

RSIMPLE マニュアル

株式会社 NTT データ数理システム

2022/6/28 更新

目次

1	はじめに	4
1.1	本書の使い方	4
1.2	RSIMPLE の構造	5
1.3	RSIMPLE に付属しているモデル	5
1.4	RSIMPLE をはじめる	6
2	数理計画モデルの記述（基本編）	8
2.1	定数（Parameter）、集合（Set）の宣言と定義	9
2.1.1	集合（Set）の宣言と初期化	11
2.1.2	定数（Parameter）の宣言と初期化	12
2.2	集合の要素（Element）の宣言と利用	12
2.3	変数（Variable）の宣言と利用	13
2.3.1	変数（Variable）に初期値を与える	13
2.4	式オブジェクト（Expression）の宣言と代入演算子による定義	14
2.5	目的関数オブジェクト（Objective）の宣言と代入演算子による定義	15
2.6	添字の範囲の和を取る演算子（Sum）	15
2.7	定式の展開（System）と最適化の実行（solve）	16
2.7.1	数理計画モデルのデータを引数として System から与える	20
2.8	制約式 Constraint の追加	21
2.8.1	制約式の名前付け, 除去, 復活, 追加 (delete.con, restore.con, add.con)	22
2.9	整数変数 IntegerVariable	24
3	数理計画問題の記述（応用編）	26
3.1	条件式と資金調達問題	26
3.2	SymmetricMatrix と最小固有値問題	29
3.3	wcsp と集合分割問題	31
3.3.1	1 指標の 2 分割問題	31
3.3.2	wcsp による 3 指標の 2 分割問題	32
3.3.3	wcsp による多分割問題 (selection とハード, セミハード, ソフト制約)	37
3.3.4	乱数の種の設定による解の違い	39
3.3.5	アルゴリズム wcsp を使うときの注意点	40
3.4	大学駅伝エントリー最適化問題	40
4	R とのデータ連携	43
4.1	R オブジェクトから RSIMPLE オブジェクトへの変換	43
4.1.1	integer / numerical	43

4.1.2	vector	43
4.1.3	array / matrix	44
4.1.4	list	44
4.1.5	data.frame	45
4.1.6	自動代入機能について	46
4.2	RSIMPLE オブジェクトから R オブジェクトへの変換	46
4.2.1	変数や式 (Variable/Expression) の値を array 型のデータとして取り出す	47
4.2.2	変数や式 (Variable/Expression) の値を list 型のデータとして取り出す	49
5	ポートフォリオ最適化	52
5.1	基本的なマルコビッツモデル	52
5.2	さまざまなリスク尺度によるポートフォリオ最適化	57
5.2.1	分散をリスクとした場合	57
5.2.2	絶対偏差をリスクとした場合	59
5.2.3	1 次の下方部分積率 (LPM) をリスクとした場合	61
5.2.4	CVaR をリスクとした場合	63
5.3	コンパクト分解 (大規模ポートフォリオ最適化)	64
5.4	端株処理	66
5.5	銘柄グルーピング	69
5.6	Maximum Drawdown	72
5.7	Sharpe Ratio	75
6	半正定値計画法の利用	78
6.1	半正定値行列の取得	78
6.2	ロバスト最適化	80
7	非線形フィッティング	84
7.1	イールドカーブのフィッティング	84
7.2	格付け推移行列の推定	87
7.3	ロジスティック回帰	89
8	最適化ソルバー NUOPT	92
8.1	nuopt.options() を使って NUOPT をカスタマイズする	92
8.1.1	特殊な大規模二次計画問題を高速に解く	93
8.1.2	線形計画問題・二次計画問題をより高精度で解く	94
8.1.3	計算機資源の利用を制限する	94
8.1.4	分枝限定法のチューニング	94
8.1.5	非線形計画法, 半正定値計画法の安定化のためのチューニング	95
8.1.6	ヒープメモリをクリアする	96
8.2	エラーメッセージ	97

8.2.1	モデリング言語解釈部からのエラー	97
8.2.2	NUOPT が出力するエラー	100
8.3	solveQP	104
9	補足 : SNUOPT をお使いの方へ	107

1 はじめに

RSIMPLE は汎用の数理計画法パッケージ Nuorium Optimizer (以下 NUOPT) の R インターフェースです。「汎用」というのは様々な分野に応用できることを示している一方で、具体的な結果を出すためには「ひと手間」かけねばならないということを同時に示しています。このマニュアルは統計や金融など、具体的な応用を抱えているユーザーの方々を想定して、その「ひと手間」をできるだけ軽減することを目的として書かれました。

RSIMPLE と R の最大の違いは、問題の定式(「数理計画モデル」と呼びます)が必要となる点です。たとえば R には重回帰を行う `lsfit` という手続きがあらかじめ用意されていて、ユーザはデータを決まったフォーマットで用意してこの手続きをコールすれば結果が得られます。一方、RSIMPLE では重回帰一つ行うのにも、何が変数で、何を最小化したいのかという数理計画モデルを記述する必要がある、一般に導入コストがかかります。ただし、モデル記述の幅は非常に柔軟でありかつ広く、通常重回帰モジュールとは異なりパラメータの範囲を設定することや、パラメータ同士の関係式(「制約」と言います)を定義すること、またパラメータを自動的に絞り込むといったことも可能です。そこで、このマニュアルでは導入部分に関してはできるだけ豊富なモデルとデータのサンプルを与え、その応用の筋道を示すことによって、RSIMPLE の可能性を実感していただくようにしました。

1.1 本書の使い方

「RSIMPLE に付属しているモデル」でこのマニュアル内で取り上げた(パッケージに含まれている)サンプルの一覧とその書いてある場所を示します。まずはご自身の興味のあるところを拾い読みして感じをつかんでいただければと思います。

最適化パッケージを利用する上で「数理計画モデル」作成は避けて通ることができません。しかしながら、サンプルで要領をつかんでしまえば、かなり少ない手間でご自身が抱えている問題に役立てるように書き換えることができる場合があります。特に NUOPT の大きな応用分野である金融ポートフォリオについては主要なものをほぼ含んでいます。このため、金融ポートフォリオと最適化の関係については本パッケージとマニュアルを用いればほぼ体感できるのではと考えています。

「数理計画モデルの記述(基本編)」では、簡単な重回帰モデルを題材として数理計画モデルを記述する基本的な道具立てについて網羅的に解説しています。

「数理計画問題の記述(応用編)」は比較的発展的な使い方を想定して、特定の場合に有効なモデル記述法・解法について解説しています。

「R とのデータ連携」では、R でのデータ解析を実際に行われている方を想定し、お持ちのデータをどのようにしたら RSIMPLE に渡して、最適化のためのデータとすることができるのかを解説します。

「ポートフォリオ最適化」・「半正定値計画法の利用」・「非線形フィッティング」はそれぞれ例題とその解説です。

「最適化ソルバー NUOPT」では、最適化ソルバー NUOPT のオプションの解説や、エラーメッセージと回避法について解説しています。

1.2 RSIMPLE の構造

RSIMPLE は汎用統計解析パッケージである R の環境の中で対話的に最適化が行えるというところに最大の特徴があります。RSIMPLE が扱うオブジェクトは大きく分けて以下の 3 つがあります。

- 数理計画モデルの入出力データ（変数の上下限や解など）
- 数理計画の定式のパターンを記述したモデルそのもの
- 数理計画モデルとデータを結び付け、解析の対象となる具体的な問題

RSIMPLE ではこれらをそれぞれデータ、モデル、システムと呼び、いずれも R 環境の中のオブジェクトとして扱います。その構造は次のように記述できます。

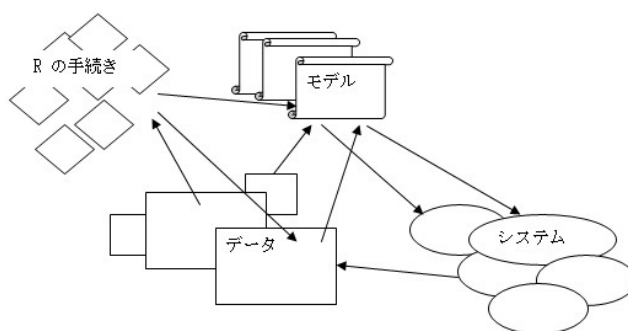


図1 RSIMPLE の構造

問題（システム）を解いた結果もまたデータとして R 環境に蓄えられることになりますので、最適化を連鎖的に行ったりすることも RSIMPLE の中で可能です。

RSIMPLE の特徴はモデルの記述が R に埋め込まれた別の言語に従って行われるということです。「数理計画モデルの記述（基本編）」や「数理計画問題の記述（応用編）」でその記述の具体的な意味と方法について解説いたします。

R には統計演算をはじめとする充実したデータ処理機能があり、その機能を使えば最適化モデルで利用するデータを素早く準備することができます。R が備えている実際的なデータ例（例えば “freeny” や “state” はこのマニュアルでも利用しています）も最適化モデルのテスト的な入力として適しています。さらに、最適化の結果も R のデータとして扱われますので、R が持つ豊富な図示機能を結果の理解に役立てることができます。

1.3 RSIMPLE に付属しているモデル

RSIMPLE には次のような例があらかじめ組み込まれています。興味のある例題を中心に試して感触をつかむのも良い方法です。

例題	内容	モデル名
重回帰	データ freeny.x/y に特化	Nlsfit
	汎用	Nlsfit.gen
	パラメータの線形制約付き	Nlsfit.eq
	名前つき制約	Nlsfit.eq.named
	パラメータの選択付き	Nlsfit.int
キャッシュフローマッチング	資金調達の問題（条件式の応用）	Cashflow
半正定値計画導入	最小固有値の取得	MinLambda
集合の分割	1 指標 2 分割	Half
	3 指標 2 分割	Half2
	1 指標多分割	Partition
ポートフォリオ最適化	マルコビッツモデル基本	Marko
	分散	MinVar
	絶対偏差	MinMad
	1 次の下方部分積率	MinLPM1
	CVaR	MinCVaR
	コンパクト分解	MinVar
	Maximum Drawdown	MinMaxDD
	Sharpe Ratio 最大化	Sharpe
離散最適化とポートフォリオ	端株処理	RoundLot
	銘柄のグルーピング	Basket
半正定値計画の応用	ロバスト最適化	Robust
	半正定値行列の生成	Cormat
非線形回帰	イールドカーブの推定	Yield
	格付け推移行列の推定	Rating
	ロジスティック回帰	LogReg

1.4 RSIMPLE をはじめる

RSIMPLE を RGui で使用する際には, R Console に次のように入力してください.

```
> library("Rnuopt")
```

次の画面のように「Rnuopt *.*.(NUOPT *.*.) for R *.*. for Microsoft Windows」とバージョン情報が表示されれば使用準備が完了したことになります.

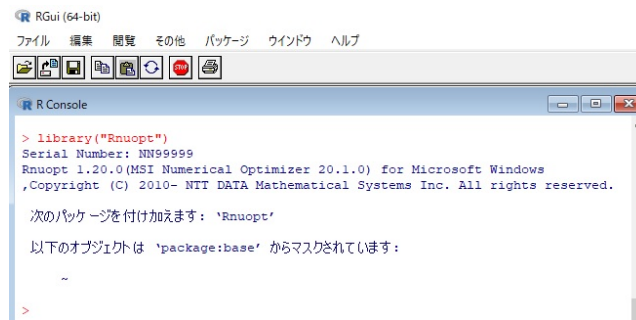


図2 Rnupt の追加

2 数理計画モデルの記述（基本編）

ここでは R の組み込み関数 `lsfit` が行っている重回帰モデルを例にとって、RSIMPLE におけるモデル記述について解説します。この章において、モデル記述の道具はほぼすべて現れます。より高度な機能や各手法に特化した記述については「数理計画問題の記述（応用編）」で触れています。

そもそも関数 `lsfit` を用いることにより、被説明変数を説明変数の線形な関数で記述したときの残差の二乗和を最小化していることになります。これを数理計画問題という視点から見ると、次のような問題を解いているとみなすことができます。

-
- 目的関数:
 - 残差の二乗和 $\sum_{i \in S} e_i^2 \rightarrow$ 最小化
 - $e_i \equiv (v_0 + \sum_{j \in P} X_{i,j} v_j) - Y_i, (i \in S : \text{観測点})$
 - 変数:
 - v_0 (定数項)
 - $v_j (j \in P : \text{線形関数の一次の係数の集合})$
 - 定数:
 - $X_{i,j}$ (観測点 $i \in S$ における説明変数 $j \in P$ の値)
 - Y_j (観測点 $i \in S$ における被説明変数の値)
-

RSIMPLE で数理計画問題を解くには、上記のような問題を記述する数式の情報を独自の記述方法で表現し、R の手続きとしておく必要があります。以降ではこの作業のことを「(数理計画) モデル記述」と呼びます。次が上記の数式に対応するモデル記述です。なお、モデル記述の中身については「定数 (Parameter)、集合 (Set) の宣言と定義」以降で順に解説します。

```
Nlsfit <- function() {  
  # freeny.y を Y として, freeny.x を X として用いる準備  
  ydata <- as.vector(freeny.y)  
  S <- Set()  
  Y <- Parameter(index=S, as.array(ydata))  
  P <- Set()  
  X <- Parameter(index=dprod(S, P), freeny.x)  
  # ここからモデル記述本体  
  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 定数 (Parameter) , 集合 (Set) の宣言と定義

最初の部分は, 重回帰に `freeny.x` および `freeny.y` という R ビルトインのデータ (説明は `help(freeny)` にあります) を用いるための準備です.

```

# freeny.y を vector 型に変換
ydata <- as.vector(freeny.y)
# 観測点の添字集合の宣言
S <- Set()
# 観測点の値の宣言と初期化
Y <- Parameter(index=S,as.array(ydata))
# 説明変数の集合の宣言
P <- Set()
# 観測点における説明変数の値の定義
X <- Parameter(index=dprod(S,P),freeny.x)

```

定数については宣言と同時に, R オブジェクトを渡して初期化を行っています. 初期化した結果, その内容が意図通りとなっているかは, コマンドを入力することで確認できます. 具体的には R Console 画面で

```

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

```

と入力した後に, Y と X を表示させます.

```

> Y
      1      2      3      4      5      6      7      8      9     10
8.79236 8.79137 8.81486 8.81301 8.90751 8.93673 8.96161 8.96044 9.00868 9.03049
     11     12     13     14     15     16     17     18     19     20
9.06906 9.05871 9.10698 9.12685 9.17096 9.18665 9.23823 9.26487 9.28436 9.31378

```

```

      21      22      23      24      25      26      27      28      29      30
9.35025 9.35835 9.39767 9.42150 9.44223 9.48721 9.52374 9.53980 9.58123 9.60048

      31      32      33      34      35      36      37      38      39
9.64496 9.64390 9.69405 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
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] "*" "*"

```

どんな R オブジェクトを定数の初期化に用いることができるかは「R とのデータ連携」で具体的に解説しています.

2.1.1 集合 (Set) の宣言と初期化

集合の宣言と初期化は次のように行ないます.

```

名前 <- Set() # 添字, 初期化なし
名前 <- Set(集合の内容) # 添字なし
名前 <- Set(集合の内容,dim=次元) # 多次元集合
名前 <- Set(集合の内容,index=添字集合/要素) # 添字あり
名前 <- Set(集合の内容,dim=次元,index=添字集合/要素)# 添字あり多次元集合

```

なお, 「集合の内容」の部分には R のベクトル型を与えることができます. 次の例では 3 つの集合 S, S1, S2 の宣言と初期化を行なっています.

```

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

```

ここで, 実際に集合の中身を確認してみましょう.

```

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

```

RSIMPLE では宣言のときに集合の内容を与えなくても、定数の入力の際に同時に定義される機能（自動代入）があります。このため、初期化については不要な場合もあります。

2.1.2 定数（Parameter）の宣言と初期化

RSIMPLE では、最適化において変化しない量はすべて定数（Parameter）と呼びます。定数の宣言と初期化の一般的な形は以下の通りです。

```
名前 <- Parameter() # 添字, 初期化なし
名前 <- Parameter(定数の内容) # 添字なし
名前 <- Parameter(index=添字集合/要素) # 初期化なし
名前 <- Parameter(index=添字集合/要素, 定数の内容) # 初期化, 添字あり
```

定数は添字を持つことが出来ます。添字を持つ定数の場合、index= の後には、添字を示す集合や要素の組を並べます。「定数の内容」は array 型もしくは list 型の R オブジェクトです。以下に定数の宣言と初期化の例をいくつか挙げます。

```
p <- Parameter(2.85)
q <- Parameter() # 0 だと解釈されます
Y <- Parameter(index=S, as.array(ydata))
i <- Element(set=S) # 集合の要素の宣言
a <- Parameter(index=i, as.array(ydata)) # 要素を添字として並べてもよい
X <- Parameter(index=dprod(S,P), freeny.x)
X <- Parameter(index=dprod(i,P), freeny.x) # 要素と集合を混ぜてもよい
```

上の例の X の記述のように、宣言において二つ以上の集合（もしくは要素）をならべるときには dprod() を用いなければならないことにご注意ください。

定数に値がうまく代入されているかどうかは R Console から確認することができます。詳しくは「R とのデータ連携」をご覧ください。

2.2 集合の要素（Element）の宣言と利用

モデル記述 Nlsfit では、Set, Parameter の宣言のあとに、数理計画問題本体の記述があります。

```
i <- Element(set=S) # 観測点の集合 S 中の要素を宣言
j <- Element(set=P) # 説明変数の集合 P 中の要素を宣言
v <- Variable(index=j) # j で添字付けられる変数 v を宣言
v0 <- Variable() # スカラーの変数 v0 を宣言
e <- Expression(index=i) # i で添字付けられる式 e を宣言
e[i] ~ Sum(X[i,j]*v[j],j) + v0 - Y[i] # e[i] の定義
```

```
esum <- Objective(type=minimize) # 目的関数の宣言
esum ~ Sum(e[i]*e[i],i) # 目的関数の定義
```

上の記述の内、式の実体の定義に対応するのは $e[i]$ の定義と目的関数 $esum$ の定義のみで、その他は宣言です。

また、最初の部分では式を表現するときの添字として用いる、集合の要素（Element）オブジェクトの宣言を行っています。

集合の要素は次のように宣言します。

```
名前 <- Element(set=集合)
```

Element は宣言されたときの集合を常に覚えており、式の中で使われるとその集合の範囲内を回ります。たとえば、上の例では i という要素オブジェクト（Element）は、

```
e[i] ~ Sum(X[i,j]*v[j],j) + v0 - Y[i] # e[i] の定義
esum ~ Sum(e[i]*e[i],i) # 目的関数の定義
```

のように使われています。 $e[i]$ の定義については、 i が定義されている集合 S の要素すべてについて記述したのと同じことになります。また、Sum については二つ目の引数に与えられた要素すべてについて和を取るものと解釈されます。

また、要素オブジェクトは他のオブジェクトの添字を定義するときに、集合の代わりに使うことができます。以下がその例です。

```
v <- Variable(index=j) # Variable(index=P) と同じ意味
e <- Expression(index=i) # Expression(index=S) と同じ意味
```

2.3 変数（Variable）の宣言と利用

変数は数理計画問題における未知の値のことです。「数理計画モデルの記述（基本編）」で取り上げた問題においては v (j という添字付き) と $v0$ (スカラー) という変数が定義されています。

変数の宣言の形式は次の通りです。

```
名前 <- Variable() # 添字なし（スカラー）
名前 <- Variable(index=添字集合/要素) # 添字あり
```

2.3.1 変数（Variable）に初期値を与える

変数の初期値について、デフォルトでは 0 となりますが、変数に対する代入演算 \sim を用いて具体的な値を与えることもできます。

次が変数に初期値を与えている例です。

```
v0 ~ 2.5 # v0 の初期値は 2.5 とする
v[j] ~ 1 # v[j] すべてに 1 という初期値を設定する
```

RSIMPLE において、変数が初期値のままでは値または微係数が評価できない式が存在する場合には、適切な初期値を自分で設定する必要があります。x を変数として、

```
1.0/x # 割り算（原点において値が評価できない）
sqrt(x) # 平方根（原点において微係数が評価できない）
```

といった関数がその例です。このような場合には 1 などの無難な初期値を与えておく工夫をすることにより、計算が安定しやすくなります。

2.4 式オブジェクト (Expression) の宣言と代入演算子による定義

モデル記述 Nlsfit において e[i] は

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

の e_i に相当します。このようにまとめた式の断片に名前を付けるときには、式オブジェクト (Expression) を使います。宣言の形は Parameter と同様です。

```
名前 <- Expression() # 添字なし
名前 <- Expression(index=添字集合/要素) # 添字あり
```

Expression は具体的な内容を代入しなければ常に値 0 と解釈されます。これは、初期化されていない Parameter でも同様です。

Expression や Parameter に値を代入するには、代入演算子 ~ を用います。= ではないことに注意してください。

代入の際の注意点として、代入の左辺には単体の Expression/Parameter オブジェクトが来なければなりません。このため、次の記述はエラーとなります。

```
x[i] + y[i] ~ z[i] # エラー
```

また、意味上右辺の添字は残らず左辺の添字に表れる必要があります。よって、次の記述は正しくなくエラーとなります。

```
z ~ x[i] # エラー
```

さらに、次のような記述も、添字 j のどの要素に関する記述なのかわからないのでエラーになります。

```
y[i] ~ u[i,j] # エラー
```

なお、左辺の添字が右辺にかならずしも表れる必要はありません。次の例は、「w[i,j] に添字 j にはよらず同じ式 x[i] を代入する」ということを表している正しい記述です。

```
w[i,j] ~ x[i] # 正しい記述
```

2.5 目的関数オブジェクト (Objective) の宣言と代入演算子による定義

目的関数とは、最小化、あるいは最大化したい式の内容で、RSIMPLE では目的関数オブジェクト (Objective) で表現します。なお、Objective は特殊な Expression と見なすことができ、目的関数の定義の際には Expression の時と同様に具体的な式を代入します。

Objective の宣言の際には、次のように最小化すべきなのかあるいは最大化すべきなのかを与えます。目的関数は一つでなければならないという性質上、Objective は添字を持つことが出来ません。

```
名前 <- Objective(type=minimize) # 最小化
```

```
名前 <- Objective(type=maximize) # 最大化
```

次は目的関数の宣言および定義の例です。

```
esum <- Objective(type=minimize) # 目的関数の宣言
```

```
esum ~ Sum(e[i]*e[i],i) # 目的関数の定義
```

2.6 添字の範囲の和を取る演算子 (Sum)

数理計画問題を記述する際には、次の式：

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

のように「添字の範囲内で和を取る」という操作がよく現れます。RSIMPLE において、上の式は Sum 演算を用いて次のように記述します。

```
Sum(e[i]*e[i],i) # e[i]*e[i] の和
```

Sum 演算は一般に次のような形式で記述します。

```
Sum(式, 添字)
```

```
Sum(式, 添字 1, 添字 2, ..., 添字 n)
```

```
Sum(式, 添字 1, 添字 2, ..., 添字 n, 条件式 1, 条件式 2, ..., 条件式 m)
```


このように記述することで、「添字」のわたる範囲で「式」の和を取ります。なお、「条件式」は、和を取る範囲を制限する場合に記述します。

なお、R 組み込みのもの（sum）とは違って、先頭が大文字（Sum）であることにご注意ください。RSIMPLE のオブジェクトに R の sum 演算を適用することはできません。

最後に、Sum 演算を用いた例を 3 つ以下に示します。

```
Sum(a[i],i) # i について a[i] の和を取る
Sum(M[i,j],i,j) # i と j について M[i,j] の和を取る
Sum(G[i,j],i,j,i>j,i!=0) # i>j かつ i!=0 である範囲内で G[i,j] の和を取る
```

2.7 定式の展開（System）と最適化の実行（solve）

定義したモデル Nlsfit を実際に解くには、次の 2 つの記述をする必要があります。

```
sys <- System(Nlsfit) # 数理計画モデルの展開
sol <- solve(sys) # 展開したシステムを解く
```

System は、数理計画モデルを定義した手続きを与え、データと連結して具体的な数理計画問題を作成します。エラー無く正しく動作した場合、戻り値として問題そのものを示す System オブジェクトを返します。

System は一般に次のように記述します。

```
名前 <- System(モデルを記述した手続き) # モデル定義に引数がない場合
名前 <- System(モデルを記述した手続き, 引数 1, 引数 2, ..., 引数 3) # モデル定義に引数がある場合
```

上記で sys を表示させてみた結果は次のようになり、式（この場合には目的関数）にデータが代入され、展開されていることがわかります。

```
> sys
:-1:info: (objective) name="esum" (8.79636*(v[lag quarterly revenue])+4.70997*(v[price
↪ index])+5.8211*(v[income level])+12.9699*(v[market potential])+v0+(-8.79236))*(8.79636*(v[lag
↪ quarterly revenue])+4.70997*(v[price index])+
...(中略)...
+(-9.77536)*(9.74924*(v[lag quarterly revenue])+4.27839*(v[price index])+6.19377*(v[income
↪ level])+13.1625*(v[market potential])+v0+(-9.77536))+(-9.77536*(v[lag quarterly
↪ revenue])+4.27789*(v[price index])+6.2003*(v[income level])+13.1664*(v[market
↪ potential])+v0+(-9.79424))*(9.77536*(v[lag quarterly revenue])+4.27789*(v[price
↪ index])+6.2003*(v[income level])+13.1664*(v[market potential])+v0+(-9.79424)) (minimize)
```

こうして生成した System オブジェクトを solve に与えると、次のように最適化の経過が出力されます。

```
> sol <- solve(sys)

[About Numerical Optimizer]
MSI Numerical Optimizer 20.1.0 (NLP/LP/IP/SDP module)
    <with META-HEURISTICS engine "wcsp"/"rcpsp">
    <with Netlib BLAS>
    , Copyright (C) 1991 NTT DATA Mathematical Systems Inc.

[Problem and Algorithm]
NUMBER_OF_VARIABLES                5
NUMBER_OF_FUNCTIONS                1
PROBLEM_TYPE                       MINIMIZATION
METHOD                             TRUST_REGION_IPM

[Progress]
<preprocess begin>.....<preprocess end>
<iteration begin>
    res=4.9e-01 ... 9.4e-12
<iteration end>

[Result]
STATUS                             OPTIMAL
VALUE_OF_OBJECTIVE                 0.007374997679
ITERATION_COUNT                    4
FUNC_EVAL_COUNT                    8
FACTORIZATION_COUNT                7
RESIDUAL                           9.399619971e-12
ELAPSED_TIME(sec.)                 0.03
```

solve の書式は次のようになります。

```
名前 <- solve(System オブジェクト) # デフォルト
名前 <- solve(System オブジェクト, trace=F) # 出力抑制
```

solve の戻り値は、最適化の結果を R のリスト型のオブジェクトとして示したものです。たとえば今解いた Nlsfit の場合、solve の戻り値のリスト型を表示すると次のようになります。

```

> sol
$variables      # 変数値
$variables$v    # 変数 v
      income level lag quarterly revenue      market potential
      0.7674609      0.1238646      1.3305577
      price index
      -0.7542401
attr(,"indexes")
[1] "j"

$variables$v0    # 変数 v0
[1] -10.47261

$objective      # 最適解における目的関数値
[1] 0.007374998

$counter
  iters fevals  vars
    4      8     5

$termination
  tolerance residual
1.00000e-08 9.39962e-12

$errorCode
[1] 0

$elapsed.time
[1] 0.02700001

```

特定の変数の値のみを取り出したいときには次のようにします。

```
sol$variables$v$current # v の現在値 (current) を表示
```

また, 変数に関するサマリ情報を表示させるには次のようにします。

```
summary(sol)
```

ここで、実際にサマリ情報を表示させて見ると以下ようになります。

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

$variables$v0
      current      dual      lower      upper
"-10.4726071080616"    "0"    "-Inf"    "Inf"
      initial
      "0"
... (以下は戻り値の表示と同様) ...
```

最後に、R 組み込みの `lsfit` の結果と `RSIMPLE` の結果が一致していることを確認するため、`lsfit` での結果を示しておきます。

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

      Estimate Std.Err t-value Pr(>|t|)
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 数理計画モデルのデータを引数として System から与える

今まで解説してきた手続き Nlsfit では、データ freeny.x および freeny.y をデータとして利用するように書かれていました。このように手続きの中で直接データを指定すると、データが代わるたびに新たな手続きを用意する必要があり不便です。

そこで、次のようにデータの部分を引数として書くことにより、同一の形のデータならどのようなものでもデータとして利用できるようになり利便性が向上します。

```
# データを問わない一般の重回帰モデル

Nlsfit.gen <- function(x,y) { # x が説明変数 y が被説明変数

  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(e[i]*e[i],i)
}
```

Nlsfit.gen ではデータの名前は System コマンドを実行する際の引数として与えます。R にビルトインされている stack.x,stack.loss (help(stack) でデータの説明が得られます) は、freeny.x,freeny.y と同様の説明変数と被説明変数の組です。同じ Nlsfit.gen から二つのデータに対してシステムを生成して、解いてみましょう。

```
sys1 <- System(Nlsfit.gen,freeny.x,freeny.y) # freeny.x/y を与える
sol1 <- solve(sys1)
sys2 <- System(Nlsfit.gen,stack.x,stack.loss) # stack.x/Loss を与える
sol2 <- solve(sys2)
```

上記のようにすることで、同じモデルから二つの数理計画問題が生成され、解かれることになります。それぞれの結果は、次のようにすると確認することができます。

```
> current(sys1,v) # freeny.x,freeny.y から生成した問題の結果

      income level lag quarterly revenue      market potential
      0.7674609           0.1238646           1.3305577
```

```

      price index
      -0.7542401
attr(,"indexes")
[1] "j"
> current(sys2,v) # stack.x, stack.loss から生成した問題の結果
Acid.Conc.   Air.Flow Water.Temp
-0.1521225  0.7156402  1.2952861
attr(,"indexes")
[1] "j"

```

2.8 制約式 Constraint の追加

制約式を追加することで、変数の取りうる範囲を設定することができます。たとえば、簡単な制約として、次の式で表されるような「すべてのフィッティングの係数の和が 0 である」ということを要求してみましょう。

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

そのためには手続き Nlsfit.gen の末尾に

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

と加えたものを内容とする手続き Nlsfit.eq を生成し、解くことになります。

一般に制約式は次の形を取ります。

```

式 1 @ 式 2 # 二項
      # @ は { ==, ==, <= } のいずれか

```

なお、< , > , != は制約式の定義に使うことはできませんのでご注意ください。

ここで、freeny.x, freeny.y をデータとして Nlsfit.eq を実際に解いてみます。

```

sys3 <- System(Nlsfit.eq,freeny.x,freeny.y)
sol3 <- solve(sys3,trace=F) # 出力を抑制して解く
current(sys3,v)

```

とすると次のように結果が出力されます。

```

> current(sys3,v)
      income level lag quarterly revenue      market potential
21

```

```

0.8026180      0.3010901      -0.1054927
price index
-0.9982154
attr(,"indexes")
[1] "j"

```

ここでは、得られた結果が制約を充足しているかどうかを検証してみましょう。

```
sum(as.array(current(sys3,v))) # as.array で array 型に変換して和を取る
```

とすると次のように出力され、制約が充足されていることを確認できます。

```

> sum(as.array(current(sys3,v)))
[1] 1.387779e-17

```

このように、重回帰問題を数理計画問題として表現すれば、求めたいパラメータの値に任意の制約を加えることができます。

2.8.1 制約式の名前付け、除去、復活、追加 (delete.con, restore.con, add.con)

RSIMPLE において、制約式は Constraint というオブジェクトに代入することで、名前をつけることができます。前項のモデル Nlsfit.eq と内容は同じですが、今度は Nlsfit.gen の末尾に

```

c1 <- Constraint()
c1 ~ Sum(v[j],j) == 0 # c1 という名前で制約式を代入

```

としたモデル Nlsfit.eq.named ならば、追加された制約式を“c1”という名前で参照することができます。具体的には、

```

sys3 <- System(Nlsfit.eq.named, freeny.x, freeny.y)
sol3 <- solve(sys3, trace=F) # 出力を抑制して解く
as.array(current(sys3,c1))

```

と問い合わせると、最適解における制約式の値が

```

> as.array(current(sys3,c1))
[1] -4.163336e-17

```

と返されます。

次が Constraint オブジェクトの宣言の形です

```
名前 <- Constraint() # 添字なし
```

```
名前 <- Constraint(index=添字集合/要素) # 添字あり
```

宣言しただけの制約式 (Constraint) オブジェクトは空の状態です、~ (代入演算子) によって具体的な内容を代入されてはじめて意味を持ちます。

名前を付けておくと、モデルを定義しなおすことなく以下のように記述することで名前の付いた制約式 `c1` を消去したり復活させたりすることができますようになります。

```
delete.con(sys3,c1) # 制約式 c1 を消去
```

```
restore.con(sys3,c1) # 制約式 c1 を復活
```

実際に

```
delete.con(sys3,c1)
```

```
sol4 <- solve(sys3,trace=F)
```

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

と記述すると次のように出力され、制約式 `c1` が消去されていることを確認できます。

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

```
[1] 1.467643
```

ここで、

```
restore.con(sys3,c1)
```

```
sol4 <- solve(sys3,trace=F)
```

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

と記述すると次のように出力され、制約式 `c1` が復活していることを確かめられます。

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

```
[1] 9.714451e-17
```

`delete.con`, `restore.con` の呼び出しの形は次のようになります。

```
delete.con(システムオブジェクト, 制約式の名前) # 制約式の除去
```

```
restore.con(システムオブジェクト, 制約式の名前) # 制約式の復活
```

上記に加えて、制約式を追加する関数 `add.con` もあります。

2.9 整数変数 IntegerVariable

整数値しか取りえない変数（整数変数）は数理計画問題のモデル化において重要な要素の一つです。ここでは、重回帰の問題を変形して、最もよくあてはまる説明変数を K 個選ぶようにしてみます。

まず、定式化は次の通りです。

- 目的関数:
 - 残差の二乗和 $\sum_{i \in S} e_i^2 \rightarrow$ 最小化
 - $e_i \equiv (v_0 + \sum_{j \in P} X_{i,j} v_j) - Y_i, (i \in S : \text{観測点})$
- 変数:
 - v_0 (定数項)
 - $v_j (j \in P : \text{線形関数の一次の係数の集合})$
 - $\delta_j \in \{0, 1\} (j \in P : \text{説明変数 } j \text{ を採用するかどうか})$
- 定数:
 - $X_{i,j}$ (観測点 $i \in S$ における説明変数 $j \in P$ の値)
 - Y_j (観測点 $i \in S$ における被説明変数の値)
- 制約:
 - $-M \cdot \delta_j \leq v_j \leq M \cdot \delta_j (j \in P)$
 - * M : 十分大きな数を設定
 - * $\delta_j = 0$ ならば $v_j = 0$ になる
 - * $\delta_j = 0$ ならば実質制約なしにする
 - $\sum_{j \in P} \delta_j \geq K$ (変数は K 個選ぶ)

これをモデルとして記述する際には、元のモデル記述 Nlsfit に、0-1 整数変数と制約を二つ追加すればよいことになります。これより、モデル記述は次のようになります。なお、データと選択数 K は引数として与え、十分大きな数 M については 10 としています。

0-1 整数変数を使った説明変数選択モデル

```
Nlsfit.int <- function(x,y,K) {
  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(e[i]*e[i],i)
# 以下, 追加した制約
delta <- IntegerVariable(index=j,type=binary)
-10*delta[j] <= v[j]
v[j] <= 10*delta[j]
Sum(delta[j],j) <= K
}

```

整数変数 (IntegerVariable) の宣言の形は次の通りです.

```

名前 <- IntegerVariable() # 添字なし
名前 <- IntegerVariable(index=添字集合/要素) # 添字付き
名前 <- IntegerVariable(type=binary) # 添字なし, 0-1 整数変数
名前 <- IntegerVariable(index=添字集合/要素,type=binary) # 添字付き, 0-1 整数変数

```

ここで, $K = 2$ として Nlsfit.int を解いてみましょう.

```

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

```

と入力すると, 以下のような結果が得られます. 結果を見ると, income level と market potential が選択されていることが分かります.

```

> current(sys,v)
      income level lag quarterly revenue      market potential
      0.6072738          0.0000000          3.7690423
      price index
      0.0000000
attr(,"indexes")
[1] "j"

```

3 数理計画問題の記述（応用編）

ここでは、数理計画問題を記述する上でのより応用的な技法について述べます。

3.1 条件式と資金調達問題

本節では次のような資金調達問題を考えます。

資金調達問題: 現時点で現金で 1000 万円を持っているとします。市場には次のようなキャッシュフローをもたらす 3 種類の債券 A,B,C があり、これらの債券はいつでも購入できるものとします。（下記データ Cashflow.bf および Cashflow.flow は RSIMPLE にサンプルとして含まれています。）

```
> Cashflow.bf
      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
```

額面はいずれの債券も 100 円ですが、満期とクーポンが債券によって異なり、A は 3 年、B は 4 年、C は 7 年です。

また、将来 10 年にわたって次のようなキャッシュフローが予測されています。

```
> Cashflow.flow
[1]    0    0  400 -200    0    0  400 -1000    0  600
```

このとき、手持ちの現金を常に 200 万円以上にしておき、10 年目の手持ちの現金を最大化する債券の購入計画を立ててください。

この問題は次のような線形な数理計画問題として定式化できます。

- 目的関数:
 - 最終時点 TL における現金 $y_{TL} \rightarrow$ 最大化

- 変数:
 - $x_{b,t}$ (債券 $b \in B$ の時点 $t \in T$ における購入量)
 - y_t (時点 t における現金)
- 制約:
 - $y_0 = P$ (初期時点の現金)
 - $y_t = y_{t-1} + \sum_{b \in B} \sum_{0 \leq k \leq L_b, t-k \geq 1} BF_{k,b} x_{b,t-k} + C_t, t \geq 1$ (現金の時間発展)
 - $\sum_{b \in B} \sum_{0 \leq k \leq L_b, t-k \geq 1} BF_{k,b} x_{b,t-k}$ (購入した債券が時点 t にもたらすキャッシュフロー)
 - C_t (予測されるキャッシュフロー)
 - L_b (債券 b の満期)
 - $x_{b,t} \geq 0$ (投資額は非負)
 - $x_{b,0} = 0$ (初期時点では投資できない)
 - $x_{b,t} = 0, t + L_b + 1 > TL$ (満期が最終時点以降になるような債券は買わない)
 - $y_t \geq C_t$ (手持ちの現金の最低必要額)

このタイプの時系列を扱う問題ではよくあることですが、「現金の時間発展」「満期が最終時点以降になるような債券は買わない」という制約式では、式が定義される範囲を条件づけています。また、「購入した債券が時点 t にもたらすキャッシュフロー」の表現では和を取る範囲を条件づけています。このような場合のモデル記述では、添字に関する条件式を定義します。以下がこのモデルの記述です。

```
Cashflow <- function(flow, bf, P, C1) {
  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 # 各債券の満期の定義
  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] >= C1
f <- Objective(type=maximize)
f ~ y[TL]
}

```

制約の添字を記述する場所に要素に関する条件式を入れると、制約を定義する範囲が制限されます。条件式は式のうちのいずれの部分の添字に含めてもかまいません。

```

y[t, t>=1] == y[t - 1] + ...

```

という記述では $t \geq 1$ という部分が条件式であり、この式全体について、定義する範囲が $t \geq 1$ となる部分であることを示しています。

```

x[b, t, t + len[b] + 1 > TL] == 0 # 満期が最終時点以降になるような債券は買わない

```

という記述では、 $t + \text{len}[b] + 1 > \text{TL}$ という部分が条件式です。条件式はこのように添字を含む定数の表現である場合にも機能します。

また、制約式の定義には $>$, $<$, $=$ のみしか利用できませんでしたが、条件式の定義に関しては以下のものを利用することも出来ます。

- $!=$ (等しくない)
- $<$ (より大きい)
- $>$ (より小さい)

条件式が Sum 関数に現れると和を取る範囲を制限する働きをします。具体的には、次の二箇所にあらわれています。

```

len[b] ~ Sum(Parameter(1), k, BF[k, b] != 0) - 1 # 各債券の満期の定義
Sum(x[b, t - k] * BF[k, b], b, k, t - k >= 1, k <= len[b]) # 過去に購入した各債券が t 時点にもたらすキ
↪ ャッシュフロー

```

条件式を含む場合について、意図通りに式が定義されているかを知るには、表示の量は多くなりますが、次のように System で展開した後 System オブジェクトを表示してみることをお勧めします。

```

> sys <- System(Cashflow, Cashflow.flow, Cashflow.bf, 1000, 200)
> sys
:-1:info: (1-1) y[0] == 1000
:-1:info: (2-1) -y[1]+y[0]-100*x[A,1]-100*x[B,1]-100*x[C,1] == 0
...(中略)...
:-1:info: (6-9) y[8] >= 200

```

```

:-1:info: (6-10) y[9] >= 200
:-1:info: (6-11) y[10] >= 200
:-1:info: (objective) name="f" y[10] (maximize)

```

次のようにすると投資計画と現金の推移をみることができます。

```

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

```

実際に実行してみると、次のように結果が出力されます。

```

> round(as.array(current(sys, x)), digits=2) # 投資計画
      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) # 現金の推移
      0      1      2      3      4      5      6      7      8      9
1000.00 200.00 200.00 200.00 200.00 200.00 200.00 636.95 200.00 1055.28
      10
1655.28
attr(,"indexes")
[1] "t"

```

3.2 SymmetricMatrix と最小固有値問題

RSIMPLE は要素が一般の式からなる行列の半正定値性を制約として与えることができます。簡単な例として次のような問題を考えましょう。

-
- 目的関数: $\lambda \rightarrow$ 最大化
 - 変数: λ
 - 制約: $A - \lambda I$ が半正定値 (A : 任意の対称行列)
-

この問題を定義するには、SymmetricMatrix というオブジェクトを使って次のように記述します。

```

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)) # 対称行列の宣言
  # 対称行列の要素の設定（下三角部分のみ）
  M[i, j, i > j] ~ A[i, j]
  M[i, i] ~ A[i, i] - lambda
  M >= 0 # 正定値性の制約
  f <- Objective(type=maximize)
  f ~ lambda
}

```

SymmetricMatrix は対称行列に対応するオブジェクトで、実体は行列の形に並んだ Expression の集合体です。SymmetricMatrix の宣言の形式は次の通りです。

```

名前 <- SymmetricMatrix(dprod(集合, 集合)) # 同じ集合を与える
名前 <- SymmetricMatrix(dprod(要素 1, 要素 2)) # 要素 1 と要素 2 は同じ集合の要素

```

本節の例では M というオブジェクトが SymmetricMatrix として定義されています。宣言では同一の集合 S の要素 i と j が与えられています。この dprod(i,j) という部分については dprod(S,S) と書くこともできます。続いて、~ 演算子を用いて対称行列の要素の設定を行っていますが、設定方法は Expression と同様です。対称行列であることは定義から保証されていますので、下三角あるいは上三角部分の要素いずれか一方のみを与えています。

それでは、例を解いてみます。サンプル行列として R 付属のデータセット stack.x の分散共分散行列を使います。

```

mat <- var(stack.x)
sys <- System(MinLambda, mat)
sol <- solve(sys, trace=F)
current(sys, lambda)

```

次のように出力されます。

```

> current(sys, lambda)
[1] 3.60468

```

この値は R を使って求めた固有値の最小値と正確に一致しています。

```
> eigen(mat)$values
[1] 99.57609 19.58114 3.60468
```

3.3 wcsp と集合分割問題

3.3.1 1 指標の 2 分割問題

R にビルトインされているデータ `state.x77` を用い、アメリカの州の面積をちょうど半分にするように部分集合を選ぶという問題を考えます。この面積を半分にするような部分集合を選ぶという問題は、次のような離散的な数理計画問題として表現できます。

-
- 変数: $\delta_s \in \{0, 1\} (s \in S)$ (部分集合に入るか否か)
 - 制約: $\sum_{s \in S} a_s \cdot \delta_s = \sum_{s \in S} a_s \cdot (1 - \delta_s)$ (a_s は州 s の面積)
-

この問題には目的関数がありませんので、厳密には制約充足問題ということになります。この問題を `RSIMPLE` で解くために、次のモデル `Half` を用意します。

```
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)
}
```

この問題は次のようにして解くことができます。

```
sys <- System(Half) # 展開
sol <- solve(sys) # 解く
```

実行してみるとほぼ瞬時に答えが返されます。次に、得られた答えを検証してみます。選ばれたものの面積の和と選ばれなかったものの面積の和をそれぞれ以下のようにして求めます。

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


すると、次のような結果が得られ RSIMPLE も解を求めることができたとわかります。

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

3.3.2 wcsp による 3 指標の 2 分割問題

面積のみならず、別の指標でもおおむね半分となるように二分割する方法を探してみることになります。元にしたデータは state.x77（下記参照）とし、ここに収められている 3 つの統計値（Area, Population, Illiteracy）を指標としてすべて半分になるように分割してみます。

```
> head(state.x77, 10)
```

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area
Alabama	3615	3624	2.1	69.05	15.1	41.3	20	50708
Alaska	365	6315	1.5	69.31	11.3	66.7	152	566432
Arizona	2212	4530	1.8	70.55	7.8	58.1	15	113417
Arkansas	2110	3378	1.9	70.66	10.1	39.9	65	51945
California	21198	5114	1.1	71.71	10.3	62.6	20	156361
Colorado	2541	4884	0.7	72.06	6.8	63.9	166	103766
Connecticut	3100	5348	1.1	72.48	3.1	56.0	139	4862
Delaware	579	4809	0.9	70.06	6.2	54.6	103	1982
Florida	8277	4815	1.3	70.66	10.7	52.6	11	54090
Georgia	4931	4091	2.0	68.54	13.9	40.6	60	58073

前項の最適化モデルを若干拡張して次のようにします。

-
- 変数: $\delta_s \in \{0, 1\} (s \in S)$ (部分集合に入るか否か)
 - 制約: $\sum_{s \in S} f_s^i \cdot \delta_s = \sum_{s \in S} f_s^i \cdot (1 - \delta_s)$ (f_s^i は州 s の $i \in F$ に関する指標値 (統計値))
-

モデルについては、Half を拡張し次のモデル Half2.infs を定義します。

```
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)
}

```

それでは、次のように Half2.infs を制限時間 5 秒として実行してみましょう。

```

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

```

すると、次のようなメッセージが表示されます。

```

> sol <- solve(sys)
...(中略)...
error: (SIMPLE 193) Problem in solve()
error: (NUOPT 16) infeasible MIP.

```

このメッセージは、解を得ることが出来なかったことを表しています。解を得られなかった理由としては、3 指標を同時にちょうど半分にするような分割が存在しないことが挙げられます。今回の例では、NUOPT が解が存在するかどうかの判定自体に時間を要しているため、このようなメッセージが出力されます。

しかし、この結果は我々の望んだものではありません。厳密解が得られなければ、近似解を求めてほしいのが普通です。そのような場合に、解法 wcsp が役立ちます。wcsp は離散最適化問題を近似的に解きます。

次が解法 wcsp を適用するための、3 指標 2 分割問題の記述です。

```

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)
}

```

```
}
```

上記について、いくつか解法 wcsp に特化した記述や工夫が含まれています。

まず, `softConstraint(1)` は `wcsp` に、この制約はソフト制約として扱う（ハード制約と呼ばれる厳密な制約ではなく、最小化する対象として扱う）ように指示する記述です。引数の値 1 はソフト制約の違反量そのものに掛ける重みであり、複数のソフト制約があった場合には、重みが大きく設定されたソフト制約が優先されることになります。

また、最後の制約式の両辺を 1000 倍していますが、これは `wcsp` が制約の違反量の計算を常に整数値で行うためです。他の指標とオーダーをそろえて整数値にするために、1000 倍しています。

それでは次のようにして `Half2` を実際に解いてみることにします。

```
sys2 <- System(Half2)
nuopt.options(method="wcsp")
sol2 <- solve(sys2)
nuopt.options(nuopt.options(NA)) # パラメータをデフォルトに戻す
```

すると、次のような実行経過が出力されます。

```
> sol2 <- solve(sys2)

[Problem and Algorithm]
NUMBER_OF_VARIABLES                50
(#INTEGER/DISCRETE)                50
NUMBER_OF_FUNCTIONS                 3
PROBLEM_TYPE                        MINIMIZATION
METHOD                             WCSP

[Progress]
<preprocess begin>...<preprocess end>
preprocessing time= 0(s)
<iteration begin>
--- TryCount = 1 ---
# random seed = 1
(hard/soft) penalty= 0/986245, time= 0.00(s)
<greedyupdate begin>.....<greedyupdate end>
greedyupdate time= 0(s)
--- End Phase-I iteration ---
(hard/soft) penalty= 0/61443, time= 0.00(s), iteration= 1
```

```

(hard/soft) penalty= 0/60119, time= 0.00(s), iteration= 2
(hard/soft) penalty= 0/49267, time= 0.00(s), iteration= 901
(hard/soft) penalty= 0/33743, time= 0.00(s), iteration= 902
(hard/soft) penalty= 0/31565, time= 0.00(s), iteration= 903
(hard/soft) penalty= 0/20577, time= 0.01(s), iteration= 2249
(hard/soft) penalty= 0/17259, time= 0.01(s), iteration= 2250
(hard/soft) penalty= 0/13587, time= 0.01(s), iteration= 2251
(hard/soft) penalty= 0/10699, time= 0.01(s), iteration= 2254
(hard/soft) penalty= 0/9271, time= 0.01(s), iteration= 3289
(hard/soft) penalty= 0/8275, time= 0.01(s), iteration= 3290
(hard/soft) penalty= 0/7065, time= 0.02(s), iteration= 4243
(hard/soft) penalty= 0/4085, time= 0.02(s), iteration= 4255
(hard/soft) penalty= 0/3821, time= 0.03(s), iteration= 6190
(hard/soft) penalty= 0/1055, time= 0.03(s), iteration= 7118
# (hard/soft) penalty= 0/1055
# cpu time = 0.03/5.00(s)
# iteration = 7118/7118
<iteration end>

```

[Result]

STATUS	NORMAL
TERMINATE_REASON	End_by_CPU_time_limit
ITERATION_COUNT	7118
PENALTY	1055
RANDOM_SEED	1
ELAPSED_TIME(sec.)	5.00

実行経過について, penalty=…という部分は探索で良い解が見つかるたびに更新されてゆきます。デフォルトでは 5 秒探索を行って止まります。

このケースでは 1055 というペナルティ値を持つ解が得られています。以下のようにして結果を検証してみると, Area 以外にも, Population, Illiteracy でもほぼ半分になっていることがわかります。

```

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])
```

出力は次のようになります。

```
> sum(state.x77[, "Area"][delta2 == 1])
[1] 1768600
> sum(state.x77[, "Area"][delta2 == 0])
[1] 1768194
> sum(state.x77[, "Population"][delta2 == 1])
[1] 106235
> sum(state.x77[, "Population"][delta2 == 0])
[1] 106086
> sum(state.x77[, "Illiteracy"][delta2 == 1])
[1] 29
> sum(state.x77[, "Illiteracy"][delta2 == 0])
[1] 29.5
```

解法 wcsp は「解の良さを判定して停止する」というロジックを持ちません。そのため、5 秒という探索時間の上限が設定されています。実行経過の出力によると、今回の例では最良の解が 0.1 秒以下で求められています。

```
(hard/soft) penalty= 0/1055, time= 0.03(s), iteration= 7118
```

この結果を見て制限時間を調整することができます。通常ペナルティは最初急速に減り、減少が緩やかになり、止まるという経過をたどります。もし制限時間が来ても更新が行われているようであれば、次のように長めの実行時間を設定してください。

```
nuopt.options(method="wcsp", maxtim=60) # 60 秒制限
```

解法 wcsp は厳密な解を求めることができませんが、多くの離散計画問題に対しかなりの短時間（1 分以下）で比較的有效な実行可能解を与えます。制限時間を延長しても、結果に満足できない場合には乱数の種の変更（方法は「乱数の種の設定による解の違い」で述べます）を検討してください。

ただ、wcsp は離散計画問題のみしか扱えない、汎用性の点では劣る解法なので、通常の簡単な数理計画問題を解く場合にエラーを発生させて混乱することがないように、wcsp を指定した後にはアルゴリズムの指定は

```
nuopt.options(nuopt.options(NA)) # デフォルトに戻す
```

としてデフォルト（自動選択）に戻すことをお勧めします。

3.3.3 wcsp による多分割問題 (selection とハード, セミハード, ソフト制約)

次は 2 分割ではなく, 面積を指標とした N 分割を行う問題の記述です.

- 変数: $u_{s,k} \in \{0, 1\} (s \in S, k \in K)$ (s をいずれのグループ k に振り分けるか)
- ハード制約: $\sum_{k \in K} u_{s,k} = 1, s \in S$ (振り分けられるのはいずれか一つ)
- ソフト制約: $\sum_{s \in S} a_s \cdot u_{s,k} = \frac{1}{|K|} \sum_{s \in S} a_s$ (a_s は州 s の面積)

この問題のモデル記述は以下の通りです.

```
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() # 次の制約がハード制約であることの定義  
  # Sum(u[s, k], k) == 1 と等価だがこの方が wcsp はより高速に動作する  
  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  
  # 各 s が何番目のグループに分類されたかという指標  
  dst[s] ~ Sum(knum[k] * u[s, k], k)  
}
```

上記モデル記述中の `hardConstraint()` は以降に定義される制約式が最も優先される制約 (ハード制約) であることを示します. すべて以降に引き続く制約がその種別であることを示します. 形式は次の通りです.

```
softConstraint(1 より大きな整数値) # ソフト制約の定義のはじまり  
semiHardConstraint() # セミハード制約の定義のはじまり  
hardConstraint() # ハード制約の定義のはじまり
```

これらの構文は何度でもコールすることができ、常に直前のコールが有効です。

selection() 構文は、0-1 変数の和が 1 であるという定式化の場合に使うことができ、解法 wcsp の動作を高速化する働きがあります。

では、wcsp を用いて 5 分割の場合を次のように解いてみます。

```
sys <- System(Partition, 5)
nuopt.options(method="wcsp", maxtim=1, wcspTryCount=1)
sol <- solve(sys)
```

得られた結果を確認するため、式 (Expression) dst を取り出して、和を計算してみます。

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

すると、次のような結果が得られ比較的均等にわかれていることがわかります。

```
> for(i in 1:5) print(sum(state.x77[, "Area"][dst == i]))
[1] 709978
[1] 707218
[1] 709735
[1] 705445
[1] 704418
```

各グループを表示させると次のようになります。

```
> for(i in 1:5) print(state.x77[, "Area"] [dst == i])
Alaska Colorado Virginia
566432    103766    39780
      Alabama      California      Florida      Hawaii      Kentucky
      50708      156361      54090      6425      39650
Mississippi      New Mexico North Carolina      Oregon      Pennsylvania
      47296      121412      48798      96184      44966
Tennessee
      41328
      Idaho      Illinois      Indiana      Iowa      Kansas
      82677      55748      36097      55941      81787
Michigan      Texas West Virginia      Wisconsin
      56817      262134      24070      54464
Arizona      Connecticut      Georgia      Maine Massachusetts
```

113417	4862	58073	30920	7826
Nebraska	New Hampshire	New Jersey	New York	Ohio
76483	9027	7521	47831	40975
Oklahoma	South Dakota	Washington	Wyoming	
68782	75955	66570	97203	
Arkansas	Delaware	Louisiana	Maryland	Minnesota
51945	1982	44930	9891	79289
Missouri	Montana	Nevada	North Dakota	Rhode Island
68995	145587	109889	69273	1049
South Carolina	Utah	Vermont		
30225	82096	9267		

3.3.4 乱数の種の設定による解の違い

wcsp は乱数を利用して解を発見しますので、乱数のシード値によって結果が異なります。前項の Partition モデルにおいて、以下のように乱数のシード値（デフォルトは 1）を 4 に変更して実行してみます。

```
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.x77[, "Area"][dst == i]))
```

出力結果を見ると、次のようになり先ほどとは別の答えが得られていることがわかります。

```
> for(i in 1:5) print(sum(state.x77[, "Area"][dst == i]))
[1] 709978
[1] 707218
[1] 709735
[1] 705445
[1] 704418
```

RSIMPLE にはシード値を変更して自動的に最も良いものを求める機能があります。wcspTryCount というパラメータを設定すると、wcspRandomSeed で設定した値から、この回数だけシード値を 1 刻みで設定しなおして最も良いものを出力します。たとえば、以下のように設定するとシード値を 1 から 5 までの値に設定し、最も良いものを出力します。

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


3.3.5 アルゴリズム wcsp を使うときの注意点

アルゴリズム wcsp は RSIMPLE に実装されているほかのアルゴリズムとはかなり違う特徴を持っています。利用上の注意点を以下にまとめておきます。

- 0-1 整数変数しか扱うことができません
- 起動したら, `nuopt.options(nuopt.options(NA))` で設定を元にもどしましょう
- 真の最適解が得られる保証はありません
- `nuopt.options(maxtim=…)` で停止時間 (デフォルト: 5 秒) をチューニングすることができます
- 制約のクラスが (ソフト/セミハード/ハード) の 3 つあります
- 与える乱数のシード値 `nuopt.options(wcspRandomSeed=…)` によって違う解を返します
- `nuopt.options(wcspTryCount=…)` で回数を与えればその回数だけ乱数のシード値を振って試行し, 最良の結果を返します
- 制約式の違反量は整数値に丸めて判定されるので制約式の定数倍が必要な場合があります

3.4 大学駅伝エントリー最適化問題

本節では次のような大学駅伝エントリー最適化問題を考えます。

大学駅伝エントリー最適化問題: 6 人が大学駅伝にエントリーしており, 区間は 5 つあります。各走者によって各区間の走るタイムが違ふ時, どの区間でどの走者が走ると最適かを決めます。

上の問題を数理計画問題として定式化すると以下のようになります。

- 目的関数:
 - タイム合計 $\sum_{(i,j) \in IJ} r_{i,j} x_{i,j} \rightarrow \text{最小化}$
- 変数:
 - $x_{i,j} \in \{0, 1\}$ (走者 i が区間 j を走るかどうか)
- 集合:
 - IJ (走者 i と区間 j のありうる組合せ)
- 定数:
 - $r_{i,j}$ (走者 i の区間 j におけるタイム)
- 制約:
 - $\sum_{i, (i,j) \in IJ} x_{i,j} = 1$ (各区間は必ず誰かが走る)
 - $\sum_{j, (i,j) \in IJ} x_{i,j} \leq 1$ (各走者が走れるのは一回まで)

この問題のモデル記述は以下の通りです.

```
Ekiden <- function(running.time){
  I <- Set()
  J <- Set()
  i <- Element(set=I)
  j <- Element(set=J)
  IJ <- Set(superSet=I * J, dim=2)
  ij <- Element(set=IJ)
  r <- Parameter(running.time, index=IJ)
  x <- IntegerVariable(index=ij, type=binary)

  Sum(x[i, j], i, dprod(i, j) < IJ) == 1
  Sum(x[i, j], j, dprod(i, j) < IJ) <= 1

  f <- Objective(type=minimize)
  f ~ Sum(r[ij] * x[ij], ij)
}
```

制約式に記述する条件式については「条件式と資金調達問題」で述べていますが,ここでは「多次元添字」に関する条件式を述べます. 多次元添字を用いた条件式を記述する際は, dprod 関数を用います. 例えば以下のように記述します.

```
Sum(x[i, j], i, dprod(i, j) < IJ) == 1
```

上の例では, 和をとる範囲を添字 (i,j) が集合 IJ の範囲内となるように条件付けしています.

次のようにすると最適な駅伝エントリーを得ることが出来ます.

```
running.time <- as.list(data.frame(
  i = c("D", "E", "F", "C", "D", "E", "F", "C", "D", "E", "A", "C", "A", "S"),
  j = c(1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4, 5, 5),
  r = c(3, 2, 0, 4, 5, 7, 0, 8, 2, 0, 0, 3, 0, 1),
  stringsAsFactors = FALSE
))
sys <- System(Ekiden, running.time)
sol <- solve(sys)
x <- as.array(current(sys, x))
y <- data.frame(which(x == 1, arr.ind=TRUE))
data.frame(i=rownames(x)[y$row], j=colnames(x)[y$col])
```

実際に実行してみると, 次のように結果が出力されます.

```
> data.frame(i=rownames(x)[y$row], j=colnames(x)[y$col])
```

```
  i j
```

```
1 A 4
```

```
2 S 5
```

```
3 F 2
```

```
4 E 3
```

```
5 D 1
```

4 R とのデータ連携

- 本章では, RSIMPLE がロードされているものとして説明を行います. なお, RSIMPLE をロードするには次のようにします.

```
library(Rnuopt)
```

4.1 R オブジェクトから RSIMPLE オブジェクトへの変換

- R オブジェクトは Parameter, Set 等の RSIMPLE オブジェクトに変換することができます
- 本節では主に R オブジェクトから Parameter への変換について述べます.
- Parameter に変換できる R のデータ型は主に以下です.
 - integer
 - numerical
 - vector
 - data.frame
 - array
 - list
- 「自動代入機能」は変換において最も注意すべき機能です. 本文書の最後で説明しています.

4.1.1 integer / numerical

- Parameter の引数に与えることによって, その値をもつ Parameter オブジェクトが返ります.
- 以下実行例です.

```
p <- Parameter(3)
print(p)
[1] 3
```

4.1.2 vector

- vector を Parameter に変換した場合, vector の names 属性が Parameter の添字に対応します.
- vector の names 属性が NULL の場合, 1 始まりの添字が Parameter に設定されます.
- Parameter の引数 index には, 添字に対応する集合を設定します.
- 注意すべきこととして, names における順番と Parameter の添字における順番は必ずしも対応しません.
- 以下実行例です.

実行例:

```

S <- Set() # 添字を入れる Set オブジェクトを準備
Y <- Parameter(index=S, c(2,3,5,7,11,13))
print(Y) # 添字は 1 はじまりの数値 (ただし 1 から出力される保証はない)
1 2 3 4 5 6
2 3 5 7 11 13

```

4.1.3 array / matrix

- array / matrix を Parameter に変換した場合、行名・列名が Parameter の添字に対応します。
- array / matrix の行名・列名が NULL の場合、1 始まりの添字が Parameter に設定されます。
- Parameter の引数 index には、添字に対応する集合を設定します。このとき、array の次元数が 2 以上の場合は dprod 関数を通じて集合を設定します。
- 注意すべきこととして、行名・列名における順番と Parameter の添字における順番は必ずしも対応しません。
- 以下実行例です。

実行例：

```

x <- array(c(1,2,3,4), c(2,2))
S <- Set()
T <- Set()
Y <- Parameter(index=dprod(S,T), x) # 自動代入機能により S には {1 2} が設定される (ただし 1 2 の順番
  ↳ になる保証はない)
print (Y)
1 2
1 1 3
2 2 4

```

4.1.4 list

- list から Parameter に変換した場合、list の最初から length(list)-1 個の要素が Parameter の添字、最後の要素が Parameter の値ベクトルに対応します。
- 注意すべきこととして、list で与えられた要素の順番と Parameter の添字の順番は必ずしも対応しません。

実行例：

```

## 一次元 Parameter の設定
x <- list(c("a", "b", "c"), 1:3)
S <- Set()
p <- Parameter(index=S, x)
print(p["a"])

```

```

a
1
## 二次元 Parameter の設定
x.twodim <- list(c("a","a","b","b"),c(1,2,1,2),7:10)
S1 <- Set()
S2 <- Set()
p <- Parameter(index=dprod(S1,S2), x.twodim)
print(p["a",1])
1
a 7
## 三次元 Parameter の設定
x.threedim<-list(c(1,2,1,2,1,2),c("a","a","b","b","c","c"),c("A","A","A","B","B","B"),1:6)
S1 <- Set()
S2 <- Set()
S3 <- Set()
p <- Parameter(index=dprod(S1,S2,S3),x.threedim)
print(p[1,"a","A"])
, , A

a
1 1

```

4.1.5 data.frame

- data.frame を Parameter に変換した場合、行名・列名が Parameter の添字に対応します。
- Parameter の引数 index には、添字に対応する集合を設定します。このとき、dprod 関数を通じて二つの集合を設定します。dprod は direct product の略です。
- 注意すべきこととして、行名・列名における順番と Parameter の添字における順番は必ずしも対応しません。
- 以下実行例です。

実行例：

```

country <- c("Japan","Germany","USA","France","China")
age <- c(20,21,18,19,23)
score <- c(98,85,72,71,80)
x <- data.frame(COUNTRY=country, AGE=age, SCORE=score)
S <- Set()
T <- Set()
Y <- Parameter(index=dprod(S,T), x) # 自動代入機能により S には {1 2 3 4 5} が設定される

```

```
print (Y)
```

4.1.6 自動代入機能について

- Set の構成要素は、明示的に定義しなくとも、Parameter から自動的に定義されます。
- 複数の Parameter と関連づけられている場合、Set の構成要素は順次加えられています。
- 本機能は RSIMPLE オブジェクトへの変換を行う際に最も注意すべき機能です。

実行例：

```
S <- Set() # 添字を入れる Set オブジェクトを準備
Y <- Parameter(index=S, c(2,3,5,7,11,13))
print (S) # S は {1 2 3 4 5 6}
Z <- Parameter(index=S, c(2,3,5,7,11,13,17,19))
print (S) # S には {7 8} が追加される
```

4.2 RSIMPLE オブジェクトから R オブジェクトへの変換

- 最適化計算を取得する時などに RSIMPLE オブジェクトを R オブジェクトに変換する必要があります
- 本節では主に Variable や Expression などの RSIMPLE オブジェクトから R オブジェクトへの変換について述べます

前項までのようにして定義した Parameter を使って、数理計画モデルを例えば次のように定義することができます。

次の Nlsfit はデータの初期化を含んだモデル記述です。

```
Nlsfit <- function() {
  # データの宣言および初期化
  ydata <- as.vector(freeny.y) # ydata を vector である ydata に
  S <- Set() # Y の添字の入れ場所として S を定義
  Y <- Parameter(index=S, as.array(ydata)) # 被説明変数 Y を作る
  P <- Set() # X の二つ目の添字の入れ場所として P を定義
  X <- Parameter(index=dprod(S,P), freeny.x) # 説明変数 X を作る
  # 以降数理計画問題の記述
  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)
}

```

このモデルを解くには、次のように式を展開して System オブジェクトを作り、最適化を実行します。

```

sys <- System(Nlsfit) # 式の展開
sol <- solve(sys) # 最適化実行

```

次の Nlsfit.gen は必要なデータを引数で渡すようにしたより汎用的なモデルです。

```

Nlsfit.gen <- function(x,y) { # x は matrix 型, y は vector 型
  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(e[i]*e[i],i)
}

```

上記の Nlsfit.gen から System オブジェクトを作成するときには、データを渡して、データとモデルを結びつける必要があります。たとえば、freeny.x,freeny.y をデータとする場合には次のように記述します。

```

sys <- System(Nlsfit.gen,freeny.x,freeny.y)
sol <- solve(sys) # 最適化実行

```

4.2.1 変数や式 (Variable/Expression) の値を array 型のデータとして取り出す

solve() を起動した後であれば、最適化の結果は System() で作成されたシステムオブジェクトに問い合わせれば得ることが出来ます。システムオブジェクトには、最適化モデルの情報がすべて集約しています。

例えば Nlsfit.gen を解いた結果 v0 および v が幾つになっているか調べる際には、次のように current という関数を用いて、数理計画問題の中の RSIMPLE のオブジェクトを取り出して表示します。


```

sys <- System(Nlsfit.gen, freeny.x, freeny.y)
sol <- solve(sys, trace=F)
current(sys, v0) # sys 中の"v0"を取り出して表示
current(sys, v) # sys 中の"v"を取り出して表示

```

v0 と v は次のように表示されます。

```

> current(sys, v0)
[1] -10.47261
> current(sys, v)
      income level lag quarterly revenue      market potential
      0.7674609           0.1238646           1.3305577
      price index
      -0.7542401
attr(,"indexes")
[1] "j"

```

しかし、current の戻り値は RSIMPLE のオブジェクト（この例では変数オブジェクト Variable）の参照なので、この後 R 側で様々な操作を行うためにはそれぞれ as.array() を使って次のように R のオブジェクトに変換する必要があります。

```

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

```

current を使って取り出した RSIMPLE のオブジェクトに対して、直接 as.vector() は使えません（意味のない結果 0 が返ります）。vector がほしい場合でも、一旦は as.array() を用いて array 型に変換する必要があります。

次の R コードはこうして得られた変数をまとめて一つのベクトル型の変数 vvec として表現します。array 型の変数である v に対して as.vector() とすると、添字の情報が落ちてしまいます。このため、添字を rownames で取り出し、names 属性としています。

```

v0 <- as.array(current(sys, v0))
v <- as.array(current(sys, v))
vvec <- as.vector(v) # vector 型に変換
names(vvec) <- rownames(v) # 落ちてしまった添字の情報を付与
vvec <- c(Intercept=v0, vvec) # v0 をつなげる
vvec

```

vvec は次のように表示されます。

```
> vvec
```

Intercept	income level lag	quarterly revenue
-10.4726071	0.7674609	0.1238646
market potential	price index	
1.3305577	-0.7542401	

各点の誤差を示す式オブジェクト (Expression) である `e` の内容を取り出す際も、変数の値を取り出す場合と同様次のように `sys` に対して `current` を適用します。

```
err <- as.array(current(sys, e))  
barplot(err, names=rownames(err))
```

この結果, `barplot` コマンドにより次の棒グラフが描かれます。

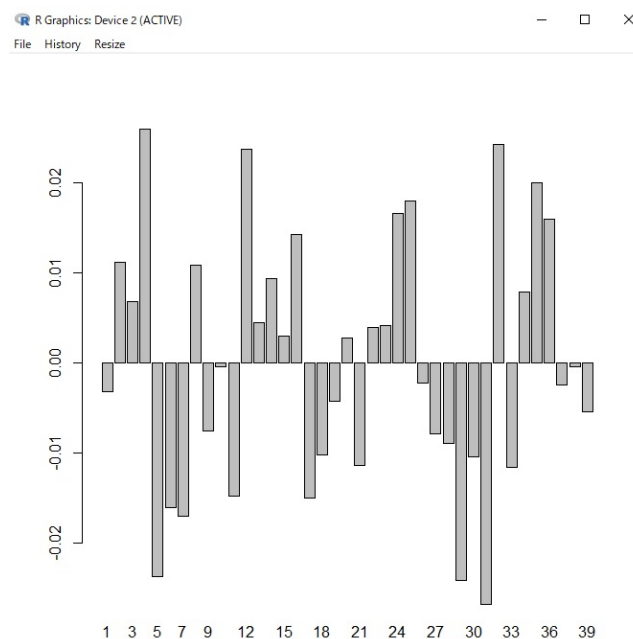


図3 RSIMPLE のオブジェクトの可視化

4.2.2 変数や式 (Variable/Expression) の値を list 型のデータとして取り出す

RSIMPLE のオブジェクトは, すべて `as.list` を用いて `list` 型に変換して処理することができます。変換の規則は入力の場合と同じで, `list` の最初から `length(list)-1` 個の要素が添字のベクトル, 最後の要素が値のベクトルとみなします。

ここでは, 次のようにして `Nlsfit.gen` 内の式オブジェクト `e` を `list` 型のデータに変換し, 表示してみます。

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

```
err <- as.list(current(sys, e))
err
```

出力結果は次のようになります。

```
> err
$indexes
$indexes$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 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39

$values
 [1] -0.0031898351  0.0111499004  0.0068457709  0.0259425872 -0.0236273474
 [6] -0.0160653570 -0.0169512730  0.0108668368 -0.0075620522 -0.0003823554
[11] -0.0147085007  0.0237528558  0.0045025340  0.0093397035  0.0029599861
[16]  0.0142621132 -0.0149334480 -0.0102146487 -0.0042357547  0.0028433162
[21] -0.0113535385  0.0039589736  0.0041339093  0.0165713011  0.0180124058
[26] -0.0022507615 -0.0078694913 -0.0089017162 -0.0241231690 -0.0104325799
[31] -0.0267124037  0.0242586200 -0.0115644134  0.0079191653  0.0200000710
[36]  0.0160024868 -0.0024336529 -0.0003717022 -0.0054385434
```

二次元のオブジェクトの場合には、添字の部分が増えます。

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

上のコマンドを実行すると、Xlist の内容は次のように表示されます。

```
> Xlist
$indexes
$indexes$`*`
 [1] "1"  "1"  "1"  "1"  "2"  "2"  "2"  "2"  "3"  "3"  "3"  "3"  "4"  "4"  "4"
[16] "4"  "5"  "5"  "5"  "5"  "6"  "6"  "6"  "6"  "7"  "7"  "7"  "7"  "8"  "8"
...(中略)...
[136] "34" "35" "35" "35" "35" "36" "36" "36" "36" "37" "37" "37" "37" "38" "38"
```

```
[151] "38" "38" "39" "39" "39" "39"
```

```
$indexes$`*`
```

```
[1] "income level"          "lag quarterly revenue" "market potential"
```

```
[4] "price index"           "income level"         "lag quarterly revenue"
```

```
...(中略)...
```

```
[151] "market potential"      "price index"          "income level"
```

```
[154] "lag quarterly revenue" "market potential"     "price index"
```

```
$values
```

```
[1] 5.82110 8.79636 12.96990 4.70997 5.82558 8.79236 12.97330 4.70217
```

```
[9] 5.83112 8.79137 12.97740 4.68944 5.84046 8.81486 12.98060 4.68558
```

```
...(中略)...
```

```
[145] 6.18768 9.71774 13.15790 4.29627 6.19377 9.74924 13.16250 4.27839
```

```
[153] 6.20030 9.77536 13.16640 4.27789
```

5 ポートフォリオ最適化

5.1 基本的なマルコビッツモデル

4 銘柄の時系列データを用いて、マルコビッツモデルによる最適ポートフォリオを求めます。まず、マルコビッツモデルは次のように記述することが出来ます。

-
- 目的関数:
 - 収益率の分散 $\sum_{i,j \in Asset} Q_{ij} x_i x_j \rightarrow \text{最小化}$
 - 変数:
 - $x_j, j \in Asset$ (銘柄 j の組み入れ比率)
 - 定数:
 - \bar{r}_j (収益率の平均)
 - Q_{ij} (収益率の分散)
 - 制約:
 - $\sum_j x_j = 1$
 - $\sum_j \bar{r}_j x_j \geq r_{\min}$ (最低期待収益率 r_{\min} の確保)
 - $x_j \geq 0$ (空売りの禁止)
-

ここでは r_{\min} を 1% とします。また、本節では 4 銘柄の 60 期間分の収益率データ $R.60 \times 4$ を利用します。 $R.60 \times 4$ は `RSIMPLE` にサンプルとして含まれているため、`RSIMPLE` をロードしていない場合、まずは次のようにロードしましょう。

```
library(Rnupt)
```

$R.60 \times 4$ について、最初に R でグラフ化し確認してみます。

```
# 60x4 の時系列の収益率推移 R.60x4 をグラフ化
matplot(rownames(R.60x4), R.60x4, pch=colnames(R.60x4), type="o")
```

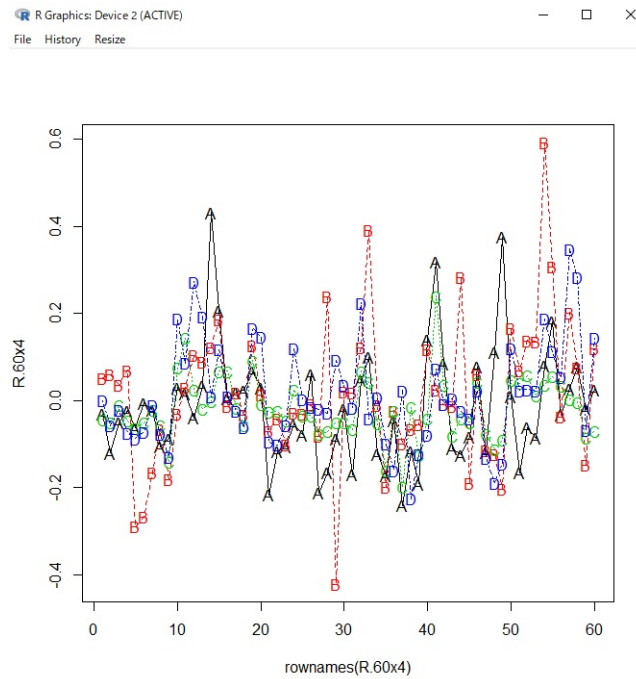


図4 時系列の収益率推移

次に R の機能を使って最適化の材料となる分散と平均を求めます。

```
> Q <- var(R.60x4)
> Q
```

	A	B	C	D
A	0.01778838	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

```
> rbar <- apply(R.60x4, 2, mean)
> rbar
```

	A	B	C	D
	-0.01483683	0.01473420	-0.01805572	0.01391076

NUOPT が認識するためには、R の vector 型オブジェクトは array 型として定義しておく必要があります。

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

最適化モデルは R の手続きとして次のように定義しておきます。

```
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
}

```

モデルにデータを代入するには次のように System コマンドを使います。この命令で最適化問題が定義されます。

```
> s <- System(model=Marko, Q, rbar, 0.01)
```

Evaluating Marko(Q,rbar,0.01) ... ok!

[Expand Constraints and Objectives]

```

objective (1/4) name="risk"
constraint (2/4) name=""
constraint (3/4) name=""
constraint (4/4) name=""

```

展開した結果は s というオブジェクトになりますので、次のように内容を表示することができます。

```

> s
:-1:info: (1-1) -0.0148368*x[A]+0.0147342*x[B]-0.0180557*x[C]+0.0139108*x[D] >= 0.01
:-1:info: (2-1) x[A]+x[B]+x[C]+x[D] == 1
:-1:info: (3-1) x[A] >= 0
:-1:info: (3-2) x[B] >= 0
:-1:info: (3-3) x[C] >= 0
:-1:info: (3-4) x[D] >= 0
:-1:info: (objective) name="risk" 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]+0.00515315*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)

```

では解いてみましょう。最適化問題を解くためには次のように solve コマンドを用います。

```

> sol <- solve(s)

[Problem and Algorithm]
NUMBER_OF_VARIABLES          4
NUMBER_OF_FUNCTIONS          3
PROBLEM_TYPE                  MINIMIZATION
METHOD                        TRUST_REGION_IPM

[Progress]
<preprocess begin>.....<preprocess end>
<iteration begin>
  res=1.1e+01 .... 6.7e-02 .... 5.4e-03 .... 1.3e-03 .... 6.6e-06 ..
      1.1e-09
<iteration end>

[Result]
STATUS                        OPTIMAL
VALUE_OF_OBJECTIVE            0.01087261417
ITERATION_COUNT               23
FUNC_EVAL_COUNT               27
FACTORIZATION_COUNT           34
RESIDUAL                      1.142691332e-09
ELAPSED_TIME(sec.)            0.03

```

次に、最適化問題を解いた結果得られた解ベクトル（組み入れ比率）を取り出します。このためには、次のようにオブジェクト `s` に対して、`current`（現在の値の取得）という手続きが必要です。

```

> xopt <- as.array(current(s, x))
> xopt
      A      B      C      D
0.07466605 0.19864172 0.06030883 0.66638339
attr(,"indexes")
[1] "*"

```

ここで、取り出した解ベクトルを棒グラフにしてみます。

```

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

```

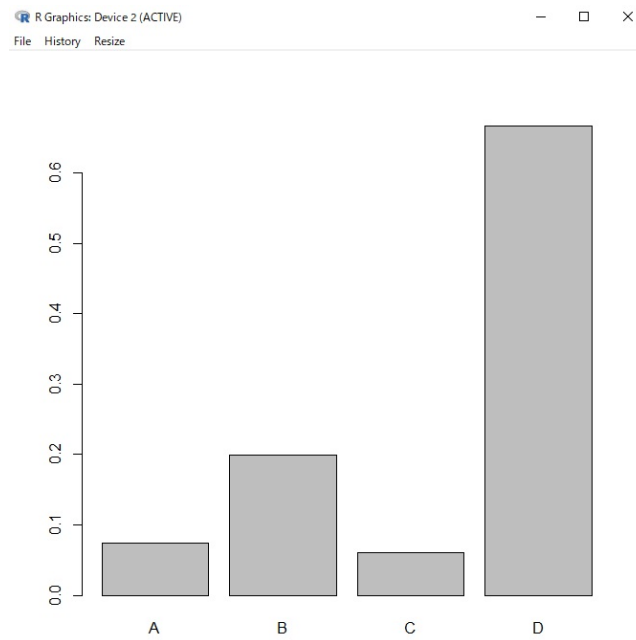



図5 解ベクトルの棒グラフ

今度はポートフォリオの組み入れ比率に従った場合の収益率の時系列 (pf) を出してみます。

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

pf の分散を表示させて見ると次のようになり、最適化問題を解いた際に表示されている目的関数値 (VALUE_OF_OBJECTIVE) とほぼ一致しています。

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

最後に、元の収益率グラフにこのポートフォリオの収益率の変動を重ねてみます。

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

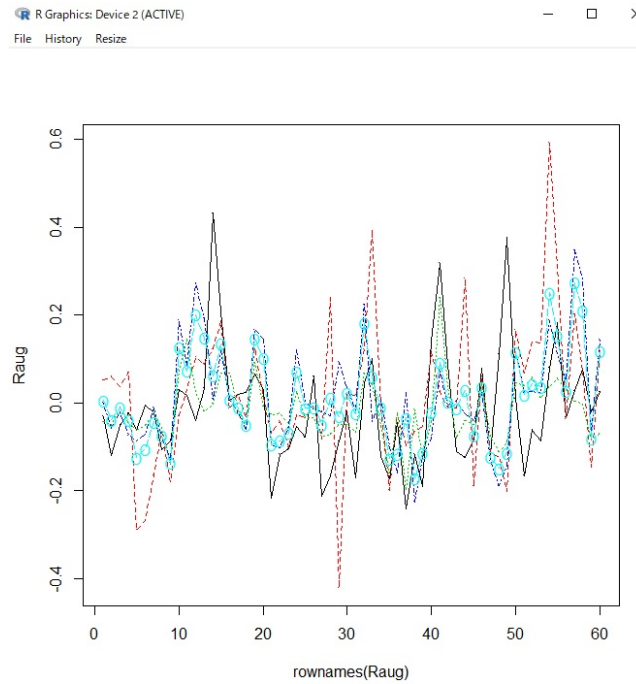


図6 ポートフォリオの収益率推移

“O” のプロットがあるのがポートフォリオで、変動が抑えられていることがわかります。

5.2 さまざまなリスク尺度によるポートフォリオ最適化

ポートフォリオ最適化におけるリスク尺度（変動の測り方）には分散以外にも様々なものがあります。ここでは、分散以外のリスク尺度として、絶対偏差、1 次の下方部分積率（LPM）、Conditional Value at Risk（CVaR）を用いた場合のポートフォリオ最適化問題を扱います。これらのリスク尺度を用いたモデルはすべて線形計画問題として記述できます。

ここで元にするデータは 8000 ケースの 5 銘柄の収益率を含む R:8000x5 です。

5.2.1 分散をリスクとした場合

まずは比較のため、分散最小化モデル（コンパクト分解表現）を解いてみましょう。RSIMPLE ではモデルを次のように定義します。

分散最小化モデル

```
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]
}

```

このモデルについて, 次のようにして最適化を行います.

```

# 展開 (データの代入)
sys <- System(MinVar, R.8000x5)

# 最適化実行
sol <- solve(sys)

# 解の取得
x <- as.array(current(sys, x))

```

さて, 結果の解析に入りましょう. まず最適ポートフォリオ x を見てみます.

```

# 図示
pie(1000 * x, labels=dimnames(x)[[1]])

```

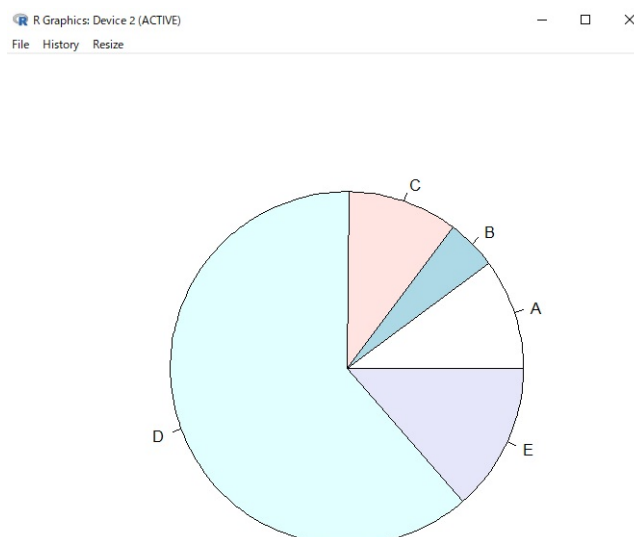


図7 分散最小化最適ポートフォリオ

次に、ポートフォリオに従った場合の収益率の分布のヒストグラムを書いてみます。

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

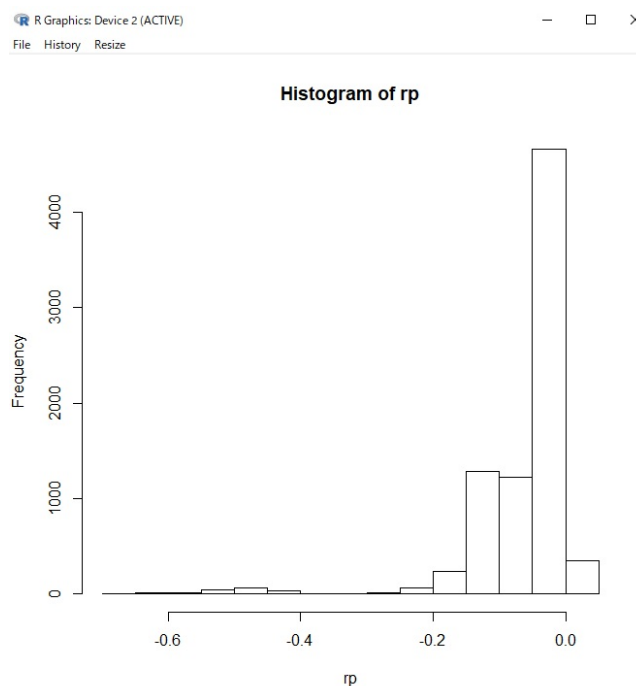


図8 分散最小化収益率ヒストグラム

ヒストグラムを見ると分かるように、8000 ケースのうちわずかですが収益率が非常に悪いケース（収益率 -0.6 ～ -0.4）が存在します。これは対象としているアセットが社債でデフォルトするリスクが含まれているような場合に起きます。このようなアセットに対して分散最小化モデルを適用するのは危険であるといえるでしょう。

5.2.2 絶対偏差をリスクとした場合

次に、リスク尺度として絶対偏差を用いた絶対偏差最小化モデルを解いてみましょう。次のモデルは絶対値関数 `abs()` を用いずに、線形計画問題として定式化するテクニックを利用しています。

```
#### 絶対偏差最小化モデル ####
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]
}

```

次は最適化の実行手順です.

```

# 展開
sys <- System(MinMad, R.8000x5)
# 実行
sol <- solve(sys)
# 解の取得
x <- as.array(current(sys, x))

```

得られた解を元にヒストグラムを表示してみましょう.

```

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

```

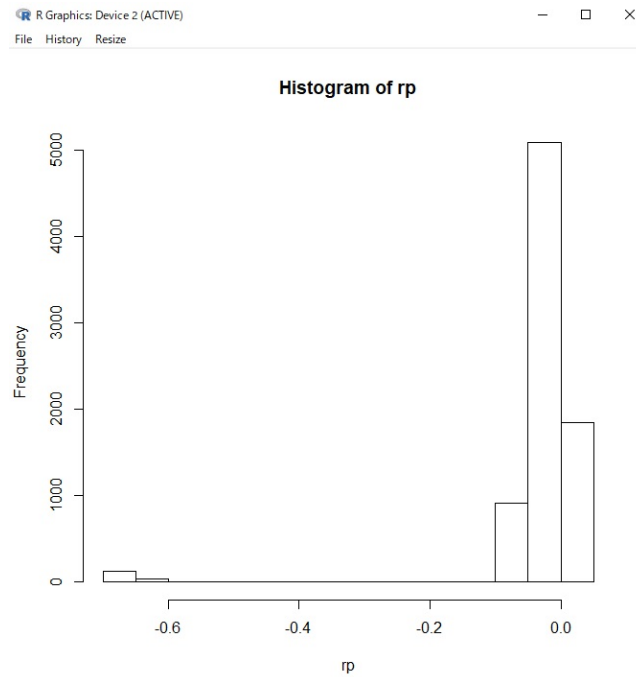


図9 絶対偏差最小化ヒストグラム

分散最小化モデルと比べて、線形計画問題として定式化できるという利点がありますが、大崩に対する感度が小さく、収益率 -0.6 以下のケースが見受けられます。

5.2.3 1 次の下方部分積率（LPM）をリスクとした場合

下方リスクモデルの一つである 1 次の下方部分積率（LPM）最小化モデルの結果を見てみましょう。このモデルは目標収益率 rG 以上のサンプル点については関知しないかわりに、目標収益率 rG を下回る部分の平均を最小化しようとしています。

1 次の下方部分積率（LPM）最小化モデル

```
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]
}

```

rG は下回ってほしくない（リスクとみなせる）収益率を与えます。ここでは -0.4 としましょう。今までと同様次のようにしてヒストグラムを表示してみます。

```

# 展開
sys <- System(MinLPM1, R.8000x5, -0.4)
# 実行
sol <- solve(sys)
# 解の取得
x <- as.array(current(sys, x))
rp <- as.matrix(R.8000x5) %*% as.vector(x)
hist(rp)

```

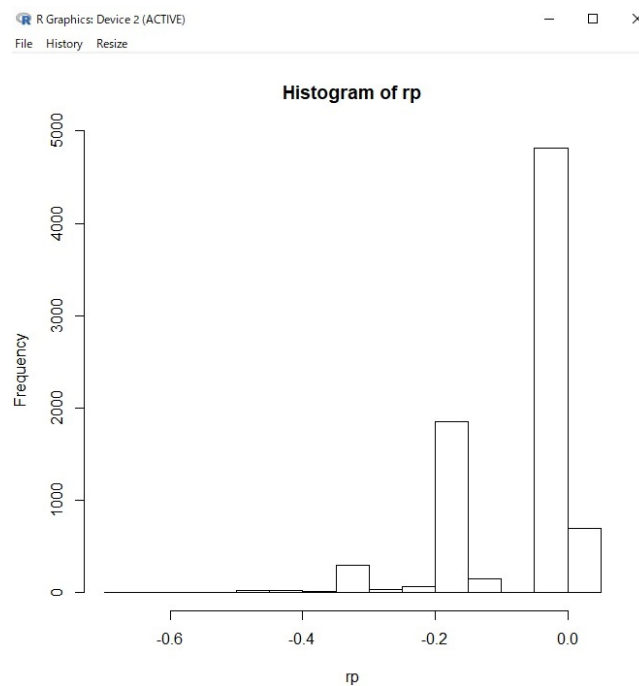


図10 LPM 最小化ヒストグラム

目標収益率 -0.4 を下回るケースがほとんどなく、結果的に分散最小化モデルや絶対偏差最小化モデルと比べて大崩れするケースが少なくなっていることがわかります。

5.2.4 CVaR をリスクとした場合

最後に Conditional Value at Risk (CVaR) 最小化モデルです. CVaR は収益率の損失がある確率水準 (パーセント点 β) を上回るときの平均損失のことです. つまり, 収益率の下位 $(1 - \beta) \times |Asset|$ ($|Asset|$ は銘柄数) の平均損失を最小化します.

```
#### Conditional Value at Risk (CVaR) 最小化モデル ####
```

```
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  
}
```

ここでは $\beta = 0.97$, $|Asset| = 8000$ とします. この場合, 収益率の下位 240 個 ($= (1 - 0.97) \times 8000$ 個) のサンプル点の平均損失を最小化します.

```
# 展開  
sys <- System(MinCVaR, R.8000x5, 0.97)  
# 実行  
sol <- solve(sys)  
# 解の取得  
x <- as.array(current(sys, x))  
# 図示  
rp <- as.matrix(R.8000x5) %*% as.vector(x)  
hist(rp)
```


実行すると次のようなヒストグラムが表示されます。

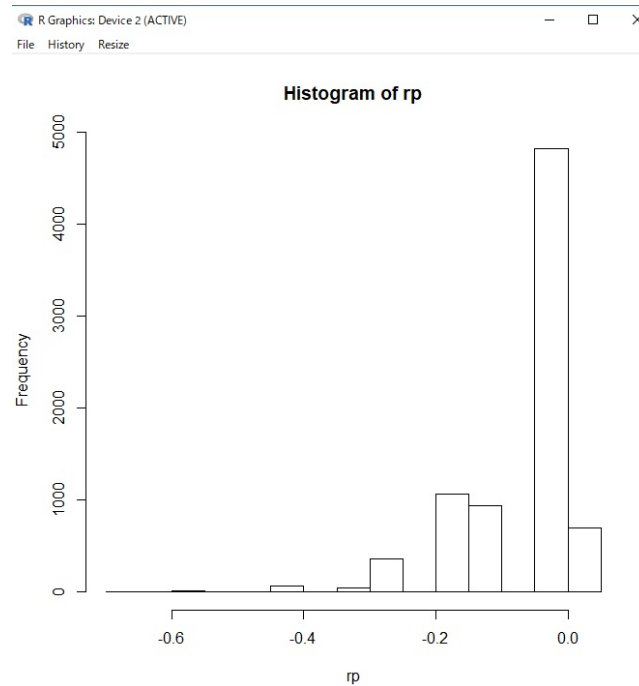


図11 CVaR ヒストグラム

1 次の下方部分積率（LPM）最小化モデル同様、分散最小化モデルや絶対偏差最小化モデルと比べて大崩れするケースが少なくなっていることがわかります。

5.3 コンパクト分解（大規模ポートフォリオ最適化）

基本的なマルコビッツモデルの弱点として、収益率の分散

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

というリスクに関する式に表れる分散・共分散行列を取得しなければならないという点があります。昨今実務家が解かねばならない 1000 ～3000 銘柄のポートフォリオでは、このような分散・共分散行列を陽に生成すると計算負荷が増大してしまいます。

そのような場合には、各銘柄の収益率のヒストリカルデータ、(サンプル点 $t \in Period$ における銘柄 j の収益率の観測値 R_{tj}) を直接データとして入力した場合、

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

と表されることを利用します。具体的には、補助変数 s を定義して次のような制約：

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

を導入し、リスクを次のように書き換えます。

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

このようにすることで、分散・共分散行列を直接生成しなくともよいので、効率的に計算を行うことができます。次が、今述べたテクニックを取り入れたモデル記述となります。

```
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]
}
```

ここでは、1000 銘柄のポートフォリオを求めてみます。データとして月次 5 年分（60 期間）のデータ R.60x1000 が得られているとします。

次のように分散共分散行列を作ることなく R.60x1000 を直接入力データとして最適化を行うことができます。

```
# 展開
sys <- System(MinVar, R.60x1000)
# 実行
sol <- solve(sys)
# 解の取得
x <- as.array(current(sys, x))
```

同様のテクニックは収益率が

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

のようにファクターリターン f_k によって表現されているファクターモデルの場合にも利用することができます。この場合収益率の分散は次のように表現されます。

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

よって、次のような中間変数 s_k を用いた線形制約式：

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

を導入することで、分散は

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

と表現することができます。

コンパクト分解は効率的であるのみならず、下方リスクモデルなど様々なリスク尺度によるポートフォリオモデルを記述する場合の基本となります。

5.4 端株処理

一般にポートフォリオ最適化問題においては、取引額を連続量として求めることが多いのですが、実際には各銘柄について最小取引額の整数倍の額で取引しなければなりません。連続量として求められた取引額を何らかの形で最小取引額の整数倍に補正する作業のことを端株処理と呼びます。

紹介するモデルでは、各銘柄において端株を切り上げるなら 1、切り下げるなら 0 を取るような 0-1 整数変数 d を用いて、取引の総額が与えられた値の $\pm 1\%$ の範囲に収まるという制約の下で各サンプル点における収益額の絶対偏差を最小化しています。

ここでは、前項で得られた解を `RoundLot.x` とし、端株処理してみましょう。ここでは、取引の総額を 50 億円とします。また、以下のような最小取引額データ `RoundLot.unit` が与えられているとします。（`RoundLot.x` および `RoundLot.unit` は `RSIMPLE` にサンプルとして含まれています。）

```
> head(RoundLot.unit)

  A0001   A0002   A0003   A0004   A0005   A0006
10880000 5400000 873000 1875000 7040000 4980000
```

次が切り上げ、切り下げを決定する最適化モデルです。端株処理は比較的規模の大きな 0-1 計画問題ですので、メタヒューリスティクス解法 wcsp を用いて解くことにします。このことに関連して、モデル中でソフト制約を指定するための記述 `softConstraint()` を用いています。

端株処理モデル

```
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)
  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
}
```

ここで実際に解を求めてみます。wcsp を用いる場合には時間制限が必要であるため、次の例では 20 秒を実行時間上限として与えていることに注意してください。

```
# 総額 (50 億円)
total <- 5e9

# 展開
sys <- System(RoundLot, R.60x1000, RoundLot.unit, RoundLot.x, total)

# 実行 (wcsp を指定し実行時間上限 20 秒で解く)
```

```

nuopt.options(method="wcsp")
nuopt.options(maxtim=20)
sol <- solve(sys)
# 設定を元に戻す
nuopt.options(nuopt.options(NA))

```

解が得られたら、次のようにして、解 d とそれを採用した場合の取引額を取得しましょう。

```

# 解の取得
d <- as.array(current(sys, d))
fund <- as.array(current(sys, fund))

```

次に、連続量から計算した理想的な投資量と具体的な投資量を以下のように定義します。

```

n <- sum(fund > 0)
ideal <- (total * RoundLot.x)[fund > 0] # 理想的な投資量
up <- d[fund > 0]
real <- fund[fund > 0] # 具体的な投資量

```

今定義した理想的な取引額 (ideal), 切り上げるか否か (up), 実際の取引額 (real) を表示すると次のようになります。

```

> cbind(ideal, up, real)

```

	ideal	up	real
A0072	2.268363e+07	0	22275000
A0113	1.688806e-01	1	3740000
A0150	4.873353e+07	0	47960000
A0461	1.050252e+07	0	10296000
A0508	9.945731e+07	0	97500000
A0647	1.502035e+07	1	15030000
A0664	1.513867e+07	1	16450000
A0712	9.665077e-01	1	3880000
A0713	1.312491e+07	1	13310000
A0717	1.757698e+07	0	17492400
A0725	3.798210e-02	1	609000
A0746	1.929550e+08	0	188160000
A0751	1.010049e+08	0	99960000
A0870	3.162495e+07	1	31625000
A0875	7.202038e+06	0	4530000
A0884	1.400708e+00	1	626000

```

A0907 2.333426e+07 0 17970000
A0919 2.447820e+07 0 21700000
A0935 7.123815e+06 1 7736000
A0945 4.145884e+07 0 38940000
A0962 1.749276e+07 0 13860000
A0965 1.691399e-01 1 6055000
A0975 1.265959e+09 0 1265850000
A0977 1.487009e+08 1 150960000
A0987 2.505418e+09 0 2505170000
A0997 3.859446e+08 0 385640000

```

最後に, ideal と real についてグラフに描いてみましょう.

```

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

```

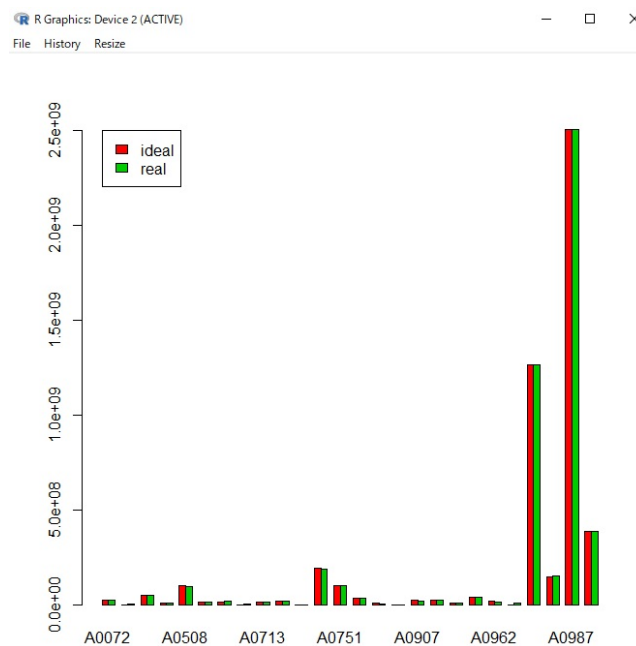


図12 端株処理取引額

5.5 銘柄グルーピング

ポートフォリオ最適化を実行して得られた結果に適当な端株処理を行うことで各銘柄に対する投資額が決まります. こうなるとあとは実際に買い／売り注文を出すだけです. しかし, マーケットインパクト等を考慮すれば, 一度に巨額の注文を出すのはあまり好ましいことではありません. そこで, 銘柄を複数のグループに分けてグループ毎に注文を出すことがあります. このとき, 「各グループの性質がそれほど大きく異ならないようにしたい」というのは自然に発生する要求です. ここでは, 各銘柄にはいくつかのファクター値 (例えば, 投資額) が与えられており, グループご

とのファクター値が均等になるように銘柄をグルーピングすることを目的とします。

例として、50 個の銘柄を 5 個のグループに分けることを考えます。各銘柄は次のように F0 から F9 までの 10 個のファクター値で特徴づけられるとします。(以下のデータ Basket.fcoef, Basket.flow, Basket.fhigh, Basket.fbar, Basket.W は RSIMPLE にサンプルとして含まれています。)

```
> 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
...	(中略)
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

なお、グループ分けについて次の 2 つ要請を受けているものとします。

- それぞれのグループに割り当てられた銘柄での F0 ~F9 の合計が、与えられた範囲に収まること
- 特に重要な F2,F3,F4,F5,F6,F9 の属性については与えられた値にできるだけ近いこと

この問題を解くための数理計画モデルは以下のようになります。

```
#### 銘柄グルーピングモデル ####  
  
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 * .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)
}

```

ここで、この問題に対して以下の 4 種類のデータが与えられているものとします。

```

>> Basket.flow # 各ファクターのグループごとの下限
F0 F1 F2 F3 F4 F5 F6 F7 F8 F9
-15 -15 -15 -15 -15 -15 -15 -15 -15 -15

> Basket.fhigh # 各ファクターのグループごとの上限
F0 F1 F2 F3 F4 F5 F6 F7 F8 F9
15 15 15 15 15 15 15 15 15 15

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

> Basket.W # 特に大事なものを定義する重みベクトル
F0 F1 F2 F3 F4 F5 F6 F7 F8 F9
0 0 1 1 1 1 1 0 0 1

```

それでは、モデルを展開して解を求めてみます。次の記述中ではグループの数は“5”としています。

```

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

# 実行
nuopt.options(method="wcsp", maxtim=10)
sol <- solve(sys)
nuopt.options(nuopt.options(NA))

# 解の取得
gnum <- as.array(current(sys, g))

```

得られた解について、グループごとのファクター値がどのようなになっているか、適当にグルーピングした場合と比較します。fopt は上記のモデルを使ってグルーピングした場合、frand は適当にグルーピングした場合の結果であり、それぞれを出力した結果は以下の通りです。

検証

```
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,]	-13.87	0.20	-0.42	0.01	-0.12	-0.27	-0.60	2.33	13.69	0.08
[2,]	-14.22	-0.84	-0.15	0.13	-0.03	-0.45	-0.12	-8.98	13.86	0.10
[3,]	-14.87	0.17	0.41	0.08	0.42	0.67	0.83	3.19	14.44	-0.66
[4,]	-14.90	-0.88	0.25	0.19	-0.51	0.60	-1.21	7.39	14.45	0.12
[5,]	-15.00	-0.71	0.72	0.12	0.71	0.06	0.87	6.58	14.93	-0.39

> 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

適当にグルーピングした場合と比べて、均質なグループが得られていることが分かります。

5.6 Maximum Drawdown

Maximum Drawdown はポートフォリオ最適化問題におけるリスク尺度の一つであり、ある特定の期間における、資産額の最も大きな減少量のことを指します。特徴として、分散や下方部分積率と異なり、入力データをサンプル点では

なく時系列として扱うという点が挙げられます。ここでは所与の期間において Maximum Drawdown が最も小さくなるような各投資対象に対する投資量を決定します。なお、投資量は初期時点でのみ決定され、その後リバランスは行われないものとします。

まずは、収益率データを次のように初期価格を 1 とした価格データに変換します。

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

以下は Maximum Drawdown 最小化モデルです。変数 x は初期時点における各投資対象に対する投資量です。変数 W は時点 t における資産額、 U は時点 t における W の累積最大値、 V は時点 t における W の時間を逆方向に見た累積最小値です。Maximum Drawdown は U と V の差の最大値として表現することができます。

```
#### Maximum Drawdown 最小化モデル ####
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
}
```

それでは、以下のようにしてシステムを作成して解き、得られた解を取得します。

```

sys <- System(MinMaxDD, P.521x95)
sol <- solve(sys)
x <- as.array(current(sys, x))

```

ここで、確認のために価格データと組入比率から Maximum Drawdown の値を求める関数を定義します。この関数では各期における資産額 W 、累積最大値 U 、時間を逆方向に見た累積最小値 V も求めています。

```

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))
}

```

それでは、今定義した関数を利用して Maximum Drawdown を計算します。

```

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

```

次に、得られた解を棒グラフとして表示します。

```

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

```

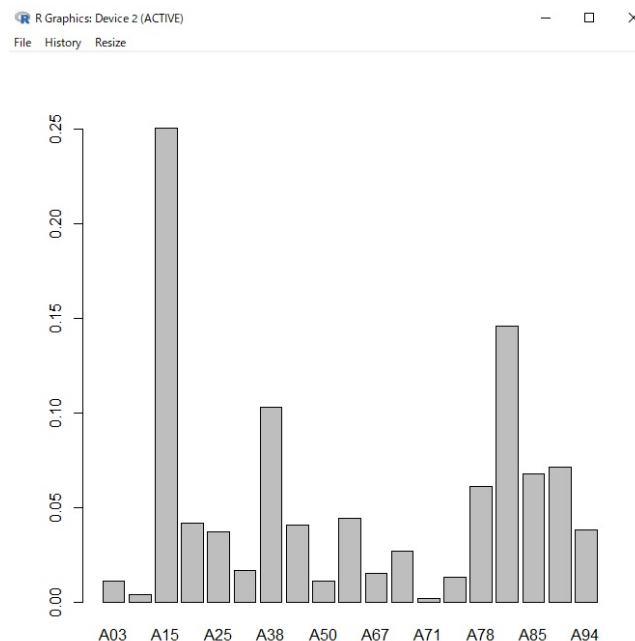


図13 Maximum Drawdown の解

最後に、先ほど計算した W, U, V を図示した結果を示します。

```
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)
```

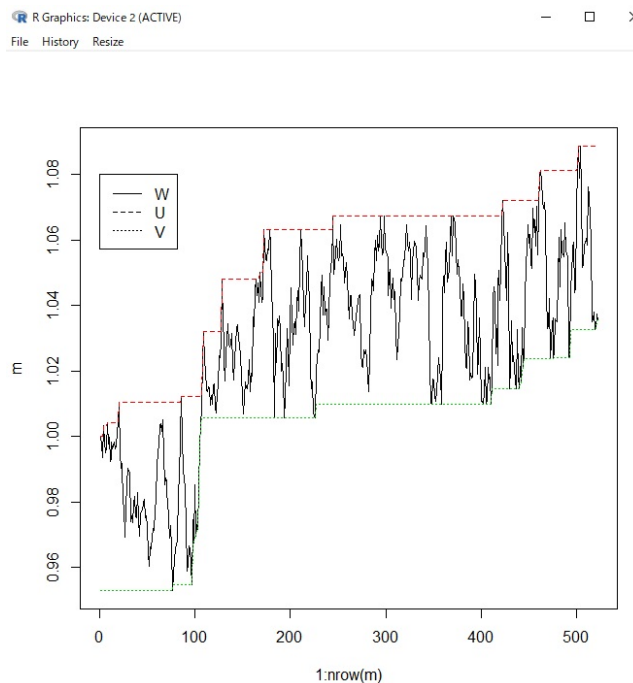


図14 Maximum Drawdown の各変数

5.7 Sharpe Ratio

ポートフォリオ最適化問題の目的関数として Sharpe Ratio と呼ばれる指標を用いることがあります。Sharpe Ratio は次のように表すことができます。

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

ただし、 A は投資対象の集合、 x_j は組入比率、 \bar{r}_j は平均収益率、 σ_{jk} は収益率の分散共分散行列、 r_f は安全資産の収益率を表します。

Sharpe Ratio 最大化問題は（目的関数が非線形となるので）非線形計画問題となりますが、RSIMPLE ではそのまま解くことができます。次がモデル記述となります。

```
#### Sharpe Ratio 最大化モデル ####
```

```
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
}

```

早速解いてみましょう。まずは収益率データから収益率の分散共分散行列と平均収益率を求めます。

```

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

```

次に、システムを作成し、求解を行い、解を得ます。

```

sys <- System(Sharpe, Q, rb)
sol <- solve(sys)
x <- as.array(current(sys, x))

```

実は Sharpe Ratio 最大化問題は目的関数の分子を $\lambda = \sum_{j \in A} \bar{r}_j x_j - r_f (> 0)$ とおいて、変数を $w_j = x_j / \lambda$ と変換することによって目的関数が二次式となるため、等価な二次計画問題に置き換えることができます。目的関数を一般の非線形な式ではなく、二次式として表現できるということは、数理計画問題を解く上で大きなメリットとなります。次が二次計画問題としてとく場合のモデル記述です。

Sharpe Ratio 最大化モデル (QP 版)

```

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
}

```

それでは解いてみましょう.

```

sys <- System(Sharpe.qp, Q, rb)
nuopt.options(eps=1e-10)
sol <- solve(sys)
w <- as.array(current(sys, w))

```

今 Sharpe.qp を解くことで得られた解 w は, 次のようにすることで元の問題 (Sharpe) の解 x に変換することができます.

```

x.qp <- w / sum(w)

```

この変換により得られた解 $x.qp$ と Sharpe により得られた解 x を比較します.

```

eps <- 1e-4
x[x >= eps]
x.qp[x.qp >= eps]

```

すると, 次のように出力され, 同等の解が得られていることがわかります.

```

> x[x >= eps]
      A071      A080      A081      A087      A103      A113
0.1317169944 0.0622462771 0.1040098858 0.0306944986 0.0256562983 0.0375669711
      A139      A189      A190      A195      A197      A200
0.0034172676 0.2357518759 0.1985072487 0.0052974246 0.1649864616 0.0001487956
> x.qp[x.qp >= eps]
      A071      A080      A081      A087      A103      A113
0.1317799005 0.0621905647 0.1040996606 0.0306578186 0.0253607173 0.0376142619
      A139      A189      A190      A195      A197      A200
0.0033814838 0.2358079302 0.1985771506 0.0051194417 0.1649651212 0.0004270562

```

6 半正定値計画法の利用

6.1 半正定値行列の取得

相関を持ついくつかのアセットの値動きをモンテカルロ法でシミュレーションする場合などには、アセット間の相関行列を与える必要があります。その際、相関行列は原理的にコレスキー分解が可能であるように正定値である必要があります。しかし、正定値であるという性質は直観的に明らかではありません。RSIMPLE に備わった半正定値計画アルゴリズムを用いれば、与えられた（正定値であるとは限らない）行列に最も近い正定値な行列を算出させることができます。

例えば次のすべて正の要素から成る 10×10 の対称行列を取り上げます。（Cormat.A は RSIMPLE にサンプルとして含まれています。）

```
> 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
```

この行列の固有値を R を使って求めると以下のようになり、最小固有値（-0.1603709）が負になっていることがわかります。

```
> eigen(Cormat.A)$values
[1]  2.6192649  2.0903778  1.2534419  1.0818527  1.0000000  0.8367970
[7]  0.7030020  0.6042132 -0.0285787 -0.1603709
```

これより、Cormat.A は正定値ではありませんのでコレスキー分解することができません。

次は与えられた行列とフロベニウスノルムの意味で近い、正定値行列を求めるための数理計画モデルです。

```
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)
# 最小固有値が minEig 以上
M >= minEig
# 初期値
X[i, i] == 1
# 対称行列であることを要請
X[i, j, i < j] == X[j, i]
}

```

このモデルを利用し、先ほどの行列 Cormat.A にフロベニウスノルムの意味で最も近い、最小固有値が 0.001 以上の正定値対称行列を求めてみましょう。

```

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

```

最適化問題を解いた結果得られた Aopt は次のような行列です。Aopt の各要素は元の行列 Cormat.A とそれなりに近い値になっていることにご注意ください。

```

> 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


```
attr("indexes")
[1] "i" "j"
```

最後に, A_{opt} は本当に最小固有値が 0.001 以上の正定値行列であることを検証してみます. A_{opt} の固有値を R を使い求めた結果は次のようになり, 最小固有値が 0.001 より大きいことから確認が出来ます.

```
> eigen(Aopt)$values
[1] 2.580801901 2.028569698 1.236808663 1.067535281 0.968330525 0.830142890
[7] 0.685764533 0.600046357 0.001000127 0.001000025
```

6.2 ロバスト最適化

ポートフォリオのリスクを投資対象の収益率の分散共分散行列を用いて計測するマルコビッツモデルにおいて, 与えられた分散共分散行列はしばしば不確実性を伴います. この不確実性に対してロバストな解を得る問題を考えます.

以下の典型的な平均・分散モデルを考えます. ここでは期待収益率とリスクの線形結合 (効用関数) を目的関数としています.

$$\begin{aligned} \max_x (\mu^T x - \lambda x^T \Sigma x) \\ \text{s.t. } x^T e = 1 \end{aligned}$$

なお, μ を期待リターン, Σ をリターンの分散共分散行列, x をポートフォリオの重み, λ をリスク回避係数とします.

ここで, 上記のモデル中の分散共分散行列 Σ に不確実性が伴うものとして, 以下のロバストポートフォリオ最適化問題を考えます. ただし, U_Σ は不確実性が伴うことによって取りうる Σ に関する行列の集合とします.

$$\begin{aligned} \max_x (\mu^T x - \lambda \max_{\Sigma \in U_\Sigma} (x^T \Sigma x)) \\ \text{s.t. } x^T e = 1 \end{aligned}$$

上記の問題に出てくる U_Σ について, 各要素が

$$\Sigma_{\text{lower}} \leq \Sigma \leq \Sigma_{\text{upper}}$$

のような区間を持つ分散共分散行列からなる集合とします. このとき問題は, 新たに行列 U, L を用いて以下のような問題に置き換えることができます.*1 なお行列 A, B に対し, $A \bullet B$ は, A と B の内積 $\sum_{i,j} A_{ij} B_{ij}$ を表すものとします.

*1 参考文献: Fabozzi, Kolm, Pachamanova, Focardi, Robust Portfolio Optimization and Management, 2007

$$\begin{aligned} & \max_{x,U,V} (\mu^T x - \lambda(\Sigma_{\text{upper}} \bullet U - \Sigma_{\text{lower}} \bullet L)) \\ & \text{s.t.} \begin{cases} x^T e = 1 \\ \begin{pmatrix} U - L & x \\ x^T & 1 \end{pmatrix} \succeq 0 \\ U \geq 0, L \geq 0 \end{cases} \end{aligned}$$

次の RSIMPLE のスクリプトは、分散共分散行列 Σ の各要素の下限を表す行列 sigL, 上限を表す行列 sigU, 収益率の期待値 mu, およびリスク回避係数 lambda が与えられているとき、上記の置き換え後の問題を解くものです。

```
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")) # Asset に "dummy" を加えた集合 V を作る
  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
}
```

ここでは、必要なデータは以下のように与えられているものとします。(RSIMPLE にサンプルとして含まれています.)

> Robust.sigU # 分散共分散行列の要素の値の上限

	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 # 分散共分散行列の要素の値の下限

	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 # 収益率の期待値

X1	X2	X3	X4	X5	X6	X7	X8
0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

リスク回避係数 λ が 1.0 の場合について、実際に解いてみます。

```
s <- System(model=Robust, Robust.sigL, Robust.sigU, Robust.mu, 1.0)
# 求解
sol <- solve(s)
# 図示
xopt <- as.array(current(s, x))
barplot(names=rownames(xopt), xopt)
```

解いた結果、次のようなポートフォリオが得られました。

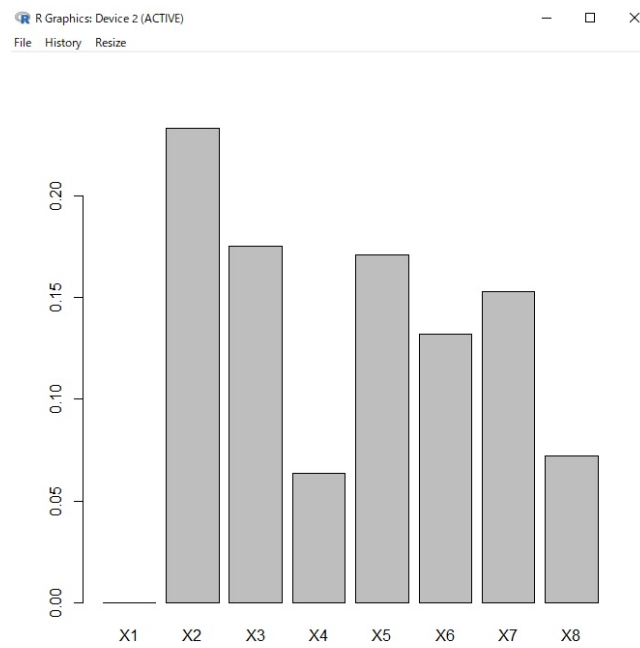


図15 ロバストポートフォリオ

7 非線形フィッティング

非線形なモデル関数をデータに合わせこむ問題は次の形の非線形な数理計画問題として一般的に記述することができます。

-
- 目的関数:
 - 誤差二乗和 $\sum_i e_i^2 \rightarrow$ 最小化
 - 変数:
 - モデルパラメータ a_1, \dots, a_n
 - 制約:
 - 各観測点の誤差の定義 $e_i = f(x_i; a_1, \dots, a_n) - y_i$
 - パラメータ値に関する制限 $g(a_1, \dots, a_n) \leq 0$
-

ここで, $f(x; a_1, \dots, a_n)$ はパラメータ a_1, \dots, a_n に対して線形あるいは非線形な式として定義されている (変数 x を持つ) モデル式で, 観測データとして, 説明変数と目的変数の m 個の組 (x_i, y_i) ($i = 1, \dots, m$) が与えられているとします。

重回帰は上記の問題の最も基本的な形であり, $f(x; a_1, \dots, a_n)$ が線形関数である場合に対応します。 $f(x; a_1, \dots, a_n)$ が線形関数であってもパラメータの値に制限 (制約) がある場合にはこの数理計画問題を解く必要があります。

7.1 イールドカーブのフィッティング

金融工学に表れるフィッティングで最も多く現れるのは, スポットレート (割引債の利回り) を観測された利付債の現在価格から推定するイールドカーブフィッティングです。

償還期間 t における額面価格 100, クーポンレート 1% の利付債の理論現在価格 $S(t)$ は, スポットレート r_i を用いて以下のように表すことができます。

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

ここで償還期間 T_i の利付債の現在価格 S_i が以下のように観測されたとします。 (データは RSIMPLE にサンプルとして含まれています。)

```
> Yield.term
```

```
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
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
```

```

27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
 5  5  5  5  6  6  6  6  6  6  7  7  7  7  7  7  8  8  8  8  8  8  9  9  9  9
53 54 55 56 57 58 59 60
 9  9 10 10 10 10 10 10
> Yield.price
      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
101.73  99.78 100.47  98.19  99.55  99.79  98.40  96.60  96.93  96.80  94.91
      23     24     25     26     27     28     29     30     31     32     33
 96.28  95.33  95.13  93.50  93.42  91.56  92.67  96.28  89.66  89.46  91.21
      34     35     36     37     38     39     40     41     42     43     44
 91.42  93.32  89.76  90.18  88.58  87.41  90.33  87.50  87.66  86.06  85.55
      45     46     47     48     49     50     51     52     53     54     55
 84.74  88.78  86.79  88.76  84.95  84.31  87.24  84.73  83.76  84.02  81.75
      56     57     58     59     60
 84.46  82.51  85.42  81.45  85.36

```

この時、各観測点 i における理論現在価格 $S(T_i)$ と観測された現在価格 S_i の差の二乗和：

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

を最小にするスポットレート r_i を求めることにします。次の数理計画モデルはこの問題を解くためのものです。なお、このモデルについて、引数で与えるデータは各債券の償還期間 (tvalue.d) および市場で観測された価格 (Svalue) となります。telem.d は $S(t)$ の各項のべきの値のベクトルです。

イールドカーブ推定モデル

```

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]
}

```

では、このモデルにデータを与えて解いてみます。

```

# 展開
sys <- System(Yield, Yield.telem, Yield.term, Yield.price)
# 実行
sol <- solve(sys)
# 解の取得
r <- as.array(current(sys, r))
# 図示
plot(r, type="b")

```

その結果として以下のイールドカーブが得られます。

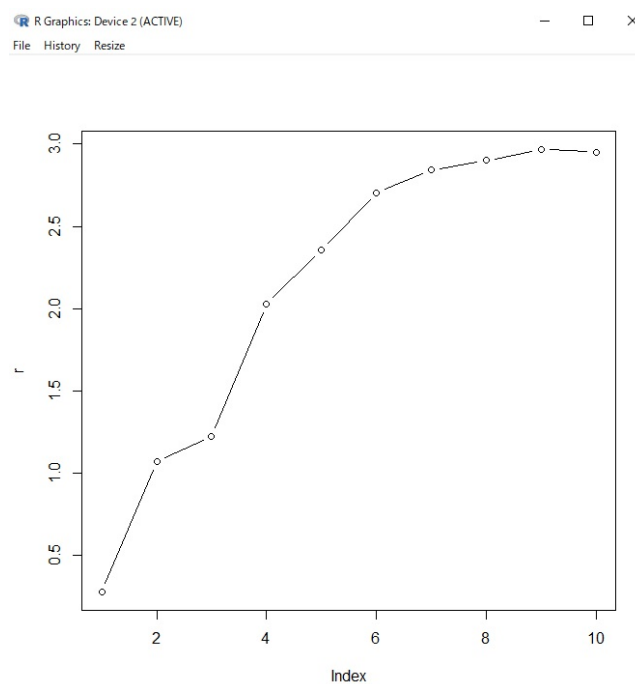


図16 イールドカーブ

7.2 格付け推移行列の推定

格付け (Rating) とは、企業の発行する社債の元本、利息の支払い能力をランク形式で表示したものです。格付け会社は独自の調査結果のもと、A, B, C などの記号を用いて対象社債のリスク度合いを示します。格付け推移行列とは、ある格付け評価を受けている企業が、一定期間後にどのような格付けとなるかについての確率を表す行列のことをいいます。

ここでは、1 年分の格付け推移行列から 1 か月分の格付け推移行列を推定する問題を扱います。この問題を数理計画モデルとして記述すると次のようになります。

格付け推移行列推定モデル

```
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] * q8[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]  
  q[i, i] <= 1  
  0 <= q[i, j, i != j]  
  q[i, j, i != j] <= 0.05  
  q[i, i] ~ 0.9  
}
```


それでは, Rating.Q0 という 1 年分の格付け推移行列が与えられた場合についてこの問題を実際に解いてみます. (データは RSIMPLE にサンプルとして含まれています.)

```
# 展開
sys <- System(Rating, Rating.Q0)

# 実行
sol <- solve(sys)

# 解の取得
q <- as.array(current(sys, q))

# 確認
q12 <- diag(1, nrow(q))
dimnames(q12) <- dimnames(q)
for(i in 1:12) q12 <- q12 %*% q

# 表示
r.order <- order(rownames(Rating.Q0)) # 格付けの順番を定義
```

解いた結果得られた 1 か月分の格付け推移行列を表示します.

```
> 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
```

次に 1 か月分の格付け推移行列を 12 乗, つまり 1 年分に変換した値を示します.

```
> round(q12[r.order, r.order], digits=4)
      AAA    AA     A   BBB    BB     B    CCC     CC      C
AAA 0.9649 0.0347 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 0.0433 0.9364 0.0186 0.0003 0.0000 0.0000 0.0000 0.0000
BBB 0.0000 0.0007 0.0349 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 0.0083 0.9675 0.0242
C    0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0001 0.0265 0.9733

```

最後に、比較のため与えられた 1 年分の格付け推移行列を表示します。

```

> 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

```

与えられた 1 年分の格付け推移行列と、推定した 1 か月分の格付け推移行列を 1 年分に変換した行列を見比べることで、1 か月分の格付け推移行列が適切に推定できていることを確かめることができます。

7.3 ロジスティック回帰

次のように、RSIMPLE を用いてロジスティック回帰を行うモデルを作成することもできます。

```

#### ロジスティック回帰モデル ####

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]), 1)
}
```

ここでは、例として Iris virginica と Iris versicolor の判別分析を扱います。まずは問題を解き、解を得ます。(データは RSIMPLE にサンプルとして含まれています。)

```
# 展開
sys <- System(LogReg, LogReg.X, LogReg.t)
# 実行
sol <- solve(sys)
# 解の取得
a <- as.array(current(sys, a))
a0 <- as.array(current(sys, a0))
```

得られた解について、学習データとは別のテストデータで予測力を評価します。

```
# 予測
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
```

RSIMPLE には、数理計画モデルを自在に設計できるというメリットがあります。例えば、推定したいパラメータに何らかの制約を課して回帰を行ったり、ロジット関数を 2 次式で表して予測力向上のために 2 次の係数行列に半正定値制約を課したりすることができます。

最後に、注意点としてロジスティック回帰の引数発散があります。関数 exp は引数の値が大きいと関数値が極端に大きくなります。このため引数に対して制約を設けると、より安定化することがあります。以下記述例です。

```
#### ロジスティック回帰モデル ####
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=1)
y[1] ~ exp(Sum(a[i] * X[1, i], i) + a0) / (1 + exp(Sum(a[i] * X[1, i], i) + a0))
Sum(a[i] * X[1, i], i) + a0 <= 20
mle <- Objective(type="maximize")
mle ~ Sum(t[1] * log(y[1]) + (1 - t[1]) * log(1 - y[1]), 1)
}

```

8 最適化ソルバー NUOPT

RSIMPLE は汎用の最適化エンジン NUOPT と接続しています。System オブジェクトを引数として solve() コマンドをコールすると NUOPT が起動します。System オブジェクトには問題のタイプ（制約式や目的関数の次数や変数の種類）の情報が含まれているため、NUOPT はその情報に基づいて適切な最適化アルゴリズムの選択やそのパラメータ設定を自動的に行うことができます。自動設定で通常は問題ありませんが、実際規模の問題や複雑な問題においては、手動でアルゴリズムの選択やパラメータの設定を行う方が有利な場合があります。

「nuopt.options() を使って NUOPT をカスタマイズする」はそうした場合に nuopt.options() という関数を使って NUOPT の動作をカスタマイズする方法について述べています。

続いて「エラーメッセージ」では RSIMPLE のご利用上によく現れるエラーの意味や回避方法について述べています。

RSIMPLE は最適化ソルバー NUOPT にモデリング言語を介さず直接データを与えて最適化を行うための関数 solveQP を備えています。「solveQP」は solveQP の使い方について解説しています。この方法は最適化問題を記述する生データである制約式を記述する係数行列やヘッセ行列を R 上のデータとして既にお持ちの場合に有効です。

8.1 nuopt.options() を使って NUOPT をカスタマイズする

NUOPT には動作をカスタマイズするための様々なチューニングパラメータがあります。以下の表がその一覧を示しています。なお、各パラメータの詳細は オンラインマニュアル の SIMPLE マニュアルをご覧ください。

名前	デフォルト値	設定可能な値	設定例
method	“auto”	“auto”, “simplex”, “asqp”, “wcsp”, “higher”, “lipm”, “tipm”, “lepm”, “tepm”, “lsqp”, “tsqp”, “line”, “trust”, “bfgs”, “lbfgs”, “lsdp”, “csdp”, “qnsdp”, “trsdp”	“simplex”
crossover	“off”	“on”, “off”	“on”
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-07	double	1.00E-8
maxitn	150	int	300
maxnod	-1	int	500000
maxtim	-1(except wcsp), 5(wcsp)	int	60
told	1.00E-06	double	1.00E-8

名前	デフォルト値	設定可能な値	設定例
tolx	1.00E-08	double	1.00E-10
maxintsol	-1	int	1
maxmem	-10	int	-100
wcspTryCount	1	int	5
wcspRandomSeed	1	int	23

これらのパラメータは `nuopt.options` コマンドから次のようにして設定することができます。

```
nuopt.options(method="simplex") # 最適化アルゴリズムを "simplex" に
nuopt.options(eps=1.0e-10) # 停止条件をより小さめに（目安は 1.0e-8）
# スケーリングをし、反復回数の上限を設定する
nuopt.options(scaling="off", maxitn=300)
```

`nuopt.options` はカンマ (,) で区切られた “名前 = 値” の列を引数としてとります。現状のパラメータ設定の内容は次のように入力することによって表示されます。

```
nuopt.options() # 現状のパラメータ設定の内容を表示
```

現状のパラメータ設定を R データに保存するには次のようにします。

```
cops <- nuopt.options() # 現状のパラメータ設定をオブジェクト cops に保存
```

保存したパラメータを読み込むには次のようにします。

```
nuopt.options(cops) # cops に保存した内容を設定する
```

パラメータ設定は RSIMPLE セッションの間中保存されることにご注意ください。デフォルトの設定は `nuopt.options()` に引数 `NA` を与えることによって得られます。したがって、次のようにするとデフォルトの設定に戻ります。

```
nuopt.options(nuopt.options(NA)) # デフォルトのパラメータに戻す
```

デフォルトの設定は通常問題が起きないように選ばれていますが、計算のパフォーマンスよりも安定性を主眼として設定されています。そのため、ユーザが問題に関する情報を元に手で調整することによってより良い結果を得ることが可能です。次からのセクションではその典型的なケースについて述べます。

8.1.1 特殊な大規模二次計画問題を高速に解く

ポートフォリオ最適化問題は大規模二次計画問題（制約は線形で、目的関数は二次関数）となります。これらの二次計画問題に関して、「変数に比べ制約式の数が少ない」という特徴をもつものが多く見られます。そのような問題は

アルゴリズム “asqp”（有効制約法）が有利です。次のように設定してみてください。より速く最適化計算が終了する可能性があります。

```
nuopt.options(method="asqp") # アルゴリズム asqp を用いる
```

8.1.2 線形計画問題・二次計画問題をより高精度で解く

特に設定を行わなければ、NUOPT は線形計画問題（目的関数と制約式はすべて線形）と二次計画問題（制約は線形で、目的関数は二次関数）を、内点法によって解きます。大規模問題に対しても高速な求解ができるという意味でこの設定は一般的に有効ですが、線形計画問題は “simplex”（単体法）、二次計画問題は “asqp”（有効制約法）を用いる方が、問題に関する有益な情報が得られる場合があります。特に問題が実行不可能に近い（制約の充足が不可能か非常に難しい）場合、また問題のスケールが悪い（変数や制約式の大きさに非常にばらつきがある）場合には内点法の動作は不安定となり、反復回数オーバー（NUOPT エラーコード 10）が起きることがあります。そのような場合には、次のように設定してみてください。

```
nuopt.options(method="simplex") # 線形計画問題の場合
nuopt.options(method="asqp")   # 二次計画問題の場合
```

変数や制約の数が一万個を超えるなどの場合には若干時間を所要しますが、これらの手法は内点法にくらべて頑健で、より正確な解を与えます。

8.1.3 計算機資源の利用を制限する

整数変数を含む問題を与えた場合、NUOPT は分枝限定法を用いて解きます。一般に分枝限定法は計算時間およびメモリーを所要するアルゴリズムです。大規模な問題など、難しい問題では次のようにして計算時間およびメモリーの利用を制限することが時に有効です。

```
nuopt.options(maxtim=600) # 計算時間を 10 分 (600 秒) に制限
nuopt.options(maxmem=10000) # メモリーの利用量を 1G byte に制限
# システムの残りメモリーが 100M byte をしたまわるまで計算を行う
nuopt.options(maxmem=-100)
```

これらの上限に到達した場合には、NUOPT は時間制限オーバー、あるいはメモリーオーバーと出力して計算を停止し、そのときまでに見つかった最良の解を返します。

アルゴリズム “wcsp” を起動している場合には、時間制限が必須です。これはアルゴリズム “wcsp” が最適性を判定することができないためです。アルゴリズム “wcsp” の利用の方法とパラメータ wcspRandomSeed, wcspTryCount の設定については「wcsp と集合分割問題」をご覧ください。

8.1.4 分枝限定法のチューニング

整数変数を含む数理計画問題を解く際に起動される分枝限定法を次のパラメータで制御することができます。

```
nuopt.options(maxnod=100000) # ノード数を 100000 以下に制限する
nuopt.options(maxintsol=1) # 実行可能解を一つ見つけたら止まる
# 目的関数についての最適性の判定条件を 0.99 だけ緩める
nuopt.options(addToCutoff=0.99)
```

maxintsol が与えられている場合、その個数の実行可能解が発見された段階で停止します。addToCutoff が与えられている場合、分枝限定法の反復の中で、addToCutoff よりも改善度の少ない解は無視します。もし目的関数が整数であることがわかっているのであれば、addtoCutoff を 0.99 (1 以下) に設定してもアルゴリズムは正確性を失いません。

8.1.5 非線形計画法, 半正定値計画法の安定化のためのチューニング

これまで述べた例にあるように、RSIMPLE は非線形計画問題、半正定値計画問題を特にパラメータの設定なく解くことができます。しかし、難しい問題になるとソルバーの計算が失敗し、解の精度が落ちたり、エラーを起こしたりすることがあります。計算の失敗の理由としては次のようなものがあり、対策としてこれらのいずれかを除去することが有力です。

1. 問題が実行不可能に近い

制約条件が厳しすぎる場合に、内点法のふるまいは不安定になります。

2. スケールのばらつきが大きい

制約および目的関数は、スケーリングされているのが好ましいといえます。すなわち絶対値において 1 程度の値に揃っているのが望まれる状況です。典型的な悪条件としては、変数や制約式の一部が他の変数や制約式と比べて突出して (1000 あるいは 1/1000 程度の) 大きな (あるいは小さな) 絶対値を持っていることです。

3. 問題が凸ではない

実行可能領域が凸であってかつ目的関数が凸関数である場合、その最適化問題は凸な問題と呼ばれます。凸な問題は解きやすく比較的解析が容易です。しかし、その保障がない場合、問題は凸なケースに比べて非常に難しくなります。

凸なケースでは局所最適解が大域的最適化である保障がありますが、凸でないケースではその保障がなく、複数の局所的最適解が存在することが解析を難しくしています。

NUOPT は凸でない問題を扱うことができますが、そのアルゴリズムはいずれか一つの局所的な最適解を求めるためのものであり、大域的な最適解を常に出力できるものではありません。

4. 問題の非線形性が高い

もし、問題が $\exp(x)$ や a/x , 高次の多項式を含んでいたら、その分問題は線形な問題から遠ざかり、扱いが難しくなります。最適化ソルバーは内部で非線形モデルを線形に近似するというを行いますので、その近似が正確であるほど性質は良好になります。

上記の対策としては、以下の手段が一般的に有効です。

- 問題の実行可能性を調べる

制約を緩めたり、スラック変数を導入したりした結果振る舞いに変化するかどうかを見ることによって、制約がどの程度厳しいか、制約の厳しさが振る舞いにどの程度影響しているかを見ることができます。

- スケールを変えてみる

手動で変数あるいは制約式をスケールしてみます。また、NUOPT が自動的に行っているスケーリングを、次のようにしてやらせないで試してみます。 `nuopt.options(scaling="off")` # 自動スケーリングを行わない

- 変数に初期値を与える

変数に「変数（Variable）に初期値を与える」において説明した方法で初期値を与えます。特に $1/x$ のように、関数が極をもち、その周辺では発散するような場合には初期値の工夫が有効です。

- アルゴリズムを変更して試してみる

NUOPT にはデフォルトでは起動されない、多数のアルゴリズムが実装されています。次の表を参考に、手動でそれらの一つを起動するように設定された方が良い結果をもたらすことはあり得ます。

	凸な問題用	非凸な問題用
非線形	“lipm”, “line”, “lepm”, “lsqp”	“tipm”, “trust”, “tepm”, “lbfgs”, “tsqp”
半正定値	“lsdp”(linear SDP only), “csqp”	“trsdp”, “qnsdp”

特にパラメータを指定しない場合には、NUOPT は問題が線形であるか非線形であるか、半正定値性の制約を含むかという情報を元にアルゴリズムの選択を行います。問題が凸であるかどうかという情報はアルゴリズムの選択上重要ですが、問題が凸であるかどうかはわからないので、NUOPT は非線形計画問題であれば常に非凸な状況に対応できる “tipm” を用いて問題を解きます。半正定値性の制約を含み、問題がすべて線形であるならば “lsdp”，そうでなければ “trsdp” を選択します。例えば、ユーザが扱っている問題が凸であることがわかっているのであれば、以下のようにしてアルゴリズムとして “lipm” を用いるのは良い選択です。なぜなら、凸性を仮定しない分、“lipm” は “tipm” よりも高速であるためです。

```
nuopt.options(method="lipm") # Lipm を方法として採用する
```

8.1.6 ヒープメモリをクリアする

RSIMPLE は最適化のためにヒープメモリを使用します。セッションの間に最適化を多数回繰り返して、ヒープメモリが積みあがってくると、NUOPT はメモリーエラー（エラーコード 1 あるいは 8）で終わってしまうようになりますことがあります。そのような場合には、次のコマンドでヒープメモリをクリアしてください。

```
nuoptAllClear()
```

8.2 エラーメッセージ

この節では RSIMPLE を使うにあたって出会うエラーの意味について解説します。解決の方法も一部示しています。

8.2.1 モデリング言語解釈部からのエラー

次はモデリング言語の解釈部から出される典型的なエラーの表です。

メッセージ	意味
エラー：予想外の ‘,’ です …	index の右辺に ‘dprod’ が抜けています
Error: Illegal use of summary function on Simple objects	‘Sum’ を使うべきところで ‘sum’ が用いられています
Error: SIMPLE object not created in this R session	前回以前のセッションで作成された、RSIMPLE 固有のオブジェクトなので内容がわからないオブジェクトです。
«SIMPLE 1» Infeasible bound for variable …	変数の上下限が矛盾しています。
«SIMPLE 67» Index error in reference of <objname> with no index but should be with index of dimension 1**	添字があると定義しているオブジェクトに添字を与えていません。
«SIMPLE 67» Index error in reference of <objname> scalar but with index of dimension 1.	添字がなしで定義しているオブジェクトに添字を与えています。
«SIMPLE 82» Subscript <elem> of <objname> out of range.	インデックスの範囲が誤っています。
«SIMPLE 168» Objective can only be assigned once.	目的関数に複数回代入しています。
«SIMPLE 214» Warning constraint#1 reduce to … (always satisfied)«SIMPLE 216» Trivial and Infeasible constraint appeared.	常に満たされないか、常に満たされる（意味のない）制約が定義されています。

以下では各メッセージについて意味や対処法を解説します。

- エラー：予想外の ‘,’ です …

このメッセージは複数の添字を持つオブジェクトの宣言で、“index=” の右辺に dprod を忘れると出力されます。

例えば、次のようにするとこのエラーメッセージが出ます。

```
S <- Set()
U <- Set()
i <- Element(set=S)
```

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

この例では x,y はそれぞれ添字を 2 つ持つにもかかわらず、宣言の際に “dprod” を忘れているためエラーとなっています。このため、次のように “dprod” を記述することでこのエラーは解消します。

```
S <- Set()
U <- Set()
i <- Element(set=S)
j <- Element(set=U)
x <- Variable(index=dprod(S,U)) # 正しい
y <- Variable(index=dprod(i,j)) # 正しい
```

- **Error: Illegal use of summary function on Simple objects**

モデリング言語の記述で、変数や式の和を定義するときには “sum” ではなく、“Sum” を用いなければなりません。このメッセージは、次の例のように “sum” が誤って用いられた場合に出力されます。

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

- **Error: SIMPLE object not created in this R session**

System, Variable, Parameter, Set, Element などのオブジェクトの内容は R のセッションをまたがっては保存されません。このメッセージは以前のセッションで作成した、既に内容が失われているオブジェクトを参照した場合に出現します。重要な最適化の結果などは、「最適化の結果 (Variable/Expression) を取り出す」で述べたように as.array や as.list を用いて R オブジェクトの形で蓄えておく必要があります。

- **«SIMPLE 1» Infeasible bound for variable …**

変数の上下限はすべての設定の共通部分をとります。このメッセージはその際に矛盾が起きており、共通部分が存在しないことを示します。

例えば、次のような x に関する共通部分が存在しないモデルについて、System 関数で System オブジェクトを生成しようとした場合にこのメッセージが表示されます。

```
error1 <- function() {
  x <- Variable()
  x >= 7
```

```

    x <= 5 # error (conflict)
}
sys <- System(error1)

```

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

これらのメッセージは添字の使い方の矛盾を指摘するものです。モデル中に次のようなことが書かれている場合に出 force されます。

```

x <- Variable()
x[3] # error (x は添字をつけてはならない)
S <- Set(1:3)
y <- Variable(index=S)
y <= 2 # error (y は添字をつけなければならない)
y[4] # error (添字の範囲が違う)

```

- «SIMPLE 168» Objective can only be assigned once

目的関数を表す Objective オブジェクトに対して、代入は一度しか出来ません。次の例のように二度目の代入を行なおうとした場合に、このメッセージは出力されます。

```

f <- Objective(type=minimize)
x <- Variable()
y <- Variable()
f ~ x+y
f ~ x-y # error (再度代入を行なおうとしています)

```

- «SIMPLE 214» Warning constraint#1 reduce to ... (always satisfied)
- «SIMPLE 216» Trivial and Infeasible constraint appeared

定数の初期化忘れのミスなどで、常に満たすことができない制約式や意味のない制約式を定義してしまうことがあります。これらのメッセージはそのような制約式が現れたことを示しています。次の例のようなモデルについて System により System オブジェクトを生成しようとするメッセージが表示されます。

```

error2 <- function() {
  S <- Set(1:3)
  i <- Element(set=S)
  x <- Variable(index=i)
  a <- Parameter(index=i) # a の中身は与えられていない (零だと解釈する)
  Sum(a[i] * x[i],i) <= 0 # 意味のない制約 (左辺が零)
}

```

```
}
sys <- System(error2)
```

8.2.2 NUOPT が出力するエラー

問題の解釈が完了した後に最適化の過程でエラーを発見すると、次のようなメッセージが現れます。

```
<<SIMPLE 193>> Error in solve():

"<<NUOPT 10>> IPM iteration limit exceeded." # ソルバーエラーの内容
```

エラーコードは、solve() の戻り値のリストの中の"errorCode" というアイテムに格納されていますので取得することができます。次の表にはメッセージの意味と解説と関連項目について記述しています。

コード	メッセージ	意味
1	memory error in preprocessing.	前処理のフェーズでメモリーエラーが起きました (「ヒープメモ리를 クリアする」を参照)。
2	infeasible(linear constraints and variable bounds)	線形制約と変数の上下限が矛盾しています。
3	no variables in this model.	変数が定義されていません。
4	no functions in this model.	制約も目的関数も定義されていません。
5	infeasible variable bounds.	変数の上下限が矛盾しています。
7	internal error.	内部エラーが発生しました。 (nuopt-support@ml.msi.co.jp にご連絡をお願いします)
8	memory error in optimization phase.	最適化アルゴリズム適用のフェーズでメモリーエラーが起きました (「ヒープメモ리를 クリアする」を参照)。
9	step reduction limit exceeded.	直線探索を行うアルゴリズムで、ステップサイズの設定に失敗し、降下方向が発見できませんでした。数値的なエラーか、問題が凸でない可能性があります (「非線形計画法, 半正定値計画法の安定化のためのチューニング」を参照)。
10	IPM iteration limit exceeded.	内点法の反復回数が上限を超えました。 (「線形計画問題・二次計画問題をより高精度で解く」および「非線形計画法, 半正定値計画法の安定化のためのチューニング」を参照)
11	infeasible	問題が実行不可能です (制約を満たす変数が存在しません)。

コード	メッセージ	意味
13	unbounded.	有界な問題ではありません (目的関数を任意に小さくあるいは大きくできる実行可能解の列があります).
14	integrality is violated.	出力された解においては整数変数が整数値以外の値をとっています (整数変数を含む問題に対して, 強制的に tipm や line などの内点法を適用した場合に現れます. これらのアルゴリズムは整数性を意識しないのでこのような結果になります).
15	simplex misapplied to QP.	二次計画問題に対してアルゴリズム "simplex" を指定しました.
15	simplex/asqp misapplied to NLP.	一般の非線形計画問題に対して, アルゴリズム "simplex", "asqp" を指定しました.
16	Infeasible MIP.	制約式と変数に対する整数条件を満たす解が存在しないことがわかりました (整数条件を緩めれば解は存在します).
17	B & B node limit reached (with feas.sol.).	実行可能解が見つかりましたが, 分枝限定法のノード数 (子問題数) がパラメータ maxnod で指定された個数を超えたので停止しました. 見つかった解が最適であるかどうかはわかりません (「分枝限定法のチューニング」参照).
18	MIP iteration failed (with feas.sol.).	実行可能解が見つかりましたが, 分枝限定法のプロセスの途中で数値的な問題が発生して停止しました. 見つかった解が最適であるかどうかはわかりません.
19	B & B node limit reached (no feas.sol.).	分枝限定法のノード数 (子問題数) がパラメータ maxnod で指定された個数を超えました. 実行可能解は見つけれませんでした (「分枝限定法のチューニング」参照).
20	MIP iter. failed (no feas.sol.).	分枝限定法のプロセスが数値的問題によって失敗し, 実行可能解も見つけれませんでした.
21	B & B itr. timeout (with feas.sol.).	分枝限定法が, パラメータ maxtim で与えた時間上限を超過したため終了しました. 実行可能解は見つっていますが, 最適であるかどうかはわかりません (「計算機資源の利用を制限する」参照).

コード	メッセージ	意味
22	B & B itr. timeout (no feas.sol.).	分枝限定法が, パラメータ maxtim で与えた時間上限を超過したため終了しました. 実行可能解も見つけられませんでした (「計算機資源の利用を制限する」参照).
27	SIMPLEX iteration limit exceeded.	単体法の反復回数が上限値を超えました.
28	higher-order method is only for LP.	アルゴリズム"higher" を非線形計画問題に適用しようとした.
29	iteration diverged.	計算が発散しました (目的関数を任意に小さくあるいは大きくできる実行可能解の列があると思われます)
33	Bound violated.	内部的に施していた自動スケーリングを戻した結果変数の上下限を満たしていないことがわかりました. 変数や制約式のスケールのばらつきがかなり大きいことが想像されます.scaling="off" として自動スケーリングを行わないか, 入力前に問題をスケーリングすることをお勧めします (「非線形計画法, 半正定値計画法の安定化のためのチューニング」参照) .
34	Bound and Constraint violated.	33 と同じですが, 変数の上下限とあわせて制約式の上下限も満たしていませんでした (「非線形計画法, 半正定値計画法の安定化のためのチューニング」参照).
35	Constraint violated.	33 と同じですが, 制約式の上下限を満たしていないことがわかりました (「非線形計画法, 半正定値計画法の安定化のためのチューニング」参照) .
36	Equality constraint violated.	33 と同じですが, 制約式の上下限を満たしていないことがわかりました (「非線形計画法, 半正定値計画法の安定化のためのチューニング」参照) .
37	B&B terminated with given # of feas.sol.	パラメータ maxintsol で指定されただけの個数の整数解が求められましたので計算を終了しました (「分枝限定法のチューニング」参照).
38	dual infeasible.	分枝限定法の途中で解く部分問題が実行不可能と判定されました. 変数や制約式のスケールのばらつきがかなり大きく, 数値的条件が悪いことが考えられます.

コード	メッセージ	意味
39	IPM iteration timeout.	内点法の反復がパラメータ <code>maxtim</code> で設定した上限を超えました. 数値的条件が悪いことや問題の規模が大きいことなどが考えられます.
40	SQP iteration limit exceeded.	逐次二次計画法の反復回数が上限を超えました (「非線形計画法, 半正定値計画法の安定化のためのチューニング」参照). 数値的条件が悪いことが考えられます.
41	SQP internal error.	逐次二次計画法の内部エラーです.(nuopt-support@ml.msi.co.jp までご連絡ください)
43	B&B memory error (with feas.sol.).	実行可能解を求めることができませんが, 分枝限定法のプロセスでメモリーを使い尽くしたために停止しました (「計算機資源の利用を制限する」参照). 求められた解が最適解であるかどうかはわかりません.
44	B&B memory error (no feas.sol.).	実行可能解を求めることができませんが, 分枝限定法のプロセスでメモリーを使い尽くしたために停止しました (「計算機資源の利用を制限する」参照). 実行可能解を求めることができませんでした.
46	trust region too small	信頼領域法が失敗しました. 数値的条件が悪いと思われます (「非線形計画法, 半正定値計画法の安定化のためのチューニング」参照).
47	Continuous Variable <varname> cannot be included in model for wcsp.	<code>wcsp</code> が扱うことができるのは 0-1 整数変数のみですが, 0-1 整数変数以外が現れました (「アルゴリズム <code>wcsp</code> を使うときの注意点」参照).
48	Variable <varname> appear in two selection()	変数 <varname> が二つ以上の <code>selection</code> 文に現れています. 変数は高々一つの <code>selection</code> 文にしか現れることはできません.
49	Variable <varname> is fixed to infeasible value.	変数 <varname> が上下限の外に固定されているので制約式を満たすことができません.
50	Both of two variables <varname1> and <varname2> cannot be 1.	変数 <varname1> と変数 <varname2> がひとつの <code>selection</code> 文に現れていますが, 両者ともに 1 に固定されています. このため, <code>selection</code> 文の意図と矛盾しています.
51	wcsp is not available without SIMPLE.	アルゴリズム <code>wcsp</code> は <code>solveQP()</code> を用いた時に使うことはできません.

コード	メッセージ	意味
52	<number>th selection() statement has no movable binary variables.	ある selection 文に入っているすべての変数は、固定されており、動くことができません。
54	Constraint <name>'s weight is <value> should be -1 or non-negative value.	ソフト制約の重みとしては -1 あるいは正の整数値のみが許されています。
55	exterior solution obtained.	外点法（アルゴリズム”tepm”/”lepm”）が実行可能領域の外の解を見つけて停止しました（「非線形計画法, 半正定値計画法の安定化のためのチューニング」参照）
110	CF failed at getcky	半正定値計画問題を解く過程で数値的な問題が起きました（「非線形計画法, 半正定値計画法の安定化のためのチューニング」参照）.
111	CF failed at logdet	110 と同じです.
112	InvMat cannot obtained at calxx1	110 と同じです.
113	GSEP failed at minevl	110 と同じです.
114	trianglization failed at minevl	110 と同じです.
115	Minimum eigenValue cannot obtained	110 と同じです.
120	PDgap is too large[PDfeasible]	双対ギャップが大きい状態で終了していますが、実行可能解は求められています.
121	PDgap is too large[Pfeasible]	双対ギャップが大きい状態で終了していますが、実行可能解は求められています.
122	minus stepsize detected	負のステップサイズが現れました. 問題の非線形性が強いことが想像されます（「非線形計画法, 半正定値計画法の安定化のためのチューニング」参照）.
123	The SDP constraint cannot be treated by specified algorithm.	指定された方法では半正定値条件を考慮することはできません.
124	No SDP constraint is detected.	半正定値条件が存在しませんが、半正定値条件を考慮するアルゴリズムが起動されました.

8.3 solveQP

solveQP という関数を通じて、モデリング言語を介すことなく R オブジェクトを直接最適化アルゴリズム NUOPT に渡すことが可能です. solveQP は次のような混合整数線形・二次計画問題を解くためのものです.

- 目的関数:

- $\frac{1}{2}x^T Qx + q^T x \rightarrow \text{最小化}$
- 変数:
 - x_i ($i \in I$: 整数変数)
 - x_j ($j \notin I$: 連続変数)
- 制約:
 - $c_{LO} \leq Ax \leq c_{UP}$ (線形制約)
 - $b_{LO} \leq x \leq b_{UP}$ (変数の上下限)

solveQP の引数並びは以下の通りです.

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

各引数の意味は以下の通りです. なお, 引数は objQ を除いて省略可能です.

- **objQ:** (matrix or list)

ヘッセ行列 Q

- **objL:** (vector) <optional>

目的関数の線形部分の係数 q . 省略すると 0 であると解釈されます.

- **A:** (matrix or list) <optional>

制約式の左辺行列 A . 省略すると 0 であると解釈されます.

- **cLO, cUP:** (vector) <optional>

制約の上下限 c_{LO}, c_{UP} . 省略するとないものと解釈されます.

- **bLO, bUP:** (vector) <optional>

変数の上下限 b_{LO}, b_{UP} . 省略するとないものと解釈されます.

- **x0:** (vector) <optional>

変数の初期値. 省略すると NUOPT が適当に設定します.

- **isint:** (vector) <optional>

変数が整数変数であるかを表す論理値ベクトル (TRUE が整数変数であることを示します). 省略すると, すべて連続変数であると解釈されます.

- **type:** ("maximize" or "minimize") <optional>

目的関数を最大化するのか最小化するのかを指定します。省略すると、最小化問題であると解釈します。

- **trace:** (TRUE or FALSE) <optional>

最適化計算の実行経過を表示するかどうかを指定します。省略すると、実行経過を表示します。

objQ と A は, R の行列あるいはリストオブジェクトのいずれとして定義することもできます。リストオブジェクトとして与える場合には、非零要素の行番号、列番号、値という三つのアイテムを持つリストとします。

A は行列として表現すると次のようになります。

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

以下のようにリストで表現しても等価です。

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

リストによる表現は行列の非零要素が少ない（ほとんどの要素が零）である場合には有効です。

9 補足 : SNUOPT をお使いの方へ

R 言語の仕様上の理由により, RSIMPLE と SNUOPT の間には制約式の書式に関して次のような違いがあります.

RSIMPLE での制約式の記述 :

```
式 1 @ 式 2 # 二項
      # @ は { ==, ==, <= } のいずれか
```

SNUOPT での制約式の記述 :

```
式 1 @ 式 2          # 二項
式 1 @ 式 2 @ 式 3 # 三項
      # @ は { ==, ==, <= } のいずれか
```

このため, SNUOPT で変数の上下限制約を

```
1 <= x <= 3
```

のように記述されていた場合, RSIMPLE では

```
1 <= x
x <= 3
```

と 2 行に分けて記述することになりますのでご注意ください.