S+NUOPT V2 manual

Contents

1. Preface

S+NUOPT is a general purpose optimization environment. 'General purpose' means that users should build optimization model for his/her problem before getting the result. This is a great drawback comparing to some other single purpose tool. For example, if you want to do linear regression from S-PLUS environment, just prepare data and call lsfit. Meanwhile, if you should do it with S+NUOPT, you first should build the linear regression model as a optimization model which states what is the 'variables' you want to know.

This manual is written to remedy such a drawback, targeted to users who have actually a problem to solve. This manual contains various real examples such as non-linear fitting, portfolio optimization, and robust portfolio optimization etc. We recommend you to first grance "1-3 examples table" to find an optimization example of your interest. All examples are proposed with complete models and datas. You can cut and paste them into your S-PLUS command prompt and see how it goes. The result will help you find what this tool can work for your job.

1-1 How to use this manual

The "1-3 examples table" will direct you which example is of your interest. We strongly recommend to "cut and paste" them into your S-PLUS command prompt to invoke the optimization to survey the possibilities of this optimization package. Optimization models are pre-built as S-PLUS procecdures and you can feel free to rename and rewrite them for your own purpose. Comparing to bulid models from the scratch, revising the examples may be the fastest way to get a desired result.

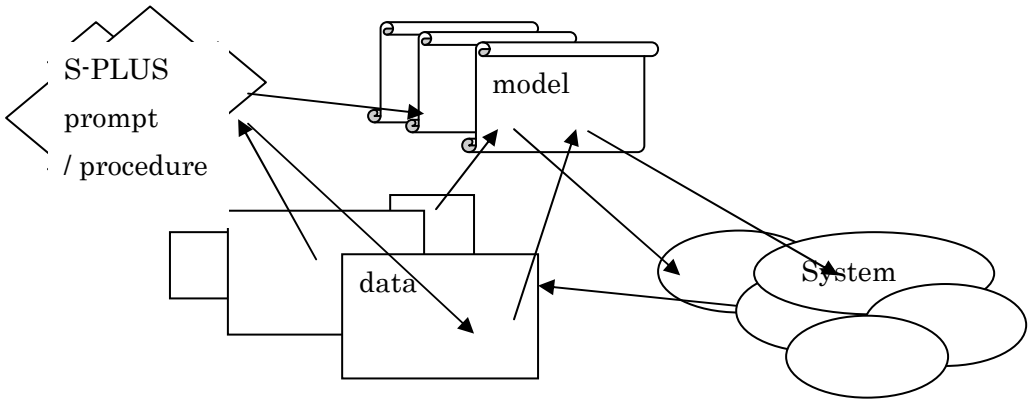
On portfolio optimization, S+NUOPT users' most common interest, we included almost all popular optimization models. If you find a suitable model, all you have to do is preparing datas (such as returns or covariances).

In the second chapter, with linear regression model as an example, basic components to build optimization models are introduced. The third chapter contains more advanced topics that are used for some special purposes like more compicated linear programming, semi-definite optimization, and usage of meta-heuristics algorithm. Forth chapter describes the interface of S+NUOPT and S-PLUS, how to put your S-PLUS data into S+NUOPT and vice versa. The fifth, sixth and seventh chapters are for examples.

1-2 How does S+NUOPT work?

S+NUOPT is an add-on package of S-PLUS. You can interactively invoke optimization engine from your command prompt. S+NUOPT deals with three types of objects. The first is the S-PLUS data (vector/matrix/array/list). They can be both input and output of the optimization. The second is the optimization model itself. It takes a form of "procecure" in S-PLUS, but for S+NUOPT it is a bunch of mathematical expression that describes optimization model. Note that, the optimization model is distinguished from data. The third is the concrete problem that is produced by combining datas and a model. We call it "system" object. The optimization engine solves the "system" objects. The result is given as S-PLUS data again, so you can invoke sequence of optimization, using a result of one optimization model as an input of the other.

The second category of S+PLUS object, the optimization models is written in other language (modeling language SIMPLE) embedded into S-PLUS. The second and the third chapter explain how you can write mathematical expressions in the language.



S-PLUS's many statistical functions and data manipulations will help you prepare the optimization inputs. And built-in real-life datas (like "freeny" or "state" used in this manual) can be used as testing purposes. Moreover, you can visualize the result of the optimization (output as S-PLUS data) easily with S-PLUS graph drawing features.

1-3 Table of Examples

S+NUOPT is with many built-in models. This table shows where to find examples in this manual. If you find an example of your interest, we strongly recommend you to test it by cut-and-pasting.

example title	description	model name	data used	Chapter and section
linea regression	specific to data "freeny"	Nlsfit	freeny.x/y	2.1
	general purpose	Nlsfit.gen	freeny.x/y,stack.x/loss	2.7.1
	with additional linear constraints	Nlsfit.eq	freeny.x/y,stack.x/loss	2.8
	with named constraints	Nlsfit.eq.named	freeny.x/y,stack.x/loss	2.8.1
	with choise of fitting parameters	Nlsfit.int	freeny.x/y,stack.x/loss	2.9
cash flow matching	cash flow matching problem (conditonal statement example)	Cashflow	Cashflow.flow/bf	3.1
SDP(semi-definite programming) introduction	find the smallest eigenvalue	MinLambda	var(air)	3.2
Set partitioning	One feature, bipartite	Half	state.x77	3.3.1
	Many features, bipartite	Half2	state.x77	3.3.2
	One feature, N-partition	Partition	state.x77	3.3.3
Portfolio optimization	Markowitz model	Marko	R.60x4	5.1
	Asset allocation (Markowitz model)	MinVar	R.8000x5	5.2.1
	Asset allocation (Mean Absolute Variance)	MinMad	R.8000x5	5.2.2
	Asset allocation (Linear Lower partial moment).	MinLPM1	R.8000x5	5.2.3
	Asset allocation (Conditional value at risk)	MinCVaR	R.8000x5	5.2.4
	Large scale portfolio (Alternative formulation)	MinVar	R.60x1000	5.3
	Maximum Drawdown	MinMaxDD	R.521x95	5.6
	Sharpe Ratiomaximization (original NLP)	Sharpe	R.60x200	5.7
	Sharpe Ratio maximization (Reforumulation as QP)	Sharpe.qp	R.60x200	5.7
Descrete Optimization and Portfolio optimization	Roudlot problem	RoundLot	RoundLot.unit/x	5.4
	A Basket portflio problem	Basket	Basket.flow/fhigh/fbar/W	5.5
SDP application	robust optimization	Robust	Robust.sigU/sigL/mu	6.2
	matrix calibration	Cormat	Cormat.A	6.1
Nonlinear regression	Yield curve fitting	Yield	Yield.telem/term/price	7.1
	Rating transition matrix estimation	Rating	Rating.Q0	7.2
	Logistic regression	LogReg	LogReg.X/t/test.X/test.t	7.3

1-4 Getting Started with S+NUOPT

Just type

```
module(nuopt)  # load S+NUOPT
```

Check if the output

```
> module(nuopt)
Initializing ... ok
NUOPT for S-PLUS Version XXXXX for Microsoft Windows
```

This completes the loading.

2. Fundamentals of the Modeling Language

To use S+NUOPT, it is required to write down the optimization model in our modeling language SIMPLE. In this chapter, we explain fundamental tools of the modeling language. Here, we take linear regression as an example and copy the output of the S-PLUS function "lsfit". Almost all of the fundamental tools to express basic optimization models appear in this chapter. More advanced and specific tools are introduced in the next chapter.

Linear regression refers to an approach to model a relationship between variable X and Y to be linear and estimate unknown coefficients from data. This is regarded as an optimization problem. That is because we minimize the description error with coefficients as variables. Below is the formulation as a mathematical programming model.

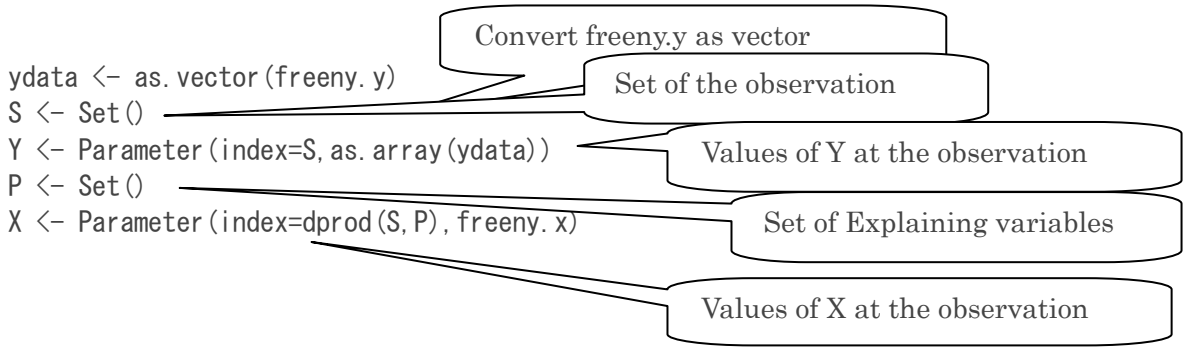
Variable	v_j	($j \in P$:linear coefficients)
	v_0	(constant term)
Minimize	$\sum_{i \in S} e_i^2$	(squeare sum of the error at the observations)
Definition	$e_i \equiv (v_0 + \sum_{j \in P} X_{i,j} v_j) - Y_i$	($i \in S$:observation)
	$X_{i,j}$:value of explaining variables ($j \in P$ at the observation $i \in S$)	
	Y_i :explained variable value at the o bservation $i \in S$	

To solve this mathematical programming model with S+NUOPT, you have to define a S-PLUS procedure that corresponds to the formulation above. Below is the one. Here, built-in data "freeny" is used as a sample data.

```
NlsfitNlsfit <- function()
{
# input freeny.y as Y, freeny.x as X
   ydata <- as.vector(freeny.y)
   S <- Set()
   Y <- Parameter(index=S,as.array(ydata))
   P <- Set()
   X <- Parameter(index=dprod(S,P),freeny.x)
#  below is the body of optimization model
   i <- Element(set=S)
   j <- Element(set=P)
   v <- Variable(index=j)
   v0 <- Variable()
   e <- Expression(index=i)
   e[i] ~ Sum(X[i,j]*v[j],j) + v0 - Y[i]
   esum <- Objective(type=minimize)
   esum ~ Sum(e[i]*e[i],i)
}
```

2-1 Parameter and Set objects

In the first part, we input freeny.x and freeny.y as Parameter objects (Parameter means constant). And we also define a Set object S.



We define Parameters Y and X as S-PLUS object freeny.y (ydata) and freeny.x as initial values. We can confirm this initialization from the command prompt. Input from the command prompt:

```
ydata <- as.vector(freeny.y)
S <- Set()
Y <- Parameter(index=S, as.array(ydata))
P <- Set()
X <- Parameter(index=dprod(S,P), freeny.x)
```

Now we can print Parameters Y and X.

```
> Y
       1       2       3       4       5       6       7       8
 8.79236 8.79137 8.81486 8.81301 8.90751 8.93673 8.96161 8.96044

       9      10      11      12      13      14      15      16
 9.00868 9.03049 9.06906 9.05871 9.10698 9.12685 9.17096 9.18665

      17      18      19      20      21      22      23      24
 9.23823 9.26487 9.28436 9.31378 9.35025 9.35835 9.39767 9.4215

      25      26      27     28      29      30      31     32      33
 9.44223 9.48721 9.52374 9.5398 9.58123 9.60048 9.64496 9.6439 9.69405

      34      35      36      37      38      39
 9.69958 9.68683 9.71774 9.74924 9.77536 9.79424
attr(, "indexes"):
[1] "*"
> X
   income level lag quarterly revenue market potential price index
 1      5.82110               8.79636          12.9699     4.70997
```

```
 2       5.82558                8.79236          12.9733      4.70217
 3       5.83112                8.79137          12.9774      4.68944
 4       5.84046                8.81486          12.9806      4.68558
 5       5.85036                8.81301          12.9831      4.64019
 6       5.86464                8.90751          12.9854      4.62553
 7       5.87769                8.93673          12.9900      4.61991
 8       5.89763                8.96161          12.9943      4.61654
 9       5.92574                8.96044          12.9992      4.61407
10       5.94232                9.00868          13.0033      4.60766
11       5.95365                9.03049          13.0099      4.60227
12       5.96120                9.06906          13.0159      4.58960
13       5.97805                9.05871          13.0212      4.57592
14       6.00377                9.10698          13.0265      4.58661
15       6.02829                9.12685          13.0351      4.57997
16       6.03475                9.17096          13.0429      4.57176
17       6.03906                9.18665          13.0497      4.56104
18       6.05046                9.23823          13.0551      4.54906
19       6.05563                9.26487          13.0634      4.53957
20       6.06093                9.28436          13.0693      4.51018
21       6.07103                9.31378          13.0737      4.50352
22       6.08018                9.35025          13.0770      4.49360
23       6.08858                9.35835          13.0849      4.46505
24       6.10199                9.39767          13.0918      4.44924
25       6.11207                9.42150          13.0950      4.43966
26       6.11596                9.44223          13.0984      4.42025
27       6.12129                9.48721          13.1089      4.41060
28       6.12200                9.52374          13.1169      4.41151
29       6.13119                9.53980          13.1222      4.39810
30       6.14705                9.58123          13.1266      4.38513
   income level lag quarterly revenue market potential price index
31       6.15336                9.60048          13.1356      4.37320
32       6.15627                9.64496          13.1415      4.32770
33       6.16274                9.64390          13.1444      4.32023
34       6.17369                9.69405          13.1459      4.30909
35       6.16135                9.69958          13.1520      4.30909
36       6.18231                9.68683          13.1593      4.30552
37       6.18768                9.71774          13.1579      4.29627
38       6.19377                9.74924          13.1625      4.27839
39       6.20030                9.77536          13.1664      4.27789
attr(, "indexes"):
[1] "*" "*"
```

You can initialize Parameters from S-PLUS vector, matrix, array, and list. Details of the data interface with S-PLUS are given in Chapter 4.

2-1-1.Set object

General form of definitions and initializations are given below.

```
name <- Set()
name <- Set(initializer)
```

The initializer can be S-PLUS vector object. Examples are given below.

```
S <- Set(1:5)
S1 <- Set(c("apple","orange","banana"))
S2 <- Set(seq(1,10,3))
```

The result is:

```
> S
{ 1 2 3 4 5 }
 > S1
{ apple orange banana }
 > S2
{ 1 4 7 10 }
```

If a Set is used as the index set of a certain Parameter initialized from data, you do not have to initialize the Set directly. Because S+NUOPT have a feature to automatically read the index set of the data, and add them to the Set defined as an index set (auto-assignment feature).

2-1-2.Parameter object

General form of definitions and initializations are given below.

```
name <- Parameter() # no index or initializer
name <- Parameter(initializer)  # no index with initializer
name <- Parameter(index=index set/element) # with index no initializer
name <- Parameter(index=index set/element,initializer)
```

S+NUOPT call objects that remain constant through optimizations, as Parameter. ”Parameter” is a generic name that means scalar/vector/matrix. That is, the name is unchanged regardless of the number of the index given. For example, a scalar constant is a Parameter without index, a vector constant is a Parameter with one index, and a constant matrix is a Parameter with two indcies.

If your Parameter has some indcies, you should give name of index Set or Element or their combination. Initializer may be S-PLUS vector, matrix, array, and list. Below is the example.

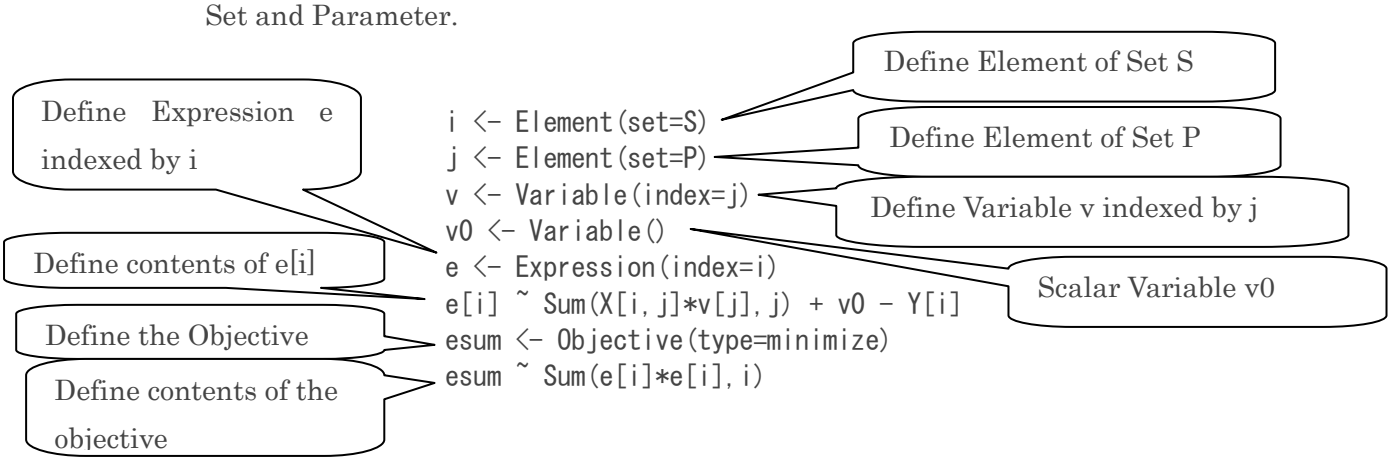
```
p <- Parameter(2.85)
q <- Parameter()  # Interpreted as zero
Y <- Parameter(index=S,as.array(ydata))
i <- Element(set=S)                    # define Element of set S
a <- Parameter(index=i,as.array())
# index may be name of Element of the index Set
X <- Parameter(index=dprod(S,P),freeny.x)
X <- Parameter(index=dprod(i,P),freeny.x)
```

```
# alternative. index may be combination of Set and Element
```

If the Parameter has two or more indecies, you should concatenate index Set or Element using dprod(). Initialization can be checked from the command prompt. Look at 2-1 or chapter 4.

2-2 Element object

In procedure "Nlsfit", the body of the formula goes after the definition/initialization of Set and Parameter.



The formulation corresponds to definitions of the contents of e[i] and esum. Every other statement is a definition of an object itself. First you should define Element object i,j used as indecies. The format of the Element definition is

```
name <- Element(set=Name of Set)
```

Element of S+NUOPT is binded to the Set given on the definition. If some Elements are used as indcies in some description, Elements are regarded to run through every Element of the binded Set and the description is expanded. For example,

```
e[i] ~ Sum(X[i,j]*v[j],j) + v0 - Y[i]  # Define contents of e[i]
esum ~ Sum(e[i]*e[i],i)                # Define contents of esum
```

The above description is expanded to mean every e[i] is defined. "Sum" operation means the sum of the 1st argument with 2nd argument running through the binded Set. Element objects can be used in the definition of index instead of Set.

```
v <- Variable(index=j)     # Same as Variable(index=P)
e <- Expression(index=i)   # Same as Expression(index=S)
```

2-3 Variable Object

Variables are unknowns in the optimization formula. In this example, v and v0 are variables. The former has index j, the latter has none. The format of the definition goes like this.

```
name <- Variable() # without index
name <- Variable(index=index Set or Element) # with index
```

2-3-1.Give variables a "initial guess"

You can give Variable "initial guess" by "~" operation. Next is an example.

```
v0 ~ 2.5   # initial guess of v0 to be 2.5
v[j] ~ 1   # initial guess of all v[j] to be 1
```

Default initial guess is zero. Giving the"initial guess" is optional. In many cases, the default setting goes fine (the S+NUOPT seeks initial guess itself). But if your model have formulations like below,

```
1.0/x          # If x = 0, function value cannot be evaluated
sqrt(x)        # If x = 0, function derivative cannot be evaluated
```

giving the "initial guess (say 1) " helps to get you out of trouble.

2-4 Expression object

In procedure "Nlsfit", e[i] corresponds the LHS of the formula:

$$e_i \equiv (v_0 + \sum_{j \in P} X_{i,j} v_j) - Y_i$$

Expression object is a fragment of formula. The definition format is just same as Parameter/Variable.

```
name <- Expression() #
name <- Expression(index=index Set or Element) # with index
```

You should not forget to assign an Expression object. If an Expression object is not assigned, it is interpreted as zero like Parameter. Please take notice that to assign **Expression or Parameter, use "~"not "=".**
The assignment should operate single object. The below is invalid.

```
x[i] + y[i] ~ z[i]   # error
```

Basically, RHS's Element should appear on the LHS. The below is invalid.

```
z ~ x[i]         # error
```

Next is also invalid because Element j is not determined.

```
y[i] ~ u[i,j]    # error
```

But RHS's do not always have LHS's index.

```
w[i,j] ~ x[i]    # valid
```

The above means, w[i,j] is assigned to be x[i] regardless of j.

2-5 Objective and its definition

Objective object can be regarded as a special Expression. The usage is just the same as Expression. Objective is a fragment of formulation you want to minimize or maximize. The format of the definition is as below. You have to define if the Objective is minimized or maximized. Objective cannot have indcies because the Objective is unity all through the formulation of an optimization model.

```
name <- Objective(type=miminize) # minimization
name <- Objective(type=maximize) # maximization
```

This is an example of definition and assignment.

```
esum <- Objective(type=minimize)
esum ~ Sum(e[i]*e[i], i)  # Sum of e[i]
```

2-6 Sum operation

In optimization problem formulations, "summation of an expression over an index" frequently appears. Below is an example.

$$\sum_{i \in S} e_i^2$$

This is expressed by a Sum operation like below.

```
Sum(e[i]*e[i], i)  # sum of e[i]
```

The format of the Sum operation is

```
Sum(formula, index)
          #  ex.  Sum(a[i], i)
Sum(formula, index1, index2, . . . , index n)
          #  ex.  Sum(M[i, j], i, j)
Sum(formula, index1, index2, …, index n, condition1, condition2, . . . , condition m)
          #  ex.  Sum(G[i, j], i, j, i>j, i!=0)
```

Sum operation takes sum of formula over area of the index given. The condition reduces the area the sum is taken. Sum is different from S-PLUS's built-in "sum". You cannot use built-in sum instead of Sum (built-in sum cannot interpret S+NUOPT object).

2-7 Create System and solve

You can solve the model by the 2 steps below.

```
sys <- System(Nlsfit)      # expand the model to create System object sys
sol <- solve(sys)          # solve the System and obtain result sol
```

System is created by giving datas to an optimization model definition. This is a concrete optimization problem solver can deal with. System command's format is below.

```
name <- System(optimization model)
name <- System(optimization model, argument1,argument 2,...,argument 3)
                    # model has some arguments
```

You can see the contents of the System object “sys”.

```
> sys

esum<objective>: (8.79636*(v[lag quarterly revenue])+4.70997*(v[price in
dex])+5.8211*(v[income level])+12.9699*(v[market potential])+v0+(-8.7923
6))*(8.79636*(v[lag quarterly revenue])+4.70997*(v[price index])+5.8211*
(v[income level])+12.9699*(v[market potential])+v0+(-8.79236))+(8.79236*
(v[lag quarterly revenue])+
                        ......
13.1664*(v[market potential])+v0+(-9.79424))*(9.77536*(v[la
g quarterly revenue])+4.27789*(v[price index])+6.2003*(v[income level])+
13.1664*(v[market potential])+v0+(-9.79424)) (minimize)
```

You can see how the objective is actually defined. Now you can drive solver on this System object and get the result, by invoking solve() command.

```
> sol <- solve(sys)
NUOPT 11.1.9a (NLP/LP/IP/SDP module)
       <with META-HEURISTICS engine "wcsp"/"rcpsp">
      , Copyright (C) 1991-2009 Mathematical Systems Inc.
NUMBER_OF_VARIABLES                                          5
NUMBER_OF_FUNCTIONS                                          1
PROBLEM_TYPE                                        MINIMIZATION
METHOD                                           TRUST_REGION_IPM
<preprocess begin>.....<preprocess end>
<iteration begin>
    res=1.6e-001 ... 4.4e-013
<iteration end>
STATUS                                                    OPTIMAL
VALUE_OF_OBJECTIVE                                0.007374997682
ITERATION_COUNT                                              4
FUNC_EVAL_COUNT                                              8
FACTORIZATION_COUNT                                          7
RESIDUAL                                       4.425550204e-013
ELAPSED_TIME(sec.)                                        0.02
```

The optimization is complete.

```
name <- solve(System object)          #  default with some output from solver
name <- solve(System object,trace=F) #  no output
```

The returned value from solve() command is a S-PLUS list object that contains the

result of the optimization. In this case that goes like this.

```
> sol
$variables:
$v:
 income level lag quarterly revenue market potential price index
    0.7674609             0.1238646         1.330558  -0.7542401
attr(, "indexes"):
[1] "j"

$v0:  #  variable v0
[1] -10.47261

$objective:  # the Objective
[1] 0.007374998

$counter:
 iters fevals vars
     3      6    5

$termination:
 tolerance      residual
  1.5e-006 1.138916e-009

$elapsed.time:
[1] 0.015

$errorCode:
[1] 0
```

If you want some specific variable's value from this list, type:

```
        sol$variables$v$current        # output v
```

summary() operation is defined to give a concise output about variables.

```
        summary(sol)
```

```
> summary(sol)
$variables:
$v:
                          current lower upper    initial dual
         [income level]  0.7674609  -Inf   Inf  0.7674609    0
[lag quarterly revenue]  0.1238646  -Inf   Inf  0.1238646    0
     [market potential]  1.3305577  -Inf   Inf  1.3305577    0
          [price index] -0.7542401  -Inf   Inf -0.7542401    0

$v0:
```

```
   current lower upper   initial dual
-10.47261  -Inf   Inf -10.47261    0
                  ......
```

Now you can crosscheck the result with the S-PLUS built-in function “lsfit” specific for linear regression.

```
> regfreeny <- lsfit(freeny.x, freeny.y)
> ls.print(regfreeny)
Residual Standard Error = 0.0147,  Multiple R-Square = 0.9981
N = 39,  F-statistic = 4354.254 on 4 and 34 df, p-value = 0

                        coef std.err  t.stat p.value
           Intercept -10.4726  6.0217 -1.7391  0.0911
lag quarterly revenue   0.1239  0.1424  0.8699  0.3904
         price index  -0.7542  0.1607 -4.6927  0.0000
        income level   0.7675  0.1339  5.7305  0.0000
    market potential   1.3306  0.5093  2.6126  0.0133
```

2-7-1.Give data name from System command and check the result

Procedure Nlsfit directly specifies freeny.x and freeny.y as input datas. By setting input datas as arguments, we can define a more generic optimization model which accepts any data set of same type.

```
Nlsfit.gen <- function(x,y)
# x (matrix) is explaining, y(vector) is explained variable
{
   S <- Set()
   Y <- Parameter(index=S,as.array(y))
   P <- Set()
   X <- Parameter(index=dprod(S,P),x)
   i <- Element(set=S)
   j <- Element(set=P)
   v <- Variable(index=j)
   v0 <- Variable()
   e <- Expression(index=i)
   e[i] ~ Sum(X[i,j]*v[j],j) + v0 - Y[i]
   esum <- Objective(type=minimize)
   esum ~ Sum(err[i]*err[i],i)
}
```

Data name can be specified through System() command. Now we demonstrate creating System object by setting data set as (freeny.x, freeny.y) and change it to (stack.x, stack.loss) with Nlsfit.gen unchanged.

```
sys1 <- System(Nlsfit.gen,freeny.x,freeny.y) # (freeny.x/y) as input
sol1 <- solve(sys1)
sys2 <- System(Nlsfit.gen,stack.x,stack.loss) # (stack.x/loss) as input
sol2 <- solve(sys2)
```

Two different problems are generated from the same procedure Nlsfit.gen, and solved. Command “current” is an alternative way to monitor the variable.

```
current(sys1, v)
# monitor Variable “v” of  “sys1” created from (freeny.x, freeny.y)
```

```
> current(sys1, v)
 income level lag quarterly revenue market potential price index
    0.7674609             0.1238646          1.330558  -0.7542401
```

```
current(sys2, v)
# monitor Variable  “v”  of  “sys2” created from (stack.x, stack.loss)
```

```
> current(sys2, v)
Acid Conc.   Air Flow Water Temp
 -0.1521225 0.7156402    1.295286
```

2-8 Add constraints to regression model

We can add general constraints on the coefficient. For example, we restrict the sum of the linear coeffieicnt to be zero.

Subject to $\sum_{j \in P} v_j = 0$

To do this, we add

```
Sum(v[j], j) == 0
```

to Nlsfit.gen and generate Nlsfit.eq. Generally constraints are expressed as below.

```
formula1 @ formula2                # two term
formula1 @ formula2 @ formula3     # three term
               @ is either { ==, =>, <= }
```

Note that you cannot use “<”, “>” or “!=” to define constraints. Solving this new model:

```
sys3 <- System(Nlsfit.eq, freeny.x, freeny.y)
sol3 <- solve(sys3, trace=F)    # suppress the output
current(sys3, v)
```

Output:

```
 income level lag quarterly revenue market potential price index
    0.802618             0.3010901        -0.1054927  -0.9982154
```

To confirm the result, we get variable “v”, convert it as array and sum. Actually,

```
sum(as.array(current(sys3,v)))
```

gives

```
[1] 4e-006
```

If you express the linear regression as an optimization model, you can set any constraints on the result.

2-8-1.Constraint object, delete or restore constraints.

You can manipulate constraints by defining Constraint object. Procedure Nlsfit.eq.named is equivalent to Nlsfit.eq but has constraint definition goes like this.

```
c1 <- Constraint()
c1 ~ Sum(v[j],j) == 0    # assign c1 to constraint
```

Now you can manipulate the constraint by its name "c1".

```
sys3 <- System(Nlsfit.eq.named,freeny.x,freeny.y)
sol3 <- solve(sys3,trace=F)   # suppress the output
current(sys3,c1)
```

gives the constraint value at the optimum.

```
[1] 4e-006
```

Below is the general format of defining Constraint object.

```
name <- Constraint()            #  no index
name <- Constraint(index=index set/element)    #  with index
```

Note that Constraint object is empty upon definition just the same as Expression object. It takes an effect when some constraint is assigned to it. We can delete or restore named constraints in a system not re-creating it.

```
delete.con(sys3,c1)   # delete constraint  "c1"
restore.con(sys3,c1)  # restore constraint  "c1"
```

For example,

```
delete.con(sys3,c1)
sol4 <- solve(sys3,trace=F)
sum(as.array(current(sys3,v)))
```

gives below.

```
[1] 1.467643
```

This means that this constraint is inactivated now. If we restore this constraint, it will

gets active again.

```
restore.con(sys3,c1)
sol4 <- solve(sys3,trace=F)
sum(as.array(current(sys3,v)))
```

gives

```
[1] 4.000003e-010
```

Below is the general format of delete.con, restore.con.

```
delete.con(System object, constraint name)   # remove constraint
restore.con(System object, constraint name)  # restore constraint
```

2-9 IntegerVariable

IntegerVariable is a Variable that is restricted to be integral and contributes greatly to the optimization modeling frexibility. Here, we modify the linear regression model to determine not only coefficient values but also the best K choice of the explaning variables. The optimization model is given as follows.

Variables	v_j	($j \in P$:linear coefficients)
	v_0	(constant term)
	$\delta_j \in \{0,1\}$	($j \in P$:use variable j or not)
Minimize	$\sum_{i \in S} e_i^2$	(squeare sum of the error at the observations)
Definition	$e_i \equiv (v_0 + \sum_{j \in P} X_{i,j} v_j) - Y_i$	($i \in S$:observation)
	$X_{i,j}$:value of explaining variables ($j \in P$ at the observation $i \in S$)	
	Y_i :explained variable value at the observation $i \in S$	
Constraints		
	$-M \cdot \delta_j \le v_j \le M \cdot \delta_j$	($j \in P$)
	:$\delta_j = 0 \Rightarrow v_j = 0$,	
	$\delta_j = 1 \Rightarrow v_j$ is not restricted	
	(M is a large number)	
	$\sum_{j \in P} \delta_j \le K$	:Select K variables

Adding binary IntegerVariables δ_j and two constraints to Nlsfit.gen yields Nlsfit.int.

```
Nlsfit.int <- function(x,y,K)
{
   S <- Set()
```

Take K from argument

```
    Y <- Parameter(index=S,as.array(y))
    P <- Set()
    X <- Parameter(index=dprod(S,P),x)
    i <- Element(set=S)
    j <- Element(set=P)
    v <- Variable(index=j)
    v0 <- Variable()
    e <- Expression(index=i)
    e[i] ~ Sum(X[i,j]*v[j],j) + v0 - Y[i]
    esum <- Objective(type=minimize)
    esum ~ Sum(e[i]*e[i],i)
    delta <- IntegerVariable(index=j,type=binary)
    -10*delta[j] <= v[j] <= 10*delta[j];
    Sum(delta[j],j) <= K
}
```

Constraints.
We set $M = 10$

The folloing is the IntegerVariable definition format.

```
name <- IntegerVariable() # no index
name <- IntegerVariable(index=index set/element) # with index
name <- IntegerVariable(type=binary) # no index,binary
name <- IntegerVariable(index=index set/element,type=binary) # with index,binary
```

Solve setting $K = 2$.

```
    sys <- System(Nlsfit.int,freeny.x,freeny.y,2)
    sol <- solve(sys)
    current(sys,v)
```

Explained variables selected.

```
  income level lag quarterly revenue market potential price index
    0.03609152             0.9792798                 0           0
attr(, "indexes"):
[1] "j"
```

3. Advanced features of the Modeling Language

In this chapter, we explain more advanced features of the modeling language. These features will be useful for smore complicated or special purpose optimization models.

3-1 Conditions for cash flow matching problem

Now you have 1000 units of cash. In the market, three bonds A, B and C exist. Their expiration dates are 3, 5 and 7 years respectively. They yields different cash flow like below, available at anytime.

```
> Cashflow.bf  # face value is 100
   A    B    C
0 -100 -100 -100
1    3    5    2
2    3    5    2
3  110    5    2
4    0  105    2
5    0    0    2
6    0    0    2
7    0    0  130
```

We expect cash flow forthcoming 10 years as below.

```
> cash.flow
1 2   3    4 5 6   7     8 9  10
0 0 400 -200 0 0 400 -1000 0 600
```

We seek a cash management plan of purchasing bonds, to maximize the cash at hand on the 10th year, remaining at least 200 units of cash every year.
This problem is expressed as a linear programming problem as below.

Variables $x_{b,t}$ ($b \in B, t \in T$:Amount of purchase of Bond b on the year t)

y_t (Cash at hand on the year t)

Maximize y_{TL} (Cash on the last year)

TL : The last year

Constraint $y_0 = P$ — Initial cash

$y_t = y_{t-1} + \sum_{b \in B} \sum_{k, t-k \geq 1, k \leq L_b} BF_{k,b} x_{b,t-k} + C_t$, $t \geq 1$ (Cash conservation) — Expected cash flow

Cash flow from bonds

$x_{b,t} \geq 0$, $b \in B, t \in T$ (purchase amount is non-negative)

$x_0 = 0$ (No purchase of bond at the initial)

$x_{b,t} = 0,\ t + L_b + 1 > TL$ (Never purchase bonds whose expiration date comes after the last year)

$y_t \geq C_l$ — Constraints of cash at hand

Constraints “cash conservation” and “Never purchase bonds whose expiration date comes after the last year” should be defined on restricted set of index. And the expression “Cash flow from bonds” has condition on the summed area. They are defined

by using conditional statement. Below is the model procedure correstponds above.

```
Cashflow <- function(flow,bf,P,CI)
{
    Time <- Set(0:length(flow))
    K <- Set()
    B <- Set()
    C <- Parameter(index=Time,as.array(flow))
    BF <- Parameter(index=dprod(K,B),bf)
    k <- Element(set=K);
    b <- Element(set=B);
    t <- Element(set=Time)
    len <- Parameter(index=b)
    len[b] ~ Sum(Parameter(1),k,BF[k,b]!=0) - 1
    TL <- length(flow)
    x <- Variable(index=dprod(b,t))
    y <- Variable(index=t)
    y[0] == P
    y[t,t>=1] == y[t-1] + Sum(x[b,t-k]*BF[k,b],b,k,t-k>=1,k<=len[b]) + C[t]
    x[b,0] == 0
    x[b,t,t+len[b]+1>TL] == 0
    x[b,t] >= 0
    y[t] >= CI
    f <- Objective(type=maximize)
    f ~ y[TL]
}
```

Expiration date definition

Underlined is the conditional statement. If a conditional statement appears in a definition of a constraint, the constraint is only defined where the conditional statement is fulfilled.
The definition of a constraint defined below

```
y[t,t>=1] == y[t-1] + Sum(x[b,t-k]*BF[k,b],b,k,t-k>=1,k<=len[b]) + C[t]
```

contains conditional statement “t>=1”, that restrict where this constraint holds.
Conditional statement can appear anywhere in the “[]” clause in the constraint.

```
x[b,t,t+len[b]+1>TL] == 0
```

Described above is also a constraint with a conditional statement. The conditional statement can include Parameter and Set object.
Conditinal statement can include >=, <=, == and

!=	(not equal)
<	(greater than)
>	(less than)

A conditinal statement in “Sum”function restricts the summed area.

```
len[b] ~ Sum(Parameter(1),k,BF[k,b]!=0) - 1  # expiration date
```

```
Sum(x[b,t-k]*BF[k,b],b,k,t-k>=1,k<=len[b])
                    # cash flow yielded by bonds.
```

Described above are examples.
We recommend printing the System object to see if it is properly expanded.

```
sys <- System(Cashflow,Cashflow.flow,Cashflow.bf,1000,200)
```

```
> sys
1-1 :  y[0] == 1000

2-1 :  -y[1]+y[0]-100*x[A,1]-100*x[B,1]-100*x[C,1] == 0
2-2 :  -y[2]+y[1]-100*x[A,2]-100*x[B,2]-100*x[C,2]+3*x[A,1]+5*x[B,1]+2*x
[C,1] == 0
2-3 :  -y[3]+y[2]-100*x[A,3]-100*x[B,3]-100*x[C,3]+3*x[A,2]+5*x[B,2]+2*x
[C,2]+3*x[A,1]+5*x[B,1]+2*x[C,1]+400 == 0
                        ...
6-7 :  y[6] >= 200
6-8 :  y[7] >= 200
6-9 :  y[8] >= 200
6-10 :  y[9] >= 200
6-11 :  y[10] >= 200

f<objective>: y[10] (maximize)
```

Now solve and check the result.

```
sol <- solve(sys)
round(as.array(current(sys,x),digits=2)
round(as.array(current(sys,y),digits=2)
```

Below is the result.

```
> round(as.array(current(sys,x),digits=2) # purchase plan
  0    1    2    3 4    5    6 7 8 9 10
A 0 1.52 0.00 4.25 0 0.00 4.9 0 0 0  0
B 0 2.37 0.00 0.00 0 2.71 0.0 0 0 0  0
C 0 4.11 0.25 0.00 0 0.00 0.0 0 0 0  0
attr(, "indexes"):
[1] "b" "t"

> round(as.array(current(sys,y)),digits=2) # cash at hand
    0   1   2   3   4   5   6      7   8       9      10
 1000 200 200 200 200 200 200 636.95 200 1055.28 1655.28
attr(, "indexes"):
```

3-2 SymmetricMatrix and minimum eigenvalue problem

You can restrict a symmetric matrix to be positive semi-definite. By using this feature, we can express minimum eigenvalue problem as an optimization model and solve.

Variable	λ
Maximize	λ
Constraint	$A-\lambda I$ is positive semi-definite
	A:Any symmetric matrix as input

To define this problem, we use SymmetricMatrix object.

```
MinLambda <- function(A)
{
    S <- Set()
    i <- Element(set=S)
    j <- Element(set=S)
    A <- Parameter(index=dprod(i,j),A)
    lambda <- Variable()
    M <- SymmetricMatrix(dprod(i,j))
    M[i,j,i>j] ~ A[i,j]
    M[i,i] ~ A[i,i] - lambda
    M >= 0
    f <- Objective(type=maximize)
    f ~ lambda
}
```

Define SymmetricMatrix M

Define elements of M (Diagonal and Lower triangular part only)

Constrain M to be Positive semi-definite constraint

SymmetricMatrix is equivalent to a collection of Expression objects as a symmetric matrix. Elements of a SymmetricMatrix can be any expression. The following is the format of definition of SymmetricMatrix.

```
name <- SymmetricMatrix(dprod(Set, Set))  # Give same 2 Sets as arguments
name <- SymmetricMatrix(dprod(Element1, Element2))
                                  # Element1 and Element2 should be of set Set
```

In this example a SymmetricMatrix object named “M” is defined. At the definition, i and j (both are elements of same Set) are given. You can give dprod(S,S)at the RHS of “index=”. Next, we define the element of M by “~” operator just as Expression. By definition, M is Symmetric, only the Diagonal and Lower triangular part is defined.
Now we solve the model and get minimum eigenvalue. We use covariance matrix of built-in data air as a sample.

```
mat <- var(air)
sys <- System(MinLambda,mat)
sol <- solve(sys,trace=F)
current(sys,lambda)
```

gives below.

```
[1] 0.2510515
```

This exactly coincides with the least eigenvalue calculated by S-PLUS.

```
> eigen(mat)$values
[1] 8317.0294527   86.5362664    9.2065573    0.2510515
```

3-3 Wcsp and set partitioning problem

3-3-1. One feature bipartite case

S-PLUS has a built-in function napsack. It gives the subset of component of a vector, on which the sum is near to the given value. According to the manual, we use knapsack to get subset of America's states whose sum of the Area is the half of the total. Here, built-in data state.x77 is used.

```
state.area <- state.x77[,"Area"]  # extract the area vector
state.half <- napsack(state.area, sum(state.area)/2) # invoke napsack
```

napsack returns multiple set of solution (vector of logical value that determines the choice). We get the first one.

```
x1 <- state.half[,1]  # Get the first solution
sum(state.area[x1])   # sum of the selected component
```

gives below.

```
[1] 1768397
```

And

```
sum(state.area[!x1])  # sum of the non-selected component
```

gives exactly the same result.

```
[1] 1768397
```

We now have an optimal solution.

This is written as a sort of Descrete Optimization Problem.

Variables $\delta_s \in \{0,1\}$ ($s \in S$:In the subset or not)

Constraint $\sum_{s\in S} a_s \cdot \delta_s = \sum_{s\in S} a_s \cdot (1-\delta_s)$

a_s :Area of $s \in S$

We have no objective. Strictly speaking, this is a Constraint Satisfaction Problem. To model this by S+NUOPT, we define a procedure named Half.

```
Half <- function()
{
    S <- Set()
```

```
    Cat <- Set()
    Data <- Parameter(index=dprod(S,Cat),state.x77)
    delta <- IntegerVariable(index=S,type=binary)
    s <- Element(set=S)
    Sum(Data[s,"Area"]*delta[s],s) == Sum(Data[s,"Area"]*(1-delta[s]),s)
}
```

We can solve this.

```
sys <- System(Half)    # expand
sol <- solve(sys)      # solve
```

It is solved without a second. Get the solution and confirm.

```
delta <- as.array(current(sys,delta))
sum(state.x77[,"Area"][delta==1])
sum(state.x77[,"Area"][delta==0])
```

We see NUOPT also have the optimal solution.

```
> sum(state.x77[,"Area"][delta==1])
[1] 1768397
> sum(state.x77[,"Area"][delta==0])
[1] 1768397
```

3-3-2. 3 features bi-partite case

In this section we step forward to more general case, we seek subset 3 features. We use data state.x77, a collection of statistics (see below).

state.x77

		1 Population	2 Income	3 Illiteracy	4 Life.Exp	5 Murder	6 HS.Grad	7 Frost	8 Area
1	Alabama	3615.00	3624.00	2.10	69.05	15.10	41.30	20.00	50708.0
2	Alaska	365.00	6315.00	1.50	69.31	11.30	66.70	152.00	566432.0
3	Arizona	2212.00	4530.00	1.80	70.55	7.80	58.10	15.00	113417.0
4	Arkansas	2110.00	3378.00	1.90	70.66	10.10	39.90	65.00	51945.0
5	California	21198.00	5114.00	1.10	71.71	10.30	62.60	20.00	156361.0
6	Colorado	2541.00	4884.00	0.70	72.06	6.80	63.90	166.00	103766.0
7	Connecticut	3100.00	5348.00	1.10	72.48	3.10	56.00	139.00	4862.0
8	Delaware	579.00	4809.00	0.90	70.06	6.20	54.60	103.00	1982.0
9	Florida	8277.00	4815.00	1.30	70.66	10.70	52.60	11.00	54090.0
10	Georgia	4931.00	4091.00	2.00	68.54	13.90	40.60	60.00	58073.0
11	Hawaii	868.00	4963.00	1.90	73.60	6.20	61.90	0.00	6425.0
12	Idaho	813.00	4119.00	0.60	71.87	5.30	59.50	126.00	82677.0
13	Illinois	11197.00	5107.00	0.90	70.14	10.30	52.60	127.00	55748.0
14	Indiana	5313.00	4458.00	0.70	70.88	7.10	52.90	122.00	36097.0
15	Iowa	2861.00	4628.00	0.50	72.56	2.30	59.00	140.00	55941.0

. . .

Among them, we choose Area, Population, Illiteracy. We expand the formulation of the previous section.

Variables $\delta_s \in \{0,1\}$ ($s \in S$:In the subset or not)

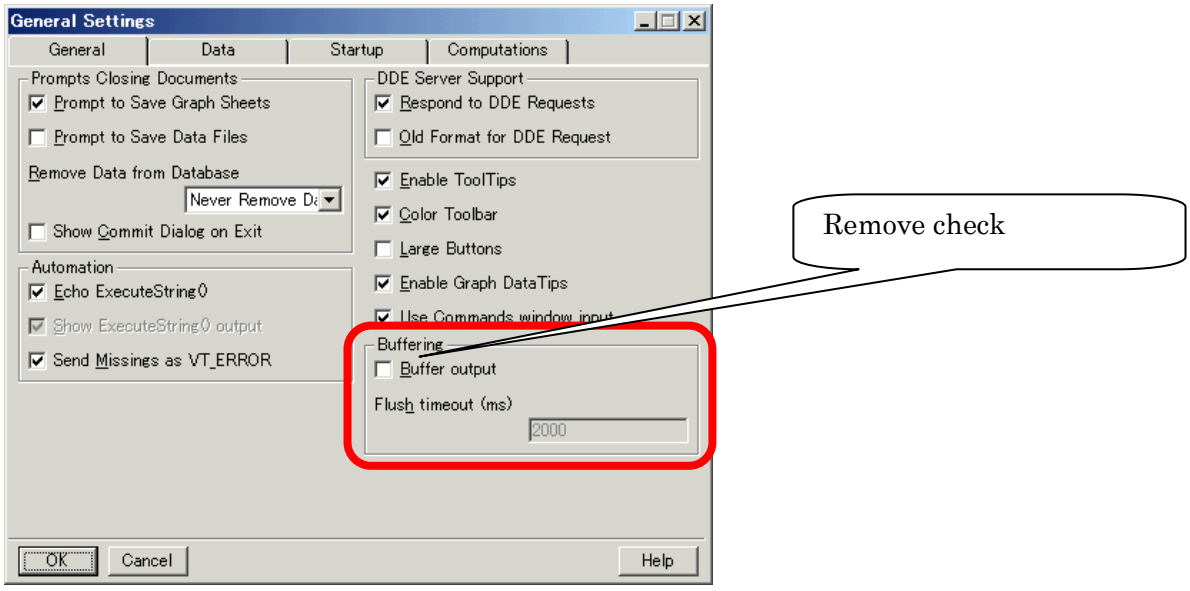
Constraint $\sum_{s\in S} f_s^i \cdot \delta_s = \sum_{s\in S} f_s^i \cdot (1-\delta_s),\ i \in F$

f_s^i :feature(Statistics) $i \in F$ of state $s \in S$

Below is the corresponding model definition.

```
Half2.infs <- function()
{
    S <- Set()
    Cat <- Set()
    Data <- Parameter(index=dprod(S,Cat),state.x77)
    delta <- IntegerVariable(index=S,type=binary)
    s <- Element(set=S)
    f <- Objective(type=minimize)
    Sum(Data[s,"Area"]*delta[s],s) == Sum(Data[s,"Area"]*(1-delta[s]),s)
    Sum(Data[s,"Population"]*delta[s],s)                                    ==
Sum(Data[s,"Population"]*(1-delta[s]),s)
    Sum(Data[s,"Illiteracy"]*delta[s],s)                                    ==
Sum(Data[s,"Illiteracy"]*(1-delta[s]),s)
}
```

Before starting the calculation, we recommend you to set the S-PLUS option to be output buffering OFF from the “Options” menu. This is to watch the course of the calculation properly.



Now give NUOPT 5 seconds to solve this problem.

```
sys <- System(Half2.infs)
nuopt.options(maxtim=5)
sol <- solve(sys)
```

But it reports an error that no solution is found.

```
<<NUOPT 22>> B & B itr. timeout (no feas.sol.).
```

It is because the **no exact solution exist** that half the 3 features (Area, Population, Illiteracy) at the same time, and NUOPT takes quite a much time to find out there's no solution.
But this output is far from what we want. In such a case, we want to solve this problem **approximately**. If the exact solution cannot be obtained, we want best possible solution. Optimization method "wcsp" is useful for such a case. It solves discrete optimization approximately. Below is the model description for wcsp.

The following is the 3 feature bi-partite model for wcsp.

```
Half2 <- function()
{
    S <- Set()
    Cat <- Set()
    Data <- Parameter(index=dprod(S,Cat),state.x77)
    delta <- IntegerVariable(index=S,type=binary)
    s <- Element(set=S)
    softConstraint(1)
    Sum(Data[s,"Area"]*delta[s],s) == Sum(Data[s,"Area"]*(1-delta[s]),s)
    Sum(Data[s,"Population"]*delta[s],s)                                    ==
Sum(Data[s,"Population"]*(1-delta[s]),s)
    1000*Sum(Data[s,"Illiteracy"]*delta[s],s)                               ==
1000*Sum(Data[s,"Illiteracy"]*(1-delta[s]),s)
}
```

The statement softConstraint(1) is specification for "wcsp", that can treat some constraints as "soft" constraints. Comparing to ordinary constraints ("Hard" constraints), "Soft" constraints are less important whose violations are subject to be minimized. The argument of softConstraint (in this case, "1") is the weight parameter multiplied to the amount of violation. If you set large weight (say "100" comparing to "1") to a certain constraint, wcsp regard its violation more serious and try to fulfill it by priority.
The last constraint is multiplied by 1000. This is to set the scale of this constraint approximately the same as the other. Note that method "wcsp" round constraint violation to the nearest integer, so you should scale-up small scaled constraint.
Now invoke the algorithm "wcsp".

```
sys2 <- System(Half2)
nuopt.options(method="wcsp")
sol2 <- solve(sys2)
nuopt.options(nuopt.options(NA))    # unset the option to the default
```

NUOPT will output the following.

```
NUOPT 11.1.9a (NLP/LP/IP/SDP module)
       <with META-HEURISTICS engine "wcsp"/"rcpsp">
```

```
      , Copyright (C) 1991-2009 Mathematical Systems Inc.
wcspMain: Invoke wcsp with NO TIME LIMIT. Set default maxtim = 5 .
wcspMain: CONSIDER SETTING nuopt.options(maxtim=<appropriate time>)
NUMBER_OF_VARIABLES                                          50
(#INTEGER/DISCRETE)                                          50
NUMBER_OF_FUNCTIONS                                           3
PROBLEM_TYPE                                        MINIMIZATION
METHOD                                                      WCSP
<preprocess begin>...<preprocess end>
preprocessing time= 0(s)
<iteration begin>
# random seed = 1
(hard/soft) penalty= 0/986245, time= 0(s)
<greedyupdate begin>.........<greedyupdate end>
greedyupdate time= 0(s)
--- End Phase-I iteration ---
(hard/soft) penalty= 0/61443, time= 0.016(s), iteration= 1
(hard/soft) penalty= 0/60119, time= 0.016(s), iteration= 2
(hard/soft) penalty= 0/39847, time= 0.031(s), iteration= 15
(hard/soft) penalty= 0/34401, time= 0.031(s), iteration= 16
(hard/soft) penalty= 0/31565, time= 0.031(s), iteration= 17
(hard/soft) penalty= 0/16107, time= 0.031(s), iteration= 231
(hard/soft) penalty= 0/13653, time= 0.047(s), iteration= 233
(hard/soft) penalty= 0/6051, time= 0.047(s), iteration= 235
(hard/soft) penalty= 0/4421, time= 0.047(s), iteration= 238
(hard/soft) penalty= 0/4203, time= 0.063(s), iteration= 626
(hard/soft) penalty= 0/3833, time= 0.063(s), iteration= 627
(hard/soft) penalty= 0/2727, time= 0.063(s), iteration= 628
(hard/soft) penalty= 0/1473, time= 0.063(s), iteration= 690
(hard/soft) penalty= 0/1073, time= 0.078(s), iteration= 899
(hard/soft) penalty= 0/991, time= 0.141(s), iteration= 3850
# (hard/soft) penalty= 0/991
# cpu time = 0.141/5.016(s)
# iteration = 3850/292808
<iteration end>
STATUS                                                     NORMAL
TERMINATE_REASON                          End_by_CPU_time_limit
ITERATION_COUNT                                           292808
PENALTY                                                      991
RANDOM_SEED                                                    1
ELAPSED_TIME(sec.)                                          5.03
> nuopt.options(nuopt.options(NA))
```

The output “penalty=...” is decreasing as the algorithm searching. At the default, it continue searching 5 second and stop. In this case, a solution with penalty “991” is found. Extract the solution to check the result.

```
delta2 <- as.array(current(sys2,delta))
sum(state.x77[,"Area"][delta2==1])
sum(state.x77[,"Area"][delta2==0])
sum(state.x77[,"Population"][delta2==1])
sum(state.x77[,"Population"][delta2==0])
sum(state.x77[,"Illiteracy"][delta2==1])
sum(state.x77[,"Illiteracy"][delta2==0])
```

The output is below. We can confirm almost the desired partition is obtained.

```
> sum(state.x77[,"Area"][delta2==1])
[1] 1768097
> sum(state.x77[,"Area"][delta2==0])
[1] 1768697
> sum(state.x77[,"Population"][delta2==1])
[1] 106306
> sum(state.x77[,"Population"][delta2==0])
[1] 106015
> sum(state.x77[,"Illiteracy"][delta2==1])
[1] 29.2
> sum(state.x77[,"Illiteracy"][delta2==0])
[1] 29.3
```

This algorithm has no way to judge the optimality of the current solution, that is why there's a default timelimit (5 seconds). In this case the output shows this solver found the best solution less than a second:

```
(hard/soft) penalty= 0/991, time= 0.125(s), iteration= 3850
```

You can tune the calculation time, seeing the searching process and quality of the solution. Typically, the penalty rapidly decreases for first iterations, and saturates and stops. If the penalty is still decreasing after the default 5 seconds, set the time longer like:

```
options.method(method=" wcsp" ,maxtim=60)  # give wcsp 60 seconds
```

At the expense of the optimality of the solution, "wcsp" find a moderately good feasible solution around 60 seconds for typical large scale optimization problem. If the solution is unsatisfactory giving long time, consider changing the random seed (see 3-3-4 for details).
Note that wcsp" can only handle binary integer variable, we recommend to reset the option by calling

```
nuopt.options(nuopt.options(NA))     # unset the option to the default
```

after specifying "wcsp" as method, otherwise you got many errors for ordinal optimization models.

3-3-3. N-partition with wcsp (selection and hard/semi-hard/soft constraints)

Below is the formulation of N-partition problem. We seek partition whose some of Area is almost same.

Variables $u_{s,k} \in \{0,1\}$ ($s \in S$, $k \in K$:

$u_{s,k} = 1 \Leftrightarrow s \in S$ is in group $k \in K$)

Hard Constraint

$$\sum_{k \in K} u_{s,k} = 1, \quad s \in S \qquad \text{(every } s \in S \text{ in one group)}$$

Soft Constraint

$$\sum_{s \in S} a_s \cdot u_{s,k} = \frac{1}{|K|} \sum_{s \in S} a_s$$

a_s:Area of $s \in S$

The following is the corresponding model description.

```
Partition <- function(N)
{
    S <- Set()
    Cat <- Set()
    Data <- Parameter(index=dprod(S,Cat),state.x77)
    K <- Set(1:N)
    u <- IntegerVariable(index=dprod(S,K),type=binary)
    k <- Element(set=K)
    s <- Element(set=S)
    hardConstraint()
    selection(u[s,k],k)
    softConstraint(1)
    Sum(Data[s,"Area"]*u[s,k],s) == Sum(Data[s,"Area"],s)/N
    dst <- Expression(index=s)
    knum <- Parameter(index=k)
    knum[k] ~ k
    dst[s] ~ Sum(knum[k]*u[s,k],k)
}
```

Subsequent constraints are hard constraint.

Equiavalent to Sum(u[s,k],k) == 1 faster for wcsp.

Soft constraints will begin

dst[s] holds the destination (group number) of s

hardConstraint() declares the subsequent constraints have first priority. The format of the declarations of this sort is as below. These declarations can be called anywhere, any number of times.

```
softConstraint(integer larger than 1)   # beginning of soft constraints
semiHardConstraint()  # beginning of semi-hard constraints
hardConstraint()    # beginning of hard constraints
```

semiHardConstraint() are avaiable to declare the subsequent ones are of second priority. This class of constraint is a sort of hard constraints to which you can impose infeasibility to make the solution being informative.

selection()statement defines constraint, sum of the binary variables to be 1. It is treated always as a hard constraint, and effective to speed-up the solution searching processes.

Now we solve 5-partition.

```
sys <- System(Partition,5)
nuopt.options(method="wcsp",maxtim=1)
sol <- solve(sys)
nuopt.options(nuopt.options(NA)) # set the options to default
```

We extract Expression"dst"and calculate the sum for each group.

```
dst <- as.array(current(sys,dst))
state.area <- state.x77[," Area" ]
for(i in 1:5) print(sum(state.area[dst==i]))
```

Output is as follows.

```
> for(i in 1:5) print(sum(state.area[dst==i]))
[1] 709978
[1] 705870
[1] 706540
[1] 707348
[1] 707058
```

The result is almost satisfactory. See the partition itself.

```
for(i in 1:5) print(state.area[dst==i])
```

```
> for(i in 1:5) print(state.area[dst==i])
 Alaska Colorado Virginia
 566432   103766    39780
 Alabama California Florida Hawaii Indiana Kentucky Mississippi
   50708     156361   54090   6425   36097    39650       47296

 New Jersey New Mexico North Carolina Oregon Tennessee
       7521     121412          48798  96184     41328
 Delaware Idaho Illinois  Iowa Kansas Maine Michigan  Texas
     1982 82677    55748 55941  81787 30920    56817 262134

 West Virginia Wisconsin
         24070     54464
 Arizona Georgia Massachusetts Nebraska New York  Ohio Oklahoma
  113417   58073          7826    76483    47831 40975    68782

 Pennsylvania South Dakota Vermont Washington Wyoming
        44966        75955    9267      66570   97203
 Arkansas Connecticut Louisiana Maryland Minnesota Missouri Montana
    51945        4862     44930     9891     79289    68995  145587
```

```
   Nevada New Hampshire North Dakota Rhode Island South Carolina  Utah
   109889          9027        69273         1049          30225 82096
```

3-3-4.Setting the random seed

Algorithm wcsp uses random values for solution seeking process. Consequently, if you set a different seed, you'll obtain a different solution. Let us change random seed to 4 (the default is 1).

```
nuopt.options(method="wcsp",maxtim=1,wcspRandomSeed=4)
sol <- solve(sys)
dst <- as.array(current(sys,dst))
for(i in 1:5) print(sum(state.area[dst==i]))
nuopt.options(nuopt.options(NA)) # set the options to default
```

We obtain a different solution.

```
> for(i in 1:5) print(sum(state.area[dst==i]))
[1] 707621
[1] 709041
[1] 706960
[1] 706429
[1] 706743
```

By setting parameter wcspTryCount, wcsp automatically repeats the seeking process by changing seed value by 1, from the value set by wcspRandomSeed. The following does 5 tries and output the best.

```
nuopt.options(method="wcsp",maxtim=1,wcspRandomSeed=1,wcspTryCount=5)
```

3-3-5. Restriction/Special features of algorithm wcsp

Algorithm "wcsp" is a method outstandingly different from others implemented in S+NUOPT. Here is a summary of the restriction and special features.

- It can handle only binary integer variables.
- After invoking, reset the options by nuopt.options (nuopt.options(NA)).
- No exact solution is guaranteed to be obtained.
- You have to tune limit time (5 seconds by default).
- Three class of constraint (soft/semi-hard/hard) exist.
- Seed of the random value will take an effect of the result.
- Giving wcspTryCount automatically change seed value and repeat.
- Constraint violation (objective) is always rounded to integral value.

4. S-PLUS and S+NUOPT data conversion

S-PLUS data (vector, matrix, array and list) can be used as the initializer of the optimization modeling language objects (mainly Set and Parameter). We can also extract optimization result (Variable and Expression) as S-PLUS data. In this chapter, we show the way through the example.
The best way is to confirm the conversion result from the command prompt. By creating Parameter or Set from the command prompt. In the subsequent section, we assume S+NUOPT is installed and loaded by typing

```
module(nuopt)
```

4-1 Initilize Parameter by S-PLUS vector

The most basic S-PLUS data type is vector. S-PLUS does not distinguish scalar and vector explicitly and both are expressed by vector. Consequently, you have to convert a vector to an array when you try to initialize indexed Parameters.

4-1-1. Initialize Parameter without index

In the following, Parameter objects with no indices (p and q) are initialized by scalar value.

```
p <- Parameter(3)      # Initialize by 3
p                       # confirm
b <- 3^5                # b is a S-PLUS object
q <- Parameter(b)      # Initialize q by b
q                       # confirm
```

Now p,q is initialized.

```
> p
[1] 3
> q
[1] 243
```

4-1-2. Initialize Parameter with one index.

In S-PLUS, vector typed object can also express scalar (scalar is a special vector with length 1).In S+NUOPT, scalar (Parameter with no index) and vector (Parameter with one index) is strictly distinguished at the declaration and cannot be exchanged. Consequently, indexed Parameter can only be initialized by S-PLUS array. To initialize indexed Parameter by S-PLUS vector, you have to convert it by as.array() procedure.
Following is the example. First, built-in data “freeny” is converted to vector.
freeny.y is a time series: (A year is partitioned by quarter, it is 1962/2Q to 1971/4Q)

```
> freeny.y
          1Q       2Q       3Q       4Q
1962:           8.79236  8.79137  8.81486
1963: 8.81301  8.90751  8.93673  8.96161
1964: 8.96044  9.00868  9.03049  9.06906
1965: 9.05871  9.10698  9.12685  9.17096
```

```
1966: 9.18665 9.23823 9.26487 9.28436
1967: 9.31378 9.35025 9.35835 9.39767
1968: 9.42150 9.44223 9.48721 9.52374
1969: 9.53980 9.58123 9.60048 9.64496
1970: 9.64390 9.69405 9.69958 9.68683
1971: 9.71774 9.74924 9.77536 9.79424
```

Convert to vector type.

```
ydata <- as.vector(freeny.y)
ydata
length(ydata)
```

The result.

```
> ydata
 [1] 8.79236 8.79137 8.81486 8.81301 8.90751 8.93673 8.96161 8.96044 9.00868
9.03049 9.06906
[12] 9.05871 9.10698 9.12685 9.17096 9.18665 9.23823 9.26487 9.28436 9.31378
9.35025 9.35835
[23] 9.39767 9.42150 9.44223 9.48721 9.52374 9.53980 9.58123 9.60048 9.64496
9.64390 9.69405
[34] 9.69958 9.68683 9.71774 9.74924 9.77536 9.79424
> length(ydata)
[1] 39
```

We initialize Parameter object Y as below.

```
S <- Set()   # Prepare Set object of the index
Y <- Parameter(index=S,as.array(ydata))  # initialize Y as.array(ydata)
Y
S
```

We first define Set object S that contains index and give it to Y as the initialization. Y is initialized by as.array(ydata), not that S-PLUS vector ydata is not directly passed.

Now confirm Y by printing

```
> Y
       1        2        3        4        5        6        7        8        9
10       11       12
 8.79236 8.79137 8.81486 8.81301 8.90751 8.93673 8.96161 8.96044 9.00868 9.03049
9.06906 9.05871

      13       14       15       16       17       18       19       20       21
22       23       24
 9.10698 9.12685 9.17096 9.18665 9.23823 9.26487 9.28436 9.31378 9.35025 9.35835
9.39767 9.4215
```

```
      25       26       27       28       29       30       31       32       33
34       35       36
 9.44223 9.48721 9.52374 9.5398 9.58123 9.60048 9.64496 9.6439 9.69405 9.69958
9.68683 9.71774

      37       38       39
 9.74924 9.77536 9.79424
attr(, "indexes"):
[1] "*"
```

Both index (1,2,3,...) and contents (8.79236,8.79137,8.81486,...) are shown. Index set S is also initialized.

```
> S
{ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 }
```

Y has index "1,2,3,...,39". You can pass any of vector objects by converting by as.array().
The following explains how to initialize Parameter r by vector "seq(1,10,3)".

```
s <- seq(1,10,3)        # s is S-PLUS vector
D <- Set()              # D is r' s index set
r <- Parameter(index=D,as.array(s))     # create r
r
D
```

r and D are output as below.

```
> r
 1 2 3  4
 1 4 7 10
attr(, "indexes"):
[1] "*"
> D
{ 1 2 3 4 }
```

4-1-3.Parameter whose Index set is not a sequence

The index set can be defined other than integer sequence1,2,3,… Give vector's names attribute and convert it to array. The following, we give names attribute c("a","b","c","d") to vector s.

```
s <- seq(1,10,3)                        # s is S-PLUS vector
names(s) <- c("a","b","c","d")  # give s names atribute a,b,c,d
D <- Set()
r <- Parameter(index=D,as.array(s))     # create r
s
r
```

```
D
```

Consequently Parameter r's index set is defined to be a,b,c,d.

```
> s
 a b c  d
 1 4 7 10
> r
 a b c  d
 1 4 7 10
attr(, "indexes"):
[1] "*"
> D
{ a b c d }
```

In this way, you can give any sequence of string as the index of Parameter objects.

4-2 Initialize Parameter by S-PLUS matrix

S-PLUS matrix is derived from array type. That is why S-PLUS matrix can initialize Parameter without conversion. The following is the example. Built-in data freeny.x is an S-PLUS matrix object.

```
> freeny.x
numeric matrix: 39 rows, 4 columns.
      lag quarterly revenue price index income level market potential
 [1,]                8.79636      4.70997       5.82110           12.9699
 [2,]                8.79236      4.70217       5.82558           12.9733
 [3,]                8.79137      4.68944       5.83112           12.9774
 [4,]                8.81486      4.68558       5.84046           12.9806
 [5,]                8.81301      4.64019       5.85036           12.9831
 [6,]                8.90751      4.62553       5.86464           12.9854
 [7,]                8.93673      4.61991       5.87769           12.9900
                    ......
[35,]                9.69958      4.30909       6.16135           13.1520
[36,]                9.68683      4.30552       6.18231           13.1593
[37,]                9.71774      4.29627       6.18768           13.1579
[38,]                9.74924      4.27839       6.19377           13.1625
[39,]                9.77536      4.27789       6.20030           13.1664
>
attr(, "indexes"):
[1] "*"
```

In the following, Parameter A is initialized directly by freeny.x. The index sets are S and P.

```
S <- Set()      # row index Set
P <- Set()      # column index Set
A <- Parameter(index=dprod(S,P),freeny.x)
```

```
S
P
A
```

S and P are shown below.

```
> S
{ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 2
7 28 29 30 31 32 33 34 35 36 37 38 39 }
> P
{ lag quarterly revenue price index income level market potential }
```

The below correspond freeny.x's row and column indecies. Now check.

```
rownames(freeny.x)              # output row index of freeny.x
colnames(freeny.x)              # output column index of freeny.x
```

The output follows like this.

```
> rownames(freeny.x)        # row index
NULL
> colnames(freeny.x)        # column index
[1] "lag quarterly revenue" "price index"
[3] "income level"          "market potential"
```

Because the row index is not defined, NUOPT supply integral sequence 1,2,3,…,39 as the index. A is printed as below.

```
> A
   income level lag quarterly revenue market potential price index
 1      5.82110               8.79636          12.9699     4.70997
 2      5.82558               8.79236          12.9733     4.70217
 3      5.83112               8.79137          12.9774     4.68944
 4      5.84046               8.81486          12.9806     4.68558
 5      5.85036               8.81301          12.9831     4.64019
                             ......
37      6.18768               9.71774          13.1579     4.29627
38      6.19377               9.74924          13.1625     4.27839
39      6.20030               9.77536          13.1664     4.27789
attr(, "indexes"):
[1] "*" "*"
```

4-3 Initialize Parameter by S-PLUS list

You can initialize Parameter by S-PLUS list object. The first length(list)-1 items are interpreted as index vectors, the last one is interpreted as the value vector.
Below is the example to initialize Parameter with one index by a list.

```
list1 <- list(c("a","b","c"),1:3)
```

```
S <- Set()
a <- Parameter(index=S,list1)
a
```

a is printed as below.

```
> a
 a b c
 1 2 3
attr(, "indexes"):
[1] "*"
```

Similary, Parameter with two indicies is initialized as below.

```
list2 <- list(c("a","a","b","b"),c(1,2,1,2),7:10)
S1 <- Set()
S2 <- Set()
b <- Parameter(index=dprod(S1,S2),list2)
b
```

b is printed as below.

```
> b
  1  2
a 7  8
b 9 10
attr(, "indexes"):
[1] "*" "*"
```

Below is the three indicies case.

```
list3                                                                <-
list(c(1,1,1,2,2,2),c("a","a","b","b","c","c"),c("A","A","A","B","B","B"
),1:6)
S1 <- Set()
S2 <- Set()
S3 <- Set()
d <- Parameter(index=dprod(S1,S2,S3),list3)
d
```

d is printed as below.

```
> d
, , A
  a b c
1 2 3 0
2 0 0 0
```

```
, , B
  a b c
1 0 0 0
2 0 4 6
attr(, "indexes"):
[1] "*" "*" "*"
```

4-4 Extract the optimization result (Variable/Expression)

You can define an optimization model by using Parameters defined above.
Optimization model description Nlsfit (see below) contains data initialization.

```
Nlsfit <- function()
{
#  data definition/initialization
   ydata <- as.vector(freeny.y)
   S <- Set()
   Y <- Parameter(index=S,as.array(ydata))
   P <- Set()
   X <- Parameter(index=dprod(S,P),freeny.x)
 # body of optimization model
   i <- Element(set=S)
   j <- Element(set=P)
   v <- Variable(index=j)
   v0 <- Variable()
   e <- Expression(index=i)
   e[i] ~ v0 + Sum(X[i,j]*v[j],j) - Y[i]
   esum <- Objective(type=minimize)
   esum ~ Sum(e[i]*e[i],i)
}
```

You can construct the System by expanding the model, and solve.

```
sys <- System(Nlsfit) # expand the model
sol <- solve(sys)     # solve
```

More generic model Nlsfit.gen is defined by getting data source from the argument.

```
Nlsfit.gen <- function(x,y)
# x (matrix) is explaining, y(vector) is explained variable
{
   S <- Set()
   Y <- Parameter(index=S,as.array(y))
   P <- Set()
   X <- Parameter(index=dprod(S,P),x)
   i <- Element(set=S)
   j <- Element(set=P)
   v <- Variable(index=j)
```

```
    v0 <- Variable()
    e <- Expression(index=i)
    e[i] ~ Sum(X[i,j]*v[j],j) + v0 - Y[i]
    esum <- Objective(type=minimize)
    esum ~ Sum(err[i]*err[i],i)
}
```

Now you bind the data to the model at the expansion, and solve.

```
sys <- System(Nlsfit.gen,freeny.x,freeny.y)  # bind freeny.x,freeny.y
sol <- solve(sys) # solve
```

4-4-1.Extract Variable/Expression asS-PLUS array

After solving the System, you can print the value of Variable/Expression by current() function.

```
sys <- System(Nlsfit.gen,freeny.x,freeny.y)
sol <- solve(sys,trace=F)
current(sys,v0)    # print  “v0”  inside sys
current(sys,v)     # print   “v”  inside sys
```

The current value of v0 and v are output as below.

```
> current(sys,v0)
[1] -10.47261
> current(sys,v)
 income level lag quarterly revenue market potential price index
    0.7674609             0.1238646         1.330558  -0.7542401
attr(, "indexes"):
[1] "j"
```

But the returned value of current() is just the reference S+NUOPT of object. You have to convert them to S-PLUS array object to allow manipulation in S-PLUS environment.

```
v0 <- as.array(current(sys,v0))
v <- as.array(current(sys,v))
```

Note that you can **not use as.vector directly** to the returned reference from current(). You got meaningless result “0”. If you want vector, you have to convert them to array by as.array() once.

Next code will generate vector “vvec” that concatenate the optimization result v0 and v. We supply index information afterwards. Because as.vector () miss them.

```
v0 <- as.array(current(sys,v0))
v <- as.array(current(sys,v))
vvec <- as.vector(v)              # convert to vector
names(vvec) <- rownames(v)        # supply index information
```

```
vvec <- c(Intercept=v0, vvec)      # concatenate v0 (name "Intercept")
vvec
```

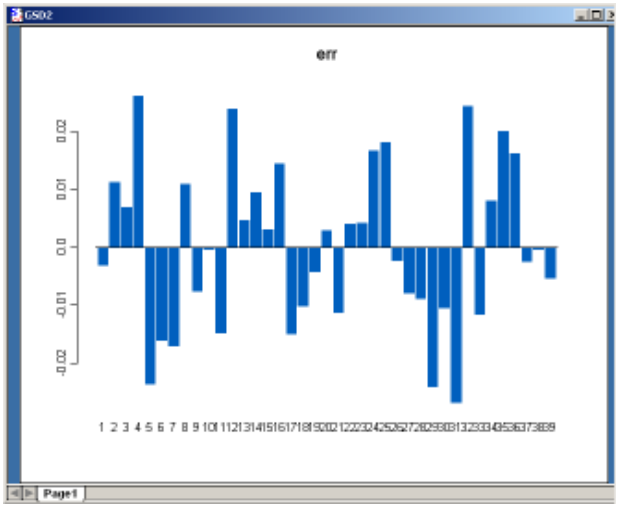
vvec is printed as below

```
> vvec
 Intercept income level lag quarterly revenue market potential price index
 -10.47261     0.7674609              0.1238646          1.330558  -0.7542401
```

Error vector 'e' is extracted just as v and v0.

```
err <- as.array(current(sys, e))
barplot(err, names=rownames(err))  # to draw bar chart
```

You can see a barchart appears as below.



4-4-2. Extract Variable/Expression as S-PLUS list.

as.list() can convert any S+NUOPT specific object to S-PLUS list. The conversion rule is the same as using the list as input, namely, the first length(list)-1 items are indices, the last is the value. In the example below, object err (Expression) is converted to a list.

```
sys <- System(Nlsfit.gen, freeny.x, freeny.y)
sol <- solve(sys)
err <- as.list(current(sys, e))
err
```

```
> err
$indexes:
$i:
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
[23] 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39

$values:
 [1] -0.0031897999  0.0111499356  0.0068458061  0.0259426225
```

```
 [5] -0.0236273123 -0.0160653218 -0.0169512378  0.0108668721
 [9] -0.0075620169 -0.0003823201 -0.0147084654  0.0237528911
[13]  0.0045025693  0.0093397389  0.0029600214  0.0142621487
[17] -0.0149334126 -0.0102146132 -0.0042357193  0.0028433516
[21] -0.0113535030  0.0039590091  0.0041339447  0.0165713365
[25]  0.0180124412 -0.0022507261 -0.0078694559 -0.0089016807
[29] -0.0241231336 -0.0104325444 -0.0267123682  0.0242586554
[33] -0.0115643780  0.0079192007  0.0200001064  0.0160025223
[37] -0.0024336174 -0.0003716668 -0.0054385080
```

This is the case of Parameter with two indices.

```
S <- Set()
P <- Set()
X <- Parameter(index=dprod(S,P),freeny.x)
Xlist <- as.list(X)
Xlist
```

Xlist is printed as below.

```
> Xlist
$indexes:
$"*":
  [1] "1" "1" "1" "1" "2" "2" "2" "2" "3" "3" "3" "3" "4"
 [14] "4" "4" "4" "5" "5" "5" "5" "6" "6" "6" "6" "7" "7"

              ......
$"*":
 [1] "income level"          "lag quarterly revenue"
 [3] "market potential"      "price index"
 [5] "income level"          "lag quarterly revenue"
 [7] "market potential"      "price index"
 [9] "income level"          "lag quarterly revenue"
[11] "market potential"      "price index"
[13] "income level"          "lag quarterly revenue"
[15] "market potential"      "price index"
[17] "income level"          "lag quarterly revenue"
[19] "market potential"      "price index"
              ......
$values:
  [1]  5.82110  8.79636 12.96990  4.70997  5.82558  8.79236 12.97330
  [8]  4.70217  5.83112  8.79137 12.97740  4.68944  5.84046  8.81486
 [15] 12.98060  4.68558  5.85036  8.81301 12.98310  4.64019  5.86464
              ......
```

5. Portofolio optimization

5-1 Markowitz model

The purpose of this chapter is to introduce some portofolio optimization models. We will begin by considering Markowitz model. Mathematical Programming style representations are described below.
Markowitz model:

$$\text{minimize} \quad \sum_{i,j \in Asset} Q_{ij} x_i x_j \qquad \text{(variance of return rates)}$$

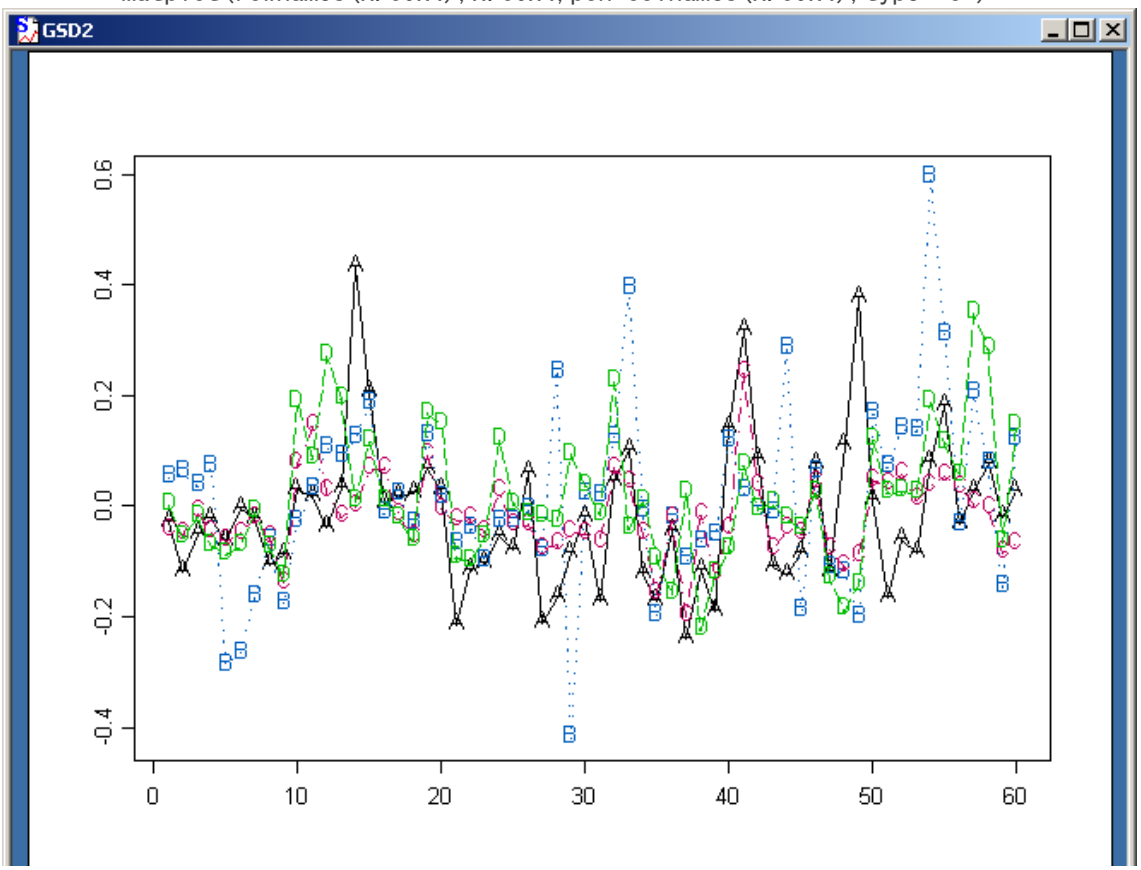
$$\text{subject to} \quad \sum_{j \in Asset} x_j = 1$$

$$\sum_{j \in Asset} \bar{r}_j x_j \geq r_{\min} \qquad \text{(secure minimum return rates)}$$

$$x_j \geq 0, \quad j \in Asset \qquad \text{(short sellings are prohibited)}$$

In our example, $r_{\min}$ is assumed to be 1%.
The sample time-series data is R.60x4(which is a sample data included in S+NUOPT and consists of 4 brands and 60 terms)
We graph out this in the following.

```
# Graph out R.60x4
matplot(rownames(R.60x4), R.60x4, pch=colnames(R.60x4), type="o")
```



Next, we want to know the variance of return rates named Q.

```
Q <- var(R.60x4)
Q
> Q
            A           B           C           D
A 0.017788838 0.005730923 0.004552905 0.003779794
B 0.005730923 0.026978759 0.005153147 0.008770707
C 0.004552905 0.005153147 0.005200091 0.004284923
D 0.003779794 0.008770707 0.004284923 0.014216287
```

In the same way, we get the average of return rates named $\overline{r}$.

```
rbar <- apply(R.60x4,2,mean)
rbar
> rbar
          A         B           C          D
 -0.01483683 0.0147342 -0.01805572 0.01391076
```

Please be careful that vector type objects need to be defined as array types in applying NUOPT.

```
rbar <- as.array(rbar)
```

An optimization model can be defined as a procedure in S-PLUS. The below is a S-PLUS procedure which describe the Markowitz model(But datas have not yet been defined).

```
Marko <- function(Q,rbar,rmin)
{
      Asset <- Set()
      Q <- Parameter(index=dprod(Asset,Asset),Q)
      rbar <- Parameter(index=Asset,rbar)
      i <- Element(set=Asset)
      j <- Element(set=Asset)
      x <- Variable(index=Asset)
      risk <- Objective(type="minimize")
      risk ~ Sum(x[j]*Q[i,j]*x[i],i,j)
      Sum(rbar[j]*x[j],j) >= rmin
      Sum(x[j],j) == 1
      x[j] >= 0
}
```

Loading NUOPT module.

```
> module(nuopt)
NUOPT for S-PLUS Version 2.11.1.9 Release 1 for Microsoft Windows
```

If you want to assign datas to a model, use “System” command.

```
> s <- System(model=Marko,Q,rbar,0.01)
Evaluating Marko(Q,rbar,0.01) ... ok!
Expanding objective  (1/4 name="risk")
Expanding constraint (2/4)
Expanding constraint (3/4)
Expanding constraint (4/4)
```

After assigning datas to a model, a new object(named “s” in the above example) will be produced. You can display the contents in the following.

```
> s
1-1 :  -0.0148368*x[A]+0.0147342*x[B]-0.0180557*x[C]+0.0139108*x[D] >= 0.01

2-1 :  x[A]+x[B]+x[C]+x[D] == 1

3-1 :  x[A] >= 0
3-2 :  x[B] >= 0
3-3 :  x[C] >= 0
3-4 :  x[D] >= 0

risk<objective>: 0.0177888*x[A]*x[A]+0.00573092*x[A]*x[B]+0.00455291*x[A
]*x[C]+0.00377979*x[A]*x[D]+0.00573092*x[A]*x[B]+0.0269788*x[B]*x[B]+0.0
0515315*x[B]*x[C]+0.00877071*x[B]*x[D]+0.00455291*x[A]*x[C]+0.00515315*x
[B]*x[C]+0.00520009*x[C]*x[C]+0.00428492*x[C]*x[D]+0.00377979*x[A]*x[D]+
0.00877071*x[B]*x[D]+0.00428492*x[C]*x[D]+0.0142163*x[D]*x[D] (minimize)
```

We try to solve the optimization problem. To solve this, you have to apply ”solve” function.

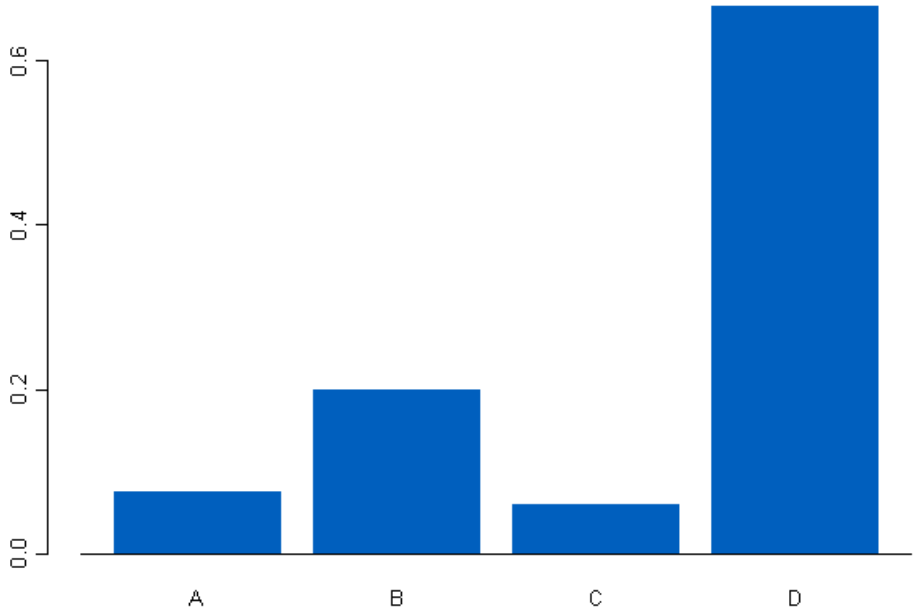
```
> sol <- solve(s)
NUOPT 11.1.9a (NLP/LP/IP/SDP module)
       <with META-HEURISTICS engine "wcsp"/"rcpsp">
      , Copyright (C) 1991-2009 Mathematical Systems Inc.
NUMBER_OF_VARIABLES                                    4
NUMBER_OF_FUNCTIONS                                    3
PROBLEM_TYPE                                  MINIMIZATION
METHOD                                     TRUST_REGION_IPM
<preprocess begin>.....<preprocess end>
<iteration begin>
    res=1.1e+001 .... 1.0e-002 .... 4.3e-008  3.2e-010
<iteration end>
STATUS                                              OPTIMAL
VALUE_OF_OBJECTIVE                         0.01087261448
ITERATION_COUNT                                       11
FUNC_EVAL_COUNT                                       15
FACTORIZATION_COUNT                                   17
RESIDUAL                                3.209329232e-010
ELAPSED_TIME(sec.)                                  0.02
```

The below is a way to know the allocation rates “x”. Command “current” is a function to know current values of a component of the object.

```
> xopt <- as.array(current(s,x))
> xopt
          A         B          C         D
 0.07466462 0.1986515 0.06031033 0.6663735
attr(, "indexes"):
[1] "*"
```

Let us display it as a histogram.

```
> barplot(names=rownames(xopt), xopt)
```



Next, let us plot the time-series return rates named “pf” which correspond to the allocation rates calculated above.

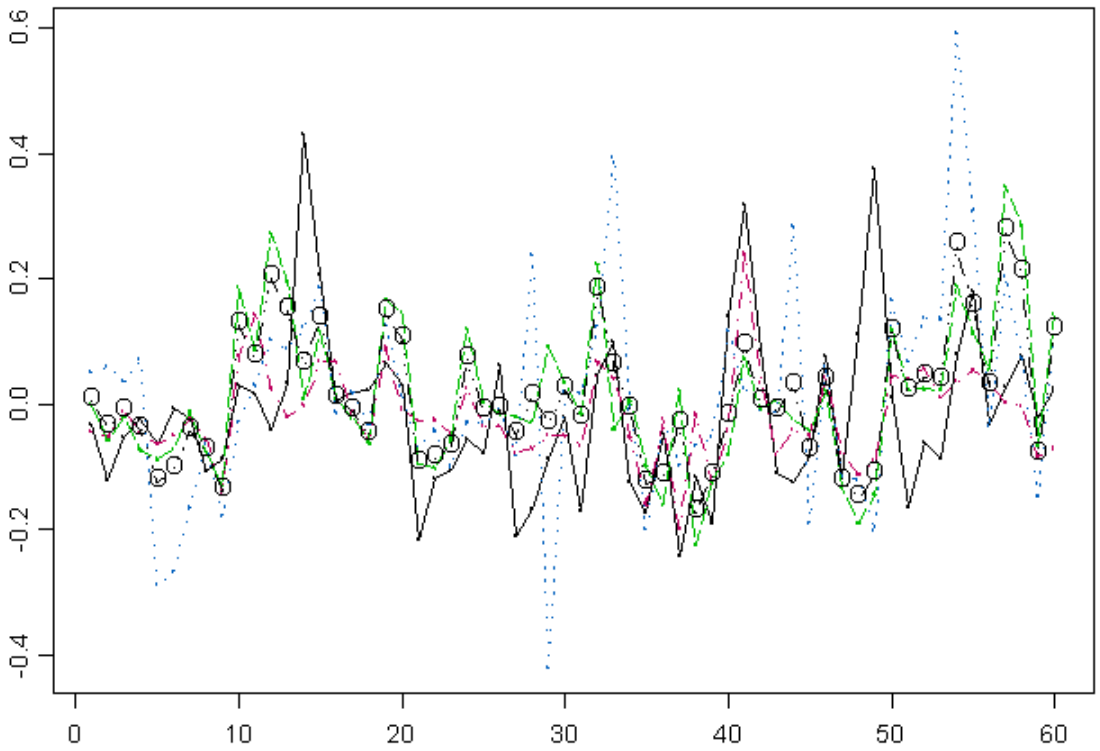
```
> pf <- R.60x4 %*% as.vector(xopt)
```

The variance is almost nearly equal to the VALUE_OF_OBJECTIVE.

```
>var(pf)
             [,1]
[1,] 0.01087261
```

Let us overlap the time-series return rates.

```
> Raug <- cbind(R.60x4, pf)
> matplot(rownames(Raug), Raug, pch=c(".",".",".",".",".","O"), type="o")
```



Plot “0”’s are return rates of the portfolio. Please confirm that the variation of return rates is controlled to be lower.

5-2 Various types of risk estimation in portfolio optimization

In the previous section, we introduce Markowitz model. In Markowitz model, risk is estimated by variance. But, there are various types of risk estimation.
In this section, we take other 3 types of risk estimation model into consideration. In the following 3 models, risk is interpreted as Absolute Deviation, Lower Partial Moments and Conditional Value at Risk. Fortunately, all these models can be described as Linear Programming Problems. Let us apply them to a sample data R.8000x5(consists of 8000 cased and 5 brands)

5-2-1.Variance(Markowitz model)

First, we consider the Markowitz model again to compare. Variance needs to be minimized.

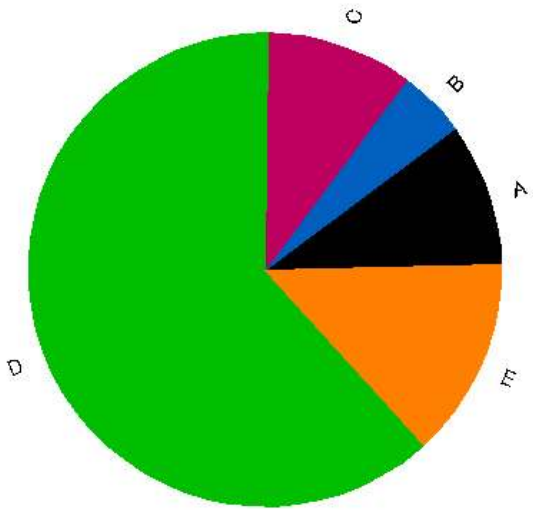
```
# minimize variance
MinVar <- function(r.d)
{
    Asset <- Set()
    j <- Element(set=Asset)
    Sample <- Set()
    t <- Element(set=Sample)
    r <- Parameter(index=dprod(t,j),r.d)
    rb <- Parameter(index=j)
    rb[j] ~ Sum(r[t,j],t)/nrow(r.d)
```

```
    x <- Variable(index=j)
    s <- Variable(index=t)
    f <- Objective(type=minimize)
    f ~ Sum(s[t]*s[t],t)/nrow(r.d)
    Sum(x[j],j) == 1
    x[j] >= 0
    Sum((r[t,j]-rb[j])*x[j],j) == s[t]
}

# Assign datas
sys <- System(MinVar, R.8000x5)
# Optimize
sol <- solve(sys)
# Get solutions
x <- as.array(current(sys,x))
```

Let us analize the result. Described below are the optimized asset allocation rates.

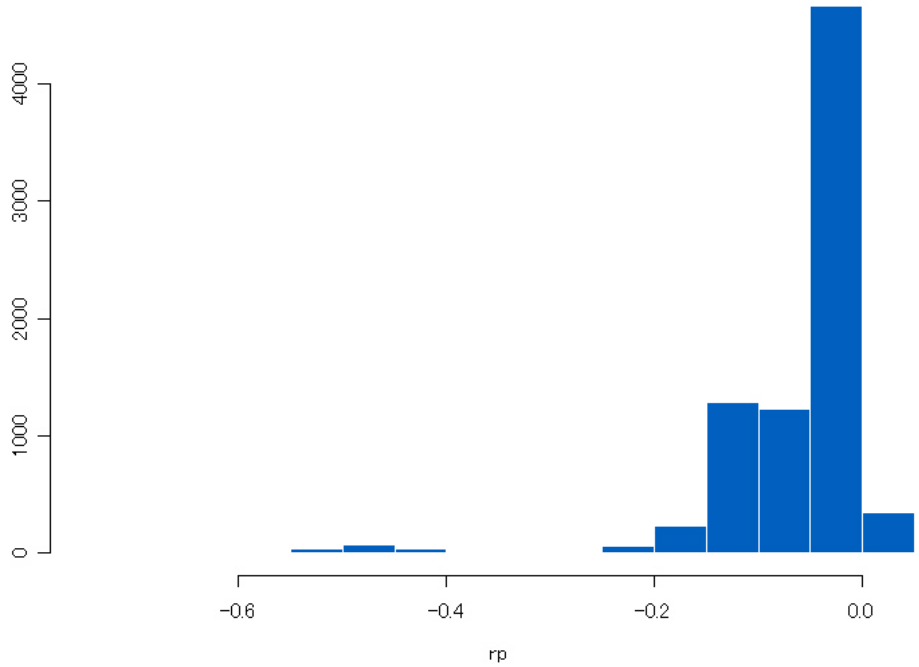
```
# Display
pie(1000*x,names=dimnames(x)[[1]],inner=1.2)
```



Next, let us graph out a histogram of return rates.

```
rp <- as.matrix(R.8000x5) %*% as.vector(x)
graphsheet()
```

```
hist(rp)
```



Please take notice that there are a few bad cases that return rates are from -0.6 to -0.4. These bad cases are interpreted as defaults. If an asset have some default risk, it is sometimes dangerous to apply the Markowitz model simpliy.

5-2-2.Absolute Deviation

In this section, let us solve Mean-Absolute Deviation Model(MAD model). Absolute Deviation needs to be minimized. We take note that function “abs” is not necessary to describe.

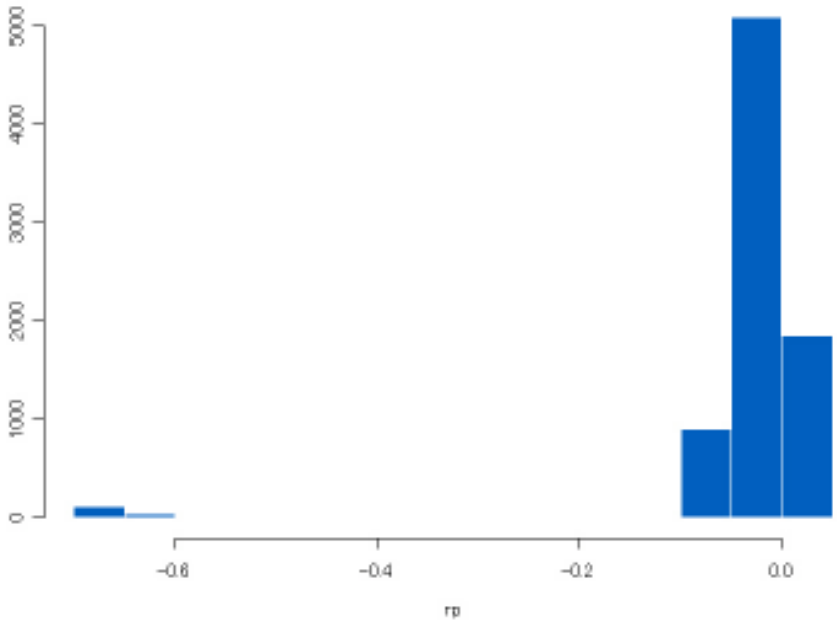
```
#### MAD model ####
MinMad <- function(r.d)
{
    Asset <- Set()
    j <- Element(set=Asset)
    Sample <- Set()
    t <- Element(set=Sample)
    r <- Parameter(index=dprod(t,j),r.d)
    rb <- Parameter(index=j)
    rb[j] ~ Sum(r[t,j],t)/nrow(r.d)
    x <- Variable(index=j)
    s <- Variable(index=t)
    u <- Variable(index=t)
    v <- Variable(index=t)
    f <- Objective(type=minimize)
    f ~ Sum(u[t]+v[t],t)/nrow(r.d)
```

```
    Sum(x[j],j) == 1
    x[j] >= 0
    Sum((r[t,j]-rb[j])*x[j],j) == s[t]
    u[t] >= 0
    v[t] >= 0
    s[t] == u[t] - v[t]
}

# Assign datas
sys <- System(MinMad, R.8000x5)
# Optimize
sol <- solve(sys)
# Get solutions
x <- as.array(current(sys,x))
```

Let us represent a histogram.

```
rp <- as.matrix(R.8000x5) %*% as.vector(x)
hist(rp)
```



MAD model can be formulated as a Linear Programming Problem, and can deal with large datas. This is an advantage of the model, but the drawback is the sensitivity of deviations. Please remark that return rates are under -0.6 in a few cases.

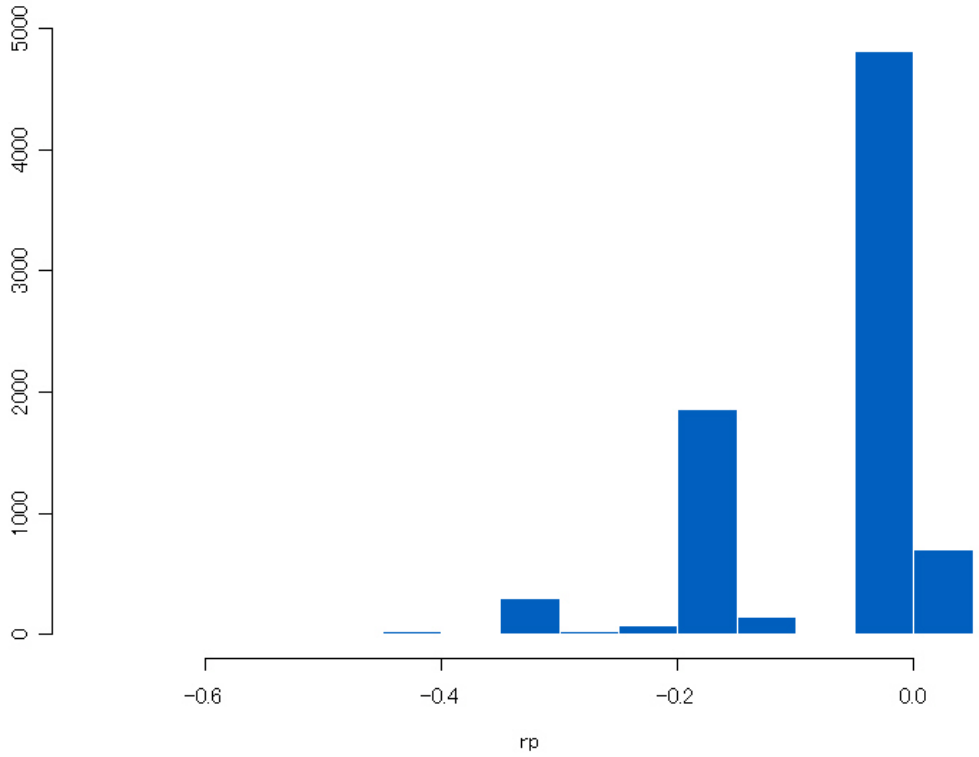
5-2-3.Lower Partial Moments(LPM)

In this section, we try to describe a model to minimize Lower Partial Moments(LPM). This model ignores sample plots whose return rates are over rG, and minimize the average under rG.

```
#### minimize LPM ####
MinLPM1 <- function(r.d, rG)
{
    Asset <- Set()
    j <- Element(set=Asset)
    Sample <- Set()
    t <- Element(set=Sample)
    r <- Parameter(index=dprod(t,j),r.d)
    rb <- Parameter(index=j)
    rb[j] ~ Sum(r[t,j],t)/nrow(r.d)
    x <- Variable(index=j)
    s <- Variable(index=t)
    f <- Objective(type=minimize)
    f ~ Sum(s[t],t)/nrow(r.d)
    Sum(x[j],j) == 1
    x[j] >= 0
    s[t] >= 0
    rG <= Sum(r[t,j]*x[j],j) + s[t]
}
```

In this example, we set rG as -0.4.

```
# Assign datas
sys <- System(MinLPM1, R.8000x5, -0.4)
# Optimize
sol <- solve(sys)
# Get solutions
x <- as.array(current(sys,x))
rp <- as.matrix(R.8000x5) %*% as.vector(x)
graphsheet()
hist(rp)
```



The chart shows that this model can avoid very bad cases compared to Markowitz Model or Mean-Absolute Deviation Model.

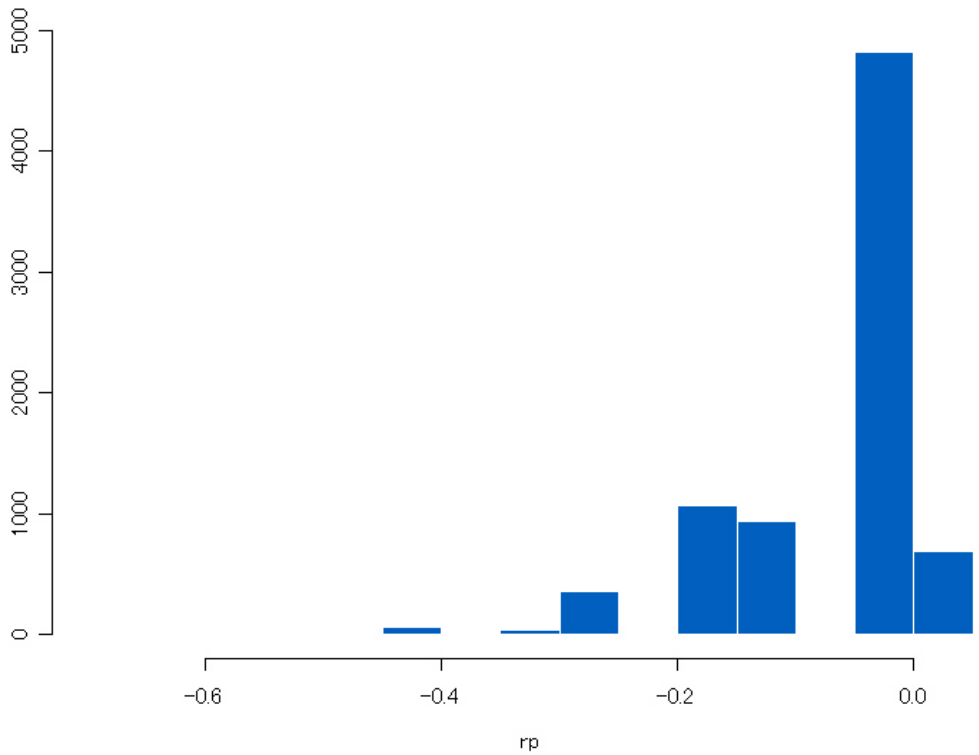
5-2-4.Conditional Value at Risk

Finally, we try to minimize Conditional Value at Risk(CVaR). Conditional Value at Risk means the average loss of assets which are low order in return rates. An asset is considered to be low when the order of the return rate is under $1-\beta$. For example, if the number of assets is 8000 and $\beta = 0.97$, then we minimize the average loss of 240 = 8000 * 0.03 assets which return rates are lower.

```
#### Minimize Conditional Value at Risk (CVaR)  ####
MinCVaR <- function(r.d, beta)
{
    Asset <- Set()
    j <- Element(set=Asset)
    Sample <- Set()
    t <- Element(set=Sample)
    r <- Parameter(index=dprod(t,j),r.d)
    rb <- Parameter(index=j)
    rb[j] ~ Sum(r[t,j],t)/nrow(r.d)
    x <- Variable(index=j)
    s <- Variable(index=t)
    VaR <- Variable()
    f <- Objective(type=minimize)
```

```
    f ~ Sum(s[t],t)/(nrow(r.d)*(1-beta)) + VaR
    Sum(x[j],j) == 1
    x[j] >= 0
    s[t] >= 0
    Sum(r[t,j]*x[j],j) + VaR + s[t] >= 0
}

# Assign datas
sys <- System(MinCVaR, R.8000x5, 0.97)
# Optimize
sol <- solve(sys)
# Get solutions
x <- as.array(current(sys,x))
# Graph out
graphsheet()
rp <- as.matrix(R.8000x5) %*% as.vector(x)
hist(rp)
```



As is the case with Lower Partial Moments Model, this model can also avoid very bad cases.

5-3 Compact Decomposition(Large Scale Optimization Technique)

The drawback of a basic Markowitz Model is that the variance-covariance matrix “Q” needs to be given in advance.

$$\sum_{i,j \in Asset} Q_{ij} x_i x_j,$$

But in real business, if the number of assets is hige, it is sometimes unaffordable to obtain the matrix beforehand. In this case, there is a sophisticated technique to reduce this burden named “Compact Decomposition”. The essence of the techinique is shown below. First, assign historical datas “R” directly.

R :Historical Asset datas of return rates

Please notice that the variance-covariance matrix is described below.

$$Q_{ij} = \frac{1}{|Period|-1} \sum_{t \in Period} (R_{ti} - \bar{r}_i)(R_{tj} - \bar{r}_j)$$

Therefore, if you introduce slack variables s and equality constraints in the following,

$$s_t = \sum_j (R_{tj} - \bar{r}_j) x_j,$$

The risk can be reformulated as follows.

$$\frac{1}{|Period|-1} \sum_{t \in Period} s_t^{\,2}$$

Applying this reformulation, you do not have to produce the variance-covariance matrix directly. The below is the example applying this technique to “R.60x1000”

```
MinVar <- function(r.d)
{
        Asset <- Set()
        j <- Element(set=Asset)
        Sample <- Set()
        t <- Element(set=Sample)
        r <- Parameter(index=dprod(t,j),r.d)
        rb <- Parameter(index=j)
        rb[j] ~ Sum(r[t,j],t)/nrow(r.d)
        x <- Variable(index=j)
        s <- Variable(index=t)
        f <- Objective(type=minimize)
        f ~ Sum(s[t]*s[t],t)/nrow(r.d)
        Sum(x[j],j) == 1
        x[j] >= 0
        Sum((r[t,j]-rb[j])*x[j],j) == s[t]
}
# Assign datas
sys <- System(MinVar, R.60x1000)
# Optimize
sol <- solve(sys)
```

```
# Get solutions
x <- as.array(current(sys, x))
```

The same technique can be applied to Multi Factor Model. In this case, return rates are given below.

$$r_j = \alpha_j + \sum_k \beta_{jk} f_k + \varepsilon_j \qquad k \in Factor,\ j \in Asset,\ \varepsilon_j \in N(0, \sigma_j^2)$$

Then, the variance of the return rates is reformulated as:

$$\sum_{i,j \in Asset} \sum_{k,l \in Factor} Q_{k,l}^f \beta_{j,k} \beta_{j,l} x_i x_j + \sum_{j \in Asset} \sigma_j^2 x_j^2$$

Therefore, if we introduce slack variables s and equality constraints in the following,

$$s_k = \sum_j \beta_{j,l} x_j$$

The variance-covariance matrix will be given below.

$$\sum_{k,l} Q_{k,l}^f s_k s_l + \sum_j \sigma_j^2 x_j^2$$

5-4 Odd Lot Consolidation

In optimization problems, asset allocation amounts are considered to be continuous. But in practice, they are integer and minimum truck number needs to be taken into consideration. This compensation procedure is named as “Odd lot consolidation”

In this section, we describe this model using an integer variable d_i which equal to 1 when rounding out and equal to 0 when rounding off for each asset i. This model minimizes Absolute deviations under a constraint that the fluctuation of the total amount of the truck is in the reach of 1%. Let’s apply this model to the solution RoundLot.x calculated in the previous section. Total amount of the track is assumed to be 5 billions.

Minimum truck price is given by “RoundLot.unit”.

```
> RoundLot.unit
    A0001    A0002  A0003   A0004   A0005   A0006   A0007    A0008   A0009
10880000 5400000 873000 1875000 7040000 4980000 3360000 15100000 5390000

   A0010    A0011   A0012  A0013   A0014    A0015   A0016   A0017    A0018
2360000 4330000 6160000 994000 1385000 15150000 2200000 4390000 18000000

   A0019    A0020   A0021   A0022   A0023   A0024   A0025   A0026   A0027
2760000 3790000 2395000 2730000 4430000 4160000 2960000 2265000 5410000
…
```

The following is the model teaches us which assets will be rounded out or off. This is a

large Mixed Integer Programming Problem, so we apply method “wcsp” to solve the problem.

```
#### Odd Lot Consolidation ####
RoundLot <- function(r.d, unit.d, x.d, total)
{
    Asset <- Set()
    j <- Element(set=Asset)
    Sample <- Set()
    t <- Element(set=Sample)
    r <- Parameter(index=dprod(t,j),r.d)
    rb <- Parameter(index=j)
    rb[j] ~ Sum(r[t,j],t)/nrow(r.d)
    unit <- Parameter(index=j,unit.d)
    base <- Parameter(index=j,(total*x.d)%/%unit.d*unit.d)
    d <- IntegerVariable(type="binary",index=j)
    fund <- Expression(index=j)
    fund[j] ~ base[j] + unit[j]*d[j]
    0.99*total <= Sum(fund[j],j) <= 1.01*total
    s <- Expression(index=t)
    s[t] ~ 0.01*Sum((r[t,j]-rb[j])*fund[j],j)
    softConstraint(1,0,1)
    s[t] == 0
}

# Total truck amount
total <- 5e9
# Assign datas
sys <- System(RoundLot, R.60x1000, RoundLot.unit, RoundLot.x, total)
# Optimize (method “wcsp” is applied)
nuopt.options(method="wcsp")
nuopt.options(maxtim=20)
sol <- solve(sys)
# Get solutions
d <- as.array(current(sys,d))
fund <- as.array(current(sys,fund))
```

The below is definitions of the volume of both the ideal investment and the practical investment.

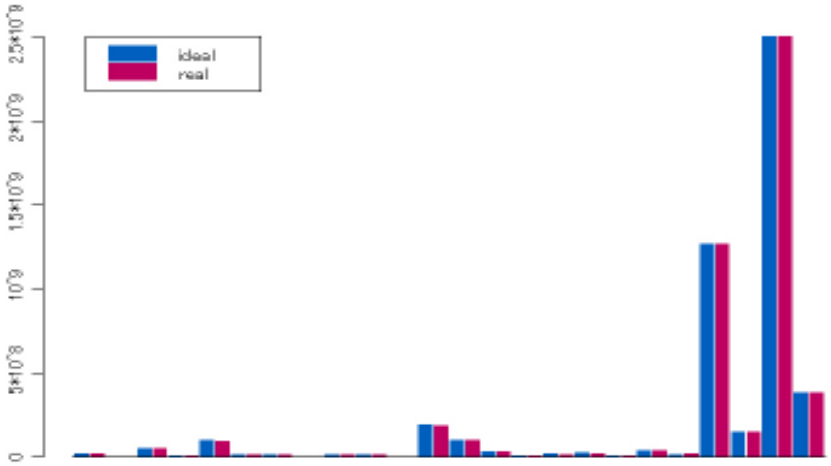
```
n <- sum(fund > 0)
ideal <- (total* RoundLot.x)[fund > 0]  # Volume of the ideal investment
up <- d[fund > 0]
real <- fund[fund > 0]  # Volume of the practical investment
```

When “up” is 0, 1 the ideal amount will be round off and out respectively.

```
> cbind(ideal, up, real)
             ideal up       real
```

```
A0072 2.268026e+007  0    22275000
A0113 1.116033e-006  1     3740000
A0150 4.873316e+007  1    50140000
A0461 1.049896e+007  0    10296000
A0508 9.945964e+007  0    97500000
A0647 1.501497e+007  1    15030000
A0664 1.513937e+007  1    16450000
A0712 1.759865e-006  1     3880000
A0713 1.312477e+007  0    12705000
A0717 1.757595e+007  0    17492400
A0725 2.784842e-007  1      609000
A0746 1.929582e+008  0   188160000
A0751 1.009987e+008  0    99960000
A0870 3.162644e+007  1    32890000
A0875 7.202727e+006  1     9060000
A0907 2.333477e+007  0    17970000
A0919 2.447410e+007  0    21700000
A0935 7.131422e+006  0     6769000
A0945 4.145754e+007  0    38940000
A0962 1.749186e+007  1    18480000
A0975 1.265938e+009  0  1265850000
A0977 1.487018e+008  1   150960000
A0987 2.505445e+009  0  2505170000
          ideal up       real
A0997 385938255  0 385640000
```

```
#Graph out
barplot(rbind(ideal,real),beside=T,col=rbind(rep(2,n),rep(3,n)))
legend(1,2.5e9,c("ideal","real"),fill=2:3)
```



5-5 Grouping of Assets

After solving optimization models, you have only to order. But in practice, giving a huge order at once is not desirable. Like this case, you may as well divide the order and assets. It is preferred that orders are not so much different each other.

In this section, our goal is to divide assets into some groups which are similar to each other. For example, 50 assets are given and we would like to divide them into 5 groups. Each asset is featured by 10 factors from F0 to F9.

```
> Basket.fcoef
        F0    F1    F2    F3    F4    F5    F6    F7     F8    F9
 1    0.81  0.29  0.42  0.00 -0.14  0.03  0.04  1.83 -14.89 -0.20
 2    1.29  0.39  0.41  0.04  0.10  0.18 -0.19  4.43 -15.35  0.23
 3    1.00 -0.16  0.32  0.02 -0.02  0.25  0.14  2.12 -26.86  0.19
 4   -2.75 -0.24 -0.40  0.01  0.29 -0.15 -0.40 -2.09  28.49 -0.18
 5   -0.64 -0.12 -0.19 -0.14  0.00  0.22 -0.05 -1.84  24.50  0.72
 6   -5.94  0.05 -0.20  0.03 -0.08 -0.24 -0.06 -1.57  49.74 -0.17
 7    1.85  0.03  0.59  0.04  0.31  0.28  0.12  2.12 -42.61 -0.88
 8    5.20  0.07  0.22  0.03 -0.07  0.01  0.19 -2.98  -2.71 -0.28
 9   12.82  0.12  0.24  0.03 -0.10  0.13  0.47  3.40 -42.22  0.21
10    4.15  0.05  0.12  0.01 -0.05  0.09  0.10 -0.35  23.63  0.22
11 -32.92  0.12 -0.11  0.03 -0.12 -0.12  0.17 -1.81  16.64 -0.17
12    4.72 -0.20 -0.08 -0.03 -0.17 -0.04  0.20  3.84  -6.97  0.34
13    4.75  0.31  0.67  0.00 -0.01  0.48 -0.45  7.06 -31.22  0.18
14 -12.12 -0.07 -0.04 -0.07  0.36 -0.10 -0.21 -4.90   4.29  0.06
15   -7.92 -0.22 -0.26 -0.01  0.06 -0.01 -0.21  0.84   8.11  0.83
16    6.30  0.03  0.23  0.02 -0.07  0.19  0.00  2.46 -16.53  0.14
17 -30.70 -0.52 -0.04 -0.02  0.33  0.17 -0.22 -2.99  81.08 -0.18
18   -5.14 -0.09 -0.46  0.01 -0.17 -0.29  0.04 -1.05  64.62 -0.35
19   -5.92  0.23 -0.02 -0.01  0.08 -0.20 -0.23 -0.04 -11.09 -0.04
20 -12.54 -0.16 -0.06  0.00  0.18  0.06 -0.07  1.78  -2.60 -0.23
21    2.46  0.29  0.56  0.06  0.06  0.41  0.14  0.99  24.71  0.17
22    2.91 -0.26  0.16  0.01 -0.02  0.00  0.03 -1.26  12.49  0.16
23    3.78  0.11  0.16  0.00 -0.06  0.18  0.10 -0.66  10.73  0.14
        F0    F1    F2    F3    F4    F5    F6    F7     F8    F9
24    1.83  0.16  0.35  0.01  0.03 -0.05  0.11 -2.04  15.40  0.17
25    3.02  0.15  0.51  0.00 -0.10  0.07  0.34  3.21 -17.92  0.35
26 -17.63  0.00  0.12  0.03 -0.05  0.03 -0.03  1.55 -39.72 -0.19
27   -1.54 -0.60 -0.54  0.01  0.13 -0.16 -0.27 -2.41  20.25 -0.18
28    1.56 -0.15 -0.22  0.03 -0.29 -0.20  0.19 -3.81  33.04 -0.01
29    1.56 -0.15 -0.22  0.03 -0.29 -0.20  0.19 -3.81  33.04 -0.01
30    5.65 -0.18 -0.28  0.02 -0.15 -0.19  0.12 -4.15  33.42  0.15
31    1.36  0.40  0.57  0.06 -0.24  0.11  0.18  2.04   0.87  0.17
32   -1.69 -1.01 -0.48 -0.01  0.01  0.01 -0.37 -1.61  -6.85 -0.20
33   -1.32  0.00 -0.27  0.01  0.07  0.02 -0.44  3.43 -20.62 -0.22
34   -1.60 -0.14 -0.44  0.02  0.09 -0.33  0.16 -0.34  15.63 -0.25
35    2.46  0.00 -0.06  0.10  0.10 -0.11  0.10  3.93 -16.98 -0.88
36   -3.28 -0.03 -0.27 -0.02  0.03 -0.25 -0.03 -1.54 -19.38 -0.30
37   -1.48 -0.17 -0.28  0.03  0.21  0.03 -0.22 -2.21   1.19 -0.12
```

```
38  -2.38 -0.24 -0.02 -0.01  0.17  0.09 -0.01 -2.26 -14.43  0.25
39  -0.50 -0.25 -0.24  0.02  0.04  0.19  0.10  2.10  -1.32 -0.21
40  -0.35 -0.44 -0.27  0.01  0.11  0.28 -0.26 -2.88  -3.57  0.66
41   0.78  0.32  0.22  0.02 -0.13  0.00 -0.71  0.42 -26.84 -0.88
42 -14.75  0.00 -0.11  0.04  0.17 -0.23 -0.10  0.30  -5.03  0.00
43   0.46  0.86  0.45  0.07  0.12 -0.09  0.37  5.16 -71.47 -0.12
44  13.48 -0.47 -0.17 -0.01  0.06  0.06  0.09  1.86  -1.98 -0.31
45  -1.83 -0.54 -0.44  0.01 -0.49 -0.21 -0.25  2.32   0.31  0.40
46  -2.56  0.25  0.31  0.00 -0.06  0.19  0.02  3.26  -6.25 -0.28
      F0    F1    F2    F3    F4    F5    F6    F7     F8    F9
47  3.83 -0.48 -0.06 -0.02 -0.09  0.10  0.20  2.18   5.77 -0.40
48 -3.67  0.41 -0.03  0.03  0.14 -0.18 -0.04 -4.86   0.09  0.56
49  6.17 -0.26  0.09 -0.01 -0.05  0.13  0.16  1.81  44.54  0.00
50  4.11  0.45  0.35  0.00  0.24 -0.03  0.52 -0.47 -15.80  0.19
```

For each asset, the sum of F0 to F9 is restricted in a given bound. And for some important factors(F2, F3, F4, F5, F6, F9), the difference between the variables and given datas needs to be minimized. The model is described in the following.

```
#### Grouping of Assets ####
Basket <- function(ng.d, fcoef.d, flow.d, fbar.d, fhigh.d, W.d)
{
    M <- Set()
    m <- Element(set=M)
    J <- Set()
    j <- Element(set=J)
    K <- Set(1:ng.d)
    k <- Element(set=K)
    f <- Parameter(index=dprod(j,m), fcoef.d)
    flow <- Parameter(index=m, flow.d)
    fbar <- Parameter(index=m, fbar.d)
    fhigh <- Parameter(index=m, fhigh.d)
    W <- Parameter(index=m, W.d)
    u <- IntegerVariable(index=dprod(j,k),type="binary")
    .F <- Expression(index=dprod(m,k))
    .F[m,k] ~ Sum(f[j,m]*u[j,k], j)
    selection(u[j,k],k)
    scf <- 1000
    scf*fhigh[m] >= scf*.F[m,k] >= scf*flow[m]
    softConstraint(scf,1)
    W[m]*(.F[m,k] - fbar[m]) == 0
    g <- Expression(index=j)
    g[j] ~ Sum(u[j,k]*k, k)
}
```

```
# Lower bounds of factors
> Basket.flow
  F0  F1  F2  F3  F4  F5  F6  F7  F8  F9
```

```
 -15 -15 -15 -15 -15 -15 -15 -15 -15 -15

# Upper bounds of factors
> Basket.fhigh
 F0 F1 F2 F3 F4 F5 F6 F7 F8 F9
 15 15 15 15 15 15 15 15 15 15

# Average
> Basket.fbar
 F0 F1 F2 F3 F4 F5 F6 F7 F8 F9
  0  0  0  0  0  0  0  0  0  0

# To define which factors are important(F2,F3,F4,F5,F6,F9 are important in this
case)
> Basket.W
 F0 F1 F2 F3 F4 F5 F6 F7 F8 F9
  0  0  1  1  1  1  1  0  0  1
```

The above are data objects given beforehand.

```
# Assign datas
sys <- System(Basket, 5, Basket.fcoef, Basket.flow, Basket.fbar,
Basket.fhigh, Basket.W)

# Set algorithm and time limit
nuopt.options(method = "wcsp",maxtim = 10)

# Optimize
sol <- solve(sys)

# Get solutions
gnum <- as.array(current(sys,g))
```

To confirm the results, we compare the values of factors. “fopt” is obtained by this model. On the other hand, “frand” is aquired at random.

```
# Confirmation
fopt <- rbind(apply(Basket.fcoef[gnum == 1, ], 2, sum),
apply(Basket.fcoef[
        gnum == 2, ], 2, sum), apply(Basket.fcoef[gnum == 3, ], 2, sum),
        apply(Basket.fcoef[gnum == 4, ], 2, sum), apply(Basket.fcoef[gnum
==
        5, ], 2, sum))
frand <- rbind(apply(Basket.fcoef[1:10, ], 2, sum),
apply(Basket.fcoef[11:
        20, ], 2, sum), apply(Basket.fcoef[21:30, ], 2, sum), apply(
        Basket.fcoef[31:40, ], 2, sum), apply(Basket.fcoef[41:50, ], 2,
        sum))
```

```
> fopt
          F0    F1    F2    F3    F4    F5    F6    F7    F8    F9
[1,] -14.65  0.76  0.36  0.16  0.09 -0.23 -0.23 -4.52 13.94  0.59
[2,] -14.13 -0.13 -0.41  0.24 -0.56 -0.32  1.01 -0.57 14.82 -0.77
[3,] -14.72 -0.24  0.06  0.16  1.02  0.35 -0.07 -2.66 14.22 -0.38
[4,] -14.60 -0.30  0.69  0.03  0.41  0.62 -0.37  8.34 14.89  0.42
[5,] -14.76 -2.15  0.11 -0.06 -0.49  0.19 -0.57  9.92 13.50 -0.61
> frand
          F0    F1    F2    F3    F4    F5    F6     F7      F8    F9
[1,]  17.79  0.48  1.53  0.07  0.24  0.80  0.36   5.07  -18.28 -0.14
[2,] -91.49 -0.57 -0.17 -0.08  0.47  0.14 -0.98   5.19  106.33  0.58
[3,]   3.60 -0.63  0.60  0.20 -0.74 -0.11  0.92 -12.39  125.44  0.75
[4,]  -8.78 -1.88 -1.76  0.21  0.59  0.04 -0.79   0.66  -65.46 -1.10
[5,]   6.02  0.54  0.61  0.13 -0.09 -0.26  0.26  11.98  -76.66 -0.84
```

It is obvious that “fopt” is superior to “frand”.

5-6 Maximum Drawdown

Maximum drawdown is one way to estimate risk. It estimates how much money could be lost at some intermediate point. The feature of maximum drawdown is to consider input date not to be sample data, but to be a time series data.
In this section, we try to decide asset allocation rates minimizing maximum drawdown at a given intermediate point. The asset allocation rates are fixed at initial point and will not be changed.
First, we convert return rate data into price data that the initial price is 1.

```
tmp <- rbind(rep(0,ncol(R.521x95)),R.521x95)+1
P.521x95 <- apply(tmp,2,cumprod)
```

The model is described as following. Variable x is asset allocation amounts of each brands, Variable W is portfolio value at each time, Variable U is cumulative maximum of W at each t, Variable V is cumulative minimum in reversed order at each t.
Maximum drawdown could be described as maximal difference of U and V.

```
#### Maximum Drawdown Minimization Model ####
MinMaxDD <- function(P.d)
{
    Asset <- Set()
    j <- Element(set=Asset)
    Period <- Set()
    t <- Element(set=Period)
    P <- Parameter(index=dprod(t,j),P.d)
    x <- Variable(index=j)
    U <- Variable(index=t)
    V <- Variable(index=t)
    W <- Variable(index=t)
    maxdd <- Variable()
    risk <- Objective()
    risk ~ maxdd
```

```
    W[t] == Sum(P[t, j]*x[j], j)
    U[t, t>1] >= U[t-1, t>1]
    U[t] >= W[t]
    V[t-1, t>1] <= V[t, t>1]
    V[t] <= W[t]
    U[t] - V[t] <= maxdd
    Sum(x[j], j) == 1
    x[j] >= 0
}

# Make system and solve
sys <- System(MinMaxDD, P.521x95)
sol <- solve(sys)
#Get solution
x <- as.array(current(sys, x))
```

Here, we define the function which calculates maximum drawdown from price data and asset allocation rates. The function also calculates W, U and V.

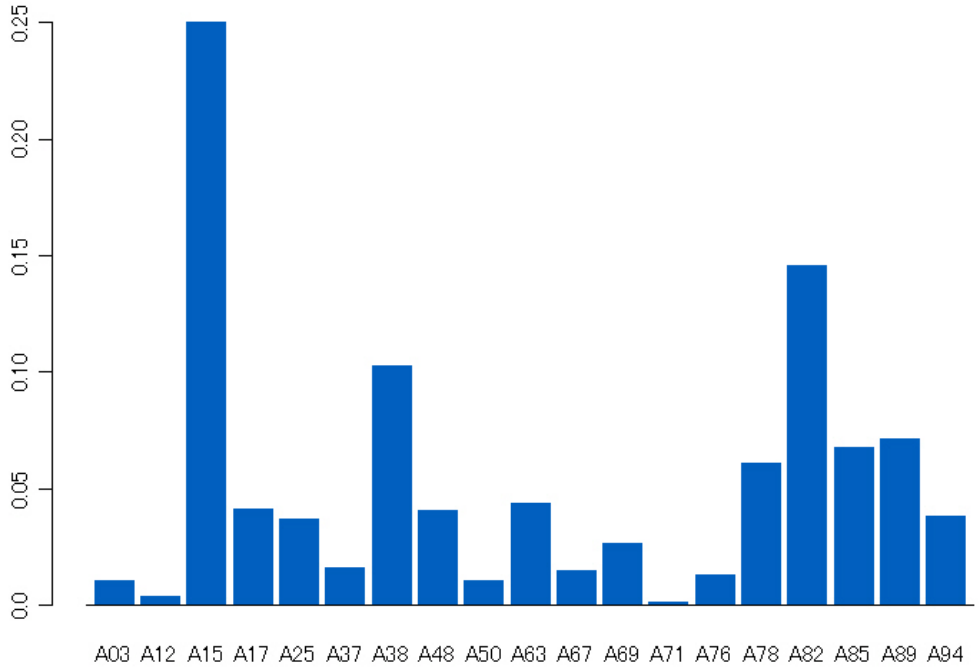
```
calc.MaxDD <- function(P, x)
{
        W <- as.vector(P %*% as.vector(x))
        U <- cummax(W)
        V <- rev(cummin(rev(W)))
        return(list(maxdd=max(U-V), W=W, U=U, V=V))
}
```

Calculate maximum drawdown.

```
res <- calc.MaxDD(P.521x95, x)
res$maxdd
```

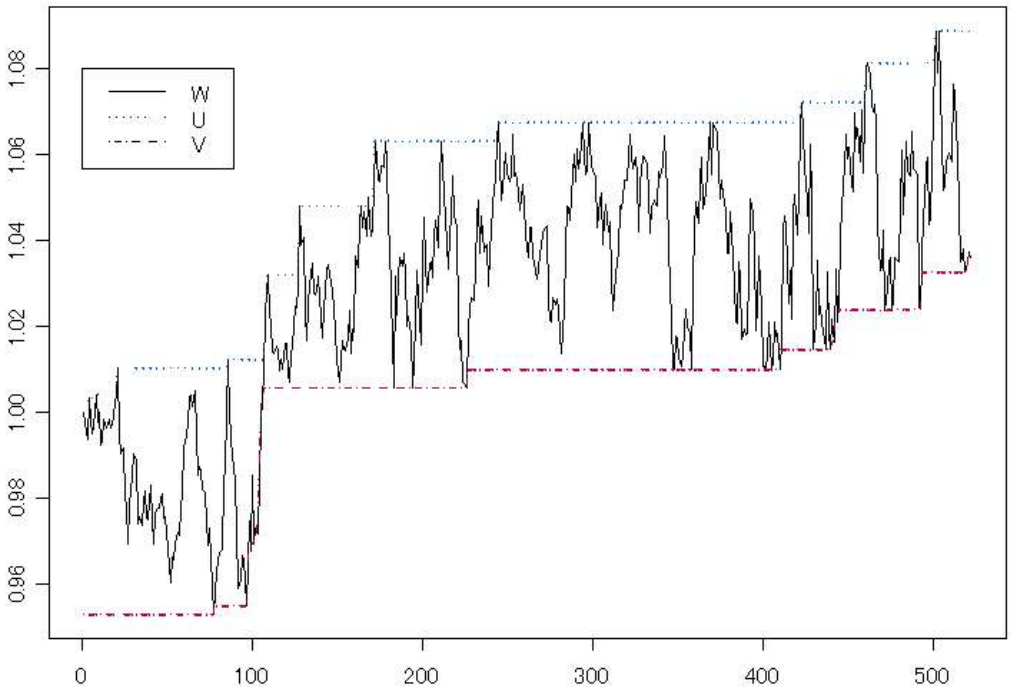
Show the solution as bar plot.

```
eps <- 1e-4
barplot(x[x>=eps], names=names(x[x>=eps]))
```



Plot W, U and V above.

```
m <- cbind(res$W,res$U,res$V)
matplot(1:nrow(m),m,name=colnames(m),type="l")
legend(0,1.08,c("W","U","V"),lty=1:3)
```


W
U
V
0.96
0.98
1.00
1.02
1.04
1.06
1.08
0
100
200
300
400
500

5-7 Sharpe Ratio

To maximize an index called "Sharpe Ratio" is one of the well-known portfolio optimization problems. Sharpe Ratio is defined as follows:

$$\frac{\sum_{j \in A} \overline{r}_j x_j - r_f}{\sqrt{\sum_{j \in A} \sum_{k \in A} \sigma_{jk} x_j x_k}}$$

A is a set of assets, x_j are asset allocation rates, $\overline{r}_j$ are average return rates of assets, σ_{jk} are variances(or covariances), and r_f are return rates of secure assets. The objective of this model is nonlinear, therefore this is a Nonlinear Optimization Problem. S+NUOPT can handle this problem.

```
#### Maximize Sharpe Ratio ####
Sharpe <- function(Q.d,rb.d)
{
    Asset <- Set()
    j <- Element(set=Asset)
    k <- Element(set=Asset)
    Q <- Parameter(index=dprod(j,k),Q.d)
    rb <- Parameter(index=j,rb.d)
    x <- Variable(index=j)
    x[j] ~ 0.1
    f <- Objective(type="maximize")
    f ~ (Sum(rb[j]*x[j],j)-0.002)/Sum(Q[j,k]*x[j]*x[k],j,k)^0.5
    Sum(x[j],j) == 1
    x[j] >= 0
}
```

In advance, we obtain variance-covariance matrix and average return rates.

```
Q <- var(R.60x200)
rb <- apply(R.60x200,2,mean)
rb <- as.array(rb)

# Assign Datas
sys <- System(Sharpe, Q, rb)
# Optimize
sol <- solve(sys)
# Get solutions
x <- as.array(current(sys,x))
```

In practical, the above model is reformulated as a Quadratic Programming Problem. The reformulation is described as follows.

$$\sum_{j\in \mathrm{A}} \overline{r}_j x_j - r_f = \lambda (> 0)$$

$$w_j = x_j / \lambda$$

The reformulated form of the model is written below.

```
#### Maximize Sharpe Ratio (QP)  ####
Sharpe.qp <- function(Q.d,rb.d)
{
    Asset <- Set()
    j <- Element(set=Asset)
    k <- Element(set=Asset)
    Q <- Parameter(index=dprod(j,k),Q.d)
    rb <- Parameter(index=j,rb.d)
    w <- Variable(index=j)
    f <- Objective(type="minimize")
    f ~ Sum(Q[j,k]*w[j]*w[k],j,k)
    Sum((rb[j]-0.002)*w[j],j) == 1
    w[j] >= 0
}
# Assign datas
sys <- System(Sharpe.qp, Q, rb)
# Set optional value
nuopt.options(eps=1e-10)
# Optimize
sol <- solve(sys)
# Get solutions
w <- as.array(current(sys,w))
# Transform
x.qp <- w/sum(w)
```

The comparison of the results of the two models is described below. You can confirm that the results are almost equal.

```
> eps <- 1e-4
> x[x>=eps]
      A071       A080       A081       A087       A103       A113       A139
 0.1317304 0.06223414 0.1040298 0.03068688 0.02559427 0.03757759 0.003409419

     A189       A190        A195      A197          A200
 0.235764 0.1985228 0.005258932 0.164983 0.0002087379
> x.qp[x.qp>=eps]
      A071       A080       A081       A087       A103       A113       A139
 0.1317538 0.06221308 0.1040643 0.03067362 0.02548632 0.03759601 0.003395785

      A189       A190        A195       A197          A200
 0.2357851 0.1985498 0.005192057 0.1649768 0.0003129249
```

6. Semidefinite Optimization

6-1 Nearest correlation matrix problem

Correlation matrices of assets are required in some cases like Monte Carlo simulations. A correlation matrix is positive by definition, but in some cases there are some deficits. The problem to reformulate such deficits as far as the matrix to be positive is sometimes very important. S+NUOPT can handle this problem using Semidefinite Optimization algorithms. Let us consider the problem.
For example, a Symmetric Matrix described below is not positive definite.

```
> Cormat.A
     X01  X02 X03 X04 X05 X06 X07 X08 X09 X10
X01 1.00 0.17 0.5 0.2 0.1 0.0 0.2 0.0 0.8 0.7
X02 0.17 1.00 0.1 0.9 0.6 0.4 0.0 0.0 0.0 0.0
X03 0.50 0.10 1.0 0.2 0.0 0.0 0.0 0.2 0.0 0.2
X04 0.20 0.90 0.2 1.0 0.2 0.2 0.2 0.2 0.0 0.0
X05 0.10 0.60 0.0 0.2 1.0 0.4 0.0 0.0 0.0 0.0
X06 0.00 0.40 0.0 0.2 0.4 1.0 0.0 0.0 0.0 0.0
X07 0.20 0.00 0.0 0.2 0.0 0.0 1.0 0.0 0.0 0.0
X08 0.00 0.00 0.2 0.2 0.0 0.0 0.0 1.0 0.0 0.0
X09 0.80 0.00 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
X10 0.70 0.00 0.2 0.0 0.0 0.0 0.0 0.0 0.0 1.0
```

These are eigenvalues of the matrix:

```
> eigen(Cormat.A)$values
[1]  2.6192649  2.0903778  1.2534419  1.0818527  1.0000000  0.836797
[7]  0.7030020  0.6042132 -0.0285787 -0.1603709
```

The minimum eigenvalue is minus (-0.1603709). To modify the matrix to be positive, please look at the following optimization model. This model teaches us a positive matrix which is nearest to the initial matrix in the context of Frobenius norm.

```
Cormat <- function(A, minEig = 0.001)
{
    N <- Set()
    i <- Element(set = N)
    j <- Element(set = N)
    A <- Parameter(A, index = dprod(i, j))
    minEig <- Parameter(minEig)
    X <- Variable(index = dprod(i, j))
    M <- SymmetricMatrix(dprod(i, j))
    M[i, j, i >= j] ~ X[i, j]
    diffnrm <- Objective(type = minimize)
    diffnrm ~ Sum((X[i, j] - A[i, j]) * (X[i, j] - A[i, j]), i, j, i >= j)
    # Minimum eigenvalue of M is over minEig
    M >= minEig
    # Diagonal components needs to be 1
```

```
    X[i, i] == 1
    # The matrix need to be symmetric
    X[i,j,i<j] == X[j,i]
}

s <- System(model=Cormat,Cormat.A)
sol <- solve(s)
Aopt <- as.array(current(s,X))
```

The positive matrix obtained above is described below.

```
> round(Aopt,2)
     X01  X02  X03  X04  X05  X06  X07   X08   X09  X10
X01 1.00 0.16 0.46 0.19 0.09 0.01 0.18  0.01  0.71 0.63
X02 0.16 1.00 0.10 0.88 0.59 0.40 0.01  0.00  0.01 0.00
X03 0.46 0.10 1.00 0.20 0.00 0.00 0.01  0.20  0.03 0.22
X04 0.19 0.88 0.20 1.00 0.21 0.20 0.20  0.20  0.00 0.00
X05 0.09 0.59 0.00 0.21 1.00 0.40 0.00  0.00  0.01 0.00
X06 0.01 0.40 0.00 0.20 0.40 1.00 0.00  0.00  0.00 0.00
X07 0.18 0.01 0.01 0.20 0.00 0.00 1.00  0.00  0.01 0.01
X08 0.01 0.00 0.20 0.20 0.00 0.00 0.00  1.00 -0.01 0.00
X09 0.71 0.01 0.03 0.00 0.01 0.00 0.01 -0.01  1.00 0.05
X10 0.63 0.00 0.22 0.00 0.00 0.00 0.01  0.00  0.05 1.00
```

This matrix is the nearest positive matrix to "Cormat.A" in the context of Frobenius norm as far as the minimum eigenvalue is over 0.001. To confirm the claim, let us show the eigenvalues.

```
> eigen(Aopt)$values
 [1] 2.580796251 2.028567070 1.236807350 1.067534416 0.968329999
 [6] 0.830140682 0.685764222 0.600045416 0.001012264 0.001002330
```

6-2 Robust Portfolio Optimization

In Markowitz model, variance-covariance matrix is given as a Parameter, but sometimes this is uncertain. In this section, we try to obtain a robust solution under this uncertainty. First, we start from describing the typical Markowitz model again.

$$\max_x \ \mu^T x - \lambda x^T \Sigma x$$
$$s.t. \quad x^T e = 1$$

μ is return rates, Σ is Variance-covariance matrix, x is asset allocation rates, λ is a risk aversion rate. If the variance-covariance matrix Σ is uncertain and the uncertainty is considerd as $\sum \in U_\Sigma$, the model is described below.

$$\max_x \left\{ \mu^T x - \lambda \max_{\Sigma \in U_\Sigma} \left\{ x^T \Sigma x \right\} \right\}$$
$$s.t. \quad x^T e = 1$$

If $\sum \in U_\Sigma$ can be interpreted as $\underline{\Sigma} \le \Sigma \le \overline{\Sigma}$, the model can be reformulated in the following[3].

$$\max_x \mu^T x - \lambda \left(\overline{\Sigma} \bullet U - \underline{\Sigma} \bullet L \right)$$
$$s.t. \quad x^T e = 1$$
$$\begin{pmatrix} U - L & x \\ x^T & 1 \end{pmatrix} \succeq 0$$
$$U \ge 0, L \ge 0$$

Please notice that the operator $\bullet$ means the inner product of the matrices defined as:

$$A \bullet B = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} B_{ij}$$

The following is a S+NUOPT procedure to handle the above model. $\underline{\Sigma}, \overline{\Sigma}, \mu$ and λ are written by “sigL”, “sigU”, “mu” and “lambda” respectively.

```
Robust <- function(sigL, sigU, mu, lambda)
{
  Asset <- Set()
  i <- Element(set = Asset)
  j <- Element(set = Asset)
  sigL <- Parameter(sigL, index = dprod(i, j))
  sigU <- Parameter(sigU, index = dprod(i, j))
  lambda <- Parameter(lambda)
  mu <- Parameter(mu, index = i)
  U <- Variable(index = dprod(i, j))
  L <- Variable(index = dprod(i, j))
  x <- Variable(index = i)
  V <- Set(c(as.list(Asset),"dummy")) # V consists of Asset and " dummy"
  v <- Element(set = V)
  w <- Element(set = V)
  M <- SymmetricMatrix(dprod(v, w))
  M[i, j] ~ U[i, j] - L[i, j]
  M[j, "dummy"] ~ x[j]
  M["dummy", "dummy"] ~ 1
  f <- Objective(type = maximize)
  f ~ Sum(mu[i] * x[i], i) - lambda * Sum(sigU[i, j] * U[i, j] - sigL[i, j]
* L[i, j], i, j)
  M >= 0
  Sum(x[i], i) == 1
  U[i, j, i > j] == U[j, i]
  L[i, j, i > j] == L[j, i]
```

```
        U[i, j] >= 0
        L[i, j] >= 0
      }
```

Datas are given below.

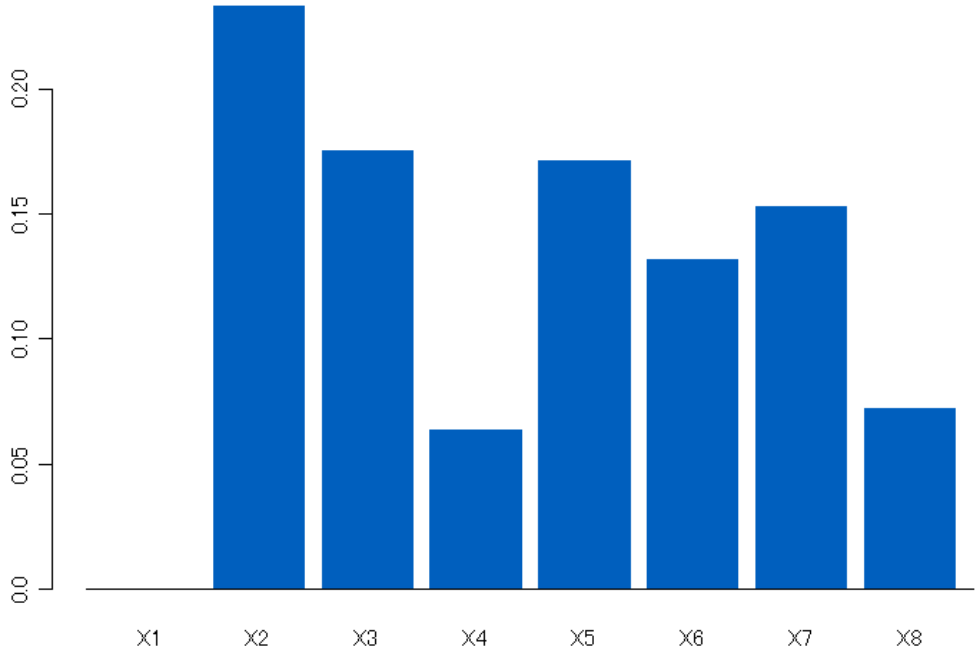
```
> Robust.sigU    # Upper values of components of the variance-covariance matrix
     X1   X2   X3   X4    X5    X6    X7    X8
X1  3.0  1.0  2.5 -0.9  1.00  -0.4  2.50   2.0
X2  1.0  4.5 -1.4  1.2  0.50  -1.1  0.50   2.6
X3  2.5 -1.4  6.0 -0.5  1.20   1.0  0.40  -0.1
X4 -1.0  1.2 -0.5  6.5  1.60   0.5  1.30  -0.8
X5  1.0  0.5  1.2  1.6  5.50  -1.3  0.16  -1.9
X6 -0.4 -1.1  1.0  0.5 -1.30  11.0 -0.90   0.6
X7  2.5  0.5  0.4  1.3  0.16  -0.9  6.50  -1.2
X8  2.0  2.6 -0.1 -0.8 -1.90   0.6 -1.20  13.5

> Robust.sigL    # Lower values of components of the variance-covariance matrix
     X1   X2     X3     X4    X5   X6    X7     X8
X1  1.9 -1.4 -1.000 -1.000  1.00 -0.5 -1.00   0.10
X2 -1.4  3.5 -1.500  0.500 -1.20 -1.2  0.40   0.60
X3 -1.0 -1.5  5.000 -1.125  1.10  0.9  0.30  -0.20
X4 -1.0  0.5 -1.125  6.000  1.50  0.4  1.20  -0.90
X5  1.0 -1.2  1.100  1.500  4.50 -1.4  0.15  -2.00
X6 -0.5 -1.2  0.900  0.400 -1.40  9.0 -1.00   0.50
X7 -1.0  0.4  0.300  1.200  0.15 -1.0  5.50  -1.25
X8  0.1  0.6 -0.200 -0.900 -2.00  0.5 -1.25  11.50

> Robust.mu      # expectations of return rates
  X1  X2  X3  X4  X5  X6  X7  X8
 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
```

```
      # Assign datas
      s <- System(model=Robust,Robust.sigL,Robust.sigU,Robust.mu,1.0)
      # Optimize
      sol <- solve(s)
      # Graph out
      xopt <- as.array(current(s,x))
      barplot(names=rownames(xopt),xopt)
```

Let us graph out the result of the asset allocation rates.


0.0
0.05
0.10
0.15
0.20
X1
X2
X3
X4
X5
X6
X7
X8

7. Nonlinear Fitting

To estimate the parameters of a model function by sample datas is a mathematical programming model known as nonlinear fitting. In general, sum of least squares of errors are minimized.

$$\begin{aligned}
&\text{Variables: } a_1,\cdots,a_n && (\text{ parameters }) \\
&\text{minimize: } \sum_i e_i{}^2 && (\text{ sum of least squares of errors }) \\
&\text{subject to: } e_i = f(x_i;a_1,\cdots,a_n) - y_i, i \in \{\text{observation points}\} && (\text{ errors }) \\
&\qquad g(a_1,\cdots,a_n) \le 0 && (\text{ constraints of parameters })
\end{aligned}$$

In the model above, $f(x;a_1,\cdots,a_n)$ is a model function. In Multiple regression models, the model function is linear (which is one of the most fundamental cases). $a_1,\cdots,a_n$ are parameters need to be estimated. x_i are observation points and y_i are observation values. $g(a_1,\cdots,a_n) \le 0$ are constraints of parameters.

7-1 Yield Curve Fitting

To estimate spot rates is a well-known problem in financial optimization. This problem is named as "Yield Curve Fitting Problem". When the face value is 100 at t redemption period and the coupon rate is 1%, the spot value $S(t)$ is described below using spot rates r_t

$$S(t) = \frac{100}{(1+0.01\cdot r_t)^t} + \sum_{k=1}^{t} \frac{1}{(1+0.01\cdot r_k)^k}$$

Following is the observation datas of S_i

```
> Yield.term.data
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
 1 1 1 1 1 1 2 2 2  2  2  2  3  3  3  3  3  3  4  4  4  4  4  4  5  5  5  5

 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
  5  5  6  6  6  6  6  6  7  7  7  7  7  7  8  8  8  8  8  8  9  9  9  9  9

 54 55 56 57 58 59 60
  9 10 10 10 10 10 10
> Yield.price.data
      1       2      3       4      5       6       7      8      9     10      11
100.39 102.15 99.24 101.32 99.72 101.51 100.62 99.34 98.27 98.31 100.97

     12     13      14     15     16     17    18    19     20    21     22     23     24
101.73 99.78 100.47 98.19 99.55 99.79 98.4 96.6 96.93 96.8 94.91 96.28 95.33
```

```
   25    26     27     28     29     30     31     32     33     34     35     36     37
95.13 93.5 93.42 91.56 92.67 96.28 89.66 89.46 91.21 91.42 93.32 89.76 90.18

   38     39     40    41     42     43     44     45     46     47     48     49     50
88.58 87.41 90.33 87.5 87.66 86.06 85.55 84.74 88.78 86.79 88.76 84.95 84.31

   51     52     53     54     55     56     57     58     59     60
87.24 84.73 83.76 84.02 81.75 84.46 82.51 85.42 81.45 85.36
```

The procedure described below teaches us spot rates r_t which minimize the difference between theorically spot values $S(T_i)$ and observated spot values S_i.

Variables: r_t

Minimize:

$$\sum_i \{S(T_i) - S_i\}^2$$

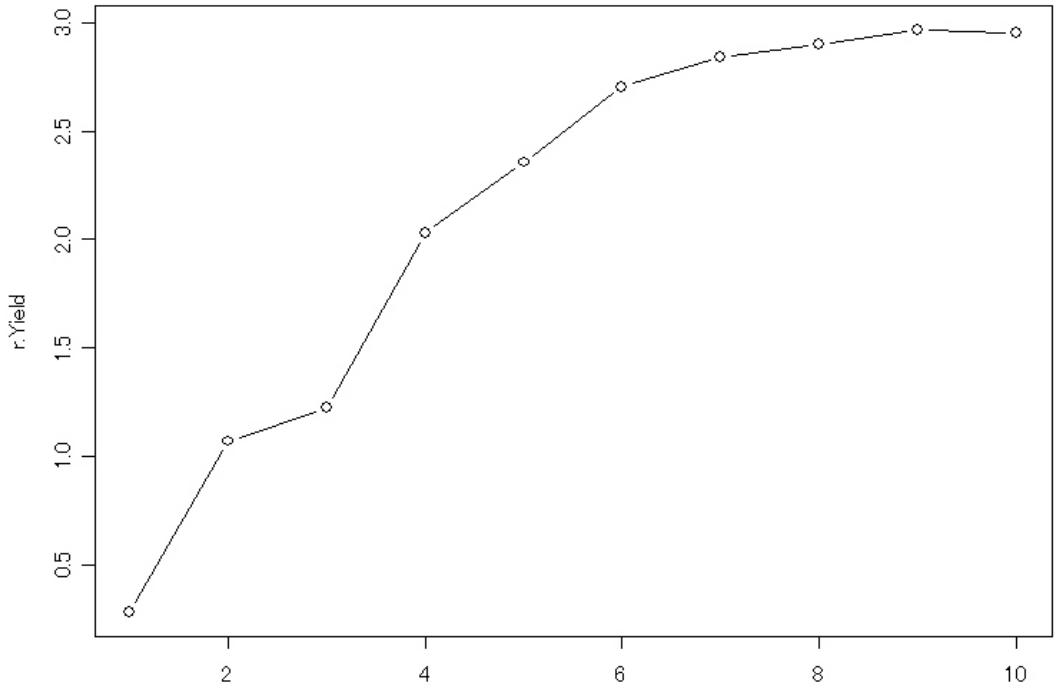
```
#### Yield Curve Estimation ####
Yield <- function(telem.d, tvalue.d, Svalue.d)
{
        Term <- Set()
        Point <- Set()
        t <- Element(set=Term)
        i <- Element(set=Point)
        r <- Variable(index=t)
        telem <- Parameter(index=t,telem.d)
        tvalue <- Parameter(index=i,tvalue.d)
        Svalue <- Parameter(index=i,Svalue.d)
        d <- Expression(index=t)
        d[t] ~ 1 / (1+0.01*r[t])^telem[t]
        S <- Expression(index=i)
        S[i]            ~           Sum(d[t],t,t<=tvalue[i])           +
Sum(100/(1+0.01*r[t])^telem[t],t,t==tvalue[i])
        diff <- Expression(index=i)
        diff[i] ~ S[i] - Svalue[i]
        err <- Objective()
        err ~ Sum(diff[i]*diff[i], i)
        0 <= r[t]
}
```

“tvalue.d”, “Svalue” and “telem.d” means redemption periods, observated values and coefficients of $S(t)$ respectively.

```
# Assign datas
sys <- System(Yield, Yield.telem, Yield.term, Yield.price)
```

```
# Optimize
sol <- solve(sys)
# Get solutions
r <- as.array(current(sys,r))
# Graph out
plot(r,type="b")
```

The below is the yield curve obtained above.



7-2 Estimation of the Rating Transit Matrix

Rating Transit Matrix is a transit matrix that describes transitions of ratings of assets. In this section, we introduce a model to estimate a rating transit matrix of a month given that of a year. Please take notice that Simple Markov Process is assumed.

```
#### Estimation of a Rating Transit Matrix ####
Rating <- function(q0.d)
{
        Rating <- Set()
        i <- Element(set=Rating)
        j <- Element(set=Rating)
        k <- Element(set=Rating)
        q0 <- Parameter(index=dprod(i,j),q0.d)
        q <- Variable(index=dprod(i,j))
```

```
        q2 <- Expression(index=dprod(i,j))
        q4 <- Expression(index=dprod(i,j))
        q8 <- Expression(index=dprod(i,j))
        q12 <- Expression(index=dprod(i,j))
        q2[i,j] ~ Sum(q[i,k]*q[k,j], k)
        q4[i,j] ~ Sum(q2[i,k]*q2[k,j], k)
        q8[i,j] ~ Sum(q4[i,k]*q4[k,j], k)
        q12[i,j] ~ Sum(q8[i,k]*q4[k,j], k)
        diff <- Expression(index=dprod(i,j))
        diff[i,j] ~ q0[i,j] - q12[i,j]
        diffnrm <- Objective()
        diffnrm ~ Sum(diff[i,j]^2, i, j)
        Sum(q[i,j], j) == 1
        0 <= q[i,i] <= 1
        0 <= q[i,j, i!=j] <= 0.05
        q[i,i] ~ 0.9
}
# Assign datas
sys <- System(Rating, Rating.Q0)
# Optimization
sol <- solve(sys)
# Get solutions
q <- as.array(current(sys,q))
# Confirmation
q12 <- diag(1,nrow(q))
dimnames(q12) <- dimnames(q)
for(i in 1:12) q12 <- q12 %*% q
# Display
r.order <- order(rownames(Rating.Q0)) # Define the order of ratings
```

The calculated monthly Rating Transit Matrix is shown below.

```
> round(q[r.order,r.order],digits=4)
       AAA     AA      A    BBB     BB      B    CCC     CC      C
AAA 0.9970 0.0030 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
 AA 0.0031 0.9946 0.0023 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
  A 0.0001 0.0038 0.9945 0.0016 0.0000 0.0000 0.0000 0.0000 0.0000
BBB 0.0000 0.0000 0.0031 0.9952 0.0014 0.0003 0.0000 0.0000 0.0000
 BB 0.0000 0.0000 0.0000 0.0098 0.9885 0.0004 0.0013 0.0000 0.0000
  B 0.0000 0.0000 0.0000 0.0000 0.0069 0.9889 0.0036 0.0006 0.0000
CCC 0.0000 0.0000 0.0000 0.0000 0.0000 0.0025 0.9970 0.0005 0.0000
 CC 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0007 0.9972 0.0021
  C 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0023 0.9977
```

The following is the 12th power of the monthly Rating Transit Matrix.

```
> round(q12[r.order,r.order],digits=4)
       AAA     AA      A    BBB     BB      B    CCC     CC      C
```

```
AAA 0.9649 0.0346 0.0004 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
 AA 0.0355 0.9381 0.0261 0.0002 0.0000 0.0000 0.0000 0.0000 0.0000
  A 0.0014 0.0432 0.9364 0.0186 0.0003 0.0001 0.0000 0.0000 0.0000
BBB 0.0000 0.0007 0.0348 0.9454 0.0152 0.0036 0.0002 0.0000 0.0000
 BB 0.0000 0.0000 0.0018 0.1071 0.8716 0.0045 0.0149 0.0001 0.0000
  B 0.0000 0.0000 0.0000 0.0041 0.0734 0.8752 0.0400 0.0071 0.0001
CCC 0.0000 0.0000 0.0000 0.0000 0.0010 0.0274 0.9650 0.0064 0.0001
 CC 0.0000 0.0000 0.0000 0.0000 0.0000 0.0001 0.0082 0.9674 0.0241
  C 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0001 0.0265 0.9733
```

The below is the given Rating Transit Matrix of a year.

```
> round(Rating.Q0,digits=4)
       AAA     AA      A    BBB     BB      B    CCC     CC      C
AAA 0.9651 0.0349 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
 AA 0.0356 0.9382 0.0262 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
  A 0.0014 0.0433 0.9364 0.0186 0.0003 0.0000 0.0000 0.0000 0.0000
BBB 0.0000 0.0000 0.0352 0.9456 0.0154 0.0038 0.0000 0.0000 0.0000
 BB 0.0000 0.0000 0.0000 0.1078 0.8720 0.0049 0.0153 0.0000 0.0000
  B 0.0000 0.0000 0.0000 0.0000 0.0747 0.8762 0.0410 0.0081 0.0000
CCC 0.0000 0.0000 0.0000 0.0000 0.0000 0.0278 0.9654 0.0068 0.0000
 CC 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0083 0.9675 0.0242
  C 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0266 0.9734
```

Please confirm that the estimation is reliable because the difference is small.

7-3 Logistic Regression

S+NUOPT can handle Logistic regression model. The procedure is given below.

```
#### Logistic Regression Model ####
LogReg <- function(X.d,t.d)
{
    M <- Set()
    i <- Element(set=M)
    L <- Set()
    l <- Element(set = L)
    X <- Parameter(index=dprod(l,i),X.d)
    t <- Parameter(index=l,t.d)
    a <- Variable(index=i)
    a0 <- Variable()
    y <- Expression(index=l)
    y[l] ~ exp(Sum(a[i]*X[l,i],i)+a0) / (1+exp(Sum(a[i]*X[l,i],i)+a0))
    mle <- Objective(type="maximize")
    mle ~ Sum(t[l]*log(y[l])+(1-t[l])*log(1-y[l]),l)
}
```

For example, we discriminate “Iris virsinica” and “Iris versicolor”

```
# Assign Datas
sys <- System(LogReg,LogReg.X,LogReg.t)

# Optimize
sol <- solve(sys)

# Get solutions
a <- as.array(current(sys,a))
a0 <- as.array(current(sys,a0))
```

Next, we evaluate the discriminate function by test datas (which are different from Supervised Datas).

```
# prediction
eta <- a0 + LogReg.test.X %*% as.vector(a)
p <- exp(eta)/(1+exp(eta))
res <- cbind(round(LogReg.test.t,0),round(p,0),eta)
dimnames(res)[[2]] <- c("answer","predict","eta")
res
```

S-PLUS can handle normal logistic regression without NUOPT. The advantage of using S+NUOPT in logistic regression is that S+NUOPT can adjust the model individually. For example, one can add other constraints for parameters, or one can introduce second order terms to the logit function (You also have an option to impose Positive Semidefinite constraints).

8. Solver NUOPT

S+NUOPT is bundled to general purpose optimizer NUOPT. It is invoked by solve() command with the argument a System object constructed by System(). Because the problem type information on the problem is embedded in the System object, NUOPT can make a proper choice of the optimization algorithm and parameter for many cases. But for real-life problems, sometimes the manual choice of the algorithm or parameter may be effective. In the section 8-1 Customizing NUOPT, the way to customize the solver is described. In the following section 8-2, described are some typical errors users encounter using S+NUOPT.

S+NUOPT has an interface (solveQP) through which you can pass S-PLUS data NUOPT directly. Usage of solveQP is described in 8-3 solveQP. This is useful when you already have your problem with "raw-data", constraint matrices, hessian matrices, as S-PLUS objects.

8-1 Set and check parameters by nuopt.options()

NUOPT has various tuning parameters you can set to customize the behavior of NUOPT. The table shows all of the parameters defined.

Name	Default	Possible value	example
method	"auto"	"auto","simplex","asqp", "wcsp","higher","lipm" ," tipm","lepm","tepm","lsq p","tsqp","line","trust"	"simplex"
scaling	"on"	"on","off"	"off"
mipfeasout	"on"	"on","off"	"off"
addToCutoff	-10	double	0.99
cutoff	1.00E+50	double	4.5
eps	-10	double	1.00E-4
epsint	1.00E-04	double	1.00E-8
maxitn	150	int	300
maxnod	-1	int	500000
maxtim	-1(except wcsp),5(wcs p)	int	60
told	1.00E-06	double	1.00E-8
tolx	1.00E-08	double	1.00E-10
maxintsol	-1	int	1
maxmem	-10	int	-100
wcspTryCount	1	int	5
wcspRandomSeed	1	int	23

These parameters are set by `nuopt.options` command, the following are examples.

```
nuopt.options(method="simplex")  # setting the algorithm "simplex"
nuopt.options(eps=1.0e-10) #  setting epsilon
nuopt.options(scaling="off",maxitn=300) # Set scaling and maxitn
```

`nuopt.options` takes ',' separated list of "name=value" as the argument. Current settings of parameter are shown by typing below.

```
nuopt.options()  # seeing current settings of parameters
```

You can save the current setting of parameters by

```
cops <- nuopt.options()  # save the current settings of parameters to cops
```

And load the parameters by

```
nuopt.options(cops)  # load the current settings of parameters from cops
```

Note that parameter settings remain valid throughout the session. You can get the default parameter settings by giving argument "NA" to nuopt.options(), therefore you can reset the parameters to be default by invoking

```
nuopt.options(nuopt.options(NA))  # Set the to be default
```

The default settings are carefully chosen to work well for most of the cases. But default setting is a "safety first" choice, that means users can make better choice if they have some special knowledge on the problem. In the following sections, some typical cases of this kind are described.

8-1-1.Solve Quadratic Programming Problem faster for a special case

Large scale Quadratic Programming Problem (QP: problem with quadratic objective with linear constraints) will appear typically in portfolio optimization. If you are solving QP relatively small number of constraints with many variables, the "asqp" algorithm may be better than the default. This is almost always the case when you are solving large scale Markowitz-type portfolio optimization.

```
nuopt.options(method="asqp")   # solve with asqp
```

8-1-2.Solve Linear/Quadratic Programming Problem more accurate

By default, NUOPT solves Linear Programming Problem (LP: problem with linear objective and all linear constraint) and Quadratic Programming Problem (QP: problem with quadratic objective with linear constraint) by interior point method. It is a better choice in general. But sometimes alternative method "simplex" (for LP) and "asqp" (for QP) may give you useful information. Especially for the case your problem is nearly infeasible (may be difficult or impossible to satisfy all the constraint) or badly scaled (constraints or variables are quite in distant order), interior point method behave unclear. It may fail by iteration limit over (error code 10). In such a case, try setting options

```
nuopt.options(method="simplex")   # for LP
```

or

```
nuopt.options(method="asqp")   # for QP
```

It may take much time for large-scale (over 10000 variables or constraints) problems, but these algorithms are tough and usually give accurate information.

8-1-3.Setting Limit of Calculation Resource

For the case you are solving problem with integer variables, by default NUOPT solve it by branch and bound algorithm. In general, the branch and bound algorithm consumes

much time and memory finding the optimal solution. In such a case you can limit the time and memory as below.

```
nuopt.options(maxtim=600)    # limit the calculation time to 600 sec.
nuopt.options(maxmem=1000)   # limit the memory to 1000M (1G) byte.
nuopt.options(maxmem=-100)   # leave the at least 100M byte of memory
```

If one of these limits has been reached before the optimal solution is found, NUOPT reports timeout or memory error. But the best solution at that time is reported.
Time limitation is required you are using algorithm “wcsp”. It is because the algorithm cannot judge the optimality of the solution. See section 3-3 and following subsection about the usage of method “wcsp” and parameter wcspRandomSeed / wcspTryCount.

8-1-4.Tune the Branch and Bound iteration

You can control branch and bound iteration for problem with integer variables, by giving the following parameters.

```
nuopt.options(maxnod=100000)    # limit the Branch and Bound node 100000
nuopt.options(maxintsol=1)   # only one feasible solution required
nuopt.options(addToCutoff=0.99) # relax the optimality condition of
objective by 0.99
```

If maxintsol is given, the solver terminates after getting specified number of feasible solutions. If addToCutoff is given, the branch and bound iteration cutoff the solution whose improvement is less than addToCutoff. If the solution is known to be integral, setting addToCutoff=0.99 does not lose the exactness of the algorithm.

8-1-5.Tune the Problem or algorithm for nonlinear or semi-definite programming

As seen the examples above, S+NUOPT solves nonlinear / semi-definite programming well by defult. But some difficult situation, it produces inaccurate result or error. In general, the solver fails one of the reasons below. It is worthwhile trying to remove the cause by users.

1. The problem is close to infeasible
   If the constraint is too tight, the algorithm behaves unclear

2. The problem has not well scaled
   Constraint or variable should be well scaled, namely, they are around 1 in magunitude at the solution. Typically bad situation is some variable or constraint is quite large or small comparing to the others.

3. The problem is not convex
   A problem is convex if the feasible region and objective are both convex. The convex problem is a well solved class of problem. But if the problem is not guaranteed to be convex, it is much more difficult than convex ones. That is because the locally optimal solution is not the global solution. S+NUOPT can treat nonconvex class of problem but only guaranteed to give one of the local solutions.

4. The problem is highly nonlinear

If the problem contains exp(x) or 1/x or polynomials of high order, the problem becomes distant from linear. The optimization algorithm more or less approximate the problem as linear inside, it will have trouble in solving highly nonlinear problems.

Now it is recommended the measures below.

A) Check the feasibility of the problem
Loose the constraint or insert slack variables to the constraints and minimize. This provides us information about the size of feasible region.

B) Change the scale of the problem
Scale the variable or constraint manually. Or try with automatic scaling options off may give better result.

```
nuopt.options(scaling="off")  # suppress automatic scaling
```

C) Give initial guess
As explained 2-3-1, give the 'initial guess' to the variables.

D) Change method
S+NUOPT is implemented with many variation of optimization algorithm. Manual choice of the method may give better result.

	Convex	*Non Covex*
Nonlinear	"lipm","line","lepm","lsqp"	"tipm","trust","tepm","lbfgs","tsqp"
Semi-definite	"lsdp"(linear SDP only) ,"csqp"	"trsdp","qnsdp"

By default, Nonlinear problem is always solved by "tipm", semi-definite "lsdp" (for linear SDP) and "trsdp" (nonlinear SDP).

For example, if you know the problem is convex nonlinear programming, setting the method "lipm" may be better.

```
nuopt.options(method="lipm")  # Choose lipm as method
```

The algorithm is generally runs fast comparing to the default algorithm "tipm" because it assumes convexity of the problem.

8-1-6.Clear the heap memory

S+NUOPT uses heap memory for optimization. If such a heap allocated memory piles-up, the optimization will fail by memory error (NUOPT error code 1 or 8). In such a case, unload and re-load NUOPT to clear.

```
module(nuopt,unload="T")      # unload
module(nuopt)                 # re-load
```

8-2 Error messages

In this chapter, some typical errors you encounter using S+NUOPT are described. Pointer to the touble shooting is also given.

8-2-1.Error from Modeling Language Interpreter

Below is the table of typical error messages from the interpreter of modeling language.

Message	*Meaning*
Problem: Syntax error: illegal "," after ..	Missing 'dprod' on RHS of index
Error: Illegal use of summary function on Simple objects	Using 'sum' instead of 'Sum'
Error: SIMPLE object not created in this SPLUS session	Obsolete object
<<SIMPLE 1>> Infeasible bound for variable x (defining infeasible bound …)	Give infeasible bound for variable
<<SIMPLE 67>> Index error in reference of <objname> with no index but should be with index of dimension 1	No index for x defined with index
<<SIMPLE 67>> Index error in reference of <objname> scalar but with index of dimension 1.	With index for x defined without index
<<SIMPLE 82>> Subscript <elem> of <objname> out of range.	Index range error
<<SIMPLE 168>> Objective can only be assigned once.	Multiple assignment to one objective
<<SIMPLE 214>> constraint#1 reduce to … (never satisfied) [/ (always satisfied)]	Infeasible or trivial constraint defined
<<SIMPLE 216>> Trivial and Infeasible constraint appeared.	

Problem: Syntax error: illegal "," after ..

The first message appears you have missed "dprod" required on the RHS of "index=". For example the statement

```
S <- Set()
U <- Set()
x <- Variable(index=(S,U)) # error
```

or

```
S <- Set()
U <- Set()
```

```
i <- Element(index=S)
j <- Element(index=U)
x <- Variable(index=(i,j))  # error
```

produces this error. If you define objects with two or more index, “dprod” is required:

```
x <- Variable(index=dprod(i,j))
```

Error: Illegal use of summary function on Simple objects

You should use “Sum” instead of “sum” in the modeling language. This message warns the confusion:

```
S <- Set()
i <- Element(set=S)
x <- Variable(index=i)
sum(x[i],i) >= 1   # error,  should be  “Sum(x[i],i) >= 1”
```

Error: SIMPLE object created in previous SPLUS session

The S+NUOPT specific objects (System, Variable, Parameter, Set, Element, etc.) do **not be preserved** if you unload S+NUOPT or terminate the S-PLUS session. This message warns you if you try to access obsolete objects. It is recommended to “save” the optimization result as S-PLUS object using as.array or as.list as described in the section 4-4.

<<SIMPLE 1>> Infeasible bound for variable

All the bounds are combined together for every variable. This message means there is a conflict:

```
x <- Variable()
x >= 7
x <= 5          # error (conflict)
```

<<SIMPLE 67>> Index error in reference of <objname>
<<SIMPLE 82>> Subscript <elem> of <objname> out of range

These indicate the inconsistency of the usage of index.

```
x <- Variable()
x[3] >= 1       # error (should not be indexed)
S <- Set(1:3)
y <- Variable(index=S)
y <= 2        # error (require index)
y[4] >= 5     # error (index out of range)
```

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

The “Objective” cannot be assigned more than once.

```
f <- Objective(type=minimize)
x <- Variable()
y <- Variable()
f ~ x+y
f ~ x-y    # error (cannot be assigned twice)
```

<<SIMPLE 214>> constraint#1 reduce to … (never satisfied) [/ (always satisfied)]
<<SIMPLE 216>> Trivial and Infeasible constraint appeared.

Sometimes trivial or infeasible constraint may be defined by mistake. This message warns such a situation.

```
S <- Set(1:3)
i <- Element(set=S)
x <- Variable(index=i)
a <- Parameter(index=i)  # not defined (interpreted as zero)
Sum(a[i]*x[i],i) == 1    # infeasible constraint (LHS is always zero)
Sum(a[i]*x[i],i) <= 0    # trivial constraint (LHS is always zero)
```

8-2-2.Errors from the solver NUOPT

If the optimization algorithm finds error it is displayed as below.

```
<<SIMPLE 193>> Error in solve():
    "<<NUOPT 10>> IPM iteration limit exceeded."    # errors from NUOPT solver
```

You can get the error code (the number after "<< NUOPT") from the returned list by solve() as the item "errorCode". Below is the table of the error messages of NUOPT and its meaning. See the specified section for the trouble shooting.

error Code	*errorMessage*	*Meaning*
1	memory error in preprocessing.	memory error happened before optimization algorithm (see 8-1-6)
2	infeasible(linear constraints and variable bounds)	No feasible solution that satisfies linear constraint and bounds
3	no variables in this model.	no variables found
4	no functions in this model.	no objective/constraint found
5	infeasible variable bounds.	infeasible bound exist
7	internal error.	internal error (report to nuopt-support@msi.co.jp)
8	memory error in optimization phase.	memory error happened in the optimization algorithm (see 8-1-6)
9	step reduction limit exceeded.	line search algorithm, step reduction limit exceeded (see 8-1-5)
10	IPM iteration limit exceeded.	Number of interior point method iteration exceeded the limit (see 8-1-2, 8-1-5)
11	infeasible	No feasible solution exist
13	unbounded.	unbounded problem (There is a sequence of feasible solution that make objective arbitrally small or large)

14	integrality is violated.	integer variable is not integral at the given solution. (You have invoked interior point method for problem with integer variables)
15	simplex misapplied to QP.	You can not apply simplex method for Quadratic Programming Problem
15	simplex/asqp misapplied to NLP.	You can not apply simplex or asqp method for general Nonlinear Programming Problem
16	Infeasible MIP.	No integer feasible solution exist (non-integer feasible solution may exist)
17	B & B node limit reached (with feas.sol.).	Number of Branch & Bound node exceed the limit, but found feasible solution (see 8-1-4)
18	MIP iteration failed (with feas.sol.).	Numerical problem occurred in the Branch & Bound iteration, but found feasible solution
19	B & B node limit reached (no feas.sol.).	Number of Branch & Bound node exceed the limit, and no feasible solution found (see 8-1-4)
20	MIP iter. failed (no feas.sol.).	Numerical problem occurred in the Branch & Bound iteration, and no feasible solution found
21	B & B itr. timeout (with feas.sol.).	Branch & Bound iteration exceeded the time limit, but found feasible solution (see 8-1-3)
22	B & B itr. timeout (no feas.sol.).	Branch & Bound iteration exceeded the time limit, and no feasible solution found (see 8-1-3)
27	SIMPLEX iteration limit exceeded.	Number of SIMPLEX iteration exceeded the limit.
28	higher-order method is only for LP.	You cannot apply "higher" to NLP
29	iteration diverged.	Iteration vector diverged (may be the problem is unbounded)

33	Bound violated.	The bound violation detected after reverting the scale(see 8-1-5). Maybe the problem is badly scaled.
34	Bound and Constraint violated.	The bound/constraint violation detected after reverting the scale(see 8-1-5). Maybe the problem is badly scaled.
35	Constraint violated.	The constraint violation detected after reverting the scale(see 8-1-5). Maybe the problem is badly scaled.
36	Equality constraint violated.	Equality constraint is violated after reverting the scale(see 8-1-5). Maybe the problem is badly scaled.
37	B&B terminated with given # of feas.sol.	Branch & Bound iteration has found given number of feasible solution and terminated (see 8-1-4)
38	dual infeasible.	The problem is dual infeasible.
39	IPM iteration timeout.	Interior point method iteration exceeded thetime limit (see 8-1-5)
40	SQP iteration limit exceeded.	Number of Sequencial Quadratic Programming iteration has reached the limit. (see 8-1-5)
41	SQP internal error.	internal error (report nuopt-support@msi.co.jp)
43	B&B memory error (with feas.sol.).	memory error happened durning the Branch & Bound iteration (see 8-1-3), but found feasible solution
44	B&B memory error (no feas.sol.).	memory error happened durning the Branch & Bound iteration (see 8-1-3), and found no feasible solution
46	trust region too small	Trust region iteration fails (see 8-1-5)
47	Continuous Variable <varname> cannot be included in model for wcsp.	wcsp cannot deal with model with continuous problem
48	Variable <varname> appear in two selection()	Variable cannot appear more than one selection statement,
49	Variable <vaname> is fixed to infeasible value.	Constraint cannot be satisfied unless <varname> is fixed to outside its bound

50	Both of two variables <varname1> and <varname2> cannot be 1.	<varname1> and <varname2> is in a same selection statement, but both of them are fixed to one.
51	wcsp is not available without SIMPLE.	wcsp cannot used from solveQP()
52	<number>th selection() statement has no movable binary variables.	All the variables in a certain selection statement is not movable
54	Constraint <name>'s weight is <value> should be -1 or non-negative value.	Soft constraint weight should be -1 (default) or positive
55	exterior solution obtained.	tepm/lepm found solution outside feasible region (see 8-1-5)
110	CF failed at getcky	numerical problem in SDP see (8-1-5)
111	CF failed at logdet	numerical problem in SDP see (8-1-5)
112	InvMat cannot obtained at calxx1	numerical problem in SDP see (8-1-5)
113	GSEP failed at minevl	numerical problem in SDP see (8-1-5)
114	trianglization failed at minevl	numerical problem in SDP see (8-1-5)
115	Minimum eigenValue cannot obtained	numerical problem in SDP see (8-1-5)
120	PDgap is too large[PDfeasible]	Iteration terminate with Primal-dual gap large, but primal/dual feasible solution is obtained.
121	PDgap is too large[Pfeasible]	Iteration terminate with Primal-dual gap large, but primal feasible solution is obtained.
122	minus stepsize detected	numerical problem in SDP see (8-1-5)
123	The SDP constraint cannot be treated by specified algorithm.	Not compatible algorithm for SDP
124	No SDP constraint is detected.	non-SDP problem but invoked SDP algorithm

8-3 solveQP

S+NUOPT has an interface solveQP, through which you can pass S-PLUS object directly to the solver kernel NUOPT. solveQP is to solve mixed integer Linear or Quadratic programming.

Minimize $\frac{1}{2}x^T Qx + q^T x$

Subject to

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

The argument is given as below.

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

The following is the description of the arguments. All arguments but objQ can be omitted.

ObjQ: (matrix or list)

Hessian matrix Q

ObjL: (vector) <optional>

Linear part of the objective q, if omitted regarded as zero.

A: (maxtrix or list) <optional>

Constraint matrix A, if omitted regarded as zero.

cLO,cUP: (vector) <optional>

Lower and upper bound of the constraints. If omitted, regarded nonexistent.

bLO,bUP: (vector) <optional>

Lower and upper bound of the variables. If omitted, regarded nonexistent.

x0:(vector)<optional>

Initial guess of the variable. If omitted regarded not given.

isint:(vector)<optional>

Logical vector if the variable is integral or not. If omitted regarded all false (continuous).

objQ and A can be of S-PLUS matrix or list. The list consists of three items, row-number, column-number and value vector. The length of the vector is the number of non-zero elements.

You can give A by

```
> A                # matrix
      [,1] [,2] [,3] [,4]
 [1,]  1.1  0.0 -1.3  0.0
 [2,]  0.0 -2.2  0.0  2.4
```

And alternatively by

```
X <- list(c(1,1,2,2), c(1,3,2,4), c(1.1,-1.3,-2.2,2.4))
```

The list expression is suitable for sparse case.