve is invoked, NUOPT prints a trace showing the progress and status of the calculations[2], and the solution is accessible via SIMPLE objects (Variable, Expression and so on). SIMPLE provides a way to control NUOPT by setting parameters through
the function nuopt.options.

In this section, we explain the following topics:

- output messages from NUOPT
- optimization methods in NUOPT
- parameters to control NUOPT

5.1 Output messages from NUOPT

When the solve command is invoked with the argument trace set to TRUE (the default), the output of NUOPT will look like the following:

```
NUOPT 4.2.2, Copyright (C) 1991-2000 Mathematical Systems Inc.
NUMBER_OF_VARIABLES                                  10
NUMBER_OF_FUNCTIONS                                   1
PROBLEM_TYPE                                 MINIMIZATION
METHOD                                      TRUST_REGION
<preprocess begin>.........<preprocess end>
<iteration begin>
   res=1.0e+000 .... 1.3e-004 . 2.5e-009
<iteration end>
STATUS                                          OPTIMAL
VALUE_OF_OBJECTIVE                           -45.77846971
ITERATION_COUNT                                       7
FUNC_EVAL_COUNT                                       9
FACTORIZATION_COUNT                                   8
```

[2]This is suppressed when the trace argument of solve is set to FALSE.

```
RESIDUAL                                  2.494695073e-009
ELAPSED_TIME(sec.)                                  0.18
```

The first three lines indicate the version of NUOPT, and the number of variables/functions. Both the constraints and the objective function are counted in the NUMBER_OF_FUNCTIONS. The next two lines show the problem type (minimization/maximization) and optimization method applied. In this case NUOPT reports that a trust region method was applied (TRUST_REGION). See Section 5.2 about the various methods available in NUOPT and their choice. The next four lines show the progress of the preprocessing and optimization phases. When an interior point method is used for optimization, the dots on the right of res indicate the progress of the iteration (one dot per iteration); also shown are successive values of the normalized residual of the optimality condition. The next line gives the status. OPTIMAL means a successful termination of the optimization method. If an error occurs (see section 7.3), the status is indicated as NON_OPTIMAL and an error message is displayed:

```
STATUS                                    NON_OPTIMAL
ERROR_TYPE          <<NUOPT 15>> simplex method misapplied to NLP.
```

A summary report of the interior point method follows in the next line, including the optimal objective function value, interior point iteration count, function evaluation count, factorization count, and final residual of optimality condition. The last line shows the over all elapsed time for the execution of NUOPT.

5.1.1 Messages from the Simplex/Active Set Method for LP/QP

When using the simplex method option for linear programming (the METHOD line indicates SIMPLEX in this case), the calculation progress report becomes:

```
<iteration begin>
      ......1.....................2
<iteration end>
```

The dots indicate the progress of the simplex method iteration (one dot per some fixed number of iterations), the letter 1 and 2 indicates the end of phase I (search for feasible solution) and phase 2 (search for optimal solution) respectively.

When the crossover switch (see section 5.3.1) is activated, NUOPT starts a simplex method

iteration from the result of an interior point method that finds a basic feasible solution. (about a "basic" solution see [2] for example). In this case a progress report of the simplex method follows that of the interior point method:

```
<preprocess begin>.........<preprocess end>
<iteration begin>
   res=1.7e+00 .... 4.6e-02 .... 2.9e-04 .... 8.2e-07 .... 5.6e-08 .
      7.5e-09
<iteration end>
<iteration begin>
      ...1...2
<iteration end>
```

When the simplex method is invoked, there is an additional line

```
SIMPLEX_PIVOT_COUNT                                (number)
```

to report the simplex pivot count. If the model contains an integer variable, a branch-and-bound enumeration scheme is invoked automatically. In this case the calculation progress report looks like:

```
<iteration begin>
     ..1..2.B.....I.I.............I..II...
<iteration end>
```

The dots and letter `p` indicate the progress of the simplex method iteration to solve the first relaxation. The letter `B` indicates the start of branch-and-bound enumeration, followed by dots that indicate the progress of the enumeration process. The letter I indicates the update of current best integer feasible solution. When you set the option:

```
> nuopt.options(mipfeasout="on") # report each feasible sol
                                          when obtained.
```

NUOPT report the objective value of the solution each time they are obtained. With obtained time and the current number of simplex pivoting(#piv) and enumeration node(#prob) as:

```
<iteration begin>
      1.2BI
feasible_sol #1        291 ( found at   0.1 sec. #piv=10:#prob=6 )
I
```

```
feasible_sol #2        323 ( found at   0.1 sec. #piv=13:#prob=8 )
I
feasible_sol #3        366 ( found at   0.1 sec. #piv=15:#prob=9 )

<iteration end>
```

In this case the following lines appear in the summary report:

```
PARTIAL_PROBLEM_COUNT                              (number)
DUAL_SIMPLEX_PIVOT_COUNT                           (number)
```

The first line gives the number of branch-and-bound enumeration nodes (subproblems) examined to find the optimal solution and prove its optimality. The next line reports total simplex pivot count required to solve sequence of subproblems (in this case we use the dual simplex method).

5.2 Choosing an Optimization Method

The various optimization methods included in NUOPT have been discussed in the introduction (section 1). The METHOD line in the output from NUOPT reports the optimization method applied,
which

higher order method for LP	HIGHER_ORDER
simplex method for LP and MIP	SIMPLEX
trust region method for NLP	TRUST_REGION
line search method for CP	LINE_SEARCH
BFGS method for NLP	BFGS_LINE_SEARCH
Active set method for CQP	ACTIVE_SET_QP

User can specify the method through the function output.options; otherwise NUOPT will make a suitable default choice. For example, a line search method can be applied to a model by typing

```
> nuopt.options(method="line") # explicitly choose line search method
```

The next sections provide a brief explanation of the characteristics and scope of the opti-

mization methods within NUOPT. They also explain NUOPT's actions when automatically selecting
a method, and give general guidelines as to when it may be appropriate to override NUOPT's choices.

The next sections provide a brief explanation of the characteristics and scope of the optimization methods within NUOPT, They also explain NUOPT's actions when automatically selecting
a method, and give general guidelines as to when it may be appropriate to override NUOPT's choices.

5.2.1 Higher-order interior point method for LP

A higher-order interior point method is available for linear programming (LP). It uses the `higher-order' correction technique proposed by Mehrotra[13] and Gondzio[6]. In comparison to more general interior point methods applicable to convex programming (such as the line search method in NUOPT), our experiments show that this method can be nearly 2-3 times faster for large-scale problems. This method is the default choice in NUOPT when the model is linear with no integer variables.

To explicitly select the higher-order method, type

```
> nuopt.options(method = "higher")
```

You can force NUOPT to apply the higher-order method to integer-programming models. However, in this case, the integrality conditions are ignored, and NUOPT will issue an error message:

```
<<NUOPT 14>> integrality is violated.
```

The higher-order method cannot be applied to nonlinear problems as indicated by the error message:

```
<<NUOPT 28>> higher-order method is only for LP.
```

Because it is an interior point method, the solution obtained by the higher-order method is not guaranteed to be a 'basic' solution. If the solution is required to be basic, select the simplex method (see section 5.2.2) by setting

```
> nuopt.options( method = "simplex" )
```

Another way to obtain a basic feasible solution is to activate the crossover switch, by typing

```
> nuopt.options(crossover = "on")
```

With this switch on, NUOPT starts the simplex method iteration from the result of an interior point (in this case the higher-order method) to construct a basic feasible solution. Our experiments show that for large-scale LP (with thousands or more variables), the higher-order method with crossover is the most efficient strategy for obtaining a basic feasible solution.

5.2.2 Simplex method for LP and MILP

The simplex method is a classical method for solving linear programming (LP) models. Simplex method of NUOPT is linked with a branch-and-bound enumeration scheme, and capable of handling mixed linear integer programming (MIP) models. When this method is selected, NUOPT automatically invokes the branch-and-bound enumeration scheme to find a globally optimal solution for MIP. The dual simplex method is used to solve the subproblems that occur in the enumeration process.

Among other methods currently incorporated in NUOPT, the simplex and active set method are capable of handling integer variables. Therefore they are the default choice in NUOPT for MIP models[3]. For MILP, NUOPT's default choice is simplex, for LP the higher-order method (see section 5.2.1).

To force to select the simplex method for LP, type:

```
> nuopt.options( method = "simplex")
```

The simplex method is effective when a basic feasible solution is required or the problem instance

is relatively small (up to hundreds of variables). Our experiments show that the simplex method solves small LP instances (up to hundreds of variables) faster than the higher-order method.

The simplex method cannot handle nonlinear problems. If the simplex method is selected

[3]Simplex method is for Mixed integer linear programming models, active set method for Mixed integer Quadratic programming models. Because none of these methods can handle general nonlinear models, NUOPT cannot handle nonlinear models with integer variables

for a nonlinear problem, NUOPT will reject it with the error message:

```
<<NUOPT 15>> simplex/asqp method misapplied to NLP.
```

5.2.3 line search method for CP

The line search method [9] is an interior point method capable of handling convex programming (CP) models. Convex programming is a class of optimization models in which a convex objective
function is minimized (or a concave objective function is maximized), and whose feasible region is also convex. Quadratic programming models fall into this category if the Hessian of the minimized (maximized) objective function is positive (negative) definite.

For nonlinear models, NUOPT's default choice is to use a trust region method (see section 5.2.4) rather than a line search method. The trust region method will work, but our experiments show that the line search method is about 1.5-2 times faster than the trust region method for convex programming models. NUOPT cannot detect convexity in nonlinear models, but if the problem is known to be convex the line search method can be selected, by typing

```
> nuopt.options( method = "line" )
```

The line search method cannot handle integrality constraints on variables.

5.2.4 trust region method for NLP

The trust region method [11] is an interior point method capable of handling general nonlinear programming (NLP) models. By default, for NLP models, NUOPT chooses the trust region method since it is reliable in all cases. This choice is not always the best when the model is convex, in which case the line search method should be selected (see section 5.2.3).

To explicitly select the trust region method for solving, type

```
> nuopt.options(method = "trust")
```

The trust region method cannot handle integrality constraint of variables.

5.2.5 BFGS method for NLP

The BFGS method is an interior point method capable of handling general NLP models. The solution framework is the same as that of the line search method (see section 5.2.3) for convex programming, but general NLP models can be solved by incorporating a quasi- Newton method (BFGS formula) that provides a positive-definite approximation to the Hessian of the Lagrangian function.

This method is not efficient for large-scale problems, because the Hessian matrix in the quasi-Newton method is dense and has memory requirements of the order of the square of the number of variables. We recommend the trust region method for large-scale NLP m odels.

Our experiments show that the BFGS method can solve small and difficult NLP models in a more stable manner than the trust region method. For small NLP models (up to 50 to 100 variables) in which the trust region method (NUOPT's default choice) takes too many iterations or fails, try the BFGS method by typing

```
> nuopt.options(method = "bfgs")
```

The BFGS method cannot handle integrality constraints on variables.

5.2.6 Active Set method for CQP and MIQP

Active Set method is a classical method to solve convex QP models that is akin to the simplex method. This is much faster than “line” (IPM based on line search) for special type of problems. This method shows clear advantage over “line” for problems such that its hessian matrix is dense (almost all of the element is non zero), and has small number of constraints. One of the examples is a portfolio optimization problem (Markowitz model) with relatively small investment constraints.

Table 5.2 comparison of “line” and “asqp” for portfolio optimization models
(Machine:Pentium400MHz,256Mbytes of memory)

# of assets	*Line search (“line”)*	*Active set method (“asqp”)*
100	0.4sec.	0.1sec.
200	2.0sec.	0.2sec.
300	6.0sec.	0.5sec.
400	14.4sec.	0.9sec.
500	24.0sec.	1.4sec.
600	42.6sec.	2.1sec.

Table 5.2 shows the comparison of “line” and “asqp” for a plane markowitz type of portfolio

optimization model as below.

$$\text{minimize} \sum_{i,j \in Brand} Q_{ij} x_i x_j ,$$

s.t.

$$\sum_{j \in Brand} x_j = 1,$$

$$\sum_{j \in Brand} \bar{r}_j x_j \geq r_{\min} ,$$

$$x_j \geq 0.\ j \in Asset$$

Q_{ij} : co-variance matrix (a dense matrix)

x_j : allocation for asset j

$\bar{r}_j$: a mean return of asset j

But for the problems with large number of constraints as many as variables, active set QP loses its advantage, and interior point method ("line") is generally faster.

Active set method is exclusively for convex QP problems. It cannot treat general nonlinear problems. If the active set method is selected for a nonlinear problem, `NUOPT` will reject it with the error message:

```
<<NUOPT 15>> simplex/asqp method misapplied to NLP.
```

Among other methods currently incorporated in `NUOPT`, the simplex and active set method are capable of handling integer variables. Therefore they are the default choice in `NUOPT` for MIP models[4]. For MIQP, `NUOPT`'s default choice is active set method, for QP the trust region method[5] (see section 5.2.4).

5.3 Setting parameters

[4]Simplex method is for Mixed integer linear programming models, active set method for Mixed integer Quadratic programming models. Because none of these method can handle general nonlinear models, `NUOPT` cannot handle nonlinear models with integer variables

[5] When you are using `NUOPT` through `solveQP` interface, line search method (see section 5.2.3) is the default choice.

It is possible to specify parameters to control the solution process of NUOPT. Although the default values are carefully chosen so that they will work in many cases, some manual tuning may be helpful. Table 5.3 lists the tuning parameters.

Some parameters are optimization-method specific. They fall into three categories:

1.Parameters for interior point method (IPM) variants
2.Parameters for the simplex method and branch-and-bound enumeration
3.General parameters (valid for all methods)

The various parameters in each category are explained in the following sections.

Table 5.3 Options for the solver NUOPT

Name	Options	Default	Meaning
method	"auto", "line", "trust", "bfgs", "simplex", "asqp"	"auto" (default choice)	optimization method:
maxitn	Integer	150	IPM iteration limit
eps	positive num.	1.4e-6(for NLP) 1.0e-8 (for LP)	IPM convergence criteria
crossover	"off" "on"	"off"	Crossover switch
mipfeasout	"off" "on"	"off"	Show the best feasible solution of MIP when obtained.
tolx	positive num.	1.0e-8	primal feasibility tolerance
told	positive num.	1.0e-6	dual feasibility tolerance
epsint	positive num.	1.0e-4	integer feasibility tolerance
maxtim	integer	-1 (no limit)	Optimization time limit (in seconds)
maxnod	integer	-1 (no limit)	branch-and-bound node number limit

cutoff	number	1.0e50 (not set)	branch-and-bound cutoff value
addToCutoff	positive num.	0	branch-and-bound cutoff value margin
maxintsol	Positive num.	-1 (not set)	The number of maximal feasible solution of MIP before stopping.
scaling	"off" "on"	"off"	Scaling switch

5.3.1 Parameters for IPM variants

The parameters described in this section are valid for all of the interior point methods (IPM) incorporated in NUOPT, including:

- higher-order method for linear programming
- trust region method for general nonlinear programming
- line search method for convex programming
- BFGS method for nonlinear programming (small- to medium- scale problems)

and not valid for:

- simplex method for linear and integer programming
- active set method for quadratic and integer quadratic programming

These parameters are specific to interior point methods, and are explained below;

1.IPM iteration limit

```
> nuopt.options(maxitn=150)
```

This parameter gives an upper limit on the number of IPM iterations. If this limit is reached before convergence is achieved, NUOPT terminates with the message:

```
<<NUOPT 10>> IPM iteration limit exceeded.
```

The default value is 150. In our experience from numerical tests, interior point iterations can usually be regarded as stalled after more than 100 iterations.

2.IPM convergence tolerance

```
> nuopt.options(eps=1.4e-6)
```

This parameter gives the convergence tolerance for interior point methods. Interior point methods are terminated successfully when the normalized optimality condition residual Yamashita[9] drops below this value. As shown in Table 2, each default value depends on the problem type[6].

Table 2: Default values of convergence tolerance

model type	default value
LP	1.0×10^{-8}
NLP	1.4×10^{-6} $\cong\sqrt{\varepsilon_{mch}}\cdot 10^{2}$

3.Crossover switch

```
> nuopt.options(crossover="off")
```

This parameter can take string value "on" or "off". With this parameter set to "on", NUOPT starts the simplex method from the solution of the IPM iteration to construct a basic feasible solution. This parameter can only be used for LP. Setting this switch to "on" for solving NLP models, will result in the following error message:

```
<<NUOPT 15>> simplex method misapplied to NLP.
```

5.3.2 Parameters for the simplex and active set method

This section deals with the parameters for solving LP, QP,MILP,MIQP with the method

- simplex method for LP and MILP
- active set method for QP and MIQP

[6] ε_{mch} : The relative machine precision in S-PLUS, it is .Machine$double.eps.

These parameters are related to the simplex algorithm and are `not` valid for IPM methods such as:

- higher-order method for linear programming
- trust region method for general non-linear programming
- line search method for convex programming
- BFGS method for small non-linear programming

The following is a list of the relevant parameters; those designated "MIP only" are valid exclusively for the bound enumeration scheme invoked when solving MILP or MIQP by the simplex or active set method.

1.primal feasibility tolerance

```
> nuopt.options(tolx = 1.0e-8)
```

This parameter gives the allowed relative error in the feasibility of primal variables. Our experiments show that the default value (10^{-8}) works well for many real-life LP applications.

2.dual feasibility tolerance

```
> nuopt.options(told = 1.0e-6)
```

This parameter gives the allowed relative error in the feasibility of dual variables, or reduced costs (shadow prices). Our experiments show the default value (10^{-6}) works well for many real-life LP applications.

3.integer feasibility tolerance (MIP only)

```
> nuopt.options(epsint = 1.0e-4)
```

This parameter gives a tolerance for integrality conditions on integer variables in MILP and MIQP. If these values are too small, they can cause numerical difficulties and inefficiencies. Values that are too large can cause the computation to give inaccurate results. The default value 10^{-4} works well for many MILP and MIQP applications.

4.limit on elapsed time of branch-and-bound calculation

```
> nuopt.options(maxtim = -1)
```

5.limit on number of branch-and-bound enumeration tree nodes (MIP only)

```
> nuopt.options(maxnod = -1)
```

In the course of the branch-and-bound enumeration, large-scale MILP/MIQP instances (with thousands of integer variables) may produce an enumeration tree that is so large as to require a prohibitive amount of time to find a global optimal solution. These parameters work as safeguards to limit the calculation time to a reasonable value.
`maxtim` sets a limit on the elapsed time in the branch-and-bound calculation process in seconds, and `maxnod` sets a limit on the number of nodes in the branch-and-bound enumeration tree. If either of these limits is reached in branch-and-bound enumeration, `NUOPT` terminates the calculation and reports the best solution obtained so far.
For both parameters, a negative value (the default) means "no limit". Related error messages are `<<NUOPT 17>>`, `<<NUOPT 19>>`, `<<NUOPT 21>>`, `<<NUOPT 22>>` (see section 7.3). Note that `maxtim` limits the total elapsed time for optimization, including data input, preprocessing, or solution of the first LP-relaxation.

6.limit on number of feasible solution of MIP before stopping.

```
> nuopt.options(maxintsol = -1) # -1 means "no value is set."
```

For some large MIP case, feasible solution is output in early stage of the branch-and-bound loop, but take quite a lot of time to prove its optimality, in such a case users wants the calculation terminate after pre-determined number of feasible solution is obtained. This specifies the maximum number of feasible solution before stopping. The number of obtained solution reaches this value, the calculation stops even the optimality of the best solution is not proved. If the solution is not proved optimality, NUOPT outputs the following error message:

```
<<NUOPT 37>> B&B terminated with given # of int.sol.
```

7.cutoff value (MIP only)

```
> nuopt.options(cutoff = 1.0e50) #  1.0e50 means "no value is set."
```

8.margin to cutoff value

```
> nuopt.options(addToCutoff = 0)
```

These parameters set or modify a cutoff value used in the branch-and-bound enumeration. By default, the cutoff value is set equal to the objective function value of the best integral solution obtained so far. Some branches of the enumeration tree are safely "pruned" and some computation is saved if the objective value of a relaxed problem on top of the branch is larger than the cutoff value (hereafter the objective is assumed to be minimized).
Finding a small cutoff value fast is very important for efficiency, but it is generally not an easy task for a solver without some knowledge of the model structure. If an approximate objective value is known beforehand, it is advisable to set the cutoff value manually to save computation. By default, `cutoff` is set to a huge value 10^{50}. Any value larger than this means that there is no effective cutoff value.
As explained above, we update the cutoff value to be the best integral solution obtained so far. `addToCutoff` is a positive value that gives a margin to the cutoff value in updating, namely:

$$\texttt{cutoffvalue} \leftarrow f^* - \texttt{addToCutoff}$$

Here f^* is the objective value of the best integral solution obtained. When the objective function is maximized, we update the cutoff value as follows:

$$\texttt{cutoffvalue} \leftarrow f^* + \texttt{addToCutoff}$$

A large value of `addToCutoff` causes a "better" cutoff value, and results in saving some computational effort. However, the larger the value of `addToCutoff`, the more the optimality of the solution is degraded it can only be guaranteed that there are is solution that gives a better optimal value within `addToCutoff`.

Parameters 4. to 8. are to limit the calculation of branch and bound loop, to set these parameters, you can consult the information of the feasible solution obtained by setting

```
> nuopt.options(mipfeasout="on") # report each feasible sol each obtained
```

By doing this, you can see the optimal value of feasible solutions, namely the objective value of the solution and time and the current number of simplex pivoting(#piv) and enumeration node(#prob) as below.

```
<iteration begin>
       1.2BI
feasible_sol #1        291 ( found at   0.1 sec. #piv=10:#prob=6 )
I
feasible_sol #2        323 ( found at   0.1 sec. #piv=13:#prob=8 )
I
feasible_sol #3        366 ( found at   0.1 sec. #piv=15:#prob=9 )

<iteration end>
```

These information is useful for setting proper parameter value to limit the branch and bound calculation.

5.3.3 General parameters

The following parameters are relevant to all methods.

1.scaling flag

```
> nuopt.options(scaling="on")
```

This parameter can take a string value "on" or "off". With this parameter set to "on", NUOPT performs scaling to balance the magnitude of objective value and constraints values in the model. Numerical experiments show that scaling may help reduce numerical difficulties and make the calculation process more stable. However for some numerically difficult problems, the scaling process may cause undesirable results such as premature termination, an inexact solution, or failure to converge

5.4 Options for the Solver

SIMPLE can be combined with various solvers. In this version of NUOPT for S-PLUS, we use a solver called NUOPT. Options for NUOPT are shown in Table 5.3. Details of the NUOPT options are
described in Section 5.3. To show the current options, we type:

```
> nuopt.options()
```

```
$method:
[1] "auto"
$crossover:
[1] "off"
$scaling:
[1] "on"
$mipfeasout:
[1] "off"
$addToCutoff:
[1] 0
$cutoff:
[1] 1e+050
$eps:
[1] -10
$epsint:
[1] 0.0001
$maxitn:
[1] 150
$maxnod:
[1] -1
$maxtim:
[1] -1
$told:
[1] 1e-006
$tolx:
[1] 1e-008
$maxintsol:
[1] -1
```

To set some options, we type:

```
> options.save <- nuopt.options()    ## save options
> nuopt.options(eps=1.7*10^-6,scaling="on")
> nuopt.options()
$method:
[1] "auto"
```

```
  $crossover:
  [1] "off"
  $scaling:
  [1] "on"
  $mipfeasout:
  [1] "off"
  $addToCutoff:
  [1] 0

  $cutoff:
  [1] 1e+050
  $eps:
  [1] 1.7e-006
  $epsint:
  [1] 0.0001
  $maxitn:
  [1] 150
  $maxnod:
  [1] -1
  $maxtim:
  [1] -1
  $told:
  [1] 1e-006
  $tolx:
  [1] 1e-008
  $maxintsol:
  [1] -1
  > nuopt.options(options.save)  ## restore options
```

To see the possible values[7] for the options, we type:

```
  > nuopt.options(NULL)
  $method:
[1] "line"    "trust"   "bfgs"    "simplex" "higher"  "auto"    "asqp"
$crossover:
```

[7] Mumeric value (integer or double) is not distinguished. Both integer and double value is displayed “double”.

```
[1] "off" "on"
$scaling:
[1] "off" "on"
$mipfeasout:
[1] "off" "on"
$addToCutoff:
[1] "double"
$cutoff:
[1] "double"

$eps:
[1] "double"
$epsint:
[1] "double"
$maxitn:
[1] "double"
$maxnod:
[1] "double"
$maxtim:
[1] "double"
$told:
[1] "double"
$tolx:
[1] "double"
$maxintsol:
[1] "double"
```

6 Examples

In this section, we define and solve several well-known models by SIMPLE.

6.1 A Simple Statistically-Oriented Optimization Problem

The first example is a modification of the constrained linear regression model of section 3, without constraints on β, and with the sum of squared residuals replaced with the sum of the Huber loss function ρ applied to each of the residuals:

$$\begin{aligned} \rho(r[i]) &= .5 * r[i]^2, && |\, r[i] \,| \leq c \\ &= c^* |\, r[i] \,| - .5 * c^2, && \text{otherwise} \end{aligned} \tag{7}$$

so that $\varphi(r[i]) = (d/dt)\rho r[i])$ is linear with slope = 1 for $|\, r[i] \,| \leq c$ and $\varphi(r[i]) = \text{sgn}(r[j]) * c$ for $|\, r[j] \,| > c$.

In SIMPLE, this model can be defined as follows:

```
> LregHuber
function(X, y)
{
       Res <- Set()
       Var <- Set()
       i <- Element(set = Res)
       j <- Element(set = Var)
       nres <- length(y)
       y <- Parameter(list(1:nres, y), index = i)
       X <- Parameter(X, index = dprod(i, j))
       beta <- Variable(index = j)
       r <- Expression(index = i)
       r[i] ~ y[i] - Sum(X[i, j] * beta[j], j)
       C <- Parameter(0.1, changeable = T)     ## changeable parameter
       rho <- Expression(index = i)    ## Huber loss function
       rho[i] ~ ife(abs(r[i]) <= C, 0.5 * r[i]^2, C * abs(r[i]) - 0.5 * C^2)
       obj <- Objective(type = "minimize")
```

```
       obj ~ Sum(rho[i], i)
}
```

Here, C is defined to be a changeable parameter in order to analyze several different cases.

```
> sys.LregHuber.stack <- System(LregHuber,stack.x,stack.loss)
> sol.LregHuber.stack <- solve(sys.LregHuber.stack,trace=F)
> summary(sol.LregHuber.stack)
$variables:
$variables$beta:
                 current lower upper initial dual
  [Air Flow]    0.9344302  -Inf   Inf        0    0
 [Water Temp]   0.3445385  -Inf   Inf       0    0
 [Acid Conc.]  -0.5349401 -Inf    Inf       0    0
$expressions:
$expressions$r:
       initial    current
1   5.55271215  5.55271215
2   0.01777207  0.01777207
           .          .
           .          .
           .          .
20 -0.35377435 -0.35377435
21 -8.62133634 -8.62133634
$expressions$rho:
        initial      current
1  0.5502712145 0.5502712145
2  0.0001579232 0.0001579232
           .          .
           .          .
           .          .
20 0.0303774345 0.0303774345
21 0.8571336341 0.8571336341

$objective:
  current
```

```
6.303447

$counter:
iters fevals vars
   16    418    3

$termination:
tolerance    residual
   1.5e-06 2.24565e-15

$elapsed.time:
 [1] 2.399994
```

Now we adjust the parameter C and solve again:

```
> current(sys.LregHuber.stack,C) ~ 1.0
> sol.LregHuber.stack <- solve(sys.LregHuber.stack, trace=F)
> summary(sol.LregHuber.stack)
$variables:
$variables$beta:
                 current  lower upper   initial dual
  [Air Flow]    0.9094374   -Inf   Inf  0.9344302    0
 [Water Temp]   0.4475915   -Inf   Inf  0.3445385    0
 [Acid Conc.]  -0.5426359   -Inf   Inf -0.5349401    0

$expressions:
$expressions$r:
      initial    current
1   5.45462732  5.45462732
2  -0.08800855 -0.08800855
           .          .
           .          .
           .          .
20 -0.38418500 -0.38418500
21 -8.23258606 -8.23258606
```

```
$expressions$rho:
      initial     current
1 4.954627315 4.954627315
2 0.003872753 0.003872753
           .           .
           .           .
           .           .
20 0.073799056 0.073799056
21 7.732586064 7.732586064


$objective:
  current
 55.08804

$counter:
iters fevals vars
    2      4    3

$termination:
tolerance    residual
   1.5e-06 8.337337e-14

$elapsed.time:
 [1] 0.1399536
```

6.2 The Diet Problem

The Diet problem (e. g. [2]) is a well-known optimization problem. Its basic form is to minimize the `cost of the menu' subject to the nutrition and serving requirements.

Variables : x_i $(i \in Food)$

Objective : $\sum_{i \in Food} \cos t_i \; x_i$

Constraints : $N\min_j \le \sum_{i \in Food} a_{ij} \; x_i \le N\max_j \qquad (j \in Nutrition)$

Bounds : $F\min_i \le x_i \le F\max_i \qquad (i \in Food)$

where

$Food$: kind of food

$Nutrition$: kind of nutrition

x_i : amount of servings of food i

$\cos t_i$: cost of food i

$N\min$: minimum amount of nutrition

$N\max$: maximum amount of nutrition

$F\min$: minimum number of servings

$F\max$: maximum number of servings

a_{ij} amount of nutrition in one serving of a certain kind of food

In SIMPLE, it can be defined by:

```
> Diet
function(Dcost, DNmin, DNmax, DFmin, DFmax, Da)
{
    Food <- Set()     ##  Kind of food
    Nutrition <- Set()       ##  Kind of nutrition
    i <- Element(set = Food)
    j <- Element(set = Nutrition)
    cost <- Parameter(Dcost, index = Food)    ## cost per serving
    Fmin <- Parameter(DFmin, index = Food)    ## minimum number of servings
    Fmax <- Parameter(DFmax, index = Food)    ## maximum number of servings
    Nmin <- Parameter(DNmin, index = Nutrition)
    ## minimum amount of nutrition
    Nmax <- Parameter(DNmax, index = Nutrition)
    ## maximum amount of nutrition
    a <- Parameter(Da, index = dprod(Food, Nutrition))
    ## amount of nutrition
    ## in one serving of
    ## a certain kind of food
    x <- Variable(index = Food)      ## amount of serving
    Fmin[i] <= x[i] <= Fmax[i]
```

```
    diet <- Constraint(index = Nutrition)
    diet[j] ~ Nmin[j] <= Sum(a[i, j] * x[i], i) <= Nmax[j]
    totalCost <- Objective(minimize)
    totalCost ~ Sum(cost[i] * x[i], i)
}
> DietInit
function()
{
    Dcost <- list(c("PotageSoup", "BeefSteak", "Eel", "leber"),
                c(500, 2000,3000, 200))
    DNmin <- list(c("VitaminA", "VitaminB"), c(20, 8))
    DNmax <- list(c("VitaminA", "VitaminB"), c(100, 200))
    DFmin <- list(c("PotageSoup", "BeefSteak", "Eel", "leber"),
                c(0, 0, 0,0))
    DFmax <- list(c("PotageSoup", "BeefSteak", "Eel", "leber"),
                c(1000,2000, 2500, 100))
    Da <- list(c("PotageSoup", "Eel", "Eel", "BeefSteak", "leber"),
              c("VitaminA", "VitaminA", "VitaminB", "VitaminB", "VitaminA"),
              c(2, 70, 3, 50, 30))
    return(Diet(Dcost, DNmin, DNmax, DFmin, DFmax, Da))
}
```

The model can be solved as follows.

```
> s3 <- System(DietInit)
> sol3 <- solve(s3,totalCost)
> summary(sol3)
$variables:
$variables$x:
                  current lower upper initial          dual
[PotageSoup]  0.00009681232     0  1000       0   486.67134048
 [BeefSteak]  0.16002252870     0  2000       0     0.29284699
      [Eel]   0.00001949195     0  2500       0  2413.51285573
    [leber]   0.66685219339     0   100       0     0.06987482
$objective:
initial  current
```

```
       0 453.5224
$constraints:
$constraints$"2":
                   current lower upper initial     dual      slack
diet[VitaminA]<2> 20.007124    20   100        0  6.664946 0.007123863
diet[VitaminB]<2>  8.001185     8   200        0 39.994389 0.001184911
$counter:
iters fevals vars
   10     12    4
$termination:
tolerance     residual
   1e-008 6.14879e-010
$elapsed.time:
 [1] 0.1796875
```

6.3 Maximal n-gon

The problem is to find a planar n-gon of maximal area inscribed in a circle of diameter 1.

Variables : ρ_i, θ_i $(i = 1 \cdots n)$

Objective : $\frac{1}{2} \sum_{2 \le i \le n} \rho_i \rho_{i-1} \sin(\theta_i - \theta_{i-1})$

Constraints: $\rho_i^{\,2} + \rho_j^{\,2} - 2\rho_i \rho_j \cos(\theta_j - \theta_i) \le 1, \quad (1 \le i < j \le n)$

$\rho_{i-1} \le \rho_i, \quad 2 \le i \le n$

Bounds : $0 \le \rho_i \le 1$

$0 \le \theta_i$

$\rho_n = 0$

$\theta_n = \pi$

where

(ρ_i, θ_i) : points of polygon (polar radius, polar angle)

The following function defines a `SIMPLE` model for the maximal n-gon problem.

```
> ngon
function(N)
{
    I <- Set(init = 1:N)
    i <- Element(set = I)
    j <- Element(set = I)
    n <- Element()
    n ~ max(i)       ## points of polygon (polar radius, polar angle)
    rho <- Variable(index = I)
    theta <- Variable(index = I)
    0 <= rho[i] <= 1
    0 <= theta[i]
    rho[i] ~ (4 * i * (n + 1 - i))/((n + 1) * (n + 1))
    theta[i] ~ (pi * i)/n
    inscribe <- Constraint(index = dprod(I, I))
    inscribe[i, j, i < j] ~ rho[i] * rho[i] + rho[j] * rho[j] - 2 * rho[i]
           rho[j] * cos(theta[j] - theta[i]) <= 1
    increasing <- Constraint(index = I)
    increasing[i, i >= 2] ~ theta[i] >= theta[i - 1]
    theta[n] == pi
    rho[n] == 0
    area <- Objective(maximize)
    area ~ 0.5 * Sum(rho[i]*rho[i-1]*sin(theta[i]-theta[i-1]),i,i>= 2)
}
```

The model can be solved as follows:

```
> s4 <- System(model=ngon,6)    ## polygon of 6 sides
> sol4 <- solve(s4,area)
> sol4
$variables:
$variables$rho:
       1         2         3         4        5 6
0.558964 0.7792118 0.9999933 0.9999907 0.6417768 0
attr($variables$rho, "indexes"):
[1] "*"
```

```
$variables$theta:
         1        2        3        4         5        6
 0.6599817 1.073387 1.38101 1.947531 2.625132 3.141593
attr($variables$theta, "indexes"):
[1] "*"
$objective:
[1] 0.6749733
$counter:
 iters fevals vars
     7      9   12
$termination:
 tolerance      residual
  1.5e-006 1.031837e-006
$elapsed.time:
[1] 0.2617188
```

The above results can be further be shown by using polygon (S-PLUSfunction) as follows.

```
> x <- as.array(current(s4,rho[i]*cos(theta[i])))        ## x - axis
> y <- as.array(current(s4,rho[i]*sin(theta[i])))        ## y - axis
> plot(c(-1,1),c(-0.5,1.5),type="n")
> polygon(x,y,density=0)
```

(see Figure 6.3).

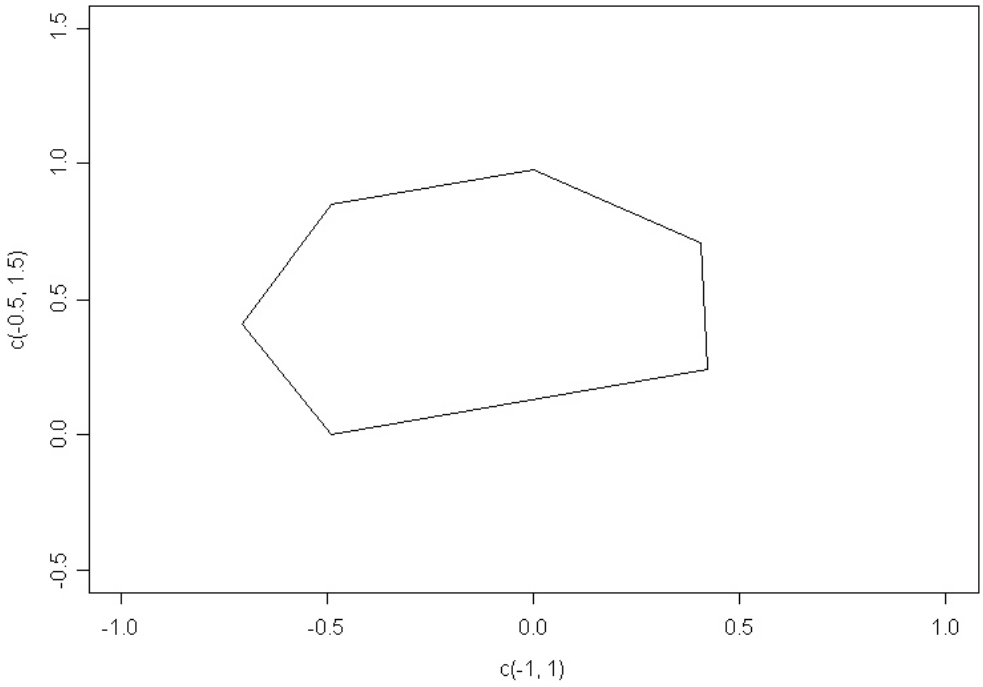

Figure 6.3: Solutions for polygon of 6 sides

6.4 Large Scale Portfolio Model

An investor wants to build a portfolio whose risk is as small as possible, but subject to the predetermined minimum accepted return. This problem can be formulated as an optimization problem called mean-variance (MV) model.

$$\text{Minimize} \quad \sum_{i,j \in Asset} Q_{ij} w_i w_j ,$$

$$\text{Subject to} \quad \sum_{j \in Asset} w_j = 1,$$

$$\sum_{j \in Asset} \bar{r}_j w_j \geq r_{\min} ,$$

$$w_j \geq 0, \quad j \in Asset$$

where w_j, r_j and Q_j correspond to an allocation for asset j (variable), a mean return of asset j and an element of a co-variance matrix respectively. $r_{\min}$ is a pre-determined minimum

accecpted return. The co-variance matrix Q is expressed as

$$Q = (1/(n-1))(R-\overline{R})^t(R-\overline{R}) \qquad (8)$$

where R is a "return matrix" whose (t, j) element is a return for asset j observed in period $t \in Period$, $\overline{R}$ is a matrix whose rows are identical to $\overline{r}$, and n is the number of periods. With `SIMPLE`, the problem is written as follows:

```
> PortfolioCor
function(rcov, averet, rmin =  0.0)
{
     Asset <- Set()
     i <- Element(set = Asset)
     j <- Element(set = Asset)
     Q <- Parameter(rcov, index = dprod(Asset, Asset))
     rBar <- Parameter(list(1:ncol(rcov), averet), index = Asset)
     w <- Variable(index = Asset)
     V <- Objective(minimize)
     V ~ Sum(w[i] * Sum(Q[i, j] * w[j], j), i)
     Sum(rBar[j]  w[j], j) >= rmin
     Sum(w[j], j) == 1
}
```

To set up and solve the system for a return matrix `portfolio.R`, you would issue the `S-PLUS` commands:

```
> portfolio.R
          [,1]        [,2]         [,3]        [,4]
 [1,] -0.7927655 -0.08728692   0.87694254 -0.59059953
 [2,]  0.7936707 -0.95924969  -0.22479839  0.18035779
 [3,] -0.8916424 -0.26851545   0.06113247 -0.99077122
 [4,]  0.1120670 -1.34384028  -0.79783496  1.85060734
 [5,]  1.3711549  0.76393502 -0.48544747  1.34083062
 [6,]  1.4169931 -0.55847085  -0.37566898  0.05630496
 [7,]  1.1665775  0.19412154  -2.17067057  1.01415727
 [8,] -0.5306316 -0.92741468  -0.27440216 -1.18712652
 [9,]  0.9211911 -0.18147267  -0.66076270  1.93023713
```

```
[10,]  -0.5769609  -1.57735867   1.42998998   0.86538263
> averet <- apply(portfolio.R, 2, mean)
> Q <- crossprod(sweep(portfolio.R, 2, averet))
> sys1 <- System(model=PortfolioCor, Q, averet)
> sol1 <- solve(sys1, V, trace=F)
```

Sometimes an alternative formulation of the optimization problem can lead to efficiency and/or stability improvements. For example, using expression 8 for the covariance matrix, and introducing internal variable $z_i \equiv (R_{tj} - \overline{R}_{ij})w_j$, we have a computationally more efficient formulation of this problem:

Minimize $(1/(n-1)) \sum_{t \in Period} z_t^2$,

Subject to $\sum_{j \in Asset} w_j = 1$,

$$\sum_{j \in Asset} \bar{r}_j w_j \geq r_{\min},$$

$$\sum_{j \in Asset} (R_{ij} - \overline{R}_{ij}) w_j - z_t = 0, \quad t \in Period$$

$$w_j \geq 0, \quad j \in Asset$$

With the modeling language `SIMPLE`, this problem is written as follows.

```
> PortfolioAlt
function(rmat, averet, rmin = 0.0)
{
     n <- nrow(rmat) # number of periods
     p <- ncol(rmat)
     Period <- Set()
     Asset <- Set()
     t <- Element(set = Period)
     j <- Element(set = Asset)
     R <- Parameter(rmat, index = dprod(t, j))
     rBar <- Parameter(list(1:p, averet), index = Asset)
     w <- Variable(index = Asset)
     z <- Variable(index = Period)
```

```
      V <- Objective(minimize)
      V ~ Sum(z[t]^2, t)
      Sum((R[t, j] - rBar[j]) * w[j], j) - z[t] == 0
      Sum(rBar[j] * w[j], j) >= rmin
      Sum(w[j], j) == 1
}
```

To set up and solve the system for a return matrix `portfolio.R`, you would issue the `S-PLUS` commands:

```
> averet <- apply(portfolio.R, 2, mean)
> sys2 <- System(model=PortfolioAlt, portfolio.R, averet)
> sol2 <- solve(sys2, V, trace=F)
```

Because the portfolio problem is a quadratic programming model, it can directly be solved by `solveQP` without building a System (see section 2) as follows.

```
## The one correspond to PortfolioCor
> PortfolioCorQP
function(rcov, averet, rmin =  0.0, trace=F)
{
      ##  build the constraint matrix and constraint bounds
      A <- rbind(averet, 1)
      cLO <- c(rmin, 1)

      ## set the bounds on the portfolio weights
      bLO <- rep(0, ncol(rcov))

      ## arguments to solveQP, multiply by 2 to correspond to PortfolioCor
       sol <- solveQP(objQ = 2*rcov, A = A, bLO = bLO, cLO  = cLO, trace = trace)
      return(sol)
}
```

To solve the model, you would issue the `S-PLUS` commands:

```
> averet <- apply(portfolio.R, 2, mean)
```

```
> Q <- crossprod(sweep(portfolio.R, 2, averet))
> sol3 <- PortfolioCorQP(Q, averet)
```

We can also use arguments to `solveQP` which correspond to the `PortfolioAlt` version.

```
> PortfolioAltQP
function(rmat, averet, rmin = 0.0, trace = F)
{
     n <- nrow(rmat)
     p <- ncol(rmat)
     ## objective function
     objQ <- diag(c(rep(0, p), rep(1, n)))   #
     ## build the constraint matrix and constraint bounds
     A <- cbind(rbind(sweep(rmat, 2, averet), averet, 1),
               rbind(diag(rep(-1,n)), matrix(0, 2, n)))
     cLO <- c(rep(0, n), rmin, 1)
     cUP <- c(rep(0, n), Inf, 1)    #
     ## set the bounds on the portfolio weights
     bLO <- c(rep(0, p), rep(-Inf, n))
     ## call the solver and extract the optimal weights
     ## multiply by 2 to correspond to PortfolioAlt
      sol <- solveQP(objQ = 2 objQ, A = A, bLO = bLO, cLO = cLO, cUP = cUP,
                  trace = trace)
     return(sol);
}
```

To solve the model, you would issue the `S-PLUS` commands:

```
> averet <- apply(portfolio.R, 2, mean)
> sol4 <- PortfolioAltQP(portfolio.R, averet)
```

References

[1] D. P. Bertsekas (1991). *Linear Network Optimization*, The MIT Press.

[2] D. Bertsimas and J. N.Tsitsiklis (1997). *Introduction to Linear Optimization*, Athena Scientific.

[3] I. Bongartz, A. R. Conn, N. I. M. Gould, and Ph. L. Toint (1995). CUTE: Constrained and Unconstrained Testing Environment, *ACM Transactions on Mathematical Software* 21, 123-160.

[4] D. M. Gay,(1991). Automatic Differentiation of Nonlinear AMPL Models, in *Automatic Differentiation of Algorithms: theory, implementation, and application* A.Griewank and G. F. Corliss (editors), SIAM (1991), 61-73.

[5] G. H. Golub and C. F. Van Loan (1996). *Matrix Computations*, 3rd edition, Johns Hopkins.

[6] J. Gondzio (1994). *Multiple Centrality Corrections in a Primal-Dual Method for Linear Programming*, Technical Report 1994.20, Logilab, HEC, Section of Management Studies, University of Geneva, (revised 1995).

[7] W. Hock and K. Schittkowski (1980). *Lecture Notes in Economics and Mathematical Systems 187: Test Examples for Nonlinear Programming Codes*, M. Beckmann and H. P. Kunzi (editors), Springer-Verlag.

[8] Statistical Science (1993). *S-PLUS Programmer's Manual, Version 3.2*. Seattle: StatSci, a division of MathSoft, Inc.

[9] H. Yamashita (1992). *A globally convergent primal-dual interior point method for constrained optimization*, Technical Report, Mathematical Systems Institute In c., Tokyo, Japan.

[10]H. Yamashita and H. Yabe (1993). *Superlinear and quadratic convergence of primal-dual interior point methods for constrained optimization* Technical Report, Mathematical

References

Systems Institute Inc., Tokyo, Japan.

[11]H. Yamashita and T. Tanabe (1993). *A primal-dual interior point trust region method for large scale constrained optimization*, Technical Report, Mathematical Systems Institute Inc., Tokyo, Japan.

[12]H. M. Markowitz (1959). *Portfolio Selection: Efficient Diversification of Investments*, Wiley.

[13]S. Mehrotra (1992). On The Implementation of a Primal-Dual Interior Point Method, SIAM J. *Optimization* 2, 575-601.

[14] George B. Dantzig; Mukund N. Thapa, Linear Programming 1:Introduction, New York : Springer, 1997 , ISBN: 0-387-94833-3

Appendix

Formal Syntax of SIMPLE

Notation:

aop denotes binary arthmetic operators +, -, *, /, %%
eop denotes logical operators <, <=, ==, >, >=
cop denotes logical operators %&&% (and), % % (or)

```
Condition = Condition cop Condition
          | ! Condition
          | Parameter eop Parameter
          | Element < Set
Constraint = Constraint eop Expression
           | Constraint eop Parameter
           | Expression eop Expression
           | Expression eop Parameter
           | Parameter  eop Expression
Element = character string
        | numeric value
        | Element aop Element
Expression = function(Expression)
           | Variable
           | Expression aop Expression
           | Expression aop Parameter
           | Parameter  aop Expression

Parameter = function(Parameter)
          numeric value
          numeric Element
          Parameter aop Parameter
  index = SetOrElement | dprod(SetOrElement,...SetOrElement)
  SetOrElement = Set | Element
 Sum(argumentList,PExpression,argumentList)
 Prod(argumentList,PExpression,argumentList)
```

```
PExpression  = Parameter | Expression
argumentList =  (empty) | CElement , argumentList
CElement     = Condition | Element (Instance)
```

Error Messages from SIMPLE

SIMPLE error messages have the following form:

```
<<SIMPLE num>> .... message ....
```

where num is an identity number.

```
<<SIMPLE 1>> Bound conflict: Lower bound > Upper Bound.
```

The lower bound and the upper bound of a variable or an expression conflict.

```
<<SIMPLE 4>> Warning: Length of a subscript exceeds 30.
```

The length of a string name used as a subscript exceeds 30. (Characters are ignored beyond position 30).

```
<<SIMPLE 6>> Illegal characters ("[", "]", "..." and so on) included
in subscript.
```

```
<<SIMPLE 15>> Internal Error!
```

An internal error has occurred (Please contact support@insightful.com).

```
<<SIMPLE 19>> Warning: No auto-assignment performed for constant set.
```

```
<<SIMPLE 24>> Attempt to find the maximum of an empty set.
```

```
<<SIMPLE 49>> A call has been made to an invalid function.
```

```
<<SIMPLE 53>> Operation between elements of different dimension.
```

An attempt has been made to perform an elementary binary operation between two subscripts belonging two different dimension sets.

`<<SIMPLE 58>> Set difference error: s1-s2 in which s1 doesn't include s2.`

An attempt has been made to do a set difference s1-s2 in which s2 is not a subset of s1.

`<<SIMPLE 59>> Fixed value (...) out of defined range (...).`

An attempt has been made to fix an element at a value out of its defined range. For example, `a<-Set(1:3);i<-Element(set=a); I¥~~ 10`, the last statement produces the error message: Fixed value (10) out of defined range (`i`).

`<<SIMPLE 60>> Illegal characters included in subscript.`

`<<SIMPLE 62>> Comparison between elements of different dimension.`

`<<SIMPLE 64>> Constraint: subscript not matched.`

`<<SIMPLE 67>> Expression: subscript not matched.`

An attempt has been made to index an object with the wrong number of subscripts (e.g. `x <- Variable(index=i); x[i,j]>=1;`).

`<<SIMPLE 70>> Unmatched or ambiguous element(s).`

There are subscripts on the right hand side of an assignment that do not occur on the left hand side. For example, `y[j]¥~~ p[i]` will produce the above error.

`<<SIMPLE 74>> Improper use of a dependent subscript.`

For example, `in j <- Element(set=S[i]); x<-Variable(index=j), j` is defined to be dependent on `i` but is not used together with `i`.

`<<SIMPLE 82>> Subscript ... of ... out of range.`

An attempt has been made to use an object with a subscript out of its defined range. For example, in the following definitions: `a <- Set(1:2); i<-Element(set=a); x<-Variable(index=i); x[3]+1`, the subscript 3 is out of `x`'s defined range.

`<<SIMPLE 91>> Operation between sets of different dimension.`

For example, the error message occurs on the following case. `a <- Set(1:5); b<-Element(1:6, dim=2); a & b;`.

`<<SIMPLE 92>> Warning: No auto-assignment performed for ...`

For example, in `a <- Set(1:2); b <- Set(3:4); d <- Set(superSet=a|b)`, the last statement produces the warning: No auto-assignment performed for Union(a,b).

`<<SIMPLE 102>> Set assignment: dimension conflict.`

An attempt is made to assign one set to another having different dimension.

`<<SIMPLE 103>> superSet not defined for sets having different dimensions.`

`<<SIMPLE 104>> ... can not be a superset of ...`

The first object `...' can not be a super set of the second one `...'.

`<<SIMPLE 106>> Argument "arcs" must be a 2-dimensional set`

`<<SIMPLE 107>> Argument "nodes" must be a 1-dimensional set.`

The argument nodes for function `Graph` must be a 1-dimensional (`dim=1`) set.

`<<SIMPLE 111>> Assignment: rhs includes free subscript.`

A free subscript occurs on the right hand side of an assignment.

`<<SIMPLE 113>> Invalid assignment.`

Any error in assignment not covered by other messages.

```
<<SIMPLE 121>> Invalid constraint specification:
 (parameter <= expr => parameter is not allowed).
```

An invalid specification fixing the value of a constraint.

`<<SIMPLE 125>> Remove/Restore: Constraint number out of range.`

A non-existent constraint number is specified in `delete.con` or `restore.con`.

`<<SIMPLE 140>> Fixed value out of range for variable.`

An attempt has been made to fix a variable at its current value that is out of range.

For example, in `x <- Variable(3); x>=10; s1 <- System(); fix.Variable(s1,x)` the current value `x` is 3 which is out of `x`'s range `x>=10`, so that `x` can not be fixed

to its current value.

<<SIMPLE 142>> Constraint object required in function call.

The constraint argument for function delete.con/restore.con must be an object defined by the Constraint function. For example, delete.con(s1, x[i]+y[i]>=6) is not allowed.

<<SIMPLE 163>> No current value for empty constraint.

<<SIMPLE 164>> No initial value for empty constraint.

<<SIMPLE 165>> No dual value for constraint that is empty or before the model is solved.

<<SIMPLE 167>> No val/dual value exists for multiply assigned Constraint id.

An attempt has been made to show the current or dual values of a multiply-defined constraint. For example, if we define co ¥~~ sin(x)<=y<=z, the current/dual values of co is not uniquely defined.

<<SIMPLE 168>> Objective can only be assigned once.

<<SIMPLE 177>> No lower bound for empty constraint.

<<SIMPLE 178>> No upper bound for empty constraint.

<<SIMPLE 191>> No assignment is allowed to Sets having both index and superSet.

An attempt has been made to assign to a set which is indexed and has a super set.

Only implicit assignment (auto-assignment) is allowed in such a case. For example, this error occurs in the following: a <- Set(); b <- Set(); d <- Set(index= a, superSet=b); d ~ 1:5;

<<SIMPLE 193>> Error in solve():
<<NUOPT xx>> xxxxxxx

Error occurred in optimization. Display an error message from NUOPT.

```
<<SIMPLE 194>> Fatal error in solve(), ......
        <<NUOPT xx>> xxxxxxx
```

Fatal error occurred in optimization. Display an error message from NUOPT. This case, NUOPT do not return any solution.

```
<<SIMPLE 214>> Warning constraint#N reduce to "xxxx" (always satisfied).
<<SIMPLE 215>> constraint#N reduce to "xxxx" (never satisfied).
<<SIMPLE 216>> Trivial and Infeasible constraint appeared.
```

Some input error may produce trivial constraints between constants. These warning and error reports such a situation. In the example below,

```
errsample <- function()
{
   S <- Set() # empty
   i <- Element(set=S)
   x <- Variable(index=i)
   Sum(x[i],i) == 0   # always satisfied but meaning less
   Sum(x[i],i) >= 5   # never satisfied
}
```

This case, S is empty then Sum(x[i],i) will be zero. Then the constraints reduce to trivial ones. The former is always satisfied the latter never satisfied. Then

```
> s <- System(errsample)
```

will produce 3 errors as below.

```
<<SIMPLE 214>> Warning constraint#1 reduce to "0 == 0" (always satisfied).
<<SIMPLE 215>> constraint#2 reduce to "0 >= 5" (never satisfied).
<<SIMPLE 216>> Trivial and Infeasible constraint appeared.
```

If Error 215 is detected, then NUOPT stops with error 216, otherwise it will continue calculation.

Error Messages from NUOPT

In this section we explain NUOPT's error messages. Error with indication "[FATA L]" are a fatal ones that cause NUOPT to terminate without any solution (the variables reported remain unchanged from the given initial values); in other cases, NUOPT returns non-optimal values for the parameters.

Each error is numbered. Explanations follow in order.

<<NUOPT 1>> memory error in preprocessing.
The required memory reaches its limit in the preprocessing phase. [FATAL]

<<NUOPT 2>> infeasible (linear constraints and variable bounds)
Conflicting linear constraints and variable bounds cause infeasibility.

<<NUOPT 3>> no variables in this model.
There are no variables in the model. [FATAL]

<<NUOPT 5>> infeasible variable bounds
Variable bounds cause infeasibility (consider the integrality of some variables). [FATAL]

<<NUOPT 6>> unbounded (linear constraints and variable bounds)
The model looks unbounded (determined by linear constraints and bounds). [FATAL]

<<NUOPT 7>> internal error. [(internal function name)]
An internal error has occurred (Please contact support@insightful.com). [FATAL]

<<NUOPT 8>> memory error in optimization phase.
The required memory reaches its limit in the optimization phase. [FATAL]

<<NUOPT 9>> step reduction limit exceeded.
A line search failed at some iteration.

<<NUOPT 10>> IPM iteration limit exceeded.
The IPM (interior point method) iteration count reaches its limit given by parameter maxitn.

<<NUOPT 11>> infeasible.

The model is determined to be infeasible.

`<<NUOPT 13>> unbounded.`

The model is determined to be unbounded.

`<<NUOPT 14>> integrality is violated.`

Some integer variables remain fractional at the solution. (If a model contains integer variables, only the simplex method can guarantee integrality.)

`<<NUOPT 15>> simplex/asqp misapplied to NLP.`

An attempt has been made to apply the simplex method to NLP (the simplex method and active set method cannot handle NLP). [FATAL]

`<<NUOPT 16>> Infeasible MIP.`

This MIP (mixed-integer LP/QP programming) model has been determined to be infeasible. (The LP/QP-relaxation solution is output.)

`<<NUOPT 17>> B & B node limit reached (with int.sol.).`

Number of nodes in branch-and-bound enumeration reaches its limit given by parameter `maxnod`. An integer feasible solution (not guaranteed to be optimal) is reported.

`<<NUOPT 18>> MIP iteration failed (with int.sol.).`

The branch-and-bound enumeration scheme failed due to numerical problems. An integer feasible solution (not guaranteed to be optimal) is reported.

`<<NUOPT 19>> B & B node limit reached (no int.sol.).`

Number of nodes in branch-and-bound enumeration reaches its limit given by parameter `maxnod`. No integer feasible solution is obtained so far. (The LP-relaxation solution is output.)

`<<NUOPT 20>> MIP iteration failed (no int.sol.).`

The branch-and-bound enumeration scheme failed due to numerical problems. No integer feasible solution is obtained. (The LP-relaxation solution is output.)

`<<NUOPT 21>> B & B iter. timeout (with int.sol.).`

The elapsed time for the branch-and-bound calculation exceeds its limit given by parameter `maxtim`. An integer feasible solution (not guaranteed to be optimal) is reported.

`<<NUOPT 22>> B & B iter. timeout (no int.sol.).`

The elapsed time for the branch-and-bound calculation exceeds its limit given by parameter `maxtim`. No integer feasible solution is obtained so far. (The LP-relaxation solution is output.)

`<<NUOPT 28>> higher-order method is only for LP.`

Try to use LP specific interior point method "higher" for NLP problems. [FATAL]

`<<NUOPT 33>> Bound violated.`

`<<NUOPT 34>> Bound and Constraints violated.`

`<<NUOPT 35>> Constraint violated.`

`<<NUOPT 36>> Equality constraints violated.`

After interior point method iteration, variable bounds or constraints or equality constraints looks violated. This happens when you are solving numerically difficult problem with interior point method("higher","line","trust","bfgs"). Try out with tighter "eps" by

```
nuopt.options(eps=1.0e-10)
```

or switch the scaling option by

```
nuopt.options(scaling="on"/"off").
```

`<<NUOPT 37>>` B&B terminated with given # of int.sol.'

Calculation terminated with feasible solution of MIP(Mixed integer LP/QP problem) without proving its optimality, because the number of the feasible solution reaches predetermined tolerance set by `maxintsol`.