Nuorium Optimizer

SIMPLE チュートリアル V24

株式会社NTTデータ数理システム

2021年12月

MATHEMATICAL SYSTEMS INCARNATION

目　次

第1章 はじめに

Nuorium Optimizer は数理最適化問題を解くための汎用ソルバであり，SIMPLE は数理最適化問題を記述するモデリング言語です．そしてそれらの専用 GUI 環境が Nuorium です．

本稿は Nuorium Optimizer/SIMPLE の基本的な機能に関するチュートリアルです．本稿を一読していただければ，Nuorium Optimizer/SIMPLE の基本的な利用方法がご理解いただけると思います．

本稿の構成は以下のようになっています．

1. はじめに
   本稿の概要の紹介です．
2. 数理最適化問題を記述する
   例題を通して SIMPLE の文法を紹介します．
3. 数理最適化問題を解く
   Nuorium Optimizer による最適化計算の実行方法を紹介します．

1.1 数理最適化問題とは

数理最適化問題とは，「与えられた条件の下で，望ましさの尺度を表す何らかの関数の最小値（最大値）を求め，さらにその最小値（最大値）を与える不特定要素の値を決定する」という問題です．

上記における，「与えられた条件」は制約条件，「望ましさの尺度を表す関数」は目的関数，「不特定要素」は変数と一般に呼ばれています．この用語を用いて書き直すと，数理最適化問題とは，「制約条件を満たす範囲における目的関数の最小値（最大値）及びその最小値（最大値）を与える変数を求める問題」といえます．

例えば，$x \geq 0$ において $3x + 2$ の最小値を求める問題は，数理最適化問題です．この場合，制約条件は $x \geq 0$，目的関数は $3x + 2$，変数は x となります．

この問題は数理最適化の世界では次のように書かれます：

- 目的関数：$3x + 2$ →最小化
- 制約条件：$x \geq 0$

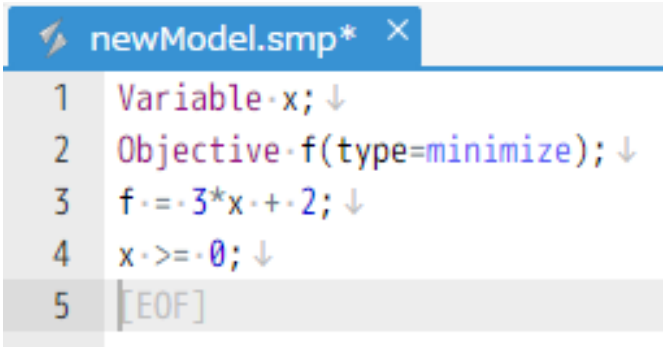
考える間もなく，上記の数理最適化問題の最もよい目的関数値は 2（$x = 0$ のとき）となります．このときの変数の値を最適解と呼びます．

最適解を求めることを「数理最適化問題を解く」あるいは「最適化する」といいます．

1.2 数理最適化問題を解く

早速 Windows 版 Nuorium Optimizer の GUI 環境である Nuorium を用いて数理最適化問題を解いてみましょう．まずは Windows のスタートメニューから「すべてのプログラム」→「MSI Solutions」→「Nuorium」を選択してください．

表示された画面左のパネルにある newModel.smp タブで次のように書き，メニューの「ファイル」→「名前を付けて保存」で適当な場所にファイルを保存します．



最後に，画面左上の「実行」ボタンを押すと Nuorium Optimizer が計算を開始し，画面右のパネルに各種出力が表示されます．

この一連の操作で，あなたは Nuorium Optimizer を使って次の数理最適化問題を解いたことになります．

- 目的関数：$3x + 2$ →最小化
- 制約条件：$x \geq 0$

第2章 数理最適化問題を記述する

本章では，具体的な例題を通して，モデリング言語 SIMPLE の文法を紹介します．

2.1 目的関数・変数・制約

次のような生産計画問題を考えます．

2 つの油田 X, Y が存在し，それぞれ一日あたり重油･ガスを次の量だけ生産する．

生産量/日

	重油	ガス
X	6t	4t
Y	1t	6t

また，重油・ガスの週あたりの生産ノルマが，次のように定められている．

	ノルマ/週
重油	12t
ガス	24t

油田 X, Y の日あたりの運転コストは，次のとおりである．

	運転コスト/日
X	180
Y	160

油田 X, Y ともに，最大で週 5 日まで運転可能である．ノルマを満たしながら運転コストを最小化するためには，それぞれの油田を週あたり何日運転すれば良いだろうか？

この問題を定式化すると，以下のようになります．

変数	
x	油田 X の運転日数/週
y	油田 Y の運転日数/週

目的関数（最小化）	
$180x + 160y$	運転コスト/週

制約条件

$6x + y \geq 12$	重油ノルマ/週
$4x + 6y \geq 24$	ガスノルマ/週
$0 \leq x \leq 5$	油田 X の週あたりの運転日数制約
$0 \leq y \leq 5$	油田 Y の週あたりの運転日数制約

それでは，この問題を SIMPLE で記述した例を見てみましょう．

```
// 油田 X，Y の運転日数/週（変数）
Variable x(name = "油田 X の運転日数");
Variable y(name = "油田 Y の運転日数");

// 運転コスト（目的関数）
Objective cost(name = "全運転コスト", type = minimize);
cost = 180 * x + 160 * y;

// 製品ノルマ
6 * x + y >= 12;      // 重油ノルマ/週
4 * x + 6 * y >= 24; // ガスノルマ/週

// 各油田の日数制約
0 <= x <= 5;        // 油田 X の週あたりの運転日数制約
0 <= y <= 5;        // 油田 Y の週あたりの運転日数制約

// 求解
solve();

// 結果出力
x.val.print();
y.val.print();
cost.val.print();
```

この SIMPLE による記述を上から順に見ていきましょう．

```
// 油田 X，Y の運転日数/週（変数）
Variable x(name = "油田 X の運転日数");
Variable y(name = "油田 Y の運転日数");
```

この部分は変数（油田の運転日数）の宣言です．モデル中で使用する変数は，使用する前に宣言す

る必要があります．name = "..."の部分には変数の名前を指定します．name = "..."は省略可能ですが，出力などで使用されますので，なるべく記述した方が良いでしょう．

"//"から行の終わりまではコメントです．

```
// 運転コスト（目的関数）
Objective cost(name = "全運転コスト", type = minimize);
```

この部分は目的関数（運転コスト）の宣言です．目的関数の内容を定義する前に，宣言する必要があります．name = "..."の部分には目的関数の名前を指定します．変数の宣言同様，name = "..."は省略可能ですが，出力などに利用されますので，なるべく記述した方が良いでしょう．type = minimize で目的関数が最小化されるべきことを指示します．type = maximize とすれば，目的関数を最大化します．

```
cost = 180 * x + 160 * y;
```

この部分は目的関数（運転コスト）の内容定義です．=の左辺に目的関数を，右辺に目的関数の内容を記述します．*は積，+は和を表す演算子です．SIMPLE では四則演算や数学関数（exp(), sin()...）などを式の記述に用いることができます．

```
// 製品ノルマ
6 * x + y >= 12;      // 重油ノルマ/週
4 * x + 6 * y >= 24; // ガスノルマ/週
```

この部分では制約式（生産ノルマ）を定義しています．関係演算子>=の左辺，右辺には，任意の式を記述できます．目的関数の内容定義の際と同様に，任意の式の中に演算子や数学関数を記述できます．左辺と右辺の関係を表す関係演算子には，以下のものを指定できます．

SIMPLE の関係演算子	定式化時の記述
>=	≥
<=	≤
==	=

```
// 各油田の日数制約
0 <= x <= 5;        // 油田 X の週あたりの運転日数制約
0 <= y <= 5;        // 油田 Y の週あたりの運転日数制約
```

この部分は制約式（運転日数の上下限）を定義しています．ここでは変数の上下限を指定していますが，SIMPLE では一般の制約式と変数の上下限制約を区別しませんので，x, y の部分に任意の式を書くことが可能です．

以上で，問題の定義の記述は完了です．

次に，これまでに定義した問題の最適解を求め，結果を出力する部分を記述します．

```
// 求解
solve();
```

solve() は，定義したモデルについて最適解の計算を行う関数です．solve() は，必ずモデル記述の後に記述する必要があります．

```
// 結果出力
x.val.print();
y.val.print();
cost.val.print();
```

この部分は，最適化計算結果の出力を指定しています．最適化計算後の値を出力するためには，最適化計算 solve() の後に記述する必要があります．

以上でこのモデルについての SIMPLE の記述は終了です．

次にこのモデルを実行してみます（実行方法については「3 数理最適化問題を解く」を参照してください）．すると，数理最適化モデルを解く経過が，以下のように出力されます．

```
[Expand Constraints and Objectives]
sample.smp:7:info: 展開中 目的関数 (1/5) name="全運転コスト"
sample.smp:10:info: 展開中 制約式  (2/5) name=""
sample.smp:11:info: 展開中 制約式  (3/5) name=""
sample.smp:14:info: 展開中 制約式  (4/5) name=""
sample.smp:15:info: 展開中 制約式  (5/5) name=""

[About Nuorium Optimizer]
Nuorium Optimizer xx.x.x (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]
PROBLEM_NAME                                        sample
NUMBER_OF_VARIABLES                                      2
NUMBER_OF_FUNCTIONS                                      3
PROBLEM_TYPE                                    MINIMIZATION
METHOD                                          HIGHER_ORDER

[Progress]
<preprocess begin>.........<preprocess end>
```

```
<iteration begin>
    res=4.0e+001 .... 2.8e-005  1.4e-007
<iteration end>


[Result]
STATUS                                              OPTIMAL
VALUE_OF_OBJECTIVE                               750.0000021
ITERATION_COUNT                                            6
FUNC_EVAL_COUNT                                            9
FACTORIZATION_COUNT                                        7
RESIDUAL                                   1.402395924e-007
ELAPSED_TIME(sec.)                                    0.20
```

最後に結果出力に対応する結果が以下のように出力されます.

```
油田 X の運転日数=1.5
油田 Y の運転日数=3
全運転コスト=750
```

=の左辺は指定した変数と目的関数の名前で，name = "..."に記述したものが出力されます．右辺には変数と目的関数の値が出力されています.

2.2 定数

現在は，モデル中に油田運転コストの値を直接記述しています．これを変更し，外部から任意の値を与えてみましょう．まず，定式化を以下のように変更します.

目的関数	
costX · x + costY · y	運転コスト/週

定数	
costX	油田 X の運転コスト/日
costY	油田 Y の運転コスト/日

costX, costY はそれぞれ油田 X, Y の運転コスト/日を表す定数です．SIMPLE では，このような定数を使用した記述が可能です.

ここでは，定数を用いて，運転コストを以下のように変更します.

```
cost = 180 * x + 160 * y;
```

↓

```
Parameter costX(name="油田 X の運転コスト");
Parameter costY(name="油田 Y の運転コスト");
cost = costX * x + costY * y;
```

まず，Parameter で定数を宣言します．モデル中で使用する定数は，使用する前に宣言する必要があります．定数の値は，モデル中で定義せず外部からデータファイルで与えます．変数，目的関数の宣言と同様に，name = "..."には，定数名を指定します．定数名は，データファイル中のデータとの対応付けに使用されます．

次にモデルに定数を与えるために，以下のデータファイルを作成します．以下のデータファイルの拡張子は.dat となります．

```
"油田 X の運転コスト" = 180;
"油田 Y の運転コスト" = 160;
```

=の左辺には，宣言時の name = "..."で与えたパラメータ名を記述します．右辺には，定数値を記述します．セミコロン; が定数データの区切りになります．データファイル中の"..."内にないスペース，改行，タブは無視されます．

では，上記データファイルを入力として，実行してみます（実行方法については「3 数理最適化問題を解く」を参照してください）．

最適化経過の出力の後，次のような実行結果が得られます．

```
油田 X の運転日数=1.5
油田 Y の運転日数=3
全運転コスト=750
```

前回と同じ結果が得られています．Parameter とデータファイルを使用することで，データファイルの変更のみで違う問題を解くことができます．

では，データファイルを変更して実行してみましょう．以下のようにデータファイルを変更します．

```
"油田 X の運転コスト" = 100;
"油田 Y の運転コスト" = 170;
```

実行すると，以下の結果が得られます．

```
油田 X の運転日数=5
油田 Y の運転日数=0.666667
全運転コスト=613.333
```

2.3 集合・添字

実は，ここまでのモデルでは，次のように各油田について同じ日数制約を定義しているので，冗長な記述になっていると言えます．

$0 \le x \le 5$	油田 X の週あたりの運転日数制約
$0 \le y \le 5$	油田 Y の週あたりの運転日数制約

そこで油田運転日数を一般的に記述することを考えてみましょう．まず油田運転日数 x, y をそれぞれ x_0, x_1 と変更し，定式化を次のように変更します．

集合	
$OilField = \{0, 1\}$	油田集合
変数	
$x_i, i \in OilField$	油田 i の運転日数/週
定数	
costX	油田 0 の運転コスト/日
costY	油田 1 の運転コスト/日
目的関数（最小化）	
$\text{costX} \cdot x_0 + \text{costY} \cdot x_1$	運転コスト/週
制約条件	
$6x_0 + x_1 \ge 12$	重油ノルマ/週
$4x_0 + 6x_1 \ge 24$	ガスノルマ/週
$0 \le x_i \le 5, \forall i \in OilField$	油田 i の週あたりの運転日数制約

運転日数の制約を一行で書き表すことができました．

対応する SIMPLE の記述は，次のようになります．

```
// 油田集合と添字の定義
Set OilField(name = "油田集合");
OilField = "0 1";
Element i(set = OilField);

// 油田 i の運転日数/週
Variable x(name = "油田の運転日数", index = i);
```

```
// 油田運転コスト/日
Parameter costX(name = "油田 X の運転コスト");
Parameter costY(name = "油田 Y の運転コスト");

// 運転コスト/週 (目的関数)
Objective cost(name = "全運転コスト", type = minimize);
cost = costX * x[0] + costY * x[1];

// 製品ノルマ
6 * x[0] + x[1] >= 12;      // 重油ノルマ
4 * x[0] + 6 * x[1] >= 24; // ガスノルマ

// 油田 i の週あたりの日数制約
0 <= x[i] <= 5;

// 求解
solve();

// 結果出力
x[i].val.print();
cost.val.print();
```

定式化と同様，日数制約を一行で書き表しています．

それでは，SIMPLE の記述の変更･追加点について，上から順に見ていきます．

```
Set OilField(name = "油田集合");
```

ここでは集合（油田の集合）を宣言しています．SIMPLE で添字を使用する場合は，まず添字の属する集合を宣言する必要があります．変数，目的関数，定数と同様に，name = "..."の部分には集合名を指定します．name = "..."は省略可能ですが，内容を出力する際などで使用されますので，記述したほうが良いでしょう．

```
OilField = "0 1";
```

ここでは油田集合の内容を定義しています．先の定式化の添字範囲が{0, 1}なので，0, 1 を集合の要素とします．

```
Element i(set = OilField);
```

ここでは集合 OilField の要素を表す添字 i を宣言しています．set = ... で添字が属する集合を定義します．

```
Variable x(name = "油田の運転日数", index = i);
```

ここでは油田の運転日数を，添字付き変数として宣言しています．index = i で添字を指定します．

```
cost = costX * x[0] + costY * x[1];
```

ここでは運転コストの内容定義をしています（制約式と内容定義は異なる点に注意してください）．添字付けは，x[添字] と記述します．

```
// 製品ノルマ
6 * x[0] + x[1] >= 12;
4 * x[0] + 6 * x[1] >= 24;
```

ここでは製品ノルマの制約を記述しています．以前に x, y と書いた変数部分を x[0], x[1] と置き換えただけです．

```
// 日数制約
0 <= x[i] <= 5;
```

ここでは日数制約を記述します．添字に i と指定することで，全ての $i \in OilField$ に関する日数制約を，かけたことになります．

```
// 結果出力
x[i].val.print();
```

結果出力も上記日数制約と同様に，添字に i と指定することで，全ての $i \in OilField$ について x[i] の値が出力されます．

次に実行してみます（実行方法については「3 数理最適化問題を解く」を参照してください）．データファイルは「2.2 定数」で使用した変更前のデータを使用してください．最適化経過が出力されたあと，x[i].val.print() に対応した，以下の出力が得られます．

```
油田の運転日数[0]=1.5
油田の運転日数[1]=3
```

変数名が添字つきで出力されているのが確認できます．

ここまでの記述の変更で，油田集合 OilField を導入し，各油田の運転日数を x[i] と簡略化することができました．次に，油田運転コスト costX, costY も添字 i を用いて簡略化してみます．運転コストを添字付けし，以下のように表すことにします．

定数	
$costX_i, i \in OilField$	油田 i の運転コスト/日

$costX_0$, $costX_1$ はそれぞれ以前の costX, costY に対応する定数です．SIMPLE でも同様に定数の添字付

けを用いて，以下のように修正します．

```
Parameter costX(name = "油田 X の運転コスト");
Parameter costY(name = "油田 Y の運転コスト");
cost = costX * x[0] + costY * x[1];
```

↓

```
Parameter costX(name = "油田運転コスト", index = i);
cost = costX[0] * x[0] + costX[1] * x[1];
```

定数の添字付けは，変数の添字付けと同様に index = i と指定します．上記変更に合わせて，データファイルの内容を以下のように修正します．

```
"油田運転コスト" = [0] 180 [1] 160;
```

添字付きの定数値を指定する右辺は，

[添字] 値 [添字] 値 ...

と記述します．

では，実行してみましょう（実行方法については「3 数理最適化問題を解く」を参照ください）．最適化経過が出力された後，以下のように以前と同様の結果が得られます．

```
油田の運転日数 [0]=1.5
油田の運転日数 [1]=3
全運転コスト=750
```

ここで，油田集合とその要素について考えます．上記のデータファイル中には，運転コストの添字として 0, 1 が記述されています．そして SIMPLE の記述中で，運転コストの添字は油田集合の要素であると明示しています．以上より，SIMPLE はこのような油田集合の要素は 0, 1 からなると推定することができますので，実は，以下の油田集合の具体的な要素を与える記述は省略することができます．

```
OilField = "0 1";
```

この記述を削除して実行してみますと，前回と同様の結果が得られるのが確認できます．

このように SIMPLE では，添字と集合の関係から集合の内容を自動的に推定する機能があります．この機能を利用すれば集合の要素を SIMPLE で陽に記述する必要がなくなります．これにより，汎用的なモデル記述が可能となりますので，是非御活用ください．

次に，重油とガスの生産ノルマの値を外部から与えることを考えます．定式化において製品集合を導入して製品ノルマを以下のように記述します．

集合	
Product = {重油, ガス}	製品集合

定数

$norma_j$, $j \in Product$	製品 j のノルマ/週

SIMPLE の記述においても同様に定数の添字付けを用いて表現し，ノルマに関する制約式を以下のように変更します．

```
6 * x[0] + x[1] >= 12;
4 * x[0] + 6 * x[1] >= 24;
```

↓

```
Set Product(name = "製品集合");
Element j(set = Product);
Parameter norma(name = "製品ノルマ", index = j);
6 * x[0] + x[1] >= norma["重油"];
4 * x[0] + 6 * x[1] >= norma["ガス"];
```

新たに製品集合の宣言を追加し，ノルマを製品を表す添字 j 付きの定数にします．上記のように文字列を添字に使用する場合は，文字列を"..."の中に記述する必要があります．次に，データファイルにノルマを与えるデータを追加しましょう．データファイルは以下のようになります．

```
"油田運転コスト" = [0] 180 [1] 160;
"製品ノルマ" = ["重油"] 12 ["ガス"] 24;
```

SIMPLE の記述中で，製品ノルマの添字は製品集合の要素であると明示しています．このことから，SIMPLE は製品集合の要素は"重油"，"ガス"であると推定することができます．ゆえに，SIMPLE の記述中に製品集合の要素を書く必要はありません．このことは，油田集合の要素の推定と同様です．実行させると以前と同様の結果が得られます．

ここまでの変更をまとめて，集合，変数，定数，制約条件，目的関数を分類し整理すると，定式化と SIMPLE の記述は次のようになります．

集合

$OilField = \{0, 1\}$	油田集合
$Product = \{$重油, ガス$\}$	製品集合

定数

$\mathrm{costX_i}$, $i \in OilField$	油田 i の運転コスト/日
$norma_j$, $j \in Product$	製品 j のノルマ/週

変数

x_i, $i \in OilField$	油田 i の運転日数/週

目的関数（最小化）

$\mathrm{costX}_0 \cdot x_0 + \mathrm{costX}_1 \cdot x_1$	運転コスト/週

制約条件

$6x_0 + x_1 \geq norma_{重油}$	重油ノルマ/週
$4x_0 + 6x_1 \geq norma_{ガス}$	ガスノルマ/週
$0 \leq x_i \leq 5, \forall i \in OilField$	油田 i の週あたりの運転日数制約

```
// 油田集合
Set OilField(name = "油田集合");
Element i(set = OilField);

// 製品集合
Set Product(name = "製品集合");
Element j(set = Product);

// 油田 i の運転コスト/日
Parameter costX(name = "油田運転コスト", index = i);

// 製品 j のノルマ/週
Parameter norma(name = "製品ノルマ", index = j);

// 油田 i の運転日数/週（変数）
Variable x(name = "油田の運転日数", index = i);

// 運転コスト/週（目的関数）
Objective cost(name = "全運転コスト", type = minimize);
cost = costX[0] * x[0] + costX[1] * x[1];

// 製品ノルマ
6 * x[0] + x[1] >= norma["重油"];     // 重油ノルマ/週
4 * x[0] + 6 * x[1] >= norma["ガス"]; // ガスノルマ/週

// 油田 i の週当りの運転日数制約
0 <= x[i] <= 5;

// 求解
```

```
solve();
// 結果出力
x[i].val.print();
cost.val.print();
```

2.4 集約・複数の添字

コスト定義式

```
cost = costX[0] * x[0] + costX[1] * x[1];
```

は，すべての油田について運転コストの和をとるという意味なので，これを一般的に記述すると，以下のようになります．

$$cost = \sum_i costX_i \cdot x_i$$

対応する SIMPLE の記述は，以下のようになります．

```
cost = sum(costX[i] * x[i], i);
```

sum() は $\sum$ に対応する関数で，

sum(和をとる式，添字)

の書式を持ちます．

次にノルマ制約についても，sum() を適用したいと考えますが，旧記述では，

```
6 * x[0] + x[1] >= norma["重油"];
4 * x[0] + 6 * x[1] >= norma["ガス"];
```

と各油田の生産量が直接数値で記述されているので，一般化できません．そこで，定式化において定数 $prodX_{i,j}$ を導入し，制約式を次のように記述します．

制約条件

$\sum_{i \in OilField} prodX_{i,j} \cdot x_i \geq norma_j, \forall j \in Product$	製品 j のノルマ/週の制約式

定数

$prodX_{i,j}, i \in OilField, j \in Product$	油田 i の製品 j 生産量/日
$norma_j, j \in Product$	製品 j のノルマ/週

対応する SIMPLE の記述は，以下のようになります．

```
Parameter prodX(name = "油田の生産量", index=(i, j));
sum(prodX[i, j] * x[i], i) >= norma[j];
```

複数の添字に依存する定数を宣言する際には，index = (i, j, ..) と指定します．上記 sum() は指定した添字 i のみの和をとります．i, j について和をとる場合は，sum(任意の式，(i, j))，と記述します．

次に油田の生産量の値を追加した以下のデータファイルを作成します．

```
"油田運転コスト" = [0] 180 [1] 160;
"製品ノルマ" = ["重油"] 12 ["ガス"] 24;
"油田の生産量" =
[0, "重油"] 6  [1, "重油"] 1
[0, "ガス"] 4  [1, "ガス"] 6
;
```

データファイル中の””に囲まれていない，スペース，タブ，改行は無視されます．従って，上記の“油田の生産量”のように，値を複数の行にわたって記述することができます．以上で，変更可能性のある全ての数値データをデータファイルから入力することができました．実行結果は以前と同様になります．

ここまでの変更をまとめて，集合，変数，定数，制約条件，目的関数を分類し整理すると，定式化と SIMPLE の記述は次のようになります．

集合	
$OilField = \{0, 1\}$	油田集合
$Product = \{$重油, ガス$\}$	製品集合

定数	
$\mathrm{costX}_i,\ i \in OilField$	油田 i の運転コスト/日
$norma_j,\ j \in Product$	製品 j のノルマ/週
$prodX_{i,j},\ i \in OilField,\ j \in Product$	油田 i の製品 j 生産量/日

変数	
$x_i,\ i \in OilField$	油田 i の運転日数/週

目的関数（最小化）	
$\sum_{i \in OilField} \mathrm{costX}_i \cdot x_i$	運転コスト/週

制約条件

$\sum_{i \in OilField} prodX_{i,j} \cdot x_i \geq norma_j, \forall j \in Product$	製品 j のノルマ/週の制約式
$0 \leq x_i \leq 5, \forall i \in OilField$	油田 i の週あたりの運転日数制約

```
// 油田集合
Set OilField(name = "油田集合");
Element i(set = OilField);

// 製品集合
Set Product(name = "製品集合");
Element j(set = Product);

// 油田 i の運転コスト/日
Parameter costX(name = "油田運転コスト", index = i);

// 製品 j のノルマ/週
Parameter norma(name = "製品ノルマ", index = j);

// 油田 i の製品 j 生産量/日
Parameter prodX(name = "油田の生産量", index = (i, j));

// 油田 i の運転日数/週（変数）
Variable x(name="油田の運転日数", index = i);

// 運転コスト/週（目的関数）
Objective cost(name = "全運転コスト", type = minimize);
cost = sum(costX[i] * x[i], i);

// 製品 j のノルマ/週の制約式
sum(prodX[i, j] * x[i], i) >= norma[j];

// 油田 i の週当りの運転日数制約
0 <= x[i] <= 5;

// 求解
solve();
```

```
// 結果出力
x[i].val.print();
cost.val.print();
```

2.5 式

ここでは，これまでの結果出力（油田運転日数/週, 全運転コスト）に加えて，各製品の生産量/週も出力してみます.

生産量/週は一般的に以下のように記述できます.

$$prod_j = \sum_{i \in OilField} prodX_{i,j} \cdot x_i, \forall j \in Product \quad \text{製品 j の生産量/週}$$

この式に対応する SIMPLE の記述は，以下のようになります.

```
Expression prod(name = "製品の生産量", index = j); // 式の宣言
prod[j] = sum(prodX[i, j] * x[i], i); // 式の定義
```

まず，Expression で式を宣言します．name, index の指定は，変数宣言時（Variable）と同様に，name で名前を指定し，index で添字を指定します．prod[j] = ... で式の内容を定義します．Expression は，任意の変数を含む式に名前を付けるためのもので，数理最適化問題の変数の数が増加することはありません.

次に生産ノルマの記述を見てみます.

```
sum(prodX[i, j] * x[i], i) >= norma[j];
```

左辺は先ほど定義した prod[j] と全く同じ内容ですので，以下のように左辺を prod[j] に置き換えることができます.

```
prod[j] >= norma[j];
```

次に結果出力部分に以下のように prod[j] を追加します.

```
prod[j].val.print();
```

これで，製品の生産量/週が出力されるようになりました．生産量の出力結果は，以下のようになります.

```
製品の生産量 [ガス]=24
製品の生産量 [重油]=12
```

2.6 整数変数

ここまでは，運転日数を連続変数とみなして解いてきました．しかし実際には油田は 1 日単位でしか運転できません．そこで，運転日数を 1 日単位の整数変数とした，整数計画問題を解くことを考えます．そのために，変数（運転日数）の宣言を以下のように変更します．

```
Variable x(name = "油田の運転日数", index = i);
```

↓

```
IntegerVariable x(name = "油田の運転日数", index = i);
```

IntegerVariable で整数変数を宣言します．整数変数として宣言された変数は，値として整数のみを取ります．以上で変更完了です．

実行すると，以下の結果が得られます．

```
（solve() の最適化経過出力）
製品の生産量 [重油]=15
製品の生産量 [ガス]=26
油田の運転日数 [0]=2
油田の運転日数 [1]=3
全運転コスト=840
...
```

運転日数が整数になっているのが確認できます．このように変数を IntegerVariable で宣言するだけで，整数計画問題を記述することができます．

2.7 結果出力関数

ここまでは，結果の出力には print() を使用してきましたが，SIMPLE は他にも書式指定出力関数 simple_printf() があります．

以下，simple_printf() の機能を簡単に紹介します．なお，説明中に C++言語の機能にふれる記述があります．C++言語については，C++言語の参考書等を参照してください．

simple_printf() は書式を細かく指定できる出力関数です．

結果の確認程度の用途ならば print() で十分ですが，出力書式を細かく指定したい場合には simple_printf() を使用すると便利です．ここでは，運転日数の出力部を以下のように変更してみます．

```
x[i].val.print();
```

↓

```
simple_printf("油田 %d の最適運転日数 = %d\n", i, x[i]);
```

対応する実行結果出力は以下のようになります.

```
油田 0 の最適運転日数 = 2
油田 1 の最適運転日数 = 3
```

関数 simple_printf() の書式指定は,

simple_printf(出力書式指定，出力対象 1，出力対象 2，...)

となります.

出力対象には，変数，式，定数，目的関数，添字，など集合以外の任意のものを任意の個数だけ指定できます．出力書式指定の指定方法は，C++言語の標準関数 printf() の書式指定と同様のものが指定できます.

2.8 デバッグ出力関数

数理最適化モデルが複雑になるほど，些細な記述ミスでも発見が困難になっていきます．そのようなミスを修正するための支援関数として showSystem() があります．showSystem() は，目的関数・制約式を実際のモデル内容に展開して出力します.

以下のように，showSystem() を最適化計算 solve() の直前に挿入してみます.

```
showSystem();
solve();
```

上記の位置に記述すれば，最適化計算を行うモデルの内容が出力できます．これを実行すると，showSystem() に対応した出力が以下のように得られます.

```
1-1 (sample.smp:26): 6*油田の運転日数 [0]+油田の運転日数 [1] >= 12
1-2 (sample.smp:26): 4*油田の運転日数 [0]+6*油田の運転日数 [1] >= 24
2-1 (sample.smp:29): 0 <= 油田の運転日数 [0] <= 5
2-2 (sample.smp:29): 0 <= 油田の運転日数 [1] <= 5
objective (sample.smp:23 name="全運転コスト"): 180*油田の運転日数 [0]+160*油田の運転日数
[1] (minimize)
```

1-1, 1-2 は次のノルマ制約式に対応しています.

```
prod[j] >= norma[j];
```

2-1, 2-2 は次の日数制約式に対応しています.

```
0 <= x[i] <= 5;
```

objective は，次のコスト定義式に対応しています.

```
cost = sum(costX[i] * x[i], i);
```

このように，showSystem() を使用することによって，定数値，添字等を実際の値に置き換えた後の目的関数・制約式を確認することができます．この機能を利用すれば，意図しない記述ミスを簡単に発見することができ，効率の良いモデル記述が可能になります．

2.9 SIMPLE を記述する際の注意点

モデリング言語 SIMPLE を用いる際に頻出する注意点を列挙します．

- 大文字と小文字は区別される
- 積演算子を省略してはならない
- 半角スペースは自由に入れてよい（等号/不等号の間は駄目）
- 改行は自由に入れてよい
- 文末には必ず半角セミコロン; を入れる
- //はコメントを意味する
- name =引数はダブルクォート"で囲む
- minimize や maximize はダブルクォート"で囲まない
- name と type のように複数の設定を行うときはカンマで区切る
- name や type を設定する順番は変えて良い
- 等式付不等号<=と>=は使用できるが，等式なし不等号<と>は使用できない
- =は代入，==は等価を表わす

第3章 数理最適化問題を解く

Windows 版／ UNIX・Linux 版に関わらず，Nuorium Optimizer/SIMPLE を用いて最適化計算を行うには，次の手順が必要となります．

1. SIMPLE モデルを記述する
2. データを作成する
3. 最適化計算を行う

以下，Windows 版／ UNIX・Linux 版別に，上記の項目について，最適化計算の一連の流れを解説します．なお，詳細については「Nuorium Optimizer/SIMPLE マニュアル」・「Nuorium Optimizer マニュアル」または「Nuorium スタートガイド」（Windows 版のみ）をご覧ください．

3.1 Windows 版

Windows 版 Nuorium Optimizer/SIMPLE を用いて最適化計算をするためには，Nuorium を用いる方法またはコマンドプロンプトを用いる方法の二通りの方法があります．ここでは，これらについて順番に紹介します．

3.1.1 Nuorium を用いる方法

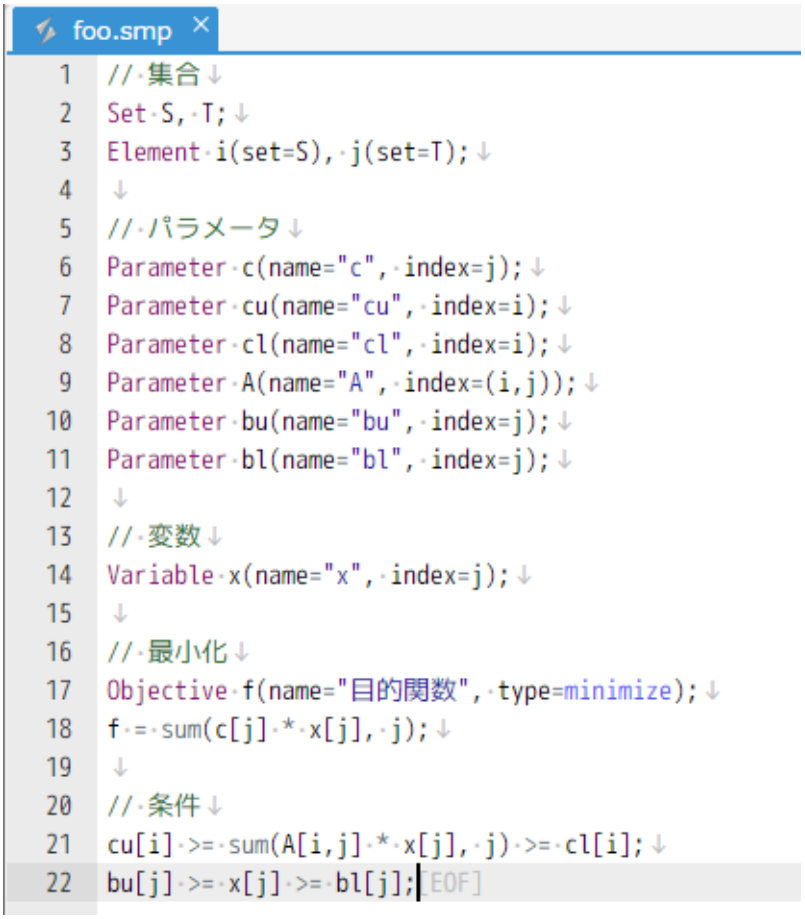
1. SIMPLE モデルを記述する

   Windows のスタートメニューから「すべてのプログラム」→「MSI Solutions」→「Nuorium」を選択し，Nuorium を起動します．表示された画面の左のパネルに以下のようなモデルを記述し，メニューの「ファイル」→「名前を付けて保存」で適当な場所にファイル名 `foo.smp` としてファイルを保存します．

   foo.smp

```
// 集合
Set S, T;
Element i(set = S), j(set = T);

// パラメータ
Parameter c(name = "c", index = j);
Parameter cu(name = "cu", index = i);
Parameter cl(name = "cl", index = i);
Parameter A(name = "A", index = (i, j));
```

```
Parameter bu(name = "bu", index = j);
Parameter bl(name = "bl", index = j);

// 変数
Variable x(name = "x", index = j);

// 最小化
Objective f(name = "目的関数", type = minimize);
f = sum(c[j] * x[j], j);

// 条件
cu[i] >= sum(A[i, j] * x[j], j) >= cl[i];
bu[j] >= x[j] >= bl[j];
```

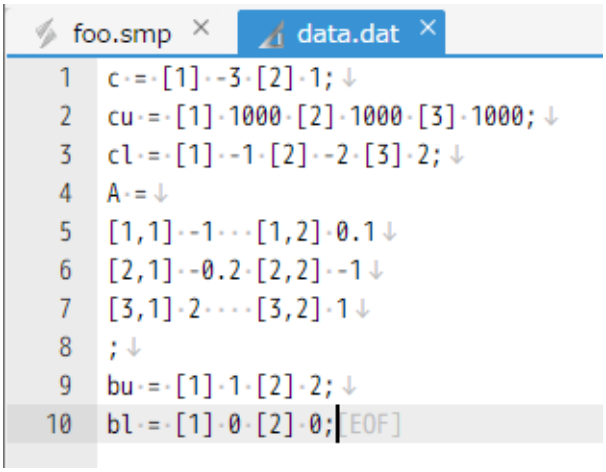


2. データを作成する

モデルに与えるデータを作成します．メニューの「ファイル」→「新規作成」→「dat」を選択し、左のパネルに以下のようなデータを記述し，メニューの「ファイル」→「名前を付けて保存」で適当な場所にファイル名 data.dat としてファイルを保存します．

data.dat

```
c = [1] -3 [2] 1;
cu = [1] 1000 [2] 1000 [3] 1000;
cl = [1] -1 [2] -2 [3] 2;
A =
[1, 1] -1   [1, 2] 0.1
[2, 1] -0.2 [2, 2] -1
[3, 1] 2    [3, 2] 1
;
bu = [1] 1 [2] 2;
bl = [1] 0 [2] 0;
```



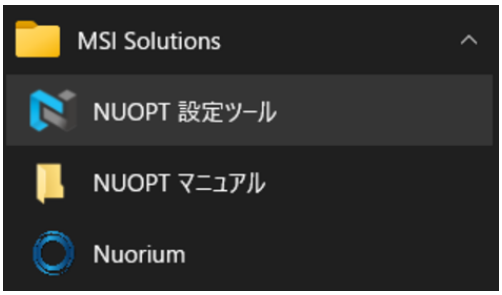
3. 最適化計算を行う

   最後に，`foo.smp` タブをアクティブにして画面左上の「実行」ボタンを押すと Nuorium Optimizer が計算を開始し，画面右のパネルに各種出力が表示されます．計算時に入力データを読み込ませる場合はそのデータファイルを Nuorium 上で開いておく必要があります．一方，不要なデータファイルは閉じてください．開いたままの場合，そのデータファイルも読み込みの対象となります．

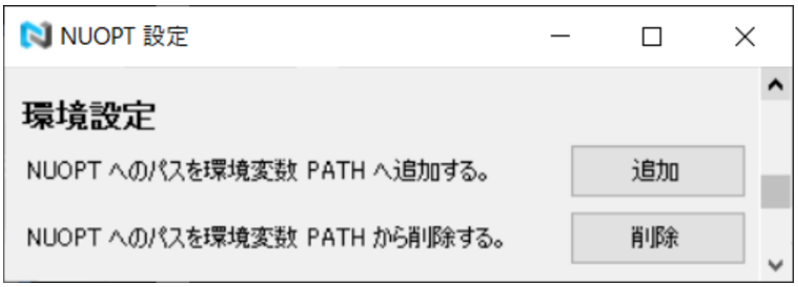
3.1.2 コマンドプロンプトを用いる方法

1. 初期設定

   Nuorium Optimizer のコマンドラインでの初回利用時は Windows のスタートメニューから「すべてのアプリ」→「MSI Solutions」から選択できる「NUOPT 設定ツール」を実行してください．



表示された画面内の「環境設定」欄にある「追加」ボタンをクリックしてください．



次に Windows の再起動をしてください．これで，コマンドプロンプトにて最適化計算実行モジュールを作成できるようになります．なお，この操作は Nuorium Optimizer をコマンドラインで初めて利用する際にのみ必要で，それ以降に再び設定する必要はありません．

2. SIMPLE モデルを作成する
3. データを作成する

   操作 2 と 3 については「3.1.1 Nuorium を用いる方法」と同様の作業を行ってください．ファイルを作成するためのエディタは Nuorium を使う必要はありません．
4. 最適化計算を行う

   それでは，準備したモデル記述ファイル `foo.smp`，データファイル `data.dat` を使用した場合の最適化計算を説明します．

   コマンドプロンプトを立ち上げ，次のように実行することで最適化計算実行モジュール `foo.exe` が作成されます．

```
> mknuopt foo.smp
```

   次に最適化計算を実行します．以下のように入力します．

```
> foo.exe data.dat
```

   実行経過・実行結果がコマンドプロンプトに出力されます．

3.2 UNIX・Linux 版

1. SIMPLE モデルを作成する

   適当なテキストエディタを用いて，SIMPLE でモデル記述し，拡張子が `.smp` となる適当なファイル名でセーブします．ここでは，以下のようなモデルを記述し，ファイル名 `foo.smp` にセーブし

ます.

foo.smp

```
// 集合
Set S, T;
Element i(set = S), j(set = T);

// パラメータ
Parameter c(name = "c", index = j);
Parameter cu(name = "cu", index = i);
Parameter cl(name = "cl", index = i);
Parameter A(name = "A", index = (i, j));
Parameter bu(name = "bu", index = j);
Parameter bl(name = "bl", index = j);

// 変数
Variable x(name = "x", index = j);

// 最小化
Objective f(name = "目的関数", type = minimize);
f = sum(c[j] * x[j], j);

// 条件
cu[i] >= sum(A[i, j] * x[j], j) >= cl[i];
bu[j] >= x[j] >= bl[j];
```

2. データを作成する

モデルに与えるデータファイルを作成します．データファイルの拡張子は，.dat とします．ここでは，以下のデータファイル data.dat を作成しました.

data.dat

```
c = [1] -3 [2] 1;
cu = [1] 1000 [2] 1000 [3] 1000;
cl = [1] -1 [2] -2 [3] 2;
A =
[1,1] -1   [1,2] 0.1
[2,1] -0.2 [2,2] -1
[3,1] 2    [3,2] 1
;
```

```
bu = [1] 1 [2] 2;
bl = [1] 0 [2] 0;
```

3. 最適化計算を行う

それでは，準備したモデル記述ファイル foo.smp，データファイル data.dat を使用した場合の最適化計算を説明します．まず最適化計算実行モジュール foo を作成します．シェル上で以下のように入力します．

```
prompt% mknuopt foo.smp
```

次に最適化計算を実行します．以下のように入力します．

```
prompt% ./foo data.dat
```

そうしますと，実行経過と実行結果が以下のように出力されます．

```
[List of Data Files]
<reading data_file: data.dat>

[Expand Constraints and Objectives]
foo.smp:18:info: 展開中 目的関数 (1/3) name="目的関数"
foo.smp:21:info: 展開中 制約式   (2/3) name=""
foo.smp:22:info: 展開中 制約式   (3/3) name=""

[About Nuorium Optimizer]
Nuorium Optimizer xx.x.x (NLP/LP/IP/SDP module)
         <with Netlib BLAS>
         , Copyright (C) 1991 NTT DATA Mathematical Systems Inc.

[Problem and Algorithm]
PROBLEM_NAME                                          foo
NUMBER_OF_VARIABLES                                     2
NUMBER_OF_FUNCTIONS                                     4
PROBLEM_TYPE                                  MINIMIZATION
METHOD                                        HIGHER_ORDER

[Progress]
<preprocess begin>.........<preprocess end>
<iteration begin>
    res=1.4e+05 .... 4.3e+02 .... 3.8e-04 . 1.3e-08
<iteration end>
```

```
[Result]
STATUS                                          OPTIMAL
VALUE_OF_OBJECTIVE                                   -3
ITERATION_COUNT                                      12
FUNC_EVAL_COUNT                                      15
FACTORIZATION_COUNT                                  13
RESIDUAL                                1.341986491e-08
ELAPSED_TIME(sec.)                                 0.03
SOLUTION_FILE                                   foo.sol
```

索　引

そ

て

へ

も