

NUOPT/SIMPLE マニュアル

株式会社数理システム

Phone: 03-3358-1701

Fax: 03-3358-1727

Email: nuopt-support@msi.co.jp

2006/07/12

第一部 モデリング言語 SIMPLE

1. はじめに.....	8
2. 機能の概要.....	10
2.1 線形計画問題.....	10
2.2 非線形計画問題.....	13
2.3 整数計画問題(添え字を用いたナップサック問題の記述).....	14
2.4 ネットワーク最適化問題(最小費用流).....	16
2.5 離散最適化(地図塗りわけ)問題.....	18
2.6 資源制約付きスケジューリング問題.....	20
2.7 ソルバ(NUOPT)との連結.....	21
3. 数理モデルを表現するクラス.....	22
3.1 クラスの一覧表.....	22
3.2 SIMPLE の仕組み.....	24
3.3 式を表現するクラス(VARIABLE/EXPRESSION/PARAMETER..).....	26
3.3.1 宣言と引数の意味.....	27
3.3.2 Expression(式)の利用.....	30
3.3.3 式の添字付け([]演算).....	31
3.3.4 範囲にわたる演算(sum, prod, ..).....	33
3.3.5 整数変数(IntegerVariable).....	36
3.3.6 離散変数(DiscreteVariable).....	37
3.3.7 制約式(Constraint).....	40
3.3.8 条件式.....	41
3.3.9 条件分岐関数(関数:ifelse(), Boolean()).....	43
3.3.10 定数値のチェック(関数:check()).....	48
3.3.11 添字の表現(Element など).....	48
3.3.12 スケジューリング問題定義の為に(Activity,ResourceRequire/Capacity).....	52
3.4 データ構造を表現するクラス(SET/GRAPH).....	54
3.4.1 宣言と引数の意味.....	55
3.4.2 集合(Set)同士の演算.....	58
3.4.3 集合(Set)の添字付け.....	59
3.4.4 添字(Element)と集合(Set)の対応.....	60
3.4.5 添字付き集合の利用.....	61
3.4.6 添字集合への自動代入.....	62
3.4.7 自動代入の禁止(lock()/unlock()).....	64
3.4.8 集合と要素から生成される条件式.....	65

3.4.9 集合同士の包含関係.....	66
3.4.10 Set の操作(要素数を知る, 要素の追加する, など).....	68
3.4.11 イテラタの表現(順序付き集合のメンバ関数).....	70
3.4.12 要素からの集合の生成(関数: setOf()).....	70
3.4.13 数列(Sequence).....	71
3.4.14 連続範囲(Interval).....	72
3.4.15 グラフ(Graph).....	74
4. データファイル.....	75
4.1 SIMPLE 形式データファイル.....	75
4.1.1 数値データ.....	77
4.1.2 集合の要素データ.....	78
4.1.3 Wild Card を使ったデータの記述.....	79
4.1.4 データファイルを使った NUOPT のパラメータの定義.....	81
4.2 CSV 形式のデータファイル.....	81
4.2.1 1D 書式.....	82
4.2.2 2D 書式.....	85
4.2.3 曖昧さについて.....	87
4.3 データ名の重複について.....	88
5. システム制御.....	89
5.1 モデルの内容の表示(SHOWSYSTEM()関数).....	90
5.2 ソルバの起動 SOLVE() 関数.....	92
5.3 オブジェクトの参照値の表示(関数 PRINT,COUT,SIMPLE_PRINTF).....	94
5.3.1 print 関数.....	95
5.3.2 << 関数.....	96
5.3.3 simple_printf 関数.....	98
5.4 オブジェクトの様々な参照値(上下限や双対変数などの参照).....	101
5.5 参照値を PARAMETER として扱う.....	103
5.6 参照値を C++ の配列に出力する.....	104
5.7 C++ の配列からオブジェクトに対する値の設定.....	109
5.8 特定の制約式の削除/復帰 DELETECO()/RESTORECO() 関数.....	111
5.9 モデル定義とシステム制御を分離して記述する.....	116
5.10 変数の固定.....	118
5.11 可変定数.....	121

第二部 最適化ソルバ NUOPT

1. はじめに	123
1.1 第二部各章の内容.....	123
1.2 内部構成.....	125
1.3 MPS ファイル用ロードモジュール NUOPT の簡単な使い方.....	127
1.4 パラメータファイル NUOPT.PRM.....	131
1.5 モデリング言語 SIMPLE との連結.....	132
2. SIMPLE 版ロードモジュール	133
2.1 SIMPLE 版ロードモジュールの構成.....	134
2.2 SIMPLE 版ロードモジュールの作成.....	135
2.3 SIMPLE 版ロードモジュールの起動.....	137
2.4 SIMPLE 版ロードモジュールの出力.....	138
2.4.1 標準出力.....	138
2.4.2 SIMPLE 版ロードモジュールの解ファイル.....	139
2.5 SIMPLE における最適値の参照, 最適化制御.....	143
2.5.1 SIMPLE による最適化の結果の表示.....	143
2.5.2 SIMPLE による最適化制御.....	146
2.6 SIMPLE のデータファイルの利用.....	149
2.6.1 目的関数に関して.....	150
2.6.2 SIMPLE 版ロードモジュールの標準出力頻度の設定.....	151
2.7 整数計画問題に関するパラメータ設定.....	153
2.8 制約充足問題ソルバのための重みおよび目標値設定.....	155
2.9 SIMPLE 版ロードモジュールの動作の制御.....	158
2.9.1 SIMPLE 版ロードモジュールの最適化手法の設定.....	163
2.9.2 SIMPLE 版ロードモジュールの解ファイル名の設定.....	163
2.10 資源制約付きスケジューリング問題のための重みの設定.....	164
3. 出力ファイルの解説	166
3.1 解ファイルのヘッダ部.....	166
3.2 解ファイルの変数値表示部.....	170
3.3 解ファイルの関数値表示部.....	172
3.4 解ファイルの上下限, 制約と対応する双対変数の表示.....	174
3.5 標準出力.....	176
3.5.1 単体法, 有効制約法, クロスオーバーの場合の実行経過表示.....	178
3.5.2 制約充足問題ソルバ(wcsp)の実行経過表示.....	178

3.5.3 分枝限定法の場合の実行経過表示.....	179
3.5.4 資源制約付きスケジューリングソルバー (rcpsp)の実行経過表示.....	181
3.5.5 標準出力の抑制.....	182
3.6 実行不可能性要因検出機能 (IISDETECT)の適用例と出力.....	183
4. パラメータ設定	187
4.1 パラメータファイル NUOPT.PRM.....	187
4.2 最適化手法に関する設定.....	189
4.2.1 解法を選択.....	194
4.2.2 クロスオーバー.....	196
4.3 最適化手法のパラメータ.....	197
4.3.1 線形計画問題専用内点法(Higher Order Method) に有効なパラメータ.....	197
4.3.2 直線探索法(Line Search Method) に有効なパラメータ.....	198
4.3.3 準ニュートン法(Line Search with BFGS)に有効なパラメータ.....	200
4.3.4 信頼領域法(Trust Region Method)に有効なパラメータ.....	201
4.3.5 単体法(SimplexMethod)/有効制約法(ActiveSetMethod)に有効なパラメータ.....	203
4.3.6 逐次二次計画法(line search SQP/trust region SQP)に有効なパラメータ.....	204
4.3.7 制約充足アルゴリズム(wcsp/rcpsp)に有効なパラメータ.....	205
4.3.8 整数変数計画, 大域的最適化 (simple/asq/global)を行う際に有効なパラメータ.....	206
4.4 MPS ファイルに関する設定.....	211
4.5 その他の設定.....	212
4.6 ソルバとのインタフェース.....	213
4.7 基底ファイルとリスタート.....	219
5. MPS ファイル入力用モジュール NUOPT	221
5.1 ロードモジュール NUOPT の構成.....	221
5.2 ロードモジュール NUOPT の起動.....	222
5.3 ロードモジュール NUOPT の出力.....	223
5.3.1 標準出力.....	223
5.3.2 nuopt の解ファイル.....	224
5.4 NUOPT の動作の制御.....	227
5.4.1 nuopt の最適化アルゴリズムの設定.....	228
5.4.2 nuopt の最小化/最大化の設定.....	228
5.4.3 目的関数行/右辺ベクトル/BOUNDS データ/RANGE データの設定.....	229
5.4.4 解ファイル名の設定.....	229
5.4.5 nuopt の標準出力モードの設定.....	230
5.5 NUOPT のリスタート機能.....	231

5.6 MPS ファイル	233
5.6.1 書式	235
5.6.2 セクション	236
5.6.3 NAME セクション	237
5.6.4 ROWS セクション	237
5.6.5 COLUMNS セクション	238
5.6.6 RHS セクション	240
5.6.7 RANGES セクション	241
5.6.8 BOUNDS セクション	243
5.6.9 HESSIAN セクション	244
5.6.10 INITIAL セクション	246
5.6.11 ENDATA セクション	247
5.7 MPS ファイル出力	248
5.7.1 MPS ファイル出力に関する制限	248
5.7.2 MPS ファイル出力時のエラー	250
付録 A NUOPT/SIMPLE のエラーメッセージ	251
A.1 SIMPLE のエラーメッセージ	251
A.2 NUOPT のエラー/警告メッセージ	278
A.2.1 NUOPT のエラー/警告	279
A.2.2 パラメータのエラー	284
A.2.3 MPS ファイルのエラー	285
付録 B NUOPT アルゴリズム概説	289
B.1 内点法	289
B.1.1 問題	289
B.1.2 直線探索を利用する方法	290
B.1.3 信頼領域を利用する方法	291
B.1.4 線形計画問題専用内点法	292
B.2 単体法・有効制約法	294
B.2.1 改訂単体法	294
B.2.2 有効制約法	294
B.2.3 分枝限定法	294
B.3 逐次二次計画 (SQP) 法	296
B.3.1 準ニュートン法を用いる方法	296
B.3.2 信頼領域法を用いる方法	296
B.4 タブー・サーチによる制約充足問題解法	298

B.5 凸緩和法に基づく大域的最適化アルゴリズム	299
B.6 タブー・サーチによる資源制約スケジューリング問題解法	300
B.7 外点法	301
付録 c 参考文献	301

第一部 モデリング言語 SIMPLE

1. はじめに

SIMPLE はシステムの記述をなるべく人間に馴染みのある数学的な記述方法で簡単に行ない、実際のシミュレータやソルバなどが認識できるような表現に翻訳して所要の解析を行うことを目的とします。

モデリング言語として、SIMPLE は次のような特徴を備えています。

- ◆ 数学的關係が自然な形で記述できる。
- ◆ 機能 (behavior) 記述ができる。
- ◆ 大規模モデルを簡便に記述できる。
- ◆ モジュール化、階層化記述ができる。
- ◆ 全体の記述が平易簡便にできる。
- ◆ システムの解析法に関してユ - ザが意識する必要がない。
- ◆ 新しい解析プログラム (ソルバ) とのリンクが簡単にできる。
- ◆ システムに対する細かい調整が簡単にできる。

SIMPLE は C++ で作成されたシステムです。モデル記述の解釈には C++ の機能である、演算子関数のオーバーロード機能を使用しています。

SIMPLE の利用分野としては

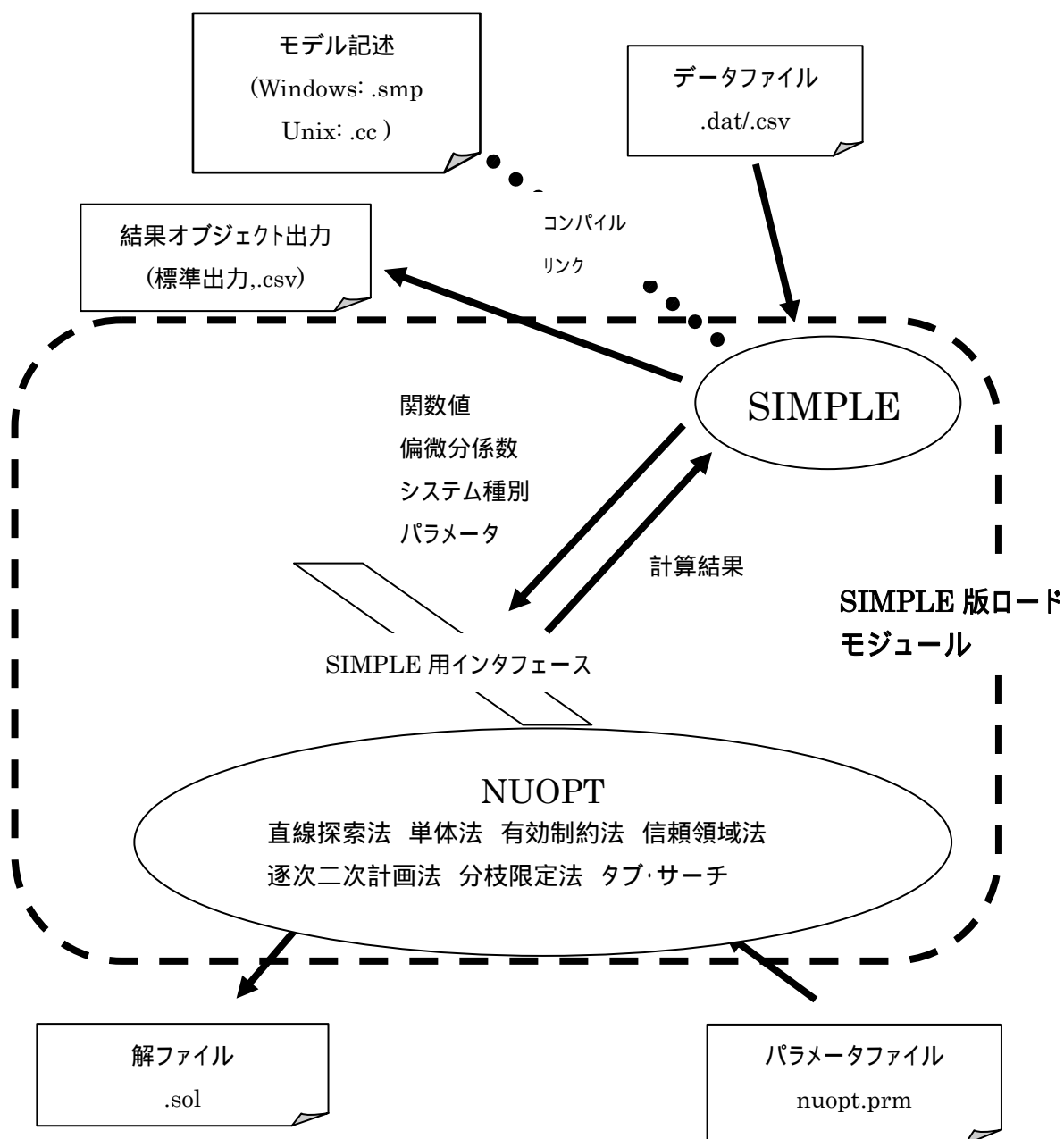
- ◆ システムの最適化
- ◆ 連続系のシミュレーション
- ◆ 離散系のシミュレーション
- ◆ 偏微分方程式の有限要素解析
- ◆ あるいはこれらの混合システム

などを想定しています。この言語を使用することによって、各種の解析が簡便に行われ、数学的手法が広く利用されることが期待されます。

SIMPLE はソルバとは独立したシステムですが、説明のため、本マニュアルではソルバを NUOPT と想定して、SIMPLE の使用方法を紹介します。

図 1 は SIMPLE と NUOPT のシステム構成を示したものです。

本マニュアルは次のように構成されます。第一部の第 2 章では、応用例をはさみながら、SIMPLE の簡単な使用方法を紹介します。第 3 章では、SIMPLE で定義されているすべてのクラスについて紹介します。第 4, 5 章ではそれぞれ、NUOPT とのインタフェース、システムの制御方法を紹介しします。



2. 機能の概要

本章では、主として例題を記述することによって本言語の基本的な機能を説明します。

2.1 線形計画問題

まず、下記のような線形計画問題の記述を例にとってみます。

これは、別冊「SIMPLE__NUOPT チュートリアル」の一章で取り上げている問題です。

変数	x, y
目的関数 (最小化)	$180x + 160y$
制約式	$6x + y \geq 12$ $4x + 6y \geq 24$
変数の上下限	$5 \geq x \geq 0$ $5 \geq y \geq 0$
(例題 1 線形計画問題)	

この問題は SIMPLE によって、上記の数学的な記述方法とほぼ同等に記述されます。

```
Variable x,y;
// 目的関数
Objective obj(name="目的関数",type=minimize);
obj = 180*x+160y;
// 制約式
6*x + y >= 12;
4*y + 6*y >= 24;
5 >= x >= 0;
5 >= y >= 0;
```

(SIMPLE 記述)

ここでは、Objective は目的関数を定義するものです。その定義の引数 type は minimize (最小化) と maximize (最大化) の 2 種類があります。

この場合のモデルの記述は変数の宣言と式の記述から成ります¹。システムの情報に関してはそれ以上の記述は必要ありません。

上の例で、演算子 \geq は等式を意味しています。この問題を解くには、問題に含まれる変数や関数の数、変数を何らかの値に設定した際の関数値や微係数の値を必要としますが、それらは本システムが C++ の演算子関数の機能と内部で生成した計算グラフ上での自動微分の手法を利用して提供します。

等式は演算子 $==$ の両側に式を置くことによって示されます。例えば、

```
x == y+z;
```

のように書くことができます。

制約式に名前を付けて参照するためには

```
Constraint g1,g2;
//制約式
g1 = 6*x + y >= 12;
g2 = 4*y + 6*y >= 24;
```

のようにします。Constraint² に対する $=$ 演算子は右辺にある等式全体を左辺のオブジェクトに代入することを意味します。従って、以後オブジェクト $g1, g2$ は代入された制約式を示すこととなります。

式の途中結果を保持する Expression というオブジェクトを定義して

```
Expression t; // 中間結果の共有などに使われる
t = 6*x + y;
t >= 12; // 6*x + y >= 12 と等価
```

とすることもできます。この例では、 t に式 $6*x+y$ が代入され、以後 t に別のものが代入されるまでは最初に代入された内容を保持します。

また、次のようにパラメータ(定数)を用いることもできます。

```
Parameter b;
b = 12;
6*x + y >= 12;
```

¹変数や目的関数に与えた `name = "文字列"` は初期値や計算結果を入出力するためのラベルとして必要なものです。

²記述対象によって、制約式 `Constraint` を方程式 `Equation` と呼ぶ時もあります。SIMPLE では `Equation` を `Constraint` の別名と見なしています。

Parameter には数字 , 文字列 , 別の Parameter などを入することができます . 例えば ,

```
Parameter b;
```

とある際には , データファイルに "b" というエントリが存在するときに , それを読み込みます . Variable 型のオブジェクトに直接にも間接にも依存しない式を定数式といいます . 定数式は変数と自由に組み合わせて式を定義するのに使用することができます .

線形計画問題の記述に Expression t, Constraint g1,g2, Parameter b を加えると , 次のようになります .

```
Variable x,y;
// 目的関数
Objective obj(name="目的関数",type=minimize);
obj = 180*x+160y;
//制約式
Constraint g1,g2;
// 式
t = 6*x + y
// 定数
Parameter b;
b = 12;

g1 = t >= b;
g2 = 4*y + 6*y >= 24;

5 >= x >= 0;
```

以上に述べた Variable, Expression, Parameter, Constraint と Objective が SIMPLE の基本的なクラスです . 大規模モデルを記述するためには , 後に述べる集合と要素の概念が必要となりますが小規模モデルには上述のクラスで十分と言えます .

2.2 非線形計画問題

SIMPLE を用いて，簡単な非線形計画問題を記述してみましょう．例えば，問題

変数	x_1, x_2
目的関数 (最小化)	$\log(1 + x_1^2) - x_2$
制約式	$(1 + x_1^2)^2 + x_2^2 = 4$
初期値	$x_1 = 2, x_2 = 2$

(Hock & Schittkowski No.7)

(例題 2 線形計画問題)

は

```
Variable x1,x2;
// 目的関数
Objective obj(name="目的関数",type=minimize);
// 式(共通して現れる部分)
Expression t;
t = 1+pow(x1,2);
obj = log(t) - x2;
// 制約式
pow(t,2) + pow(x2,2) == 4;
// 初期値
x1 =2;   x2 = 2;
```

(SIMPLE 記述)

のようになります．初期値の設定のためには変数に代入します．問題が非線形になっても用いる演算が異なるのみで，用いるクラスは変化しないことがわかります．

2.3 整数計画問題（添え字を用いたナップザック問題の記述）

整数計画問題にするためには、変数 (Variable) を整数変数 (IntegerVariable) に変更するのみで可能ですが、ここでは、添え字を用いたナップザック問題

変数	
目的関数（最大化）	$\sum_{i \in S} C_i x_i$
制約条件	$\sum_{i \in S} a_i x_i \leq b, x_i \in \{0,1\}$
（例題 3 ナップザック問題）	

のモデル定義を簡便に行うための解説をあわせて行います。

これまでに紹介した機能のみでは大規模モデルに対応しることができません。大規模モデルの特徴として、同じ種類のものの繰り返しパターンがあげられます（そもそも、このような性質のない大規模モデルの取り扱いが絶望的です）。そこで、SIMPLE では「集合とその要素」という概念を導入してその繰り返しを記述します。集合 S とその要素 i の定義は

```
Set S;
Element i (set = S);
```

$x_i (i \in S)$

によってなされます。集合は互いに異なる要素からなる集まりと考えます。通常、 S の要素の実際の値（文字列あるいは数）は、データファイルから読み込む、またはプログラムの中で以下のような代入操作によって定義します。

```
Set Cities = "Tokyo Osaka Kyoto";
Set Years;
Years = "1990 1991 1992 1993";
```

次の方程式

$$x_i + y_i = 3.0, \quad i \in S$$

は

```
Set S;
Element i(set = S);
Variable x(index = S), y(index = S);
x[i] + y[i] == 3.0;
```

と記述されます。Variable x と y の宣言は、それらが S の要素を添字（インデックスとして持つことを示します。また、上の x と y の定義において

```
Variable x(index = i), y(index = i);
```

のように Set の要素を使用して等価な宣言もできます。一般にインデックスを含む数学的式を表現するときの書き方になるべく沿った記述の方法を可能にしたいというのがこの言語の目的の一つです。要素を含む式はその要素の値が特に指定されていなければすべての要素にわたって成立しているものと解釈されます。したがって、通常の場合は特別なイテラタを使用する必要はありません。

例えば、 S とは異なる集合 T に対して

$$x_i + y_i = 3.0, \quad i \in T$$

が成立することを記述するためには

```
x[i] + y[i] == 3.0, i < T;
```

あるいは

```
i < T;
x[i] + y[i] == 3.0;
```

と書きます。演算子 $<$ は \in の意味に使用されます³。上の第 1 の例では、要素 i が集合 T に属するという条件はその行にコンマ (,) で結ばれて記述された内容にのみ適用されます。第 2 の例では条件式 $i < T$ は別の行に書かれていて、その場合にはこの条件は次に i を含む別な条件式が現れるまで有効となります。コンマで区切って条件を表わす記述法が通常の数式表現に似ていることに注意して下さい。

この機能を用いるとナップサック問題の記述は次のようになります。このモデルの中では、 S は

³ $i \in T$ は $i < T$ と記述されます。

添字の集合です．この問題は SIMPLE で次のように記述されます．

```
Set S;
Element i(set=S);
Parameter c(name="係数 c",index=i);
Parameter a(name="係数 a",index=i);
Parameter b(name="係数 b");

// 変数
IntegerVariable x(name="変数 x",index=i,type=binary);

// 目的関数
Objective obj(name="目的関数",type=maximize);
obj = sum(c[i]*x[i],i);

// 制約条件
sum(a[i]*x[i],i) <= b;

(SIMPLE 記述)
```

この問題のためのデータファイルの内容は，例えば次のようになります．

```
係数 c = [1] 42 [2] 12 [3] 45 [4] 5   [5] 2
          [6] 61 [7] 89 [8] 32 [9] 47 [10] 18;
係数 a = [1] 39 [2] 13 [3] 68 [4] 15 [5] 10
          [6] 20 [7] 31 [8] 15 [9] 41 [10] 16;
係数 b = 121;
```

集合 S の内容は，その集合の要素を添字としてもつ Parameter c ，および a のデータの添字によって間接的に示されることに注意して下さい．モデルの記述より， a, c の添字は S に属さねばならないので SIMPLE は a, c の添字から S の内容を推定します（これを自動代入と言います）．そのため S をデータファイルから陽に設定する必要はありません．Parameter とそれに関連する Set の内容はそれらの宣言が行われた時点で代入されます．

この例題で分かるように，モデル記述は一般的規則を記述すればよく，具体的な要素はデータとして与えられます．従って，特定の目的に対しては 1 度モデル記述をすれば，後はデータを更新するだけで色々な場合が解析できます．添え字（Set と Element）の使い方については，別冊「NUOPT_SIMPLE チュートリアル」でステップを踏んで具体的に解説していますので，より詳しくはそちらをご覧ください．

2.4 ネットワーク最適化問題(最小費用流)

ネットワークの最小コスト経路問題は，保存則を満たしながらコストを最小化する経路の一部

となる辺の組を求める問題で、一般的には、次のように記述されます。

$$\begin{aligned}
 &\text{変数} && x_{ij} \\
 &\text{目的関数 (最小化):} && \sum_{(i,j) \in A} a_{ij} x_{ij} \\
 &\text{制約式:} && \sum_{\{j|(i,j) \in A\}} x_{ij} - \sum_{\{j|(j,i) \in A\}} x_{ji} = s_i, \forall i \in N \\
 &&& b_{ij} \leq x_{ij} \leq c_{ij}, \forall (i,j) \in A
 \end{aligned}$$

(例題 4 ネットワークの最小コスト経路問題)

ここでは、 N は節点の集合を、 A は辺の集合をそれぞれ表しています。 x_{ij} は変数で、 a_{ij}, b_{ij}

と s_i はすべて定数でスカラーです。以下の例では、簡単のため、 $b_{ij} = 0; c_{ij} = \infty$ とします。こ

のようなネットワーク上の問題に対処するため、SIMPLE は、グラフ構造に対応するデータ構造を備えています。グラフは 2 つの集合: 節点 (nodes) の集合と辺 (arcs) の集合で構成されます。さらに、グラフノード上の保存則などを表現するのに便利のようにグラフの上の関数 `in(Graph, Element)` と関数 `out(Graph, Element)` を用意しました。前者は、節点 `Element` に進入する辺の集合を、後者は節点 `Element` から出発する辺の集合を返すような関数です。

SIMPLE のグラフ構造を用いると、最小コスト経路問題は次のように定義されます。

```

Graph      g;
Element    i(set=g.nodes);           // i はグラフのある節点
Element    e(set=g.arcs);            // e はグラフのある辺
Element    eout(set=out(g,i));       // 節点 i から出発する辺
Element    ein(set=in(g,i));         // 節点 i へ進入する辺
Variable x(name="流量", index=g.arcs);
Parameter a(name="費用", index=g.arcs); // 辺のコスト
Parameter s(name="供給", index=g.nodes);
//目的関数
Objective obj(name="目的関数", type=minimize);
obj = sum(a[e]*x[e], e);
//制約条件
sum(x[eout], eout) - sum(x[ein], ein) == s[i];
x[e] >= 0;

```

(SIMPLE 記述)

このモデルに、データファイル

```
費用= [1,2] 1 [2,4] 3 [1,3] 3 [3,4] 1 [2,3] 1 [3,2] 1 ;
共有= [1] 1 [2] 0 [3] 0 [4] -1;
```

を与えて実行（問題を解く）すると、次のような結果が得られます。

```
流量[1,2]=1      流量[1,3]=0
流量[2,3]=1      流量[2,4]=0
流量[3,4]=1      流量[3,2]=0
```

この結果により、与えられたグラフ（データファイル）の節点 1 から 4 までの最小コスト経路は 1→2→3→4 となることが分かります。

2.5 離散最適化（地図塗りわけ）問題

地図塗り分け問題の例を挙げます。領域のあらゆる隣り合った領域が異なる色になるような4色の色を割り当てるという問題を記述する例は次の通りです。

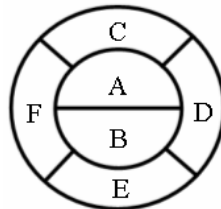
```
//
// 地図塗り分け問題
//
Set Node;
Set Edge(dim = 2, superSet = (Node, Node));
Set Color;

Element n(set = Node), n1(set = Node), n2(set = Node);
Element c(set = Color);

DiscreteVariable NodeColor(index = n, dom = Color);

Constraint DiffColor(index = (n1, n2));
DiffColor[n1, n2] = NodeColor[n1] != NodeColor[n2], (n1, n2) < Edge;
solve(); // 制約充足エンジンの起動
NodeColor.val.print(); // 解の表示
(SIMPLE 記述)
```

このモデル記述は特定の領域に依存しません。DiscreteVariable(離散変数)は一定の集合の要素(ここでは Color の要素)のいずれかを値として取る変数で、!= (異なる)という状況が表現できます。このように記述された問題を NUOPT に組み込みの制約充足問題解法ソルバーで解析することができます。領域



に対応するデータは以下ようになります。

```
Color = Red Blue Yellow Grey;
Edge =
A B
A C
A D
A F
B D
B E
B F
C D
C F
D E
E F;
```

(SIMPLE 記述)

結果として、

```
NodeColor["A"]="Blue"
NodeColor["B"]="Grey"
NodeColor["C"]="Yellow"
NodeColor["D"]="Red"
NodeColor["E"]="Blue"
NodeColor["F"]="Red"
```

が得られます。

2.6 資源制約付きスケジューリング問題

Activity, ResourceRequire, ResourceCapacity という特殊なクラスを利用することによって, 資源制約付スケジューリング問題を記述することができます. 次は人員スケジューリング問題の記述です. この例は「NUOPT_SIMPLE チュートリアル」の4章で例として説明しているものです. 詳しくはそちらをご参照ください.

```
//
// 例題 1 (人員スケジューリング問題基本形)
//
Set M = "A_does B_does C_does"; // モード
Element m(set=M);
Set R = "A B C"; // 資源
Element r(set=R);
Set D = "1 .. 11"; // 各モードの作業時間の最大
Element d(set=D);
// モードと資源消費の連関
ResourceRequire req(mode=M, resource=R, duration=D);
req["A_does,A",d] = 1, 1 <= d <= 6;
req["B_does,B",d] = 1, 1 <= d <= 8;
req["C_does,C",d] = 1, 1 <= d <= 11;
// アクティビティ
Set J = "1 .. 6";
Element j(set=J);
Activity act(name="act", index=j, mode=M); // 作業 (j=1,..6)
// 利用可能な資源の定義
Set T = "0 .. 40"; // スケジューリング全体の時間(日単位)
Element t(set=T);
ResourceCapacity cap(resource=R, timeStep=T);
cap[r,t] = 1;
Objective f(type="minimize");
f = completionTime;
options.maxtim = 2;
solve();
// 解の表示
simple_printf("job=%d %s %2d %2d¥n", j, act[j], act[j].startTime, act[j].endTime);
```

2.7 ソルバ(NUOPT)との連結

SIMPLE は数理計画法ライブラリ (NUOPT) と連結して動作します。NUOPT の GUI を用いる場合には連結操作は自動化されており、ユーザは意識する必要はありません。モデルを適当な名前のファイル (Windows 版では `model.smp`, UNIX/LINUX 版では `model.cc`) に記述し GUI を操作 (Windows 版), あるいはコマンド `mknuopt` を起動 (UNIX/LINUX 版) すると、背後でコンパイラが起動、ライブラリとリンクして実行モジュールが作成されます。この実行モジュールを実行すると数理計画法の実行が行われます。

Windows 版における基本的かつ具体的な操作手順は別冊「使ってみよう NUOPT」にありますので、まずそちらをご一読されることをお勧めします。

NUOPT のパッケージには例題集が付属しています (インストールガイドに場所が示されています) のでそちらをご参照いただくのも良い方法です。

SIMPLE での記述によって解や解を加工した値を自由に出力することも可能です。詳細は 5.3 オブジェクトの参照値の表示(関数 `print`, `cout`, `simple_printf`)をご覧ください。

3. 数理モデルを表現するクラス

SIMPLE のクラスオブジェクトは数理モデルに現れる

- ◆ 式（数式，変数，定数，制約式など）
- ◆ 添字
- ◆ データ構造（集合，グラフなど）

などを記述することができ，また構成要素としての働きをします．

3.1 クラスの一覧表

式を表現するクラスには表 1 のようなクラスがあります⁴．

クラス名称	機能概要
Expression	数式
Objective	目的関数; Expression のサブクラス
Variable	変数
IntegerVariable	整数変数; Variable のサブクラス
DiscreteVariable	離散変数; Variable のサブクラス
Activity	アクティビティ; Variable のサブクラス
Parameter	定数
ResourceRequire	作業モードの資源所要量; Parameter のサブクラス
ResourceCapacity	資源利用可能量; Parameter のサブクラス
VariableParameter	可変定数; Variable のサブクラス
Constraint	制約条件 (別名: 方程式 Equation)

表 1 式を構成するクラス

添字は表 2 のクラスで表現します．

クラス名称	機能概要
Element	集合の要素，オブジェクトの添字

表 2 添字

⁴VariableParameter は Parameter の全て性質を受け継ぐ以外に，システムを微調整する作用があります．詳しい説明は 5.11 可変定数を参照して下さい．

データ構造（集合とその亜種）を構成するクラスには次のようなクラスがあります．

クラス名称	機能概要
Set	集合
OrderedSet	順序集合，Set のサブクラス
CyclicSet	循環集合，Set のサブクラス
Sequence	数列，例えば， $1 \dots 10$
Interval	連続範囲，例えば， $(0, 1)$ ， $[0, 1]$
Graph	グラフ，ネットワークモデルなど

表 3 データ構造を構成するクラス

3.2 SIMPLE の仕組み

SIMPLE の実体は C++ のクラスライブラリです。式を特定のパーザプログラムによって解釈するのではなく、記述をコンパイル、実行して、ユーザのモデル記述内容の情報を取得します。その際に利用する仕組みは C++ の演算オーバーロード機能です。具体的には SIMPLE 特有のクラスのオブジェクト間の演算を各クラス間の演算に対応して定義し、対応する特定の実装を適宜コールさせることによって、式の解釈を行います。

そのために、定義された範囲外の演算を記述すると、まずコンパイル時にエラーとなります。例えば、2 つの引数を取る加算 (+) という演算は式の構成要素となるクラス `Expression` , `Variable` , `Parameter` を引数として取る

```
(Expression)  +  (Expression)  →  (Expression)
                                     <戻り値>
(Expression)  +  (Variable)    →  (Expression)
                                     <戻り値>
(Expression)  +  (Parameter)   →  (Expression)
                                     <戻り値>
```

等が定義されていますが、集合を表すクラス `Set` を取る

```
(Set)  +  (Expression)
(Set)  +  (Variable)
```

等は定義されていないのでエラーになります。これは集合と式の加算が定義できないことに対応しています。他の算術演算や初等関数についても同様です。例えば、

```
Expression e;  // 一般の式
Variable  x;   // 変数
Parameter a;   // 定数
Set       S;   // 集合
```

とオブジェクトが定義されているとき

```
x+a    sin(x)    a*sin(x)/e    x+e*log(x+a)
```

はそれぞれ合法に定義された演算で、いずれもその結果は一般の式 (クラス `Expression` のオブジェクト) となりますが、

```
S+a    S*x    S+sin(x+a)
```

は演算が定義されていないので違法になります。

SIMPLE の式の記述の規則は、クラスオブジェクトの間の演算規則として記述され、コンパイラによって厳格にチェックを受けます。SIMPLE の演算仕様ではクラス Parameter のオブジェクト同士、あるいは Parameter のオブジェクトとクラス Element のオブジェクトの演算結果は Parameter のオブジェクトとなりますが、Parameter とクラス Variable の演算結果はクラス Expression のオブジェクトとなります。再び加算 (+) を例にとると

(Parameter)	+	(Parameter)	→	(Parameter)
				<戻り値>
(Parameter)	+	(Element)	→	(Parameter)
				<戻り値>

であるのに対し、

(Parameter)	+	(Variable)	→	(Expression)
				<戻り値>

となります。これは変数をその構成要素として持つ式は定数とは見なせないことに対応しています。例えば

Variable	x;	// 変数
Parameter	a,b;	// パラメータ
Element	i,j;	// 集合の要素

と定義されているとき

a+b	a+i-b*j	a*b*i
-----	---------	-------

はいずれも定数(クラス Parameter のオブジェクト)を表しますが、

a/x	a*x+i	b/j+x
-----	-------	-------

は一般の式(クラス Expression のオブジェクト)になります。

以降の 3.3, 3.4 では、

- ◆ 式を表現するクラス
- ◆ 式の添字(集合の要素)を表現するクラス
- ◆ データ構造を表現するクラス

の順に、オブジェクトの生成方法や定義された演算や関数の仕様についての詳細を解説します。

3.3 式を表現するクラス (Variable/Expression/Parameter..)

SIMPLE では、式は以下のクラス

変数	Variable, IntegerVariable, DiscreteVariable Activity
数式	Expression, Objective
定数	Parameter, VariableParameter ResourceRequire, ResourceCapacity
集合の要素	Element
数値	int, double, char*

の組み合わせで構成されます。上で、数式はその表現に変数 (Variable, IntegerVariable, DiscreteVariable, Activity) を含む式を指すもので、定数式はその表現に変数を含まない式を指すものです。これらのクラスのオブジェクトを総称して以下では「式オブジェクト」と呼びます。

数式と条件付けの演算⁵

==	>=	<=	!=	<	<<
----	----	----	----	---	----

とを組み合わせで作成されたものが

制約式 Constraint (別名: 方程式 Equation)

というクラスに属するオブジェクトで、システムを記述する制約式、方程式に対応します。条件付けの演算の意味は C++ におけるものと同一です。

一方、変数を含まない定数式と条件付けの演算を組み合わせたものは

条件式 Condition

と呼ばれ、式に現れた要素 (添字) の範囲を限定する範囲を示すために使用されます⁶。

また、変数、数式と定数式に対しては演算子 ([]) を使用して添字付けることができ、

⁵ != は DiscreteVariable に対してのみ使用できます。< (先行制約), << (直前先行制約) は Activity に対してのみ使用できます。

⁶ 条件式には条件付けの演算として ==, >=, <= に加えて !=, <, > を使用することができます。

```
a[i]      x[i]*y[j]      sin(a*x[i]+y[j])
```

のような式の表現が可能です。式の添字付けについての詳細は 3.3.3 式の添字付け ([1]演算) を参照して下さい。

3.3.1 宣言と引数の意味

式あるいは式の構成要素となるクラスのオブジェクトは

```
Variable    x;
IntegerVariable y;
Expression  e1,e2;
Parameter   a,b,c;
Constraint  Co;
```

のように宣言 (コンストラクタを起動) することによって作成されます⁷。また、

```
Variable x(name="amount");
IntegerVariable y(type="binary");
DiscreteVariable x(index=(j,t),dom=A);
Activity act(index=j,mode=M);

Expression e(index=S);
Parameter a(index=i);
ResourceRequire req(mode=M,resource=R,duration=D);
ResourceCapacity cap(resource=R,timeStep=T);
Constraint Co(name="positivity",index=S);
```

のように引数付きの宣言によって、生成されるインスタンスへと付加情報を与えることもできます⁸。式について宣言の引数の書式と意味は次のように定められています。

オブジェクト種類	クラス名	引数
数式	Expression	name, index
目的関数	Objective	name, type, target
変数	Variable	name, index

⁷ 「コンストラクタの起動」とはこの様な宣言文によりオブジェクトを宣言することと等価です。

⁸ 「引数付きのコンストラクタの起動」とはこの様な引数付きの宣言文によってオブジェクトを宣言することと等価です。DiscreteVariable, Activity, ResourceRequire, ResourceCapacity で太字になっている引数はかならず与える必要があるものです。

整数変数	IntegerVariable	name,index,type,until
離散変数	DiscreteVariable	name,index,dom
アクティビティ	Activity	name,index,mode,duedate
定数	Parameter	name,index
資源所要量	ResourceRequire	name,mode,resource,duration,defaultval
資源利用可能量	ResourceCapacity	name,resource,timeStep,weight
可変定数	VariableParameter	name,index
制約式	Constraint	name,index

表 4 宣言時に与えることのできる引数

◆ **name = "文字列"**

生成されるオブジェクトの名前を定めます。名前はオブジェクトが外部に参照される際のラベルとして用いられ、特に、データファイルによってデータを与える場合には必ず必要になります。データファイルについての詳細は 4 データファイルを参照して下さい。

例えば、定義

```
Variable x(name="amount");
```

では、amount という名前を持つ x というオブジェクトを生成し、このオブジェクトと外部とがこの名前でデータの受け渡しができるようにしています。

◆ **index = 添字集合並び**

オブジェクトが添字付けられる範囲(添字集合)を定めます。index= の右辺の「添字集合並び」の表す集合が添字集合となります。「添字集合並び」は全体として集合を示すもので、集合または要素オブジェクト自身が

集合/要素オブジェクト
(例えば、index=i, index=S など)

あるいは集合または要素オブジェクトを、(コンマ)演算子で連結した

(集合/要素オブジェクト 1, ..., 集合/要素オブジェクト n)
(例えば、index=(i,j,k) など)

という書式を持ちます。集合並びに現れた集合オブジェクトはその集合そのもの、要素オブジェクトはその定義集合(要素の動く範囲)を表します。、(コンマ)で区切って並べた全体は、集

合の直積を表します。例えば、定義

```
Expression e(index=S);
```

では、集合 S を添字集合とする (S の要素で添字付けられる) e という Expression のオブジェクトを生成しています。これを、 S を定義集合とする要素 i を使用して

```
Element i(set=S);  
Expression e(index=i);
```

としても意味は同じです。

```
Parameter length(index=(N,N));
```

上記の例では 2 つの集合 N の直積を添字集合とする Parameter のオブジェクトを生成しています。

```
Element i(set=N);  
Set neighbours(index=i);  
Parameter flux(index=(i,neighbours[i]));
```

は要素オブジェクトと集合オブジェクトを混在させた表現の例で、

```
(i, neighbours[i])
```

が定数式 $flux$ の添字集合を表しています。この例で $neighbours$ は S の要素 i において添字付けられています。ただし、添字付けされた集合「添字集合並び」に使用する際には添字集合並び全体が完結している (他の要素に依存していない) が必要です。詳細については 3.4.5 添字付き集合の利用を参照して下さい。

◆ **type = minimize, maximize, integer, binary, partial, semicont**

最小化 (minimize) または最大化 (maximize) の指定は、目的関数 (Objective) の定義に用いられます。

integer, binary, partial, semicont の指定は、整数変数 (IntegerVariable) の定義の際、その種類を示すために使われます。それらの意味については 3.3.5 整数変数を参照して下さい。

◆ **until = 整数値**

type=partial である様な整数変数(IntegerVariable)の属性の定義に用いられます．詳しくは 3.3.5 整数変数を参照して下さい．

以上の宣言の引数の現われる順番は任意です．例えば

```
Constraint Co(index=S,name="poisson");
```

を

```
Constraint Co(name="positivity", index=S);
```

と書いても等価です．

◆ **dom = 集合**

制約充足ソルバー(wcsp)で扱うことのできる DiscreteVariable の定義域(有限の定数集合)を与える際に用います．詳しくは 3.3.53.3.6 を参照してください．

◆ **target = 整数値**

制約充足ソルバー(wcsp)でモデルを解く際に，目的関数のターゲット値です．詳しくは 3.3.53.3.6 を参照して下さい．

◆ **duration/resource/mode/timeStep = 集合/要素,**

◆ **duedate/defaultval/weight=定数値**

資源制約付スケジューリングエンジン(rcpsp)でモデルを解く際に問題設定を与えるためのクラス Activity, ResourceRequire, ResourceCapacity に与える引数です．詳しくは 3.3.5 を参照して下さい．

3.3.2 Expression (式) の利用

式は，宣言したオブジェクトそのもの，あるいはオブジェクト同士の演算結果を他のオブジェクトに逐次代入することによって記述されます．代入は C++ における場合と同様，すべて逐次的に解釈され，左辺は右辺のその時点における内容となります．それ以前に定義した式へと代入の影響がさかのぼることはありません．

```
Variable x,y;
Expression f,g;
f = x ;
f = f + y ;    // f = x + y となる． ( f が上書きされる)
g = f * f ;    // g = (x+y) (x+y) となる．
```

式を構成するオブジェクト同士に関しては以下の演算と初等関数が定義されています．意味はプログラミング言語 C/C++におけるものと同じです．

```

+      -      /      *
sin    cos    tan    asin  acos  atan
sec    csc    cot    asec  acsc  acot
sinh   cosh   tanh   asinh  acosh  atanh
sech   coth   asech  acsch  acoth
atan2  hypot
exp    log    log10  pow    sqrt
ceil   floor  fabs   fmod

```

変数への代入は初期値を設定することに相当します．同一の変数への代入は最も後のものが有効です．変数へは定数式以外のものを代入できません．

```

Expression f;
Parameter a;
Variable x,y;
x = 1.0;
x = 2.0*a;    // この初期値が有効
x = f;        // 違法

```

初期化されていない，オブジェクト(定数，式，変数)の値は 0 です．

次の例の p のように値並びを記述した文字列を代入することによっても値を与えることができます．文字列の内容が数値に変換可能な場合には常に数値として解釈されます．定数 (Parameter) に文字列を与えて条件判定に用いるなどのことができますが，文字の入った定

```

Parameter p;
Set S = "a b c";
Parameter q(set = S);    // 添字集合付きの定数
p = 2;                   // p = "2" と書いても同じ
q["a"] = 1;
q["b"] = 2;
q["c"] = 3;

```

数を演算に使用すると実行時エラーになります．

3.3.3 式の添字付け ([] 演算)

宣言時に“index=”で添字集合を与えられたオブジェクトは、演算子 ([]) を使用して、Element で添字付けることができます。添字付けられたオブジェクトについても式の記述の方法は全く同様です。

```
Set S = "1 2";           // 集合 S は"1" と"2" 2つの要素を持つ
Element i(set = S);      // S 上を動く要素 i を定義
Expression f(index = S); // f は S を添字集合とする
Variable x(index = i);   // x は S を添字集合とする
Parameter a(name = "coef");

f[i] = a*x[i];           // f[1] = a*x[1], f[2] = a*x[2] と等価
x[i] = 5 ;               // x[1], x[2] の初期値が 5 となる
```

上記の例に見るように、添字付きのオブジェクトへの代入は添字となっている要素の取り得る全範囲にわたって代入を書き並べたものと等価です。添字(集合の要素)を表現するオブジェクトである Element については 3.3.11 添字で、添字の動く範囲を表現する集合オブジェクトについては 3.4 データ構造を表現するクラス でそれぞれ詳しく述べます。また別冊「NUOPT_SIMPLE チュートリアル」の 2 章では集合と要素の利用例を段階的に説明しておりますので、そちらもあわせてご参照ください。

また、上記の例における

```
f[i] = a*x[i];
```

のように、添字付けられたオブジェクトと付けられていないオブジェクトを式の中に混在させることも自由です。しかし、右辺が不定になるような表現をした場合には実行時エラーとなります。例えば上記の例で

```
Expression e; // 添字付けられない式を表現する e を生成
e = f[i];     // 実行時エラー (i が定まらない)
```

とした場合には実行時エラーとなります。

生成時に添字集合を与えられていないオブジェクトを添字付けようとする、あるいは生成時に添字集合を与えられたオブジェクトを添字付けなかった場合には実行時エラーになります。

```
e[i] = f[i]; // 実行時エラー (e は添字集合を与えられていない)
f = 2.0 ;    // 実行時エラー (f は添字集合を与えられている)
```

添字には、オブジェクト Element の他、int, double, char を直接使用することも

できます．添字の表現自体についての詳細は 3.3.11 添字を参照して下さい．

```
Set T = "a b";           // 集合 T は"a" と"b" 2つの要素を持つ
Expression g(index = T); // g は T を添字集合とする
g["b"] = 5*x[2];         // g と x を char* と int それぞれで
                        // 添字付ける
```

添字として用いられた char* は数字に変換可能な場合，数字に変換されて解釈されます．double (実数) は int (整数) に変換可能な場合，整数値に変換されて解釈されます (これは 4 データファイルで述べるデータファイルの解釈についても同じです) ．

例えば，以下は同じ意味を持ちます．

```
f[5] = 1;
f[5.0] = 1;           // double が int に変換
f["5.0"] = 1 ;       // char* が double を経て int に変換
```

3.3.4 範囲にわたる演算 (sum, prod, ..)

添字の動く範囲にわたって和や積を取る演算である sum() , prod() , max() , min() , selection() が用意されています．まず sum() , prod() は以下の仕様を持つ関数です．

```
// 添字の範囲にわたる和
Expression sum(式, 範囲指定並び) // 戻り値は一般の式
Parameter sum(定数式, 範囲指定並び) // 戻り値は定数式
// 添字の範囲にわたる積
Expression prod(式, 範囲指定並び) // 戻り値は一般の式
Parameter prod(定数式, 範囲指定並び) // 戻り値は定数式
```

「範囲指定並び」とは添字同士あるいは添字と条件式の組合せで，

```
添字
(添字 1, 添字 2)
(添字, 条件式)
(添字, 条件式 1, 条件式 2)
(添字 1, ..., 添字 n, 条件式 1, ..., 条件式 n)
```

等の書式を持ちます⁹。

添字同士の組み合わせはそれらの定義集合の直積を示します。条件式は添字の動く範囲 (演算が行われる範囲) を限定する働きをします。sum() , prod() それぞれについて戻り値の異なる 2 種類があるのは演算が行われる式に通常の式と定数式の区別があるためで、定数式の和がやはり定数であることに対応しています。範囲指定並びに用いた添字の定義集合が空である等、演算を行う範囲が存在しない場合には sum() の結果 (戻り値) 0 に、prod() の結果は 1 になります。

以下は sum() の使用例です。

```
Set S = "1 2 3 4 5" ;    // S は要素として 1 ... 5 を持つ
Element i(set = S); // S 上を動く要素
Parameter a(index = i);
Variable x(index = i);
Parameter asum;
Expression xsum;

asum = sum(a[i], i); // asum =  $\sum_i a_i$  (定数)

xsum = sum(a[i], i); // xsum =  $\sum_i x_i$  (通常の式)
```

範囲指定並びとして添字と条件式の組み合わせを使用すると、その条件式によって添字の動く範囲が限定されます。条件式についての詳細は 3.3.8 条件式を参照して下さい。例えば上記の例で、

```
Expression xsum2;
xsum2 = sum(x[i], (i, i<3)); // xsum =  $\sum_{i<3} x[i] = x[1]+x[2]$ 
```

という表現が可能です。

⁹添字同士あるいは、添字と条件式を組み合わせる場合には上記の様に範囲指定並び全体を () で括る必要があります。

```

Element    j(set=S);           // j は i と独立な S の要素
Variable    m(index=(i,j));
Expression  msum,msum2;

msum  = sum(m[i,j],(i,j));      // msum  =  $\sum_{i,j} m_{ij}$ 

msum2 = sum(m[i,j],(i,j,i<j)) // msum2 =  $\sum_{i<j} m_{ij}$ 

```

添字同士の組み合わせを範囲指定並びとして使用すると、複数の添字の定義集合の直積上の演算を示します。以下はその記述例です。

また、`sum()`、`prod()`と同様に添字の動く範囲にわたっての最大、最小を求める関数 `max()`、`min()` が用意されています。以下の仕様を持つ関数です。

```

// 添字の範囲にわたる和
Expression  max(式, 範囲指定並び)      // 戻り値は一般の式
Parameter   max(定数式, 範囲指定並び)  // 戻り値は定数式
// 添字の範囲にわたる積
Expression  min(式, 範囲指定並び)      // 戻り値は一般の式
Parameter   min(定数式, 範囲指定並び)  // 戻り値は定数式

```

利用方法は `sum`、`prod` と全く同様です。

ただ、`Expression` を返す `max`、`min` は非線形な演算でかつ、一般に微分不可能な点を持つため、これらを含めて問題を記述すると、性質の悪い非線形計画問題となってしまうので、制約充足アルゴリズム (`wcsp`) をご利用する場合のみとされることを推奨します。その他の場合に `max`、`min` が必要な状況が現れた場合には、非線形な記述を用いない等価な書き換えが存在しますのでそちらをご利用下さい¹⁰。

さらに、制約充足問題ソルバー (`wcsp`) のみが解釈することができる `selection()` という演算があります。0-1 型の整数変数 (`type=binary` として宣言された `IntegerVariable`) のみを引数として取り、複数の 0-1 変数が「選択」を示すことを明示的に宣言するものです。引数として与えられた 0-1 整数変数すべての和が 1 であるという制約に等価ですが、制約充足問題ソルバー `wcsp` はこの情報をもとに、複数の 0-1 変数から離散変数一つを対応付けて効率良く処理します。

```
selection(IntegerVariable[添字]);
```

¹⁰ 具体的な定式については (株) 数理システムのサポート (nuopt-support@msi.co.jp) までご連絡下さい。

```
selection(IntegerVariable,範囲指定並び);
```

離散変数を用いた表現：

```
Set Agent = "a b c";
Set Job = "1 .. 5";
Element j(set=Job);
DiscreteVariable job(name = "job", dom = Agent, index = j);
Objective totalCost(name = "total", type = minimize);
totalCost = sum( cost[i,j] * Boolean(job[j] == i) , (i,j) );
```

等価な IntegerVariable を用いた表現：

```
Set Agent = "a b c";
Set Job = "1 .. 5";
Element i(set=Agent),j(set=Job);
IntegerVariable job(name = "job", index = (i,j), type=binary );
selection(job[i,j],i); // sum(job[i,j],i) == 1 に等価
Objective totalCost(name = "total", type = minimize);
totalCost = sum( cost[i,j] * job[i,j] , (i,j) );
```

後者の表現を用いることによって，Boolean 演算（ 3.3.9 条件分岐関数（関数：ifelse()，Boolean()））を消去，式を線形にすることができます．このような式の線形化は式を解釈する時間や，前処理時間の向上など，大規模問題を取り扱う際に有効です

3.3.5 整数変数 (IntegerVariable)

整数変数(IntegerVariable)は整数計画問題を記述するためのものです．整数変数の定義は，例えば以下ようになります．

```
Set a(name="a");
Element i(set=a);
IntegerVariable y(name="y", index= i, type=binary);
IntegerVariable x(name="x", index= i, type=partial, until=3);
```

IntegerVariable の式の中における扱いは通常の Variable と全く同じですが宣言の際に与えた引数，type と until によって，その属性を定義することができます．

```

type = integer 普通の整数値変数，type を指定しなければ integer になります．
      = binary   2 進変数，値は 0 と 1 しか取らない変数．
      = partial  until と併用で，指定された限界 (until) まで整数値を取り
                  それ以上は任意の値を取る変数．
      = semicont 0 であるか，1 以上の変数．

```

until は整数で，type=partial のときのみ有効です．整数変数の上下限の指定は，普通変数と同様，以下のように行なわれます．

```

1 <= y[3] <= 10;
-5 <= x[i] <= 5;

```

整数変数には分岐限定法を行うソルバに渡すパラメータとなる次のようなメンバが用意されています．

```

double pri   ( Priority )   分岐優先順位
double upc   ( Up PC      ) 押し上げ擬コスト
double dpc   ( Down PC   ) 押し下げ擬コスト
double dir   ( Direction ) 分岐方向

```

この意味については第二部の 2.7 整数計画問題に関するパラメータ設定を御覧下さい．値の設定は

```

Set S = " 1 2 3 ";
Element i(set=S);
IntegerVariable x(index=i);
x[1].pri   = 10;
x[2].pri   = 5;
x[3].pri   = 2;
x[i].upc   = 0.05;
x[1].dpc   = 1.35;
x[2].dpc   = 2.04;
x[3].dpc   = 6.66;
x[i].dir   = 1;

```

のように行います．

3.3.6 離散変数 (DiscreteVariable)

離散変数 (DiscreteVariable) 与えられた集合の要素のいずれかを取る変数です¹¹。集合の要素の型によって数あるいは文字列を値として取ります。以下が定義の例です。

```
Set Color = "red black yellow";
DiscreteVariable roof(name = "roof", dom = Color);
DiscreteVariable wall(name = "wall", dom = Color);
DiscreteVariable door(name = "door", dom = Color);
```

宣言の後に与える値 `dom` は集合オブジェクトを取り、この離散変数の取る範囲を示します。

```
Set Agent = "a b c";
Set Job = "1 .. 5";
Element j(set=Job);
DiscreteVariable job(name = "job", dom = Agent, index = j);
```

以下の例で示すように、離散変数は一般の整数変数の拡張概念として考えることができます。なお、DiscreteVariable の値の範囲を示す集合オブジェクトの要素は正の整数または文字列である必要があります(実数、あるいは負の整数を要素とすることはできません)。

```
Set S,Binary = "0 1";
Element j(set=S);
DiscreteVariable cover(name = "cover", dom = Binary, index = j);
//0-1 整数変数と等価
```

DiscreteVariable の式の中における扱いは通常の Variable と全く同じで、計算に用いることができます。ただし、`dom` の設定により、文字列を値として取る場合には、その値を式に用いると実行時エラーになります。

ここでは DiscreteVariable に対してのみ定義されている操作について述べます。

1. 添字付け

次の構文を用いることにより、DiscreteVariable の値で

```
Table(Parameter)
DiscreteVariable
```

を添字付けることができます。Table は Parameter の別名です。Table のクラス定義は Parameter とまったく同一です¹²が、Parameter (定数) という本来の意味から外れるため、ここではあえて Table として記述しています。

¹¹現状の NUOPT (Ver.8) では離散変数 (DiscreteVariable) を含むモデルの求解は制約充足問題ソルバー (wcsp) のみで行うことができます。

¹² C++の機能である typedef を用いて等価なものとして宣言されています。

a) Table の添字付け

```
Table[DiscreteVariable]
Table[DiscreteVariable,DiscreteVariable]
Table[DiscreteVariable,DiscreteVariable,DiscreteVariable]
```

```
Set Place;
Element i(set=Place),j(set=Place);
Table dist(name="距離",index=(i,j));
DiscreteVariable x(dom=Place),y(dom=Place);
dist[x,y] <= 35; // 距離が 35 以下であるように x,y を制約
```

b) DiscreteVariable の添字付け

```
DiscreteVariable[DiscreteVariable]
```

```
Set S = "1 .. 8";
Element i(set=S);
DiscreteVariable perm(index=i,dom=S);
DiscreteVariable inverse_perm(index=i,dom=S);
perm[inverse_perm[i]] == i;
//perm と inverse_perm が逆写像であることを定義
```

2. alldiff() 制約式の定義

この構文で「値がすべて異なる」という特殊な制約の定義を行うことができます¹³。

a) alldiff(DiscreteVariable[添字]);

```
Set S = "1 2 3";
Element i(set=S);
DiscreteVariable x(index=i,dom=S); //インデックスと範囲が同一の離散変数
alldiff(x[i]); // x[1],x[2],x[3]が 1,2,3 のいずれかですべて異なる
```

b) alldiff(DiscreteVariable,範囲指定並び);

¹³ 現在 NUOPT (Ver. 7) でこの制約を扱うことができるのは、制約充足問題ソルバー (wcsp) のみです。

```

Set S = "1 .. 10";
Set T = "a b c";
Element i(set=S);
DiscreteVariable y(index=i,dom=T);
alldiff(y[i],(i,i<=3));
// y[1],y[2],y[3]が a,b,c のいずれかですべて異なる

```

3.3.7 制約式 (Constraint)

条件付けの演算子のうち

```

==    >=    <=

```

は複数の式をつなげて制約式を定義する働きがあります。制約式は `Constraint` というクラスオブジェクトです。制約式の定義において条件付けの演算子である `<` と `>` 及び `!=` は使用できません。

また、`!=` は、微分不可能な特殊な関数を用いて解釈されますので、通常の整数計画問題などで、`!=` が必要になった場合には、他の条件付けの演算子を用いて表現して下さい。条件付けの演算 `!=` のご利用は微係数の情報を利用しない制約充足アルゴリズム (wcsp) をご利用する場合のみとされることを推奨します。

例えば

```

x*sin(y)>= z;
sum(f[i],i)==2*x;
x[i]>=y[i]>=z[i];

```

は制約式で、クラス `Constraint` のオブジェクトです。

コード中に記述された制約式はすべて記述順にシステムに認知されます。

```

Set S = " 1 2 3 ";
Element i(set = S);
Variable x(index = i);
Constraint t;
x[1] <= 5; // 最初の制約式と認知される
t = x[1]+x[2] - x[1]*x[2];
x[3]*sin(t) >= x[1] >= 2; // 2番目の制約式と認知される

```

添字付けされているオブジェクトを含む制約式は添字の取り得る範囲すべてにわたって展開されて解釈されます。

```
x[i]*x[i] >= 3;
// 添字 i の取り得る値を{ 1 2 3 } とするなら
//    x[1]*x[1] >= 3 ;
//    x[2]*x[2] >= 3 ;
//    x[3]*x[3] >= 3 ;
// を並べたのと同じ
```

Constraint 型のオブジェクトを陽に宣言して、生成された制約式を代入することもできます。例えば上記の例に現れる制約式を

```
Constraint co1,co2,cost(index = i);
co1 = x[1] <= 5;
co2 = x[3]*sin(t) >= x[1] >= 2;
cost[i] = x[i] x[i] >= 3;
```

と陽に宣言したオブジェクトに代入することができます。

3.3.8 条件式

変数を含まない式(定数式)を条件付けの演算子

```
==      >=      <=      <      >
```

を用いてつなげたものは条件式と呼ばれ、代入文の左辺や制約式に現れる添字の動く範囲を限定する働きをします。条件付けの演算子の意味は C++ のものと同一です。例えば

```
i > j      a[i]*b[i] == 1      a[i] + j < i
```

は条件式で、クラス `Condition` のオブジェクトです。さらに条件式同士は

```
! (not)      && (and)      || (or)
```

によって連結することができます。例えば

```
(i > j) && ( a[i] > 0 )      !(a[i]*b[i] == 1)
```

も同じく条件式で、クラス `Condition` のオブジェクトです。

条件式は, (コンマ) 演算子によって代入文や制約式に付加することができます. 付加された条件式は代入文の左辺や制約式に現れる添字の展開される範囲を限定する働きをします. (コンマ) 演算子の仕様は以下の通りです.

代入文, 条件式	// 代入文の左辺に現れる添字の範囲の限定
制約式, 条件式	// 制約式に現れる添字の範囲の限定

条件式が付加された代入文や制約式は条件式が真となるような添字の範囲にわたってのみ展開されます.

```
Set S = " 1 2 3 ";
Element i(set = S);
Expression f(index = i);
Variable x(index = i);
f[i] = x[i+1]/x[i] , i+1 <= 3 ; // a[1],a[2] のみへの代入
x[i] x[i] <= 3 , i != 1           // i=2,3 に対してのみ制約式を定義
```

また,

代入文, 条件式 1, 条件式 2, ..., 条件式 n
制約式, 条件式 1, 条件式 2, ..., 条件式 n

のように, 条件式を連結することも可能です. 連結された条件式はすべての and として解釈されます. 例えば上の例で

```
x[i]*x[i] >= 1, i != 1 , i <= 2
//i は{1 2 3}内をまわるのでこの制約式は i = 2 の場合に
//対してのみ定義されることになる
```

とすることができます.

条件式は, 前述した `sum()`, `prod()` 関数の範囲並びの中に, 例えば次のようにして使用することができます.

Parameter asum;
asum = sum(a[i], (i,a[i]>0)); // a のうち正のもののみの和

また, 条件式を単独で文中に書くと, その中に現れる添字が以降の式中において展開される範囲すべてを限定します.

```

a[i] < 0;          // 以降, i の範囲を a[i]<0 である範囲に限定
x[i] = 0;          // x[i] = 0, a[i] < 0; と等価
asum = sum(a[i],i); // 右辺は sum(a[i], (i,a[i]<0)) と等価

```

単独で書いた条件式は、同じ要素を使用した他の条件式が現れるか、その要素が代入によって限定されるまで有効となります。

```

Element j(set = S);
i > 2 ;
x[i] == 1;          // x[i] == 1, i > 2 と等価
i > j ;             // 条件式 i > 2 はここで無効となる
x[i] >= x[j];       // x[i] >= x[j] , i > j ; と等価

```

添字を含まない条件式は `int` 値へとキャストすることができます。キャストを行うとその時点での条件式の成立がチェックされ、真偽を示す `int` 値を返しますので、次の様に条件式を `if` や `for` 等、真偽値を要求する部分に使用したシステムの記述を行うこともできます。ただし添字を含む条件式は添字の値によって真偽が変わりますのでこのような表現はできません。また変数を含む条件式は、`SIMPLE` は数理計画の制約式だと解釈するのでこのような使い方ができないことに注意して下さい。変数を含む条件を使って式を記述するためには次項で解説する `ifelse` を用います。

```

Parameter a(name = "a");
if ( a > 0 ) {
    x[i] = x[i] + a;
} else {
    x[i] = x[i] - a;
}

```

条件式は式からのみでなく、集合を表現するオブジェクト (`Set` 等) と集合の要素の組み合わせからも生成することができます。その詳細については **3.4.8 集合と要素から生成される条件式**を参照して下さい。

3.3.9 条件分岐関数(関数:`ifelse()`, `Boolean()`)

以下の仕様の関数 `ifelse()` を用いると、変数を含む条件式 (すなわち、制約式:`Constraint` オブジェクト) の成立によって値が分岐する関数を表現することができます。

関数 `ifelse()` は非線形式として扱われるため、関数 `ifelse()` を利用できるのは NLP モジュールのみとなります。

```
Expression ifelse(制約式, 式 1, 式 2);
```

```
Expression Boolean(制約式);
```

最初の引数の「制約式」とは変数を含む複数の式を条件付けの演算子

```
==    >=    <=    !=
```

によってつなげたもの (Constraint オブジェクト) です。式 1, 式 2 の部分には式を表す任意のオブジェクトを書くことができます。具体的には `int` や `double` の値、定数を表す `Parameter`、あるいは変数を含む `Expression` のいずれをも書くことができます。

関数 `ifelse` は制約式が示す条件が成立する際に式 1 の値、成立しない場合には式 2 の値となります。

関数 `Boolean` は制約式が成立する際に 1、成立しない場合に 0 となります。

```

Set S = "1 .. 10";
Element i(set=S);
Set Val = "a b c";
DiscreteVariable x(dom=Val,index=i);
sum( Boolean(x[i] != x[i-1]), (i, i>=2 ) ) == 3;
    // x[1],x[2],x[3],...,x[10] の隣り合うもので異なるものが3つ

Set Color = "red black yellow";
DiscreteVariable roof(name = "roof", dom = Color);
DiscreteVariable wall(name = "wall", dom = Color);
DiscreteVariable door(name = "door", dom = Color);

Boolean(roof == "blue") + Boolean(door == "blue") == 1;
// roof か door のいずれかが"blue"
Set M = " a b c ";
Activity A(mode=M);
Activity B(mode=M);
Boolean(A=="a")*A.startTime >= Boolean(B=="a")*B.endTime
    - Boolean(A!="a")*1000;
// A と B がいずれもモード"a"で処理されるのであれば,BはAに先行する
// 1000 は全体のスケジュール期間の長さよりも大きな数

```

Boolean は ifelse の式 1 を 1 に、式 2 を 0 に置き換えた特殊ケースとしてみなすことができますが、制約式を 1, 0 に変換する演算は、文字列も値として取り得る離散変数 (DiscreteVariable)、アクティビティ (Activity) を用いたモデル記述に、頻繁に現れるので便宜のために用意しています。以下、Boolean の利用例です。

同様の記述が ifelse を用いても可能ですが、Boolean には DiscreteVariable を用いた制約式が引数となっている場合を前提として処理を高速化しておりますので、DiscreteVariable を用いたモデル化の際には Boolean() を推奨いたします。

ifelse も Boolean も最初の引数として使用することのできるのは、変数を含む式 (Variable, IntegerVariable, DiscreteVariable, Activity または Expression オブジェクト) を条件付けた式のみです。

例えば最初の引数に

```
// 条件式によって置き換えるのが適切な例
Set S;
Element i(set=S);
Expression f(index=i);
Variable x(index=i), y(index=i);
Parameter a(index=i);

f[i] = ifelse( a[i] >= 0, x[i], y[i]); // エラー
```

のように変数を含まない式(条件式)を用いることはできません。最初の引数が前項で説明した「条件式」であると解釈されるのでコンパイル時エラーになります。このような場合には条件式を使った表現に置き換えて記述して下さい。上記は

```
f[i] = x[i], a[i] >= 0;
f[i] = y[i], a[i] < 0;
```

という表現と等価になります。
次が ifelse 関数の記述例です。

```
// 数値的に安定な ベルヌーイ (Bernui) 関数の計算
Variable x(index=i);
Expression B(index=i);
B[i]=ifelse(fabs(x[i])<1.0e-4
            ,1/(x[i]+pow(x[i],2)/2+pow(x[i],3)/3)
            ,x[i]/(exp(x[i])-1));
```

ifelse() 関数を用いると非常に多岐にわたる式を記述することが可能です。たとえば次のように連続微分不可能な関数を記述することも可能になってしまいます。

```
// 不連続関数の記述
// (以下のような関数が含まれる問題は大域的最適化(global)以外で
// NUOPT の動作は保障されない)
Variable x(index=i);
Expression f;
f = ifelse(x <= 1.0, x , 1/x ); // 連続だが微係数は不連続
g = ifelse(x <= 1.0, x , x*x + 5 ); // 関数値も不連続
```

NUOPT のアルゴリズムの多く(大域的最適化"global", 制約充足問題ソルバー"wccsp"を除

くすべて) はモデルに現れる関数が連続微分可能であることを前提としておりますので、このことはアルゴリズムに深刻な影響を及ぼす可能性があります。

`ifelse()` 関数は、主に数値的安定性を図るために準備されているものです。上に述べた「ベルヌーイ関数」の例では `ifelse()` を使って数値的な安定性を図った例です。

ベルヌーイ関数と呼ばれる以下の関数は

$$B(x) = \frac{x}{e^x - 1}$$

$x=0$ の付近では、計算機上では $0/0$ に近くなり、数値的に安定性を欠いた式になります。しかしながら、解析的には

$$\lim_{x \rightarrow 0} B(x) = 1$$

となります。これを計算機上で近似的に扱うため、 x の絶対値が一定の数(コード例では 10^{-4}) よりも小さいときには

$$B(x) = \frac{x}{e^x - 1} \approx \frac{1}{1 + \frac{x}{2} + \frac{x^2}{3}}$$

のように近似しています。

しかしながら、次の「不連続関数の記述」の例にあるように、関数としての連続性が考慮されない式を `ifelse()/Boolean()` を用いて表現した場合は、式の記述はできても `NUOPT` が正しく機能する保証はありません。予めご了承下さい。

以上まとめると、`ifelse` もしくは `Boolean` が問題なく適用できるのは主に以下のケースに限られます。

1. `ifelse` によって定義されている関数が二階連続微分可能である場合
2. 大域的最適化アルゴリズム (`global`) を用いる場合¹⁴
3. 制約充足問題アルゴリズム (`wcsp`) を用いる場合

2. は大域的最適化が `ifelse` の分岐を考慮するため、3. は制約充足問題アルゴリズム (`wcsp`) が微係数の情報を利用しないためです。

なお、その他のケースでも関数の性質が明らかな場合には `NUOPT` を慎重に適用することにより有効な結果を出すことができるケースも存在します。`ifelse()/Boolean()` 関数の使い方についてご相談がある場合は nuopt-support@msi.co.jp までご連絡下さい。

¹⁴ `NUOPT (Ver. 7)` ではこの場合、`ifelse` の制約式に `!=` が含まれるものはサポートしておりません。ご了承下さい。

3.3.10 定数値のチェック (関数:check())

以下の仕様の関数 `check()` を呼び出すことにより条件式の成立を確認することができます。

```
void check(条件式 1[, 条件式 2]); // 条件式の成立をチェックする
```

条件式 1 が成立していなければ実行時エラーとなります。

```
Parameter p(name = "p");
check(p > 0); // p が正であることの確認
```

条件式が添字づけられている場合には、添字はその動く範囲において展開されて成立が確認されます。2 つめの引数に条件式を与えると、その添字の展開の範囲が限定されます。

```
Set S = "1 2 3";
Element i(set=S);
Parameter a(index = i);
check(a[i] > 0);           // a[1],a[2],a[3] が正であることの確認
check(a[i] > 0, i <= 2); // a[1],a[2] が正であることの確認
```

この機能は入力データが正常なものをチェックする際に便利です。

3.3.11 添字の表現 (Element など)

数理モデルを記述する式に現れる添字は、以下のオブジェクト

集合の要素	Element
文字列	char*
数値	int double
定数値	Parameter

自身や、これらを演算

, (直積)	+	(和)	-	(差)	/	(商)	*	(積)
% (余り)	ceil	(切り上げ)	floor	(切り下げ)				

で組み合わせて構成したものです。

要素オブジェクトの生成

他のオブジェクトと同様、要素は宣言して利用します。

```
Element i,j;
```

要素がどの集合オブジェクトに属するものであるかという情報は引数 `set` を用いて与えます。

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

◆ `set` = 集合オブジェクト

生成される要素が属する集合 (定義集合) を定めます。生成される要素はこの集合内のみを動くことになります。

```
Element i, j;
```

のように、引数のない宣言で生成された要素は特に属する集合を持たず、値を代入してのみ使用されます。要素への値の代入については本節の後半で説明します。定義集合を持たない要素は、要素への代入 (以降で説明します) によって値を持たせる前には何も表しません (定義集合が空集合)。

添字の表現

`Element` オブジェクトは直接コンストラクタで生成する他、演算子

```
, (直積)    + (和)    - (差)    / (商)    * (積)
% (余り)    ceil (切り上げ)  floor (切り下げ)
```

によって生成することができ、いずれも添字として使用することができます¹⁵。例えば `i, j` がそれぞれ要素オブジェクトとすると、

```
i, j      2*i-j      i%3+j      floor(i/j)
```

はそれぞれ要素オブジェクトであり、

```
f[i, j]    g[2*i-j]    h[i%3+j]    x[floor(i/j)]
```

のように添字として使用することができます。集合の直積を添字集合として持つオブジェクトを添字付けるのには

¹⁵ `%` を除く演算は両辺の要素の値を実数として扱います。演算 `%` の両辺は整数であるものとして扱います。文字列を表す要素にこれらの演算を施すと実行時エラーになります。

```
a[i,j], b[i+1,j-1]
```

のように、要素の直積を使用します。ただし、SIMPLE の実装上の制約上、C++ の組み込み型 (int, double, char*) 同士の、(コンマ:直積) 演算は正しく解釈されず、最後に現れたものを単独で書いたのと同じ意味になってしまいますので、ご注意下さい。

```
a[1,2] = ... ;           // 注意! a[2] と同じ意味になる
b["apple",3] = ... ;     // 注意! b[3] と同じ意味になる
c["orange",2.5] = ... ;  // 注意! c[2.5] と同じ意味になる
```

この場合には以下のように直積演算全体を 1 つの文字列として記述すれば所望の結果を得ることができます。

```
a["1,2"] = ... ;        // a[1,2]
b["apple,3"] = ... ;    // b[apple,3]
c["orange,2.5"] = ... ; // c[orange,2.5]
```

添字としての要素オブジェクト

宣言において属する集合 (定義集合) が与えられた要素は、添字として代入の左辺や制約式に現れると、その取り得る値の全範囲にわたって展開されて解釈されます。

```
Set S = "1 2";           // 集合 S = 1, 2
Element i(set = S);      // S を定義集合とする要素 i
Expression f(index = S); // f は S を添字集合とする
Variable x(index = i);   // x は S を添字集合とする
Parameter a(name = "coef");

f[i] = a*x[i];           // f[1] = a*x[1], f[2] = a*x[2] と等価
x[i] = 5 ;               // x[1], x[2] の初期値が 5 となる
x[i]*x[i] >= 0 ;         // 2 つの制約式が定義されたことになる
```

上記の例では i という Element オブジェクトが集合 S のものとして生成され、以降 i が添字として現れる式は定義集合 S の内容 1, 2 にわたって展開されています。

```

Set S = "1 2";
Element i(set=S),j(set = S); // i,j は S 上を動く
Set T = "2 3";
Set U = "0 1";
Expression g(index = (T,U)); // (T,U) という直積集合を添字集合とする

g[i+1,j-1] = 0; // g[2,0],g[2,1],g[3,0],g[3,1] をすべて 0 とする

```

演算子によって生成された要素についても同様です．その値の取り得るすべての範囲を動き，代入の左辺や制約式に現れると展開されて解釈されます．

上記がその例で，代入は演算によって生成された要素 $(i+1, j-1)$ が取り得る範囲すべてに関して展開されます．

要素への代入

要素は，通常その定義集合の要素全体の総称としての意味を持ちますが，代入演算によって単一の要素の値に限定することができます．

```

Set S = "1 2 3";
Element i(set = S);
Expression f(index = i);
i = 2;
f[i] = 3 ; // f[2] = 3 と同じ

```

この場合，代入の右辺として用いることができるのは

集合の要素 `Element` のオブジェクト自身

文字列 `char*`

値 `int, double`

のいずれかです．ただし，右辺として `Element` を使用する場合には，その `Element` がそれ以前に代入操作によって単一の要素の値に決定されている必要があります．

```

Set T = " a b c";
Element i,j,k;
j = "a";
i = j; // i も"a" となる
i = k; // k は代入によって決定されていないので違法

```

定義集合を持つ `Element` は定義集合の要素以外の値に固定することができません．例えば，

以下は実行時エラーとなります。

```
Set S = "1 2 3";
Element i(set=S);
i = 4; // 違法
```

要素への値の代入と集合についての条件式，及び順序付き集合 (OrderedSet, CyclicSet) のメンバ関数によって for ループや while ループを使用したシステムの記述が可能です。詳細は 3.4.11 イテレータの表現 (順序付き集合のメンバ関数) を参照して下さい。

3.3.12 スケジューリング問題定義の為に (Activity, ResourceRequire/Capacity)

クラス Activity, ResourceRequire, Capacity は NUOPT のタブー・サーチによる資源制約付きスケジューリングエンジン (rcpsp) にスケジューリング問題の定義を与えるために利用します。各クラスは以下のような意味ですが, SIMPLE の既存のクラスと等価なものとして宣言されています。したがって, ResourceRequire, ResourceCapacity は Parameter と同様の方法で値を設定したり, 値を表示することができます。また, Activity も Variable と同様の方法で値を表示したり, 演算に用いたりすることができます。

クラス名	等価なクラス	意味
ResourceRequire	Parameter	各モードと資源の消費
Activity	Variable	アクティビティとモードの対応
ResourceCapacity	Parameter	資源の利用可能量とスケジューリング全体時間の対応

クラス ResourceRequire

書式	意味
mode=集合または要素	作業モード
resource=集合または要素	所要リソース量
duration=集合または要素	経過時間
[defaultval=定数値]	デフォルトのリソース消費量

クラス Activity

書式	意味
mode=集合または要素	作業に適用可能な作業モード
[duedate=パラメータ]	納期

クラス ResourceCapacity

書式	意味
resource=集合または要素	可能消費量を定義する資源
timeStep=集合または要素	全体時刻の定義
[defaultval=定数値]	デフォルトのリソース消費量
[weight=重み]	資源制約をソフト制約にする際の重み

これらクラスの実際の利用については、別冊「NUOPT__SIMPLE チュートリアル」の 4 章「4.2rcpsp モデル記述の実際」をご覧ください。

目的関数

rcpsp において目的関数に設定出来るものとして、最後の作業の完了時刻、納期遅れの 2 つがあります。ただし、これらは最小化のみ(type=minimize)の設定となります。

最後の作業の完了時刻最小化：

```
Objective = completionTime;
```

納期遅れ最小化：

```
Objective = tardiness;
```

先行制約、直前先行制約

rcpsp のみが解釈する特殊な演算としては、Activity 同士の先行制約と直前先行制約があります。いずれも Activity 同士の演算として定義されています。

先行制約：

```
Activity < Activity [, 条件式][,Parameter];
```

直前先行制約：

```
Activity << Activity , Element[,条件式];
// Element はリソース集合の要素
```

この演算の実際の利用については別冊「NUOPT__SIMPLE チュートリアル」の 4 章「4.5.2 先行制約 4.5.3 直前先行制約」をご覧ください。

一般の演算

以下のコンポーネント

Activity.startTime

Activity.endTime

Activity.processTime

Boolean(Activity==文字列) と Activity.startTime との積

Boolean(Activity==文字列) と Activity.endTime との積

の定数倍を足し合わせて一般の制約式を記述することができます。係数となる定数には通常のSIMPLEによる数理モデルと定義と同じく一般の数字や Parameter が利用可能です。Activity のメンバーである

```

startTime
endTime
processTime

```

は Variable として扱うことができます。ただし、最後の二つについて、Activity は同一のインスタンスを指さねばなりません。実際の利用については別冊「NUOPT__SIMPLE チュートリアル」の4章「4.6 一般の考慮制約の導入」をご覧ください。

3.4 データ構造を表現するクラス (Set/Graph)

次のようなクラスは集合とその亜種であるデータ構造を表現するためのものです。

集合	Set, OrderedSet, CyclicSet
数列	Sequence
連続範囲	Interval
グラフ	Graph

この章では、上に挙げたクラスを順に紹介します。

添字の値の動く範囲は一般に集合 (Set) というクラスのオブジェクトとして表現されます。

Set には

順序付き集合	OrderedSet
周期的順序付き集合	CyclicSet

というサブクラスがあります。

集合は任意の要素の集まりとして定義されます。集合に属する要素の型は int, double, char* またはそれらの組み合わせのいずれかです。集合を定義する時に、その要素の次元数 (dim) の指定を行ないます。要素の次元数とは、要素が組である場合の構成要素の数です。例えば、

```

Set S1(dim=1);           // S1 の次元数は 1
                          // (dim=1 の指定は省略可)

S1 = " 2 string 1.234 ";  // S1 = {2, "string", 1.234}

Set S2(dim=2);           // S2 の次元数は 2

S2 = " 1 2 a 3 1.234 4 "; // S2 = {(1,2), ("a",3), (1.234,4)}

Set S3(dim=3);           // S2 の次元数は 3

S3 = " 1 a 2 1.234 4 3 "; // S3 = {(1,"a",2), (1.234,4,3)}

```

は全て有効な定義で、データの並びは次元数個単位に区切って解釈されます。ただし、次元数の違う要素が同一の集合に混在することはできません。例えば次のような集合代入

```
Set S4(dim=2);
S4 = " 1 2 3 ";    // S4 の2つ目の要素が構成できない
```

はエラーとなります。

OrderedSet や CyclicSet は属する要素の順序を保持しますので、以下のメンバ関数によって要素の順序に依存する操作ができます。

```
//
// OrderedSet, CyclicSet のメンバ関数
//
Element first();    // 最初の要素を返す
Element last();     // 最後の要素を返す
Element next(const Element& i); // 要素 i の次の要素を返す
Element prev(const Element& i); // 要素 i の前の要素を返す
int position(const Element& i); // 要素 i の場所(最初は 1) を返す
Element elementAt(int p); // 場所 p にある要素を返す
```

これらのメンバ関数と、要素と集合から生成される条件式を使用することによって、要素についての for や while ループが実現されます。詳細は 3.4.11 イテレータの表現(順序付き集合のメンバ関数)を参照して下さい。代入によって初期化されていない集合オブジェクトは空集合となります。

3.4.1 宣言と引数の意味

他のオブジェクトと同様、集合は宣言すると生成されます。

```
Set S;
OrderedSet I;
CyclicSet C;
```

集合にはコピーコンストラクタ¹⁶が定義されており、宣言時に初期値を文字列として与えることができます。

¹⁶宣言と値の設定を同時に行うものです。

```
Set S = "a b c ";          // a b c という要素を持つ集合
OrderedSet I = "1 2 3"; // 1 2 3 という要素を持つ集合
CyclicSet C = "d e f"; // d e f という要素を持つ集合
```

コピーコンストラクタに与える文字列の書式はデータファイルと同一です。データファイルの書式については **4 データファイル** を参照して下さい。また、

```
Set S(name = "points");
Set T(name = "neighbours", index = S);
Set U(name = "adjacent", index = i);
```

のように引数付きの宣言によって、生成されるインスタンスへと付加情報を与えることもできます。クラス `Set`、`OrderedSet`、`CyclicSet` について引数付き宣言に渡す引数の書式と意味は次の様に定められています。以下に示す以外の書式の引数を与えることはできません。

◆ `name = "文字列"`

生成されるオブジェクトの名前を定めます。名前はオブジェクトが外部に参照される際のラベルとして用いられ、外部からデータファイルによってデータを与える場合には必ず必要になります。データファイルからのデータ入力については **4 データファイル** を参照して下さい。

```
Set S(name="points");
```

この例では、`points` という名前を持つ `S` というオブジェクトを生成し、このオブジェクトが外部からこの名前で認識できるようにしています。

◆ `index = 添字集合並び`

オブジェクトが添字付けられる範囲 (添字集合) を定めます。`index=` の右辺の「添字集合並び」の表す集合が添字集合となります。添字集合並びは全体として集合を示すもので、集合または要素オブジェクト自身

集合/要素オブジェクト

あるいは集合または要素オブジェクトを、(コンマ) 演算子で連結した

(集合/要素オブジェクト₁, ..., 集合/要素オブジェクト_n)

という書式を持ちます。

集合並びに現れた集合オブジェクトはその集合そのもの、要素オブジェクトはその定義集合(要素の動く範囲)を表します。、(コンマ)で区切って並べた全体は、集合の直積を表します。

```
Set N(name = "neighbours",index = S);
```

この例では、集合 S を添字集合として持つ (S の要素で添字付けられる) N という Set のオブジェクトを生成しています。

ここで、 S を定義集合とする要素を使用して

```
Element i(set=S);  
Set N(name = "neighbours",index = i);
```

としても意味は同じです。

```
Set S,T;  
Set V(index = (S,T));
```

この例では集合 S と集合 T の直積を添字集合として持つ Set のオブジェクト V を生成しています。

添字集合並びに要素集合と集合を

```
Element i(set=S);  
Set V(index = (i,T));
```

のように混在させることも可能です。

添字付けされた集合を添字集合並びに使用する際には添字集合並びが全体で完結している(他の要素に依存していない)ことが必要です。詳細については 3.4.53.4.5 を参照して下さい。

◆ dim = 整数値

集合に属する要素の次元を設定します。省略すると次元は 1 になります。集合の内容を文字列やデータファイルから入力する場合、要素並びはここで与えた次元に従って解釈されます。

```
Set S;
S = " 1 2 3 4 "; // S = 1 2 3 4 となる
Set T(dim = 2);
T = " 1 2 3 4 "; // T = (1,2) (3,4) となる
```

この引数によって指定された要素と異なる次元の要素を格納することはできません。

♦ superSet = 集合

集合は他の集合の部分集合となっている場合に、その親集合 (superSet) を指定します。詳細は 3.4.9 集合同士の包含関係を参照して下さい。

なお、コンストラクタの引数の順番は任意です。例えば

```
Set T(index = i, name = "neighbours");
```

を

```
Set T(name = "neighbours", index = i );
```

としても同じ意味になります。

3.4.2 集合 (Set) 同士の演算

集合演算を用いて新たな集合を定義することができます。代入は C++における場合と同様、すべて逐次的に解釈され、左辺は右辺のその時点における内容となります。それ以前に定義した式へと代入の影響がさかのぼることはありません。

集合同士の演算には

& (積集合)	(和集合)	- (差集合)	* (直積)
---------	-------	---------	--------

があります。

```

Set S = "a b c";
Set T = "c d ";
Set U,ST(dim=2);

U= S & T ; // U  = { c }
U= S | U ; // U  = { a b c d }
ST = S * T ; // ST = { (a,c) (a,d) (b,c) (b,d) (c,c) (c,d) }

```

演算の両辺に `OrderedSet` や `CyclicSet` が現れた場合にも ,これらの演算で生成されるオブジェクトはいずれも `Set` です .これは ,上記の演算によって生成される集合の順序付けが定義できないことに対応しています .

また ,要素並びを記述した文字列を代入することによって ,集合の内容を与えることができます .

```

Set S;
Set T(index = S);
S = "1 2"; // S = { 1 2 }
T = "[1] a b c [2] e f"; // T[1] = { a b c } T[2] = { e f }
T[3] = " g h "; // これも合法

```

要素並びの書式は集合の内容をデータファイルから与える場合と同じです .データファイルからの集合の内容の与え方と要素並びの書式についての詳細は **4 データファイル**を参照して下さい .しかし ,上記の例で

```

Element i(set=S),j(set=S);
Set R(index = (i,j));

R[1] = "[2] 1 2"; // エラー

```

のように ,等号の両側に `[]` が現われるのは実行時エラーとなります .

3.4.3 集合 (Set) の添字付け

式を構成するオブジェクトと同様に ,宣言において添字集合を与えた集合については添字集合の要素で添字付けることができます .

添字付けされた集合によって添字集合の要素から 集合への射像を表現することができます .

```

Set S = "1 2";
Set T(index=S); // T の添字集合を S とする
Element i(set=S); // i は T の添字集合 S を定義集合とする要素

T[1] = " a b "; // T の要素 1 に集合 a b が対応
T[2] = " c d "; // T の要素 2 に集合 c d が対応

```

添字が代入文の左辺に現れると添字は展開されて解釈されます。

```

Set U = " x y z ";
T[i] = T[i] | U; // U の内容を T[i] に加える:
// T[1] = (T[1] U) = { a b x y z }
// T[2] = (T[2] U) = { c d x y z }

```

また、添字付けされた集合を要素の定義集合として

```

Element j(set = S[i]); // j は S[i] の要素

```

のように使用したり、オブジェクトのコンストラクタ中の添字集合並びとして

```

Element i(set=N);
Set neighbours(index=i);
Parameter flux(index=(i,neighbours[i]));

```

のように使用することができます。添字付きの集合を添字集合並びに使用する場合について、詳しくは 3.4.5 添字付き集合の利用を参照して下さい。

3.4.4 添字(Element)と集合(Set)の対応

Element は式の添え字として用いられますが、その宣言を行うとき特定の Set に結び付けられます。要素にとって結び付けられた集合のことを、要素の定義集合と呼びます。その要素が代入文の左辺や制約式に現れた場合、定義集合の内容にわたって展開されます。

```

Set S = "1 2"; // 集合 S = { 1, 2 }
Element i(set = S); // S を定義集合とする要素 i を生成
Expression f(index = i);
Parameter a(name = "coef", index = i);
Variable x(index = i);
f[i] = a[i]*x[i]; // i は S の内容に渡って展開される

```

定義集合への操作は Element へと逐次影響します。例えば上の例で

```
S.add("3"); // S に 3 を加える
```

とすると、それ以後において i の展開される範囲は変化し、

```
x[i] >= 2 ; // i = 1, 2, 3 と展開される:
           // x[1] >= 2;
           // x[2] >= 2;
           // x[3] >= 2; と等価
```

となります。

3.4.5 添字付き集合の利用

以下の例のように、添字付けた集合 `neighbours` も他の集合オブジェクトと同様、オブジェクトの添字集合とすることができます。

```
Element i(set=N);
Set neighbours(index=i);
Parameter flux(index=(i,neighbours[i]));
```

その際、添字 `index` の指定は全体で完結している (別の要素に依存していない) ことが必要です。例えば、`flux` の `index` の指定を

```
Parameter flux(index=(neighbours[i]));
```

とすることはできません。`neighbours[i]` は集合オブジェクトですが、外で決定される要素 i に依存しておりますので、単独で完結した集合を示さないからです。これは要素オブジェクトによって添字集合を指定する場合も同様です。

添字付きの集合が定義集合として与えられた要素は他の要素に間接的に依存することになります。例えば、

```
Set S = "a b"; // 集合 S = { a, b }
Set U(index = S); // 集合 U は a, b によって添字付けられる
Element i(set = S);
Element j(set = U[i]); // U[i] を定義集合とする要素 j の生成
```

では i が決定して初めて j が動く範囲が決定されますので j は i に間接的に依存しています。

```
Parameter p(index = j); // j の範囲は不定 (エラー)
```

このような要素を単独では添字集合として使用することはできません．通常の要素オブジェクトと異なり， j 単独では完結した集合を示さないからです．使用すると以下のように実行エラーになります．

これに対して，

```
Parameter a(index = (i,j)); // (i,j) (i と j の直積) の範囲は定まる
```

は可能です． i と j との直積は，全体で完結した集合を示しているからです．

3.4.6 添字集合への自動代入

SIMPLE では代入の左辺に現れた添字は，そのオブジェクトの添字集合に必ず含まれるものと考えます．もし，そのオブジェクトの添字集合が代入の左辺に現われた要素を含まない場合には，添字集合に自動的に要素の追加が行われます．

例えば

```
Set S;
Element i(set = S);
Parameter a(index = S); // 添字集合の定義
```

という場合には a の添字集合は S ですが，ここで

```
a[1] = 1.0; // S には 1 が追加される
a[2] = 2.0; // S には 2 が追加される
```

とすると， S には要素 1 2 の追加が行なわれ， i の示す範囲もそれに従います．

添字集合が複数の構成要素から成っている場合には，要素の追加は添字集合の構成要素へと行われます．例えば

```
Set Node;
Parameter edgeLength(name = "edgeLength", index = (Node, Node));
```

とあり，`edgeLength` が入力データから

```
edgeLength = [1,3] 2.0 [2,4] 4.0 ; // edgeLength のデータ
```

と与えられる場合,このデータが読み込まれた時点(edgeLength が生成された時点)で Node の内容は 1 2 3 4 となります。データファイルの書式については 4 データファイルを参照して下さい。

ただし,自動追加は,添字集合あるいはその構成要素となっている集合の内容が一意に決定される以下の場合にのみ適用されます。

◆ 添字集合が集合オブジェクトそのものである場合

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

Parameter f(index = S); // f[1]=1; で, 1 を S に追加
Expression g(index = i); // g[1]=1; で, 1 を T に追加
```

◆ 添字集合が集合オブジェクトの直積である場合

```
Set S,T;
Set ST(dim = 2);
ST = S*T;
Element i(set=ST);

Variable x(index = (S,T)); // x["1,2"]=1; で, 1 を S ; 2 を T に追加
Variable y(index = i);      // y["3,4"]=2; で, (3,4) を ST に追加
                             // (この時, S と T は変わらない)
```

◆ 添字集合が集合オブジェクトの積集合である場合

```
Parameter b(index = (S & T)); // b[1]=1; で, 1 を S と T 両方に追加
```

しかし,次のように添字集合が集合オブジェクトの和集合である場合には添字集合の構成要素になっている集合(ここでは S, T)の内容が一意に決定されませんので,要素の自動追加は行われません。

```

Set S = "1 2 3";
Set T = "3 4 5";
Parameter c(index = (S|T));
Set V = "1 2";
Element i(set = V);    // i の動く範囲は{ 1 2 } となる

c[i] = 1;              // i は S|T の範囲内なので合法
V = "6 7";            // i の動く範囲は{ 6 7 } となる
c[i] = 2;              // i は S|T の範囲外，しかも自動追加は行われないのでエラー

```

3.4.7 自動代入の禁止 (lock()/unlock())

自動追加は集合のデータを陽に与える必要を省き効率的ですが、誤ったデータを与えた際に、それによって集合の内容が変わってしまいますので、例えば以下のように、モデルにとって悪影響を与える場合があります。

```

Set S = "1 2 3";
Parameter c(index=S);
Element i(set = V);
c = "[1] 7.2 [2] 8.2 [5] 1.2 "; // 誤って添字 5 を与える
                                // ここで S は "1 2 3 5"となってしまう
Variable x(index=i);
sum(x[i],i) >= 5; // 和の取られる範囲が変わってしまう

```

そのような場合に有効なのは集合に対する lock() という操作です。集合に対して lock() を呼び出すと、以降の自動追加を禁じます。例えば上記の例で

```

Set S = "1 2 3";
S.lock(); // 集合をロック
Parameter c(index=S);
Element i(set = V);
c = "[1] 7.2 [2] 8.2 [5] 1.2 "; // 誤って添字 5 を与える
                                // ここで S は "1 2 3 5"となってしまうがここでエラー

```

となるので、データのチェックが容易です。lock() の逆（自動追加を許す）の操作として unlock() という手続きもあります。

3.4.8 集合と要素から生成される条件式

要素オブジェクトと集合オブジェクトを条件付けの演算子

$$< \quad >$$

でつなげたものも条件式と見なされます．この条件式は集合と要素の包含関係を示します．例えば i を Element , S を Set とするとき

$i < S$	$S > i$	\rightarrow	i は S に含まれる
$i > S$	$S < i$	\rightarrow	i は S に含まれない

という意味となります．以下はこの条件式の使用例です．

```
Set N;
Element i(set=N);
Set boundary(name = "boundary");
Variable x(index = N);

x[i] = 0 , i < boundary; // boundary に含まれる i について
x[i] = 2 , i > boundary; // boundary に含まれない i について
```

3.4.9 集合同士の包含関係

集合同士の包含関係は集合を定義するときのコンストラクタの引数として `superSet` を与えることによって記述されます。

```
Set S;
Set T(superSet=S); // T が S の部分集合であることを示す
```

`superSet` の右辺には 1 つまたは複数の集合オブジェクトを記述することができます。複数の集合オブジェクトを記述するのは直積集合の部分集合を定義する場合です。

```
Set S,T;
Set ST(dim=2,superSet=(S,T)); // ST は S と T の直積集合の部分集合
                                // (S,T) の代わりに S*T としても同様
```

包含関係を指定し、包含されている集合へ要素を追加すると、包含している集合へも自動的に要素が追加されます。

```
Set S;
Set T(superSet=S); // T が S の部分集合であることを示す
T = "1 2 3";       // S にも"1 2 3" が代入される
```

包含している集合が複数の集合から構成されるときには要素の追加は構成要素に対して行われます。

```
Set S,T;
Set ST(dim=2,superSet=(S,T)); // ST は S の直積集合の部分集合

Element i(set=ST);
Parameter c(index = i);

c["1,2"] = 1; // "1,2" を ST に追加, 更に, 1 を S; 2 を T に追加
c["3,5"] = 2; // "1,2" を ST に追加, 更に, 3 を S; 5 を T に追加
```

ただし、自動的に要素が追加されるのは 3.4.6 添字集合への自動の場合と同じく、包含している集合自身、あるいはその構成要素の内容が一意に決定する以下の場合のみです。

- ◆ 親集合 (superSet の右辺) が集合オブジェクトそのものである場合

```
Set S;
Set T(superSet=S);
```

- ◆ 親集合が集合オブジェクトの直積である場合

```
Set S,T;
Set ST(dim=2,superSet=(S,T));
```

- ◆ 親集合が集合オブジェクトの積集合である場合

```
Set S,T;
Set U(superSet=(S&T));
```

親集合の構成要素の内容が一意に決定できない場合，例えば以下の例の様に親集合が和集合である場合には要素の自動追加は行われず，その関係に違反する代入が行なわれると実行時エラーになります．

```
Set S,T; S=" 1 2 "; T="2";
Set U(superSet=(S|T));

U = " 1 2 3 "; // S,T への自動追加は行われない
```

しかし，実装の効率上の理由により，親集合を代入操作によって縮小した場合には，集合の包含関係 $\text{Set } T(\text{superSet}=S);$ を定義しても，常に $T \subseteq S$ が成り立つとは限らない場合があります．

```
Set S, T(superSet=S);

T = "1 2 3"; // この時点で S にも"1 2 3" を追加して S=T={1 2 3} となる
S = "1 2"; // S のメンバを減らした
//しかし T は"1 2 3"のままで T S は保たれない (S={1 2};T={1 2 3})
T.add(4); // ただし，その後も自動追加は行なわれる：
// T に 4 を追加， S にも 4 を追加：
// T={1 2 3 4}; S={1 2 4} となる
```

実装上, Set T (superSet= S) という定義は, 「 T に対する要素の追加は常に S の要素の追加にもなる」ということと等価になります.

3.4.10 Set の操作(要素数を知る, 要素の追加する, など)

集合オブジェクトのメンバ関数

```
//
// Set, OrderedSet, CyclicSet に共通なメンバ関数
//
int contains(Element& i); // 要素 i を含んでいるかどうか
int card();               // 要素の数を返す
```

を使用して集合の内容についての問い合わせを行うことができます.

```
Set S(name="S");
Element i(set=S);
Variable x(index=i);
Objective f;
sum(x,i) <= S.card();
if( S.contains("a") ){
    f = x["a"];
}else{
    f = x["b"];
}
```

また以下のメンバ関数は集合への要素の追加や削除を行います.

```
//
// Set, OrderedSet, CyclicSet に共通なメンバ関数
//
add(要素オブジェクト[, 条件式]); // 要素の追加
remove(要素オブジェクト[, 条件式]); // 要素の削除
```

`add()` / `remove()` の引数である要素オブジェクトは単一の要素を指す場合に限りません. 引数に与えられた要素オブジェクトが単一の値に限定されていない場合にはそれを展開したものが追加されます.

```

Set S = "1 2 3";
Set T = "4 5 6";
Element i(set = S);
S.add(7);    // S には 7 が追加される
T.add(i);    // T には{1 2 3 7} が追加される

```

条件式を 2 番目の引数として与えると、引数である要素オブジェクトの展開される範囲が限定されます。例えば上の例では

```

Set U;
U.add(i,i<3);    // U には{ 1 2 } が追加される

```

となります。また次のメンバ関数 `slice()` を用いて、多次元の要素よりなる集合の要素それぞれから一部分を取り出し、新しい集合を構成することができます。

```

slice(int, ..., int)

```

引数である `int` は 1 からはじまる整数で、集合の要素を構成する直積のメンバを左から数えた順番を表します。例えば、

```

Set a(name="a", dim=3); // a の要素は 3 つのメンバで構成される
a = "1,1,1 1,3,1 b,2,1 1,3,1 1,4,a 1,1,1 1,5,1"; // 内容設定
Set a1(name="a1");
a1 = a.slice(1);        // a の要素の 1 番目のメンバで構成される集合
a1.val.print()          // 結果: a1=(1 b)
Set a12(name="a12",dim=2);
a12 = a.slice(1,2);     // a の要素の 1,2 番目のメンバで構成される集合
a12.val.print()          // 結果: a12=(1 1 1 3 b 2 1 4 1 5)
Set a121(name="a121",dim=3);
a121 = a.slice(1,2,1);  // 要素の 1,2,1 番目のメンバで構成される集合
a121.val.print();        // a121=(1 1 1 1 3 1 b 2 b 1 4 1 1 5 1)
Set a13(name="a13",dim=2);
a13 = a.slice(1,3);     // 要素の 1,3 番目のメンバで構成される集合
a13.val.print();         // a13=(1 1 b 1 1 a)

```

となります。

3.4.11 イテラタの表現(順序付き集合のメンバ関数)

集合の要素にわたるループを表現する場合には, `OrderedSet`, `CyclicSet` のメンバ関数

```
//
// OrderedSet, CyclicSet のメンバ関数
//
Element first();    // 最初の要素を返す
Element last();     // 最後の要素を返す
Element next(const Element& i); // 要素 i の次の要素を返す
Element prev(const Element& i); // 要素 i の前の要素を返す
int position(const Element& i); // 要素 i の前の場所を返す
Element elementAt(int p); // 場所 p にある要素を返す
```

を使用し, 例えば以下のように記述します.

```
OrderedSet S = "1 2 3";
Element i;
// s 中のすべての要素に関するループ
for ( i = S.first() ; i < S ; i = S.next(i) ) {
    ...
}
// s 中のすべての要素に関するループ(上と等価)
for ( int p = 1 ; p <= S.card() ; ++p ) {
    i = S.elementAt(p);
    ...
}
```

`i` が最後の要素を示す場合, `OrderedSet` の `next(i)` の戻り値はヌル要素値になり, `CyclicSet` の `next(i)` の戻り値は最初の要素(`first()`の戻り値)になります.

3.4.12 要素からの集合の生成(関数: `setOf()`)

次に示す仕様の関数 `setOf` を使用すると要素オブジェクトからその要素の動く範囲に対応する集合オブジェクトを生成することができます.

```
Set setOf(要素[, 条件式]); // 要素の動く範囲の集合を返す
```

2 つ目の引数として条件式を付加した場合には, 要素の動く範囲をその条件式によって限定

した場合の結果を返します。

```
Set S;
Element i(set=S);
Parameter a(name="a", index = S);

Set T;
T = setOf(i, a[i]>0); // T = { i | a[i] > 0 }
```

setOf の戻り値は通常の集合オブジェクト Set と同じように要素の定義集合やオブジェクトの添字集合に使用することができます。

```
Element j(set = setOf(a[i] < 0));
// j は集合{ i | a[i] < 0 }の要素
Variable x(index = setOf(a[i] > 0));
// x の添字集合は{ i | a[i] > 0 }
```

3.4.13 数列 (Sequence)

数列 (Sequence) の要素は常に定数値で from と to の間に by 分の増減量で定義されます。数列は集合の特別なもので、通常の集合と同じように Element の定義として使用することができます。

数列の仕様は次の通りです。

```
Sequence a(from=定数, to=定数, by=定数);
```

by は省略することもでき、そのときには 1 と見なされます。ここで「定数として用いることができるのは int, double, 添字を持たない Parameter, またはその組み合わせです。たとえば次のようになります。

```
Sequence a1(from=0, to=3);
// a1 = { 0, 1.0, 2.0, 3 } となる
Sequence a2(from=0, to=3, by=0.5);
// a2 = { 0, 0.5, 1, 1.5, 2, 2.5, 3 } となる
```

普通の集合に対して可能な代入操作などは、数列に対しては無効となります。数列には次のようなメンバ関数があります。

```
int card();           //要素数
Element first();      //最初の要素を返す
Element last();       //最後の要素を返す
Element next(Element); //要素の次の要素を返す
Element prev(Element); //要素の前の要素を返す
int position(Element); //要素の順番数を返す
Element elementAt(int); //int 番目の要素を返す
```

次はその使用例です。

```
Sequence sq(from=0, to=3, by=0.5);
sq.card();           //要素数: 7 を返す
sq.first();          //最初の要素 0 を返す
sq.last();           //最後の要素 3 を返す
sq.next(1.0);        //要素 1.0 の次の要素 1.5 を返す
sq.prev(1.0);        //要素 1.0 の前の要素 0.5 を返す
sq.position(1.0);    //要素順番数 3 を返す
sq.elementAt(5);     //5 番目の要素 2 を返す

Element i(set=sq);
Parameter p(index=sq);
p[i] = i;            // p[0]=0, p[0.5]=0.5 ... となる
```

3.4.14 連続範囲(Interval)

数列と同様に、連続範囲(Interval)も集合の特別形態と見なすことができます。ただし、連続範囲の要素数は、通常無限個あるので、Element の定義集合やオブジェクトの添字集合としては使えません。連続範囲は、すでに他の集合の上に定義された添字の範囲をさらに限定するために使われます。

連続範囲の指定は、double の上に left (または oleft) と right (または oright) で構成されます。範囲の定義引数を次に挙げます。

```
Interval a([o]left, [o]right);
left:   範囲の下限 (closed: left を含む)
oleft:  範囲の下限 (open:   left を含まない)
right:  範囲の上限 (closed: left を含む)
oright: 範囲の上限 (open:   left を含まない)
```

範囲として次のようなメンバ関数があります .

```
int contains(Element); // (=1) 存在; (=0) 存在せず
```

以下は使用例です .

```
Interval it1(left=0,right=1); // 範囲 : [0,1]
Interval it2(left=0,oright=1); // 範囲 : [0,1)
Interval it3(oleft=0,right=1); // 範囲 : (0,1]
Interval it4(oleft=0,oright=1); // 範囲 : (0,1)

it1.contains(0); //範囲内, 1 を返す
it1.contains(0.356); //範囲内, 1 を返す
it1.contains(1); //範囲内, 1 を返す
it2.contains(0); //範囲内, 1 を返す
it2.contains(0.356); //範囲内, 1 を返す
it2.contains(1); //範囲外, 0 を返す
it3.contains(0); //範囲外, 0 を返す
it3.contains(0.356); //範囲内, 1 を返す
it3.contains(1); //範囲内, 1 を返す
it4.contains(0); //範囲外, 0 を返す
it4.contains(0.356); //範囲内, 1 を返す
it4.contains(1); //範囲外, 0 を返す

Set S(name="S");
S = "0.5 1.5 0.8";
Element i(set=S);
Parameter p(index=S);
p[i] = i, i<it1; // p[0.5]=0.5, p[0.8]=0.8 となる
```

3.4.15 グラフ(Graph)

ネットワーク問題などのモデルを記述するには、グラフ構造を使用すると便利です。SIMPLEでは、グラフは2つの集合 `node (dim=1)` と `arc (dim=2)` によって構成され、次のような形で定義されます。

```
Set n(name="n");
Set a(name="a",dim=2);
Graph g(nodes=n,arcs=a);    // グラフの定義
Element i(set=n);
```

グラフの定義引数の仕様は以下の通りです。

```
Graph g(name=文字列,arcs=集合,nodes=集合);
```

`name` は 3.3.1 宣言と引数の意味で述べたように、データファイルのラベル名となります。グラフをデータファイルから定義するときは、次のように行います。

```
//モデルの中の定義
Graph g(name="network");

//データファイルでの記述
network.nodes = 1 a b 2 3;
network.arcs   = 1,a b,2 1,2;
```

グラフのメンバ(参照可能な値)と、メンバ関数としては

<code>nodes</code>	:dim=1 の集合, グラフの節点
<code>arcs</code>	:dim=2 の集合, グラフの辺
<code>in(Graph,i)</code>	:グラフの節点 <code>i</code> へ進入する辺の集合
<code>out(Graph,i)</code>	:グラフの節点 <code>i</code> から出発する辺の集合

があります。グラフを用いた問題の記述例については、2.4 ネットワーク最適化問題(最小費用流)を参照して下さい。

4. データファイル

SIMPLE のオブジェクトの具体的な内容は

```
Variable x;
Parameter a;
Set S;

x = 2;           // 変数の初期値の設定
a = 3;           // 定数の値の設定
S = " 1 2 3 ";   // 集合の内容の設定
```

等とコード中に代入を記述する以外に、データファイルから与えることができます。単一のモデルに対するデータが複数ある場合や、モデルが大規模な場合にはデータファイルから与えるのが便利です。データファイルの形式は

1. SIMPLE に固有の形式 (SIMPLE データファイル)
2. CSV 形式 (コンマ区切りの表)

のいずれかを用いることができます。2. は数値データのみを与えることができる、補助的な形式です。UNIX/LINUX 版、あるいは DOS プロンプトから Windows 版をお使いの場合には、ファイルの拡張子が CSV/csv であるときに 2. の形式だと判定されます。Windows 版の GUI では GUI 上のデータアイコン自体がこのどちらかの属性を持ちます。

また、特例として SIMPLE に固有の形式ではデータファイルから NUOPT のパラメータを与えることができます。それについては 4.1.4 データファイルを使った NUOPT のパラメータの定義をご参照下さい。

4.1 SIMPLE 形式データファイル

データファイルからデータを与える場合には、宣言の引数の書式

```
name = "文字列"
```

によって、オブジェクトに名前を与え、その名前を使用してデータファイルに

```
名前= 数値データ (または集合の要素データ);
```

と記述します。名前は、データファイルとモデルに書かれたオブジェクトとの対応を取るた

めに使用されます。ただし、Windows 版をお使いの場合には名前を陽に与えなくとも、オブジェクトの名前そのものが `name =` の後に与えられているのと同じになります。そのため、必ずしも `name = "文字列"` として宣言する必要はありません。

データファイルの内容はオブジェクトの生成（宣言された時点）時に初期値として代入されます。

データファイルの一般的な書式は以下の通りです。

```
// コメント
/* コメント */
名前 1 = 数値データ（集合の要素データ） 1 ;
名前 2 = 数値データ（集合の要素データ） 2 ;
...
名前 n = 数値データ（集合の要素データ） n ;
```

データファイル中、`/*` と `*/` で囲まれた部分や `//` 以降から行末まではコメントと見なされます。データ記述は、`;`（セミコロン）で区切られた記述までが1つのオブジェクトに対するものです。二重引用符`"`で囲まれていないスペース、タブ、改行、`,`（コンマ）は値の区切りと見なされます。

「数値データ」は `Parameter`、`VariableParameter`、`Variable`、`IntegerVariable` の値、「集合の要素データ」は `Set/OrderedSet/CyclicSet` の内容を具体的に記述した部分です。「数値データ」の書式の詳細については 4.1.1 数値データに、「集合の要素データ」の書式の詳細については 4.1.2 集合の要素データにそれぞれ記述があります。

上記の例では

```
Variable x(name="x");
Parameter a(name="a");
Set S(name="S");
```

としてオブジェクトに名前を付け、データファイルに

```
x = 2;          // x の値
a = 3;          // a の値
S = 1 2 3;      // S の内容
```

と書くとオブジェクトに値が与えられます。データファイルの値はオブジェクトが生成（宣言）された際にオブジェクトの初期値として与えられます（オブジェクトの初期値が与えられない

式オブジェクトの初期値は 0 , 集合オブジェクトの初期値は空集合です) .

4.1.1 数値データ

Parameter, Variable, IntegerVariable, VariableParameter に与える数値データを記述する書式です . 添字付けられないオブジェクトに対する値並びの書式は

値

です . 添字付けられる (添字集合を与えられている) オブジェクトに対する値並びの書式は

[添字値 1] 値 1 [添字値 2] 値 2 ... [添字値 n] 値 n

で , 添字の値とその添字に対するオブジェクトの値を交互に書きます . 二重引用符 " で囲まれていないスペース , タブ , 改行 , , (コンマ) は無視されます .

例えば

```
Set S;
Parameter a(name = "coef", index = S);
Variable x(name = "amount", index = S);
```

とした場合 , データファイルを

```
coef    = [1] 2.1 [2] 3.5 [3] 7.2 ;
amount  = [1] 3.0 [2] 5.0 [3] 1.0 ;
```

と書くと ,

```
Set S;
Parameter a(index = S);
Variable x(index = S);

a[1] = 2.1; a[2] = 3.5; a[3] = 7.2;
x[1] = 3   ; x[2] = 5   ; x[3] = 1;
```

と等価になります .

```
"this is a value"  "this value contains space" " a,b,c "
```

二重引用符で囲まれた添字値や値は必ず文字列として解釈されます．それ以外の添字値や値が数字へと変換できる場合には数字として解釈されます．特に整数値に変換可能な場合には，整数値に変換されて解釈されます．

データファイルの記述	解釈
1.0234	double 値
2.00	int 値
"2.234"	文字列

表 5 データファイルの記述と解釈

4.1.2 集合の要素データ

Set/OrderedSet/CyclicSet の要素の値を記述する書式です．添字付けられない集合オブジェクトに対する要素並びの書式は

```
要素 1  要素 2  ...  要素 n
```

です．添字付けられる (添字集合を与えられている) 集合オブジェクトに対する要素並びの書式は

```
[添字値 1]  要素 1.1  ...  要素 1.n1  [添字値 2]  要素 2.1  ...
```

です．添字の値とその添字に対する集合の要素の並びを交互に書きます．二重引用符"で囲まれていないスペース，タブ，改行，,(コンマ)は値の区切と見なされ，無視されます．例えば

```
Set S;  
Set T(name = "T",index = S);
```

とした場合，データファイルを

```
T = [a] 1 2 3 [b] 4 5 ;
```

と書くと，

```
Set S;
Set T(name = "T", index = S);

T["a"] = " 1 2 3 ";
T["b"] = " 4 5";
```

と記述した場合と等価になります．要素並びは宣言時に与えられた集合の次元に従って解釈されます．例えば

```
Set S(name = "S");
Set T(name = "T", dim = 2);
```

とするとデータファイルは以下のように解釈されます．

```
S = 1 2 3 4 ; // S = { 1 2 3 4 } となる
T = 1 2 3 4 ; // T = { (1,2) (3,4) } となる
```

連続する整数値の並びを表現する場合には以下の略記法が用意されています．

```
S = 1 .. 100; // S は 1 から 100 の整数の集合

S = 1 ... 100; // S は 1 から 100 の整数の集合 (上に同じ)
```

4.1.3 Wild Card を使ったデータの記述

大規模データの定義を簡便化するために，数値データと集合の要素データに関して，Wild Card を使ったデータ記述を行うことができます．先頭に<*>のようなパターンを定義すると，その後のデータは * にあてはめて解釈されます．

以下は宣言

```
Set R,S,T;
Set SS(dim=2,name="SS");
Set SSS(dim=3,name="SSS");
Set U(name="U", dim=3, index=R);
```

に対する Wild Card を使ったデータファイルの例です．

```

SS   = <1*> 1 .. 3 <*2> 4 .. 5;
                                     //SS={ (1,1), (1,2), (1,3), (4,2), (5,2) }

SSS = <1*2> 10 15 20;
                                     // SSS={ (1,10,2), (1,15,2), (1,20,2) }

U   = [1] <1*2> 10 11 12 [2] <1 2*> 20 21;
                                     // U[1]={ (1,10,2), (1,11,2), (1,12,2) }
                                     // U[2]={ (1,2,20), (1,2,21) }

```

また, [] の中にも * を含むことができます. 以下のようなコード

```

Set R,S,T;
Parameter a(name="a",index=R);
Parameter b(name="b",index=(R,S));
Parameter c(name="c",index=(R,S));
Parameter d(name="d",index=(R,S));
Parameter e(name="e",index=(R,S,T));
Parameter f(name="f",index=(R,S,T));

```

に対して, [] 内の * を使ったデータファイルの例を挙げます.

```

a = [*] 1 7 2 8 3 9;      // a[1]=7; a[2]=8; a[3]=9
b = [1*] 1 3 2 4 3 5;    // b[1,1]=3; b[1,2]=4; b[1,3]=5
c = [*7] 1 3 2 4 3 5;    // c[1,7]=3; c[2,7]=4; c[3,7]=5
d = [1*] <*100> 1 2 3;    // d[1,1]=100; d[1,2]=100; d[1,3]=100
e = [1*2] 8 1 9 2 10 3;  // e[1,8,2]=1; e[1,9,2]=2; e[1,10,2]=3
f = [1*2] <*0.1> 3 .. 5; // f[1,3,2]=0.1; f[1,4,2]=0.1; f[1,5,2]=0.1

```

上記例の中の d,e,f の設定のように, 各書式を混在させて使うこともできます. その時に, 書式を処理する優先順位は次のようになります.

1. ... あるいは ... の展開
2. <*> の展開
3. [*] の展開

例えば, データ [1*2] <*0.1> 3 .. 5 は次の段階を踏んで展開されます.

```

step1:    [1*2] <* 0.1> 3 4 5
step2:    [1*2] 3 0.1 4 0.1 5 0.1
step3:    [1 3 2] 0.1 [1 4 2] 0.1 [1 5 2] 0.1

```

4.1.4 データファイルを使った NUOPT のパラメータの定義

特殊なデータ名として

`nuopt.prm`

という名前が定義されており、この内容（文字列）の設定は NUOPT のパラメータファイルの内容であると解釈されます。具体的にはデータファイル中に

```

nuopt.prm = "
begin
method:simplex
end
";

```

のように書くとこの値の文字列と等価な NUOPT のパラメータファイルを与えたこととなります。

4.2 csv 形式のデータファイル

CSV 形式のデータファイルは数値データのみを与えることができる形式です。主に Microsoft Excel からのファイルとの親和性を目的としています。CSV 形式データの書式は

例 1 :

```

a
5

```

例 2 :

```

i,a,b,c,d
1,1.5,2.8,7.2
2,3.5,4.2,9.3

```

例 3 :

```

Q,1,2,3
1,2.7,8.3,9.1
2,1.2,9.1,2.5
3,3.5,6.2,7.3
4,8.1,4.5,6.2

```

のように、コンマで区切られた行からなっています。コンマで区切られた間の部分をフィールドと呼びます。フィールドの数はすべての行で同じである必要があります（例 1 のケースではフィールドの数はすべて 1、例 2 は 5、例 3 は 4 です）、同じでない場合にはエラーになります。各

```
1, 2.7, 8.3, 9.1
```

フィールドの前後の空白は無視されます。例えば例 2 の 2 行目は

と書いても等価です。

行がコンマで終わっている場合には最後に空のフィールドがあると解釈されます。

```
1,2.7,8.2, // 最後のフィールドが空
```

コンマが連続している場合には空のフィールドがあると解釈されます。

```
1,,3,5 // 2 番目のフィールドが空
```

空のフィールドは 1 行目にのみ許され、それ以外の場所に現れるとエラーになります。CSV 形式のデータにも、SIMPLE 形式のデータと同じく、

```
// コメント
```

```
/* コメント*/
```

という形のコメントを任意の場所に入れることができます。

```
//コメント
Q,1,2,3 // コメント
1,2.7,8.3,9.1
2,1.2, /* コメント */ 9.1,2.5
3,3.5,6.2,7.3
4,8.1,4.5,6.2
```

4.2.1 1D 書式

モデル中で同一の添字を持つ複数のパラメータが定義されている場合があります。例えば

```
Set S;
Element i(set=S);
Parameter lower(name = "lower",index = i);
Parameter upper(name = "upper",index = i);
Parameter initial(name = "initial",index = i);
```



```

Set S,T;
Element i(set=S),j(set=T);
Parameter cost(name = "cost",index = (i,j));
Parameter slack(name = "slack",index = (i,j));
Parameter demand(name = "demand",index = j);

```

のようなケースですと cost, slack の添字は 2 つですので , 以下の CSV 形式データで値を与えることができます .

i, j,	cost, slack
1, A,	9.2, 7.6
1, B,	3.5, 5.0
1, C,	7.2, 8.3
2, A,	2.1, 7.2
2, B,	1.1, 2.2
2, C,	0.1, 6.2

データの名前

添字

添字の列の先頭には「i,j」と書かれていますが, 添字の列の最初のフィールドは空白を含む任意の文字が許されます . SIMPLE の処理系の解釈は, この名前のマッチングではなく, 添字がいくつあるかという数で決定されることにご注意下さい . このため, 1 行目を

```
, , cost,slack
```

と書いて, 添字に相当する名前を空白にしても正しく解釈されます .

添字の並びは列の並びの順番で解釈されます . したがって上記のデータは SIMPLE 形式で

```

cost[1,A] = 9.2;      slack[1,A] = 7.6;
cost[1,B] = 3.5;      slack[1,B] = 5.0;
cost[1,C] = 7.2;      slack[1,C] = 8.3;
cost[2,A] = 2.1;      slack[2,A] = 7.2;
cost[2,B] = 1.1;      slack[2,B] = 2.2;
cost[2,C] = 0.1;      slack[2,C] = 6.2;

```

と書いた場合と等価です .

次の例のように添字の列に同一の組が現れた場合 ,

```

i, j, cost,slack
1 ,A, 9.2, 7.6
1 ,B, 3.5, 5.0
1, A, 7.1, 6.2      //こちらの行のデータが有効

```

後の行に現れたものが有効になります .

添字の数が違うオブジェクトを同一の CSV ファイルで定義することはできません。添字の異なるオブジェクトが同一の表から読み取られようとした場合には、エラーになります。上記の場合には demand というデータがありますが、

```
i, j, cost, slack, demand
1 ,A, 9.2, 7.6, 9.4
1 ,B, 3.5, 5.0, 4.5
```

と書くとエラーとなります。これは demand のみが 1 次元で添字の数が 1 つであるからです。

この書式で添字のない値 (スカラー) を与えることもできます。この場合添字に相当する行はありません。

```
Parameter A (name="A"), B (name="B"), C (name="C");
```

なるスカラーを

```
A, B, C
4.3, 5.2, 7.6
```

と与えることができます。これは、SIMPLE データ形式で

```
A = 4.3;
B = 5.2;
C = 7.6;
```

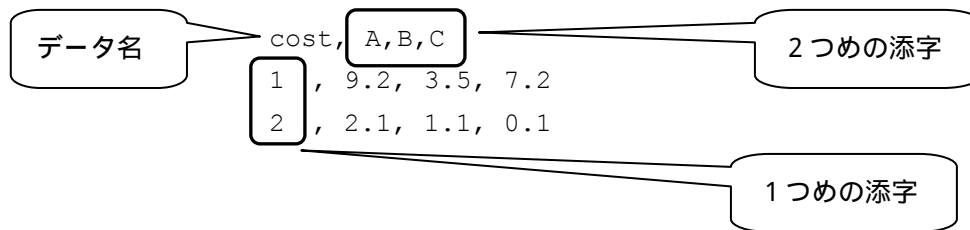
と書いた場合と等価です。

4.2.2 2D 書式

添字を 2 つ以上持つオブジェクトを 1 つの表で表現するのが便利な場合があります。その場合の書式が CSV ファイルの 2D 書式です。

```
Set S, T;
Element i (set=S), j (set=T);
Parameter cost (name = "cost", index = (i, j));
```

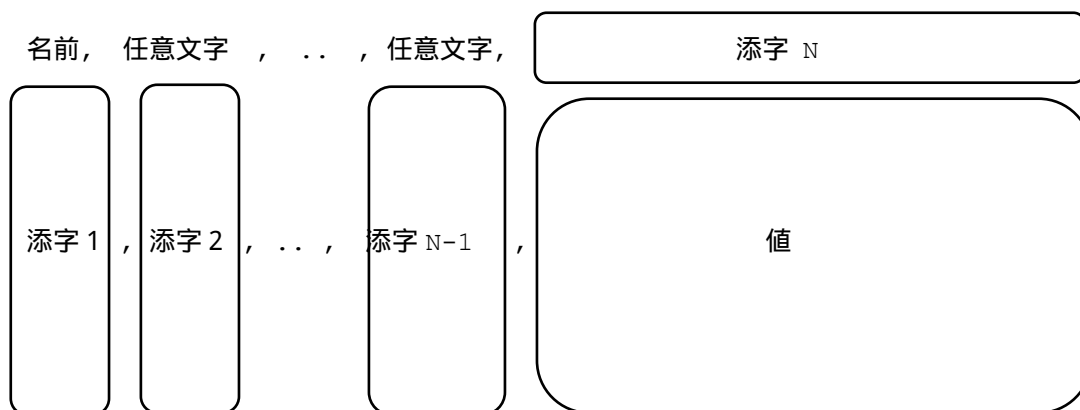
と定義されている cost を 2D 書式で与えると次のようになります。



最初の行の第 1 フィールドはデータの名前を示します。最初の列に書かれているのが最初の添字、最初の行に書かれているのが 2 つ目の添字です。したがって、このデータは以下の SIMPLE データ形式の記述と等価です。

```
cost[1,A] = 9.2;
cost[1,B] = 3.5;
cost[1,C] = 7.2;
cost[2,A] = 2.1;
cost[2,B] = 1.1;
cost[2,C] = 0.1;
```

一般に 2D 書式は添字 N 個を持つオブジェクトを次のように表現する形式です。原理的に添字の数は 2 以上のオブジェクトのみが表現可能です。



このデータが 2D 形式であるか否か、前半のどこまでが添字を示す列であるかは、データに記述されません。SIMPLE の処理系が第 1 行目の第 1 フィールドの名前を見て判定します。添字の並びは列の並びの順番で解釈され、1 行目に並んでいる添字は最後の添字となります。2D 書式で 1 行目の添字に対応する部分に空白があるとエラーになります。また、1 行目の添字に対応する部分に重複があるとエラーになります。1 ~ N-1 までの添字に重複があると、後の行に現れた行のデータが有効になります。

```
cost, A,B,C
```

```

1 , 9.2,3.5,7.2
2 , 2.1,1.1,0.1
1 , 1.8,0.7,4.5 // こちらの指定が有効

```

4.2.3 曖昧さについて

CSV 形式はデータに書式の情報が含まれず、解釈は SIMPLE のモデル記述に依存しますので、思わぬ曖昧さを生むことがあります。例えば

```

Q, A,B,C
1 , 9.2,3.5,7.2
2 , 2.1,1.1,0.1

```

という表があったとき、添字の値とオブジェクト名が偶然一致している場合に曖昧さが生じます。

```

Set i(set=S),j(set=T);
Parameter Q(name="Q",index=(i,j));
Parameter A(name="A",index=i);

```

なるモデル記述で、Q に対してはこの表は Q の値を 2D 形式で記述したように見えますが、A に対しては 1D 形式で A の値を与えていると読むことができます。

このような曖昧さによる混乱を禁じるため、2D 形式で読み込まれた CSV ファイルから 1D 形式で別のデータが読み込まれる、あるいはその逆のケースが起きた場合にはエラーを出します。その場合には読み込まれるオブジェクトの名前を `name = "..."` で変更するなどして対処して下さい。

4.3 データ名の重複について

SIMPLE の処理系は SIMPLE データ形式と CSV 形式のファイルをまずすべて読み込み、そこからデータの初期設定を行います。同じ名前のオブジェクトに対して 2 つ以上のファイルからデータが与えられた次のような場合にはエラーとなります。

```
a = [1] 1 [2] 2;  
a = [3] 3 [4] 4;
```

ただし、モデル中にて

```
options.multDataPolicy = 1; // 重複データを許す設定
```

と設定すると、エラーにはならず、警告扱いとなります。したがって、上記は $a[1] = 1$, $a[2] = 2$, $a[3] = 3$, $a[4] = 4$ がすべて有効と解釈します。その場合には次のように上書きも可能です。

```
a = [2] 2;  
a = [2] 3; // こちらの指定が有効  
b = 1 ;  
b = 5 ; // こちらの指定が有効
```

警告が出るのは大規模データが重複していた場合、深刻なパフォーマンス下落を招くケースがあるためです。したがって、大規模データの場合には重複がない形にデータを定義しなおすことをお勧めします。

5. システム制御

いままでは、`SIMPLE` によるいろいろなタイプのモデルの定義方法を紹介しました。本章では以下のような機能を紹介します。

- ◆ 定式の具体的な内容の表示 (`showSystem()`)
- ◆ 最適化 (システムの解析プログラム) の実行の指示 (`solve`)
- ◆ 各オブジェクトの値の参照 (`<<, print, simple_printf`)
- ◆ オブジェクトの値を C++ の配列に書き出し (`dump`)
- ◆ C++ の配列の内容をオブジェクトに読み込み (`readD, readS`)
- ◆ 制約式の付け外し (`deleteCo/restoreCo` 等)
- ◆ 変数の固定 (`fixVariable/unfixVariable`)
- ◆ 外部から制御できるパラメータを用いた記述 (`VariableParameter`)

Windows 版をお使いいただいている場合、`showSystem()`, `<<`, `print`, `simple_printf` が出力する表示は「メッセージウインドウ」に、UNIX/LINU 版では標準出力に現れます。

5.1 モデルの内容の表示(`showSystem()`関数)

`SIMPLE` によるモデルの定義は一般的な式の形を与えるのみで、具体的な内容についての情報を含んでいません。その際、式の具体的な内容を表示させるのが `showSystem()` 関数です。例えば、次のモデル定義があるとしましょう。

```
Set S(name="S");
Element i(set=S);
Parameter c(name="c",index=i);
Parameter a(name="a",index=i);
Parameter b(name="b");

// 変数
IntegerVariable x(name="x",index=i,type=binary);

// 目的関数
Objective obj(type=maximize);
obj = sum(c[i]*x[i],i);

// 制約条件
sum(a[i]*x[i],i) <= b;
```

これは定式の形についての情報を与えていますが、パラメータ a, b, c の具体的な内容が欠けています。`SIMPLE` は数理計画法プログラム(ソルバ)を起動する際に、以下のようなデータ

```
c = [1] 42 [2] 12 [3] 45 [4] 5   [5] 2
      [6] 61 [7] 89 [8] 32 [9] 47 [10] 18;
a = [1] 39 [2] 13 [3] 68 [4] 15 [5] 10
      [6] 20 [7] 31 [8] 15 [9] 41 [10] 16;
b = 121;
```

を読み込んで、モデル定義にあてはめます。このように抽象的なモデル記述にデータをあてはめる操作を「展開」と呼び、展開によって作成された結果を特に「モデル」と区別して「システム」と呼びます。そのシステムの内容を表示させるのが

```
showSystem();
```

という関数で、モデルやデータが正しく入力されているか確認するために有効です。モデル記述の終りにこの文を挿入すると、`SIMPLE` は展開された形のシステムを次のような形式で表示

します¹⁷ .

(目的関数)

```
obj = 42*x[1]+12*x[2]+45*x[3]+5*x[4]+2*x[5]
      +61*x[6]+89*x[7]+32*x[8]+47*x[9]+18*x[10]
```

(制約条件)

```
39*x[1]+13*x[2]+68*x[3]+15*x[4]+10*x[5]+20*x[6]
+31*x[7]+15*x[8]+41*x[9]+16*x[10] <= 121
```

showSystem の出力では，名前が name= によって与えられない変数の名前は“ ”となります．オブジェクトに name= で名前を与えてから行くと，表示にその名前が使われますのでわかりやすくなります．

showSystem には次のように，指定された制約式のみを表示する機能もあります．

```
showSystem( Constraint ); //指定された制約式のみを表示
showSystem( Constraint,Condition,... );
                        //指定された制約式かつある条件を満たすもののみを表示
```

この利用例は 5.8 特定の制約式の削除/復帰 deleteCo()/restoreCo() 関数の解説にあります．

¹⁷ 大規模問題に関してはこの出力は膨大になりますので注意して下さい．

5.2 ソルバの起動 `solve()` 関数

2.7 ソルバ(NUOPT)との連結で述べたように、定義したモデルについて一度だけ最適化を行う場合には、最適化の実行を明示的に指示する必要はありません。モデルの記述の終わりに自動的に最適化が行われます。

しかし、

- ◆ 解いた後のオブジェクトの内容を表示させる。
- ◆ 複数の目的関数について連続して最適化を行う。
- ◆ モデルの定義を変更しながら複数回の最適化を行う。

などの場合には最適化の実行を

```
solve(); // 最適化を行う
```

という関数で明示的に指示する必要があります。

具体例で説明しましょう。ここでは、最適化を実行した後の表示を、`SIMPLE` の表示関数を使って行わせるために `solve()` の呼び出しを明示的に行っています。`solve()` がなければ `f, x, y` の初期値が表示される結果となってしまいます。

```
Variable x(name="x"), y(name="y"); // 変数 x, y の宣言
Objective f(name="f", type=maximize); // 目的関数の宣言

f = x + y;
pow(x-1, 2) + pow(y-1, 2) <= pow(0.5, 2);

solve(); // 最適化の実行

// 最適化後の値の表示
cout << " f = " << f.val << "\n";
cout << " x = " << x.val << "\n";
cout << " y = " << y.val << "\n";

(Solve 関数の使用例)
```

また `solve()` には複数の目的関数が定義されている場合のために

```
solve(Objective); // Objective についての最適化を行う
```

と、明示的に目的関数を指定する呼び出し形式もあります。次のモデルでは複数の目的関数を

定義し、目的関数を引数とする `solve()` 関数を使っています。

```
Variable x(name="x"), y(name="y"); // 変数 x, y の宣言
Objective f(name="f", type=maximize); // 目的関数の宣言 (1)
Objective g(name="g", type=maximize); // 目的関数の宣言 (2)

f = x+y; // 目的関数 f の定義
g = x-y; // 目的関数 g の定義
pow(x-1,2)+pow(y-1,2) <= pow(0.5,2);

solve(); // 最後に代入された g に関する最大化を行う

solve(f); // 目的関数 f に関する最大化を行う
solve(g); // 目的関数 g に関する最大化を行う
(目的関数を指定する solve 関数の利用例)
```

最初の `solve()` は目的関数の指定を省略した最適化の実行で、この場合には最後に代入された目的関数について最適化が行われます (デフォルトルール)。この例では `g` の方が `f` よりも後に代入されておりますので `solve()` と指示した際には `g` を目的関数とした最適化が行われます。この場合、目的関数の宣言ではなく、代入された順番のみが影響することに注意して下さい。したがって `f` と `g` の宣言の順番を入れ換えても結果は同じです。

次は明示的に目的関数を指定した最適化の実行です。`solve(f)` で、目的関数 `f` に関する最適化が、`solve(g)` で目的関数 `g` に関する最適化がそれぞれ行われます。`solve()` の呼び出しによって最適化が行われるのは、その時点以前に定義されたモデルの内容に対してです。このことを利用して、モデルを逐次変更しながら最適化を行うことができます。

例えば

```
Variable x(name="x"), y(name="y"); // 変数 x, y の宣言
Objective f(name="f", type=maximize);
f = x+y;
pow(x-1,2)+pow(y-1,2) <= pow(0.5,2);
solve(); // 上記までのモデルを解く
x-y >= 0.5; // モデルへの制約式の追加
solve(); // x-y >= 0.5 を含めたモデルを解く
(モデルの変更を逐次行う例)
```

という例では、制約式 `x-y` を 1 つ加える前と後で最適化を行っています。

5.3 オブジェクトの参照値の表示(関数 `print`, `cout`, `simple_printf`)

最適化によって求められた答を表示したり, その値にしたがってモデルを操作するには, `SIMPLE` のオブジェクト `obj` について

<code>obj.val</code>	現在の値
<code>obj.init</code>	初期値
<code>obj.dual</code>	双対変数値
<code>obj.lb</code>	上限値
<code>obj.ub</code>	下限値

という記法を用います¹⁸. これらはオブジェクトの参照値と呼ばれます.

前項の例で, 目的関数 (Objective) である `f` の値を表示させる際に

```
cout << "f = " << f.val << "¥n";
```

と書きましたが, ここで `f.val` は `f` というオブジェクトの現在の値に対応する参照値です. `obj.val` は他にもクラス `Set`, `Expression`, `Variable`, `Constraint` 及び `Parameter` など, すべてのクラスについて許される記法です. `<<` は値を左側の `ostream` オブジェクト (ここでは, 標準出力 (`cout`)) に出力するという演算子です.

`obj.val` などの参照値の出力を行う関数には以下の 3 つがあります.

- ♦ `cout` 関数
- ♦ `print()` 関数
- ♦ `simple_printf()` 関数

これらの関数の使い方について述べます.

¹⁸ C++ オブジェクトのメンバを参照するということに相当します.

5.3.1 print 関数

参照値を出力する最も基本的な方法は `print` 関数を用いることです。 `print` 関数はオブジェクトの添字の値と内容を改行して表示します。

```
// 簡単な LP
Set S = " 1 2 3 ";
Element i(set=S);
Constraint co(name="co",index=i);
Objective f(type=maximize);
Variable x(name="x"),y(name="y");

f = y;
co[1] = x + 2*y <= 4;
co[2] = x - 2*y >= -3;
co[3] = -2*x - y >= -6 ;
x >= 0;
y >= 0;

solve(); // 解く

// 目的関数値を表示
f.val.print("f");
// 変数値を表示
x.val.print();
y.val.print();
// 集合の内容を表示
S.val.print("S");
// 制約式の値を表示
co[i].val.print("co");
```

上の `print()` 文は次の出力に対応します。

```
f=1.75
x=0.5
y=1.75
S=(1 2 3)
co[1]=4
co[2]=-3
co[3]=-2.75
```

最後の制約式の値の表示のように、添字 i を含む記述に対して `print()` を実行すると、すべての i に対応する `co[i]` の値が並んで表示されます。その際の順序は並びは i の内容の辞書順にソートされます。すなわち最後の表示は `co[1]`, `co[2]`, `co[3]` 順に表示されることになります。

表示する範囲を、条件式によって制限することもできます。例えば、`co[2]`, `co[3]` の値のみの表示を行いたい場合には

```
(co[i].val, i >= 2).print("co");
```

と書きます。

ただし、`print` では書式の制御などの機能がありません。細かな書式設定などを行うためには `<<` 関数や `simple_printf` 関数を用いて下さい。

5.3.2 << 関数

<< 関数は一般に

```
ostream << obj.---
```

という形で使用することができます。左辺の `ostream` には標準出力 `cout` の他、一般の文字列型ストリーム (`stringstream`)、出力用ファイルストリーム (`ofstream`) などを用いることができます。

以下はいろいろなオブジェクトの現状値 (`obj.val`) を `<<` を使って表示している例です。

```
// 簡単な LP
Set S = " 1 2 3 ";
Element i(set=S);
Constraint co(name="co", index=i);
Objective f(type=maximize);
Variable x(name="x"), y(name="y");
f = y;
co[1] = x + 2*y <= 4;
co[2] = x - 2*y >= -3;
co[3] = -2*x - y >= -6 ;
x >= 0;
y >= 0;
```

```

solve(); // 解く

// 目的関数値を表示
cout << "f = " << f.val << "\n";
// 変数値を表示
cout << "x = " << x.val << "\n";
cout << "y = " << y.val << "\n";
// 集合の内容を表示
cout << "S  = " << S.val << "\n";
// 制約式の値を表示
cout << "co = " << co[i].val << "\n";

```

`solve()` の後の記述に対応して次のような出力が現れます。

```

f = 1.75
x = 0.5
y = 1.75
S  = (1 2 3)
co = (4 -3 -2.75)

```

最後の制約式の値の表示のように、添字 i を含む記述を行うと、すべての i に対応する `co[i]` の値が並んで表示されます。その際の順序は並びは i の内容の辞書順にソートされます。すなわち最後の表示は `co[1]`, `co[2]`, `co[3]` が並ぶことになります。

表示の際に添字の範囲を制限したい場合には条件式と組みあわせて

```

cout << "co = " << (co[i].val, i >= 2) << "\n";

```

などと記述します。

表示形式を指定するには `ostream` 型のオブジェクトである `cout` に対して

```

cout.precision(10); // 小数点以下の桁数を指定
cout.width(17);     // 表示の幅を指定
cout.setf(ios::scientific, ios::floatfield); // 浮動小数点での表示指定

```

などの表示編集を行います¹⁹。例えば

¹⁹ この機能は通常の C++ に共通のものです。

```
cout << "co = " << co[i].val << "¥n";
```

の前に

```
cout.precision(10); // 小数点以下の桁数を指定
cout.width(17);      // 表示の幅を指定
cout.setf(ios::scientific, ios::floatfield); // 浮動小数点での表示指定
```

と記述すると

```
co = (3.9999999980e+00 -2.9999999980e+00 -2.7499999990e+00)
```

という形式の表示になります。

5.3.3 simple_printf 関数

simple_printf は C++ の組み込み関数の printf のように、細かな書式設定によって参照値を出力するための関数です。simple_printf は変数や目的関数など一般の式の他に、添字の値を表示することができます。

例えば、前項の末尾で

```
// 目的関数値を表示
simple_printf("f          = %17.10e¥n", f.val);
// 変数値を表示
simple_printf("x = %17.10e, y = %17.10e¥n", x.val, y.val);
```

と記述すると次のような出力が得られます。

```
f = 1.7499999990e+00
x = 5.0000000000e-01, y = 1.7499999990e+00
```

simple_printf は最初にフォーマット文字列を、次に任意個のオブジェクトの参照値を並べます。フォーマット文字列の書式は通常の C++ の組み込み関数である printf と同じです²⁰。

²⁰ただし、simple_printf は拡張仕様となっており、変数などの値(通常実数)を %d 編集に対応付けることができます。また、添字の値などの整数値を %f, %e 編集に対応付けることもできます。

フォーマット文字列の後にはフォーマット文字列中の%...に対応した数だけ, `SIMPLE` のオブジェクトの参照値を並べます。参照値で, `.val` は省略が可能です。

オブジェクト並びには(添字)を並べることもできます。`simple_printf` のオブジェクトの並びに添字を含んだ式を記述すると, その添字の取り得るすべての範囲について繰り返し表示が得られます。その際, 添字の出力される順番は辞書順にソートされます。この機能を用いることにより書式が固定された表を容易に作成することができます。

例えば

```
Set S(name="S");
Element i(set=S);
Parameter c(name="c",index=i);
Parameter a(name="a",index=i);
Parameter b(name="b");

// 変数
IntegerVariable x(name="x",index=i,type=binary);

// 目的関数
Objective obj(type=maximize);
obj = sum(c[i]*x[i],i);

// 制約条件
sum(a[i]*x[i],i) <= b;

// 求解
solve();

// simple_printf を使った表示
simple printf("obj = %d, b = %d¥n",obj.val,b.val);
simple printf("a[%2d] = %2d c[%2d] = %2d x[%2d] = %2d¥n"
              ,i,a[i],i,c[i],i,x[i]);
```

というモデル記述に

```
c = [1] 42 [2] 12 [3] 45 [4] 5   [5] 2
      [6] 61 [7] 89 [8] 32 [9] 47 [10] 18;
a = [1] 39 [2] 13 [3] 68 [4] 15 [5] 10
      [6] 20 [7] 31 [8] 15 [9] 41 [10] 16;
b = 121;
```

なるデータを与えて最適化を行うと

```
obj = 242, b = 121
a[ 1] = 39 c[ 1] = 42 x[ 1] =  1
a[ 2] = 13 c[ 2] = 12 x[ 2] =  0
a[ 3] = 68 c[ 3] = 45 x[ 3] =  0
a[ 4] = 15 c[ 4] =  5 x[ 4] =  0
a[ 5] = 10 c[ 5] =  2 x[ 5] =  0
a[ 6] = 20 c[ 6] = 61 x[ 6] =  1
a[ 7] = 31 c[ 7] = 89 x[ 7] =  1
a[ 8] = 15 c[ 8] = 32 x[ 8] =  1
a[ 9] = 41 c[ 9] = 47 x[ 9] =  0
a[10] = 16 c[10] = 18 x[10] =  1
```

なる出力が得られます。モデル中「// simple printfを使った表示」の2行目のsimple_printfでは、.valを省略しています。2行目のオブジェクト並びにはiを含んだ式が現れていますので、iについて繰り返し表示が行われます。

また、一般のファイル構造体FILEに出力を行うsimple_fprintfも用意されています。呼び出し形式は

```
simple_fprintf(FILE fp, フォーマット文字列, オブジェクト1,...);
```

です²¹。

²¹simple_printf は simple_fprintf の最初の引数として標準出力に対応する stdout を渡したものと定義されています。

5.4 オブジェクトの様々な参照値(上下限や双対変数などの参照)

obj.---として参照可能なもの(参照値)には, 前項で使った val(現在値)の他にも, 以下のようなものがあります.

obj.val	現在の値
obj.init	初期値
obj.dual	双対変数値
obj.ub	上限値
obj.lb	下限値

双対変数(シャドウプライス)を示す dual や上下限値を示す ub, lb, 最適化によって変数の値が変更される前に式がどんな値であったかを参照するための init というメンバが定義されており, val と同じ方法で参照することができます. 以下のようにして各種参照値を表示してみましょう.

```
// 簡単な LP
Set S = " 1 2 3 ";
Element i(set=S);
Constraint co(name="co",index=i);
Objective f(type=maximize);
Variable x(name="x"), y(name="y");
f = y;
co[1] = x + 2*y <= 4;
co[2] = x - 2*y >= -3;
co[3] = -2*x - y >= -6 ;
x >= 0;
y >= 0;

solve(); // 解く

// 変数の初期値
cout << "(x init,y ini) = "
    << "(" << x.init << ", " << y.init << ")" << "¥n";

// 変数の双対変数値
cout << "(x dual,y dual) = "
    << "(" << x.dual << ", " << y.dual << ")" << "¥n";

// 制約式の初期値
co[i].init.print("co ini");
```

```
// 制約式の双対変数値
co[i].dual.print("co dual");

// 制約式の上下限
co[i].ub.print("co ub");
co[i].lb.print("co lb");
```

次の出力が得られます .

```
(x init,y ini) = (0,0)
(x dual,y dual) = (4.24588e-09,1.21296e-09)
co ini[1]=0
co ini[2]=0
co ini[3]=0
co dual[1]=-0.25
co dual[2]=0.25
co dual[3]=6.53281e-10
co ub[1]=4
co ub[2]=Infinity
co ub[3]=Infinity
co lb[1]=-Infinity
co lb[2]=-3
co lb[3]=-6
```

SIMPLE の各クラスオブジェクトの意味によって参照値の種類は異なります . 以下の表はその対応を示します .

メンバ	意味	参照可能なクラス
val	現在の値	Set, OrderedSet, CyclicSet, Sequence, Expression, Objective, Variable, IntegerVariable, Parameter, VariableParameter, Constraint
init	解く前の値	Expression, Objective, Variable, IntegerVariable, Constraint
dual	双対変数値	Variable, IntegerVariable, Constraint
ub	上限値	Variable, IntegerVariable, Constraint
lb	下限値	Variable, IntegerVariable, Constraint

表 6 オブジェクトと参照値

5.5 参照値を Parameter として扱う

SIMPLE は参照値を Parameter として扱うため²², これらを Parameter のように使って条件式を組み立てることができます²³. 例えば 5.4 オブジェクトの様々な参照値(上下限や双対変数などの参照)の例で求解の後に

```
// 双対変数の絶対値が 1.0e-4 以上の制約式の集合を定義する
Set Active = setOf(i, fabs(co[i].dual) >= 1.0e-4 );

// Active を表示する
Active.val.print("Active");

// 制約式で, Active に属さないものの値を表示する
(co[i].val, i > Active ).print("co inactive");

// 制約式で Active なものの形を表示する
showSystem(co[i], i < Active);
```

と記述すると

```
Active=(1 2)
co_inactive[3]=-2.75
(co[1]): x+2*y <= 4
(co[2]): x-2*y >= -3
```

なる結果が得られます. また, モデル定義にも例えば

```
// y を現状値に最も近い整数以下の値にするという制約を加える.
y <= floor(y.val);
```

として用いることができます.

²²実際は ValTempData という Parameter とは別のクラスオブジェクトですが, 代入ができない点を除けば Parameter と全く同一の振舞いをしますので, 使用上は ValTempData と Parameter とを区別する必要はありません.

²³ただし Set の val は Parameter として扱うことはできません. 表示するのみです.

5.6 参照値を C++ の配列に出力する

これまでは参照値を表示する方法について説明しましたが、以下に説明する `dump` という関数を用いることによって参照値を C++ の基本型の配列や文字列へと格納することができます。

```
// 簡単な LP
Set S = " 1 2 3 ";
Element i(set=S);
Constraint co(name="co",index=i);
Objective f(type=maximize);
Variable x(name="x"),y(name="y");
f = y;
co[1] = x + 2*y <= 4;
co[2] = x - 2*y >= -3;
co[3] = -2*x - y >= -6 ;
x >= 0;
y >= 0;

solve(); // 解く
// 制約式の値を double の配列にダンプ
int len;
int *idx;
double *valueAry;
co[i].val.dump(len,idx,valueAry);
// ダンプした配列の内容を確認
cout << "len = " << len << "\n";
for ( int k = 0 ; k < len ; ++k ) {
    cout << "valueAry[" << k << "] = " << valueAry[k]
        << "\t idx[" << k << "] = " << idx[k] << "\n";
}
```

上記のように、求解を行ったあとに `dump` 命令を行うと、`co[i]` の値が C++ の配列となって `double` の配列である `valueAry` に書き込まれます。`idx` にはインデックスの値が整数となって書き込まれ、データの長さが `int` 値である `len` に設定されます²⁴。上記のコードから得られる出力は以下のようになります。

²⁴`dump` の引数は C++ のリファレンス型で定義されていますので、通常この種の操作を C の関数で行う場合の様に引数に `&` を掛けてアドレス値とする必要はありません。

```
len = 3
valueAry[0] = 4   idx[0] = 1
valueAry[1] = -3   idx[1] = 2
valueAry[2] = -2.75 idx[2] = 3
```

さらに、SIMPLE のオブジェクトの添字が単一の整数とは限らず、一般の文字の組み合わせであることや、値が実数とは限らないことに対応して、dump には次に挙げる様々なバージョン(18 種類)が用意されています。

一般形:

```
void dump(len, idx, res, s) const
```

引数の型:

len :	int	データ長さ
idx :	char**	index を文字列に組合わせたもの
	int*	index 第 1 次元の integer 列
	int* , int*	index 第 1 , 2 次元の integer 列
res :	char**	結果に相当する文字列の配列
	double*	結果に相当する double の配列
	int*	結果に相当する int の配列
s :	なし	
	char*	SIMPLE データ形式の文字列

例えば上記の例で、valueAry を文字型のポインタの配列として宣言して

```
int len;
char** idx;
char** valueAry;
co[i].val.dump(len,idx,valueAry);
```

とすると、idx、valueAry は添字と値を文字列に変換したものの配列となります。idx が多次元の場合にはこのようにしなければ正確に表現することはできません。また

```
char* s;
co[i].val.dump(len,idx,valueAry,s);
```

のように dump の引数の最後に文字列を渡すと、SIMPLE のデータファイルの書式を持つ文字

列となります．この場合 s の内容は

[1] 4	[2] -3	[3] -2.75
-------	--------	-----------

となり，これを出力することにより，SIMPLE で読み込み可能なデータファイルを作成することも可能です．dump のうち，idx が 2 つあるものを使うと，添字を 2 つ持つ行列等のデータを，例えば

```
Set row;
Set col;
Element i(set=row),j(set=col);
Paramter A(index=(i,j);
    ...
int len;
int* idx1; // 行番号が入る
int* idx2; // 列番号が入る
double* value; //

A[i,j].val.dump(len,idx1,idx2,value);
```

として C++ の配列へと出力することができます．

dump を使うと，SIMPLE のオブジェクトの値をそのまま他の C プログラムに渡す等の操作が可能となります．dump の戻り値となっている配列は SIMPLE 内部で確保 (new) されたものですが，破壊に関して SIMPLE は責任を持ちません．不要となった際には呼び出し側で破壊 (delete[]) する必要があることに注意して下さい．

制約式 Constraint に関する参照値に関しては以下のような取り決めがありますのでご注意下さい．

1. 2 回以上代入された Constraint や 2 つ以上の式に分割されて解釈される Constraint に対しての定義の仕方

- a) 同一の制約式 (Constraint) オブジェクトに 2 度代入する

<pre>Constraint co; Variable x,y; co = x+y; co = x-y; // 両方モデルの中の式として解釈される</pre>

- b) 2 つ以上の式に分割されて解釈すべき制約式 (Constraint) オブジェクトを定義

する

```
Constraint co;
Variable x,y;
co = x*x >= y*y >= x*y; //x*x >= y*y と y*y >= x*y に分けて解釈される
```

上記のいずれの場合にも制約式に関する参照値が一意に決定しません．このように定義した `Constraint` の参照値を出力したり，`dump` を行ったりするときには，a) のケースでは最初に代入されたものの参照値が出力されます．b) のケースでは，分割して解釈される最も左の式の参照値が出力されます．

2. 制約式の参照可能な値の意味

(以下で `co` は `Constraint` のオブジェクト，`?` は参照可能な値すなわち `val` , `dual` , `ub` , または `lb` とします．)

- ◆ 式(変数を含む式)が定数を含む式と隣接している制約式である場合

```
co = 式 >= 定数式;
co = 式 <= 定数式;
co = 定数式 >= 式 >= 定数式;
co = 定数式 <= 式 <= 定数式;
co = 定数式 == 式;
co = 式 == 定数式;
```

この場合には `co.` に対する参照値は「式」に対する参照値と同じ意味になります．例えば

```
co = 5 >= x+y >= 3;
```

という時，`co.val` は `(x+y).val` と同じ意味です．

- ◆ 式が連続している制約式である場合

```
co = 式1 => 式2;
co = 式1 == 式2;
co = 式1 <= 式2;
```

この場合には `co.参照値 = (式 2-式 1).参照値` となります。例えば

```
co = x+y <= x*y;
```

という時、`co.val` は `((x*y)-(x+y)).val` と同じ意味です。

`IntegerVariable` は親クラスである `Variable` から引き継いだ参照可能な値を持ちますが、その他に分枝限定法を備えたソルバ用のパラメータ値を保持する領域である、`pri`、`upc`、`dpc` 及び `dir` というメンバを持ちます。これらは上記と同じように参照が可能であるのみならず、定義も可能です。それらの詳細については 3.3.5 整数変数を参照して下さい。

5.7 C++ の配列からオブジェクトに対する値の設定

この節では、C++の基本型の配列の内容を SIMPLE のオブジェクトに入力する方法を説明します。設定の形式により 2 つの設定関数 readD と readS が用意されています。readD の D は"Dense" 配列を、readS の S は"Sparse"配列をそれぞれ意味します。

これらの関数の一般的な形は次のようになります。

```
void readD(len1,...,lenM, cont) const
```

引数の型:

len? : int	"cont" の? 番目の添字の次元
cont : double*	実際のデータ

```
void readS(len, idx1,...idxM, cont) const
```

引数の型:

len : int	"cont" の長さ
idx?: char**	? 番目の添字列 (文字列)
int*	? 番目の添字列 (整数)
cont: double*	実際のデータ

len1...lenM と idx1...idxM の数は最小 1 で、最大 5 となっています。

readS の idx? はすべて文字列かすべて整数かのいずれかでなければなりません。関数 readD を用いて 配列の内容を一括に SIMPLE の Expression, Parameter または Variable に値を設定することができます。例えば,

```
// 配列内容
double cont[27]={1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7,
                 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7,
                 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7,
                 4.1, 4.2, 4.3, 4.4, 4.5, 4.6};

Set a1(name="a1"); Element i1(set=a1);
Parameter p1(index=i1); // 1 次元
p1.readD(5, cont);
// 結果: p1: [1]=1.1, [2]=1.2, [3]=1.3, [4]=1.4, [5]=1.5

Set a2(name="a2",dim=2); Element i2(set=a2);
Variable p2(index=i2); // 2 次元
```

```

p2.readD(2, 3, cont);
// 結果: p2: [1,1]=1.1, [1,2]=1.2, [1,3]=1.3
//           [2,1]=1.4, [2,2]=1.5, [2,3]=1.6

Set a3(name="a3",dim=3); Element i3(set=a3);
Expression p3(index=i3);
p3.readD(2, 3, 2, cont);
// 結果: p3: [1,1,1]=1.1, [1,1,2]=1.2, [1,2,1]=1.3, [1,2,2]=1.4
//           [1,3,1]=1.5, [1,3,2]=1.6, [2,1,1]=1.7, [2,1,2]=2.1
//           [2,2,1]=2.2, [2,2,2]=2.3, [2,3,1]=2.4, [2,3,2]=2.5

```

となります。

一方、関数 readS を用いて、疎行列の形式で SIMPLE の Expression, Parameter または Variable に値を設定することができます。上記の例では、次のような結果が得られます。

```

int idxI1[6]={10, 11, 12, 13, 14, 15}; // 整数添字配列
int idxI2[6]={20, 21, 22, 23, 24, 25}; // 整数添字配列
int idxI3[6]={30, 31, 32, 33, 34, 35}; // 整数添字配列
p1.readS(5, idxI1, cont);
// 結果: p1: [10]=1.1, [11]=1.2, [12]=1.3, [13]=1.4, [14]=1.5
p2.readS(5,idxI1, idxI2, cont);
// 結果: p2: [10,20]=1.1, [11,21]=1.2, [12,22]=1.3,
//           [13,23]=1.4, [14,24]=1.5
p3.readS(5,idxI1, idxI2, idxI3, cont);
// 結果: p3: [10,20,30]=1.1, [11,21,31]=1.2, [12,22,32]=1.3,
//           [13,23,33]=1.4, [14,24,34]=1.5
// 文字列添字配列
char* idxC1[5]={"idx11", "idx12", "idx13", "idx14", "idx15"};
char* idxC2[5]={"idx21", "idx22", "idx23", "idx24", "idx25"};
p2.readS(5, idxC1, idxC2, cont);
// 結果:
//   p2:[idx11,idx21]=1.1, [idx12,idx22]=1.2, [idx13,idx23]=1.3
//       [idx14,idx24]=1.4, [idx15,idx25]=1.5

```

5.8 特定の制約式の削除/復帰 deleteCo()/restoreCo() 関数

ここでは、制約式を追加したり除去したり、システムを微小に変更する方法について述べます。次は以下の説明の出発点となる簡単な線形計画問題の定義とその問題に対する最適化を指示する記述です。

```
Set S = " 1 2 3 ";
Element i(set=S);
Objective f(type=maximize);
Variable x(name="x"), y(name="y");

// 目的関数の定義
f = y;

// 制約式の設定
Constraint co(name="co", index=i);

co[1] = x + 2*y <= 4;
co[2] = x - 2*y >= -3;
co[3] = -2*x - y >= -6 ;

x >= 0;
y >= 0;

//
// モデル記述は上記で終了。 以下はシステム制御
//
// ソルバ側の出力を行わない様にする (パラメータ設定)
options.outputMode = "silent";

// co[i] の内容を出力する
showSystem(co[i]);

// 求解
solve();

// 目的関数値を表示する
cout << "f = " << f.val << "¥n";
```

実行させると次のような出力が得られます。

```
SIMPLE 2.9.6, Copyright (C) 1994-2003 Mathematical Systems Inc.
<system code file name: lp1.cc>
1-1 (co[1]): x+2*y <= 4          <---
1-2 (co[2]): x-2*y >= -3        <--- showSystem(co[i]) に対応
1-3 (co[3]): -2*x-y >= -6      <---
f = 1.75                        <--- cout << ... に対応
```

`options.outputMode = "silent"`はソルバ側に与えるパラメータの設定で、ソルバ側が出力するメッセージを抑制するためのものです。通常 `NUOPT` は、最適化が起動される度に

```
...
PROBLEM NAME                                lp
NUMBER OF VARIABLES                         2
NUMBER OF FUNCTIONS                        4
PROBLEM TYPE                               MAXIMIZATION
METHOD                                     LINE SEARCH
<preprocess begin>.....<preprocess end>
<iteration begin>
    res=2.7e+00 .... 3.2e-03 ... 9.6e-10
<iteration end>
STATUS                                     OPTIMAL
VALUE OF OBJECTIVE                         1.749999996
ITERATION COUNT                            9
FUNC EVAL COUNT                           12
FACTORIZATION COUNT                        9
RESIDUAL                                  9.610294641e-10
ELAPSED TIME(sec.)                         0.02
SOLUTION FILE                             lp.sol
f = 1.75
...
```

のようなソルバ側からのメッセージを標準出力に表示しますが、この設定はこれを抑制するためのものです。このように `options.` パラメータ名に値を代入することによって、ソルバ側へパラメータ指定を行うことができます。詳しくは、**第二部 4.6 ソルバとのインタフェース**をご参照下さい。

出力中 `1-1 (co[1]): ...` という部分がシステムを表示する関数である `showSystem()` による表示です。最後に

```
cout << "f = " << f.val << "\n";
```

という記述により，目的関数 f の値が表示されています． $f.val$ は「 f の現在値」という意味で，このように $<<$ 演算子で表示したり，定数 (Parameter) としてモデル定義に使うことができます．詳細は 5.3 オブジェクトの参照値の表示 (関数 `print`, `cout`, `simple_printf`) にございます．

この出発点となるモデルから最初の制約式を削除して結果の変化を見るためには，システムから制約式を削除する命令 `deleteCo` を用いて末尾に次のように記述します．

```
// co[1] を削除
deleteCo(co[1]);

// 削除した状態でのシステムの内容表示
showSystem(co[i]);

// 再求解
solve();

// 目的関数値を表示
cout << "f(without co[1]) = " << f.val << "\n";
}
```

削除の確認をするため途中に `showSystem` を挟んでいます．出力の末尾は次のようになります．

```
f = 1.75
1-1 (co[1]): x+2*y <= 4   <<<deleted>>>
1-2 (co[2]): x-2*y >= -3
1-3 (co[3]): -2*x-y >= -6
f(without co[1]) = 2.4
```

`showSystem()` の表示において， $co[1]$ が除去されたことが示され (<<<deleted>>>)，目的関数の値は 2.4 と変化していることがわかります．`deleteCo` の呼び出し形式は一般に要素や条件式を使うことができます．

```
//指定された制約式をシステムから除外
deleteCo( Constraint );

//指定された制約式かつある条件を満たすもののみ除外
deleteCo( Constraint,Condition,... );
```

例えば,

```
deleteCo(co[i], i >= 2); // co[2], co[3] を除去
```

とすることにより, `co[2], co[3]` の両方を除去することができます.

`deleteCo` は引数として制約式 (Constraint) オブジェクトを指定することが必要で,

```
deleteCo(x >= 0); // エラー
deleteCo(y >= 0); // エラー
```

などを書くことはできません. 除去したい場合には以下のように, Constraint オブジェクトを作って代入し, それに対して `deleteCo` を起動します²⁵.

```
Constraint xBound, yBound;

xBound = x >= 0;
yBound = y >= 0;

deleteCo(xBound);
...
```

`co[1]` のかわりに, `co[2]` を除去した効果を調べるには, 続けて以下のように記述します.

```
... (上記と同じ) ...

// co[1] を復帰
restoreCo(co[1]);
// co[2] を削除
deleteCo(co[2]);

// 現状でのシステムの内容表示
showSystem();

// 再求解
solve();

cout << "f(without co[2]) = " << f.val << "\n";
```

²⁵SIMPLE のモデル定義では変数の上下限も特別な制約として扱います.

`restoreCo` はシステムに制約式を復帰させる命令です．これを使って最初の `deleteCo` で除去された制約式を復帰させてから，`co[2]` を除去しています．出力は以下のようになり，`co[2]` を除去すると最適解は 2 になることがわかります．

```

...
f = 1.75
1-1 (co[1]): x+2*y <= 4   <<<deleted>>>
1-2 (co[2]): x-2*y >= -3
1-3 (co[3]): -2*x-y >= -6

2-1 ( 3): x >= 0

3-1 ( 4): y >= 0

<objective>: y (maximize)
f(without co[1]) = 2.4
1-1 (co[1]): x+2*y <= 4
1-2 (co[2]): x-2*y >= -3   <<<deleted>>>
1-3 (co[3]): -2*x-y >= -6

2-1 ( 3): x >= 0

3-1 ( 4): y >= 0

<objective>: y (maximize)
f(without co[2]) = 2

restoreCo の呼び出し形式は次の通りです．

//指定された制約式をシステムに復元
restoreCo( Constraint );
//指定された制約式かつある条件を満たすものののみ復元
restoreCo( Constraint,Condition,... );

```

`deleteCo` と同様に要素や条件式を使って，ここで

```
restoreCo(co[i],i>=2); // co[2],co[3] を復帰
```

と書くことも可能です．

5.9 モデル定義とシステム制御を分離して記述する

上記のコードではモデル定義とシステム制御部分が同一の関数に書かれていましたが，これを次の様にして分離して書くこともできます．

```
Set S = " 1 2 3 ";
Element i(set=S);
Constraint co(name="co",index=i);
Objective f(type=maximize);

// モデル定義
lpModel(f,co);

// ソルバ側の出力を行わない様にする
options.outputMode = "silent";

// 求解
solve();

// 目的関数値を表示
cout << "f = " << f.val << "¥n";

deleteCo(co[1]);
solve();
cout << "f(without co[1]) = " << f.val << "¥n";
restoreCo(co[1]);

deleteCo(co[2]);
solve();
cout << "f(without co[2]) = " << f.val << "¥n";
restoreCo(co[2]);

deleteCo(co[3]);
solve();
cout << "f(without co[3]) = " << f.val << "¥n";
restoreCo(co[3]);
```

(システム制御を含むメイン)

```

void lpModel(Objective f, Constraint co)
{
    Variable x(name="x"), y(name="y");

    f = y;

    // 制約式の設定
    co[1] = x + 2*y <= 4;
    co[2] = x - 2*y >= -3;
    co[3] = -2*x - y >= -6 ;

    x >= 0;
    y >= 0;

```

(モデル定義を行う関数)

ただし, その際には C++ のスコープ (通用範囲) ルールの制約上, deleteCo 等を行いたい制約式 Constraint 等のオブジェクトはシステム制御を行う部分 (ここでは ufun()) にて宣言して, モデルを定義している部分 (ここでは lpModel()) へ引数として渡す必要があります. 上記の実行結果は以下のようになります.

```

SIMPLE 2.9.6, Copyright (C) 1994-2003 Mathematical Systems Inc.
<system code file name: lp4.cc>
f = 1.75
f(without co[1]) = 2.4
f(without co[2]) = 2
f(without co[3]) = 1.75

```

5.10 変数の固定

システムに行う頻度の高い操作として、変数の値を求解プロセスの途中で適宜固定することが挙げられますが、SIMPLE ではそれを `fixVariable` と `unfixVariable` の 2 つの関数を用いて、動的に行うことができます。

例としてネットワークの最小コスト経路問題を挙げます。与えられたネットワーク上のある開始節点 `start` から終了節点 `end` までの最小コストで、かつ短い経路を求めることが要求されているとしましょう。最初最小コストの経路を求め、それが複数存在するならそのうちで一番短い経路を選ぶ、という方法でこの問題に対処します。

例えば、次のようなネットワークを考えます。開始節点は 1 で、終了節点は 6 です。`a[i]` は辺のコストです。

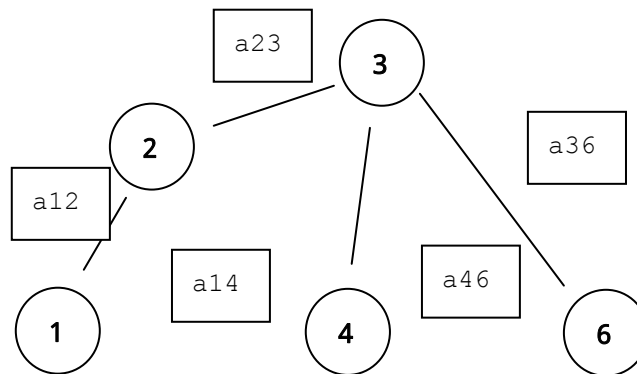


図 1 ネットワークシステム構成

この問題に対するモデル定義とシステム制御は次のようになります。

```

Graph      g;
Variable x(name="x",index=g.arcs);
Parameter a(name="cost",    index=g.arcs);    // 辺のコスト
Parameter b(name="dis",    index=g.arcs);    // 辺の距離(全員 1)
Parameter s(name="supply", index=g.nodes);    // supply
Objective obj1(name="obj1",type=minimize);
Objective obj2(name="obj2",type=minimize);
Element i(set=g.nodes);          // i はグラフのある節点
Element e(set=g.arcs);          // e はグラフのある辺
Element eout(set=out(g,i));      // 節点 i から出発する辺
Element ein(set=in(g,i));        // 節点 i へ進入する辺
  
```

```
// ソルバ側の出力を行わない様にする
options.outputMode = "silent";

//目的関数
obj1 = sum(a[e] * x[e], e);           // 最小コスト
obj2 = sum(b[e] * x[e], e);           // 最短距離
//制約条件
sum(x[eout], eout) - sum(x[ein], ein) == s[i];
x[e] >= 0;

solve(obj1);                          //最小コストを求める
obj1.val.print("obj1");               //結果表示
x[e].val.print("x1");                 //結果表示

fixVariable(x[e], x[e].val <= 0.1);    // 解でない経路を固定する
solve(obj2);                          //最短距離を求める
obj2.val.print("obj2");               //結果表示
x[e].val.print("x2");                 //結果表示
```

次のような経路コストのデータファイル

```
cost    = [1,2] 0.25 [2,3] 0.25 [3,6] 0.5 [1,4] 0.5
          [1,5] 0.5   [5,6] 0.5;
dis      = [1,2] 1 [2,3] 1 [3,6] 1 [1,4] 1 [1,5] 1 [5,6] 1;
supply = [1] 1 [2] 0 [3] 0 [4] [5] 0 [6] -1;
```

を用意してモデルを解くと次のような結果が得られます。

```

obj1=1
x1[1,2]=6.000e-01
x1[2,3]=6.000e-01
x1[3,6]=6.000e-01
x1[1,4]=0
x1[1,5]=4.000e-01
x1[5,6]=4.000e-01

x2[1,2]=4.564e-08
x2[2,3]=4.564e-08
x2[3,6]=4.564e-08
x2[1,4]=0
x2[1,5]=1.000e+00
x2[5,6]=1.000e+00

```

この定義では求解を二度行っています。2 回目のモデル (最小コスト問題) を解いたあとの解 x_1 は上記に見るように、最小コストを与える 2 つの経路 (1→2→3→6 と 1→5→6) に分れてしまっています。ここから求められた解のなかの最短経路のものを選ぶために、解でない辺 (辺の変数値が小さいもの) を固定 (`fixVariable(x[e], x[e].val <= 0.1)`) して目的関数をコストから経路長さ (`obj2`) に変更し、再度解を求めて解 x_2 を得ています。結果として 1→5→6 となります。

変数値を固定するのと、2 回目の `solve()` の前に

```
x[e] <= 0.1, x[e].val <= 0.1;
```

という制約式を追加するのは等価な意味になりますが、この場合にはこの記述に関する展開を行う必要が生じますので、`fixVariable` を用いて変数を固定の方が効率的です。もし、`fixVariable` によって指定された値がその変数の上下限を越えた場合は、`SIMPLE` はそれを検出しエラーとします。`unfixVariable` は `fixVariable` と全く同様の呼び出し形式を持ち、`fixVariable` によって指定された変数の固定を解除します。

5.11 可変定数

モデル中の定数 `Parameter` は、モデルをシステムに展開する段階で固定され、展開後には変更不可能となります。しかし、パラメトリック最適化等の場合には変更可能な定数を含めてモデルを定義して、後で変更しながら解析を行う場合が生じます。そのために `SIMPLE` は `VariableParameter` というオブジェクトを提供しています。

次は簡単な可変定数の使用例です。線形な目的関数の係数を変更しながら、定円内が実行可能領域である二次計画問題を逐次的に解きます。この問題に対するモデル定義とシステム制御は次のようになります。

```
Variable x,y;
VariableParameter a;           // 可変定数
Objective obj(type=maximize);
//目的関数
obj = -a*x+y;                  // 直線
//制約条件
pow(x-1,2)+pow(y+0.5,2) <= 0.25;

options.outputMode = "silent"; // NUOPT の出力を抑制
for(int i=-5; i<5; i++){ // 異なる a の値について繰り返し解く
    a = i;
    solve();
    cout << "a = " << a.val << ", ";
    cout << "x = " << x.val << ", ";
    cout << "y = " << y.val << ", ";
    cout << "obj = " << obj.val << "\n";
}
```

上記のように `VariableParameter` は通常の `Parameter` と全く同様にモデル定義に使用することができます。

最適値は下図のように定円の接線(傾き a)の y 切片となりますので、目的関数として各 a に対応する y 切片の値が得られることになるはずです。上記からは次のような結果が得られます。

```

SIMPLE 2.9.6, Copyright (C) 1994-2003 Mathematical Systems Inc.
<system code file name: tangent.cc>
a = -5, x = 1.49029, y = -0.401942, obj = 7.04951
a = -4, x = 1.48507, y = -0.378732, obj = 5.56155
a = -3, x = 1.47434, y = -0.341886, obj = 4.08114
a = -2, x = 1.44721, y = -0.276393, obj = 2.61803
a = -1, x = 1.35355, y = -0.146447, obj = 1.20711
a = 0, x = 1, y = -2.58616e-08, obj = -2.58616e-08
a = 1, x = 0.646447, y = -0.146447, obj = -0.792893
a = 2, x = 0.552786, y = -0.276393, obj = -1.38197
a = 3, x = 0.525658, y = -0.341886, obj = -1.91886
a = 4, x = 0.514929, y = -0.378732, obj = -2.43845

```

ただし, `VariableParameter` は通常の `Parameter` と比べて, メモリを多く所要します. モデルを定義するときに, 必要以上の `Parameter` を `VariableParameter` とすることは避けて下さい.

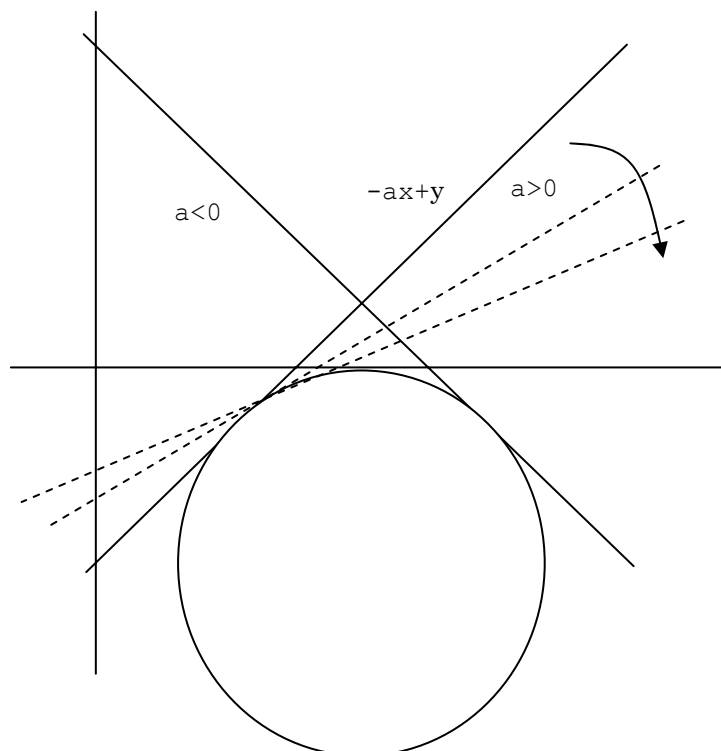


図 2 接線と切片

第二部 最適化ソルバ NUOPT

1. はじめに

1.1 第二部各章の内容

本マニュアル第二部では数理計画法求解プログラム NUOPT の機能について述べています。1～3, 5 章は UNIX/LINUX 版のコマンドラインインタフェースに特化した解説となっています。Windows 版では GUI と別に、MS-DOS プロンプト（コマンドプロンプト）から起動するコマンドラインを利用しない限り、1～3 章を直接参照する必要はありません。Windows 版をお使いの方は、最初に、別冊「NUOPT Windows 版マニュアル」をお読みいただき、必要に応じて下記の箇所を参照していただくのが効率的です。

NUOPT が備えているアルゴリズム、パラメータの設定方法について知りたい

4 パラメータ設定

解ファイル (solfile.txt) と標準出力の意味について知りたい

3 出力ファイルの解説

MPS ファイルによる問題入力方法、MPS ファイル自体について知りたい

5MPS ファイル入力用モジュール nuopt

エラーメッセージの意味について知りたい

A.1, A.2NUOPT/SIMPLE エラーメッセージ

NUOPT が備えているアルゴリズムの概要について知りたい

B. NUOPT アルゴリズム概説, C 参考文献

なお、参照箇所にある章節番号は「第一部」と断りのない限り、第二部のものを指します。

各章の内容は以下のようになっています。

1 はじめに

UNIX/LINUX のコマンドラインインタフェース機能の紹介です。

2SIMPLE 版ロードモジュール

モデリング言語 SIMPLE によって記述された問題の最適化を、UNIX/LINU 版において行う場合の具体的な解説です。UNIX/LINUX 版に特化した内容となっているので、ご注意下さい。通常 Windows 版では、同様の操作を GUI 環境から行います。方法は別冊「NUOPT Windows 版マニュアル」にございます。

3 出力ファイルの解説

NUOPT 本体部が結果として出力する解ファイル(モデル名 `.sol`, Windows 版では `solfile.txt`)と標準出力の内容についての解説です。Windows 版の GUI 実行時には解ファイルが `solfile.txt` なる固定された名前で生成されます。標準出力は GUI のメッセージウインドウに現れます。

4 パラメータ設定

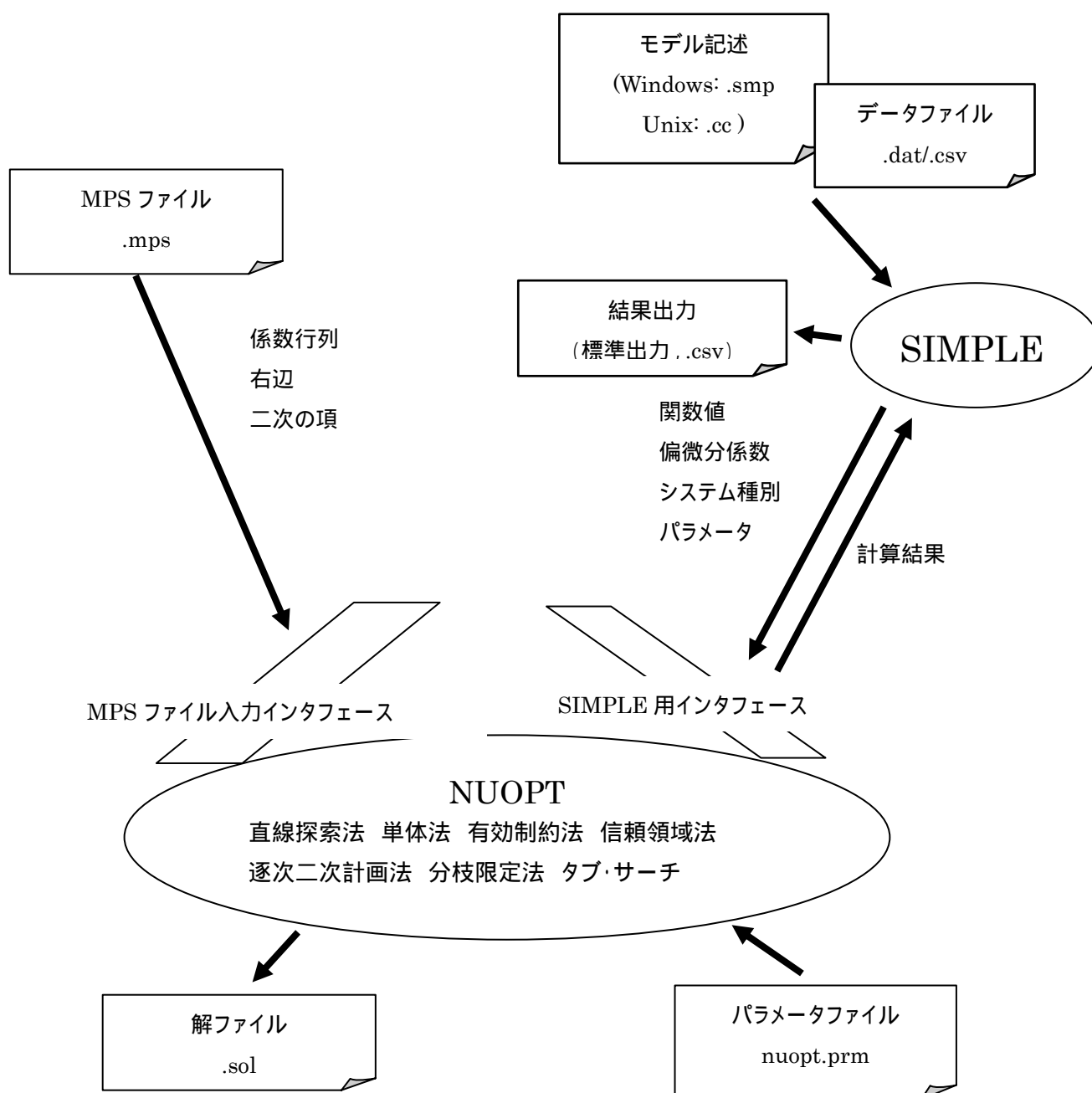
NUOPT の備えているアルゴリズムの開設や, NUIOPT が最適化の際に適用する解法を選択など, 動作を制御するパラメータの設定方法について述べています。

5 MPS ファイル入力用モジュール `nuopt`

コマンドラインインタフェースによって MPS ファイルを入力とした最適化を行うためのコマンドである `nuopt` の使用方法に関する解説です。Windows 版のコマンドラインインタフェースも全く同一の仕様です。

1.2 内部構成

下図は NUOPT のシステム内部構造を模式的に示したものです。



NUOPT の本体 (前頁図の中央) は種々の最適化手法を備えたライブラリであり ,以下の二種類のインタフェースと連結して動作します .

- ◆ MPS ファイル用インタフェース (図の左上部分)

拡張 MPS ファイル書式によって書かれた線形計画問題や二次計画問題の入力を行う .

- ◆ モデリング言語 SIMPLE 用インタフェース (図の右上部分)

モデリング言語 SIMPLE で書かれた数理計画問題 (線形 / 整数 / 二次 / 非線形計画問題) の入力を行う .

NUOPT 本体に , SIMPLE 用インタフェースを連結して `mknuopt` によって作成されたものが SIMPLE 版ロードモジュール (第 2 章) , MPS ファイル用インタフェースを連結したものが MPS ファイル入力用ロードモジュール `nuopt` (第 5 章) です .

1.3 MPS ファイル用ロードモジュール nuopt の簡単な使い方

ここでは UNIX/LINUX プロンプトからロードモジュール nuopt を使って、(例題 5 線形計画問題)

$$\begin{array}{ll}
 \text{最小化} & -3x_1 - 2x_2 - 4x_3 \\
 \text{条件} & x_1 + x_2 + 2x_3 \leq 4 \\
 & 2x_1 + 2x_3 \leq 5 \\
 & 2x_1 + x_2 + 3x_3 \leq 7 \\
 & x_1 \geq 0, \ x_2 \geq 0, \ x_3 \geq 0
 \end{array}$$

(例題 5 線形計画問題)

を解く例について説明します。まずはこの問題を 5.6MPS ファイルで説明されている MPS ファイル書式で記述した下記のようなもの (ex1.mps) を用意します。

```

NAME                EXAMPLE1
ROWS
  N   F
  L   G1
  L   G2
  L   G3
COLUMNS
  X1      F          -3.
  X1      G1          1.
  X1      G2          2.
  X1      G3          2.
  X2      F          -2.
  X2      G1          1.
  X2      G3          1.
  X3      F          -4.
  X3      G1          2.
  X3      G2          2.
  X3      G3          3.
RHS
  B      G1          4.
  B      G2          5.
  B      G3          7.
ENDATA

```

ファイル名は `ex1.mps` とします。MPS ファイルの準備ができたならファイル名を引数として与え、`nuopt` を実行します。

```
prompt% nuopt ex1.mps
```

正常にインストールされている場合には正常に最適化が行われ、最適値として -10.5 が得られたことを示す以下の様な出力が標準出力されます。

```
NUOPT 6.0.0, Copyright (C) 1991-2003 Mathematical Systems Inc.
<reading MPS file: ex1.mps >
PROBLEM_NAME(TITLE)                example1
ROWS                                4
COLUMNS                            3
NONZEROS                            11
OBJECTIVE                           f
RHS                                  b
NUMBER_OF_VARIABLES                  3
NUMBER_OF_FUNCTIONS                  4
PROBLEM_TYPE                         MINIMIZATION
METHOD                              HIGHER_ORDER
<preprocess begin>.....<preprocess end>
<iteration begin>
    res=2.4e+001 .... 2.7e-005  1.4e-008
<iteration end>
STATUS                              OPTIMAL
VALUE_OF_OBJECTIVE                   -10.5
ITERATION_COUNT                      6
FUNC_EVAL_COUNT                      9
FACTORIZATION_COUNT                  7
RESIDUAL                             1.354127444e-008
ELAPSED_TIME(sec.)                   0.01
SOLUTION_FILE                        ex1.sol
```

起動メッセージ (NUOPT 6.0.0 は NUIPT のバージョンを示しています) の後の標準出力には、MPS ファイル `ex1.mps` を読んだこと (`<reading MPS file: ex1.mps >...`)、`PROBLEM_NAME` 以降に MPS ファイルの内容に関する記述がなされます。`METHOD` で始まる行は最適化手法として線形計画問題用の内点法 (`HIGHER_ORDER`) を適用したことを示しています²⁶。

標準出力の意味に関する詳しい解説は 5.3.1 標準出力を参照して下さい。標準出力の最後

²⁶NUOPT は特に指定がないときには問題の種類に応じ、ある規則に従って最適化手法を自動的に選択します。MPS ファイル用ロードモジュールの最適化手法の自動選択については 5.4.1 `nuopt` の最適化アルゴリズムの設定に解説されています。

の行は MPS ファイルの名前に従って名づけられた

SOLUTION FILE (解ファイル)	ex1.sol
-----------------------	---------

に計算結果が書き出されたことを示します 解ファイル ex1.sol の内容を見てみましょう .

MPS_FILE_NAME	ex1.mps					
PROBLEM_NAME (TITLE)	example1					
ROWS	4					
COLUMNS	3					
NONZEROS	11					
OBJECTIVE	f					
RHS	b					
NUMBER_OF_VARIABLES	3					
NUMBER_OF_FUNCTIONS	4					
PROBLEM_TYPE	MINIMIZATION					
METHOD	HIGHER_ORDER					
STATUS	OPTIMAL					
VALUE_OF_OBJECTIVE	-10.5					
ITERATION_COUNT	6					
FUNC_EVAL_COUNT	9					
FACTORIZATION_COUNT	7					
RESIDUAL	1.354127444e-008					
ELAPSED_TIME (sec.)	0.01					
%%						
%% VARIABLES						
%%						
NAME	VALUE	STATUS	[BOUND	TYPE]
V# 1 x1	2.499999998	FREE	[0	<= x1]
V# 2 x2	1.499999999	FREE	[0	<= x2]
V# 3 x3	1.223480816e-009	LOWER	[0	<= x3]

```
%%
%% FUNCTIONS
%%
```

	NAME	VALUE	STATUS	[CONSTRAINT/OBJECTIVE	TYPE]
F# 1	f	-10.5	FREE	[OBJECTIVE (MINIMIZE)]
F# 2	g1	3.999999999	UPPER	[g1 <=	4]
F# 3	g2	4.999999998	UPPER	[g2 <=	5]
F# 4	g3	6.499999998	FREE	[g3 <=	7]

最適化全体に関する情報 (MPS FILE NAME ...) に続いて変数と関数²⁷の値とその上下限が出力されます。続いて

```
%%
%% BOUNDS
%%
```

	[BOUND	TYPE]	DUAL	VALUE
B# 1	[0	<=	x1]	4.891837406e-010	
B# 2	[0	<=	x2]	8.151927281e-010	
B# 3	[0	<=	x3]	1.0000000001	

```
%%
%% CONSTRAINTS
%%
```

	[CONSTRAINT/OBJECTIVE	TYPE]	DUAL	VALUE
C# 1	[OBJECTIVE (MINIMIZE)]	0	
C# 2	[g1 <=	4]	-1.999999998	
C# 3	[g2 <=	5]	-0.4999999986	
C# 4	[g3 <=	7]	-2.443858094e-009	

と変数の上下限や制約式に関する双対変数値の情報が出力されます。双対変数はシャドウプライスあるいは `reduced cost` と呼ばれ、制約式の右辺や変数の上下限を単位あたり変化したときの目的関数の変動を示しています。NUOPT が出力するファイルの内容について詳しくは 3 出力ファイルの解説で解説しています。

²⁷ このマニュアルでは目的関数と制約式を総称してこう呼びます。

1.4 パラメータファイル `nuopt.prm`

通常は上記の操作のみで `nuopt` を動作させることができますが、最適化手法を指定する、問題に関する付加情報を与えるといった場合にはパラメータファイルと呼ばれる `nuopt.prm` という名前²⁸のファイルから付加情報を与えます。例えば、解法としてこの場合に自動選択される内点法ではなく、単体法を適用することを指定したい場合には

```
begin
method:simplex
end
```

という内容を記述したファイル `nuopt.prm` を、`nuopt` を起動するディレクトリに置きます。`nuopt` を実行すると標準出力の中の解法を示す部分の表示が

METHOD	SIMPLEX
--------	---------

に变化します。内点法によるものとは誤差の範囲で微妙に異なりますが、同一の解が得られます。ここで `nuopt.prm` に書いた `method:` という行は適用する最適化手法を `nuopt` に指示するためのものです。詳細は 4 パラメータ設定に解説があります。設定可能なパラメータとしてその他には以下があります。

- MPS ファイルの付加情報の指定（4.4MPS ファイルに関する設定）
 - a) MPS ファイルを読み込むパス名の指定
 - b) 目的関数行の名前の指定
 - c) 計算で実際に使用する `RANGE`, `RHS`, `BOUNDS` データのラベル名の指定
 - d) 最大化/最小化の変更
- スケーリングの有無の指定
- 最適化手法に依存した各種パラメータの設定

²⁸この名前は固定されています。

1.5 モデリング言語 SIMPLE との連結

SIMPLE 用のインタフェースを用いると以下の機能を実現することができます。

- ◆ モデリング言語 SIMPLE による、任意の数値計画問題の記述と求解
- ◆ NUOPT を制御するパラメータの SIMPLE 内部からの設定
(2.9SIMPLE 版ロードモジュールの動作の制御)
- ◆ 最適化の結果として得られた変数/関数の値の表示や、それらを参照したモデル定義
(2.5.1SIMPLE による最適化の結果の表示)
- ◆ 複数回の最適化実行制御
(2.5.2SIMPLE による最適化制御)

SIMPLE のインタフェースを結合したロードモジュール(以下では SIMPLE 版ロードモジュールと総称)はモデリング言語 SIMPLE による数値計画問題の記述(C++ のソースファイル)からシェルスクリプト `mknuopt` によって作成します。SIMPLE で記述した最適化問題の求解はこの SIMPLE 版ロードモジュールを実行することによって行われます。モデリング言語 SIMPLE 自体についての解説は第一部モデリング言語 SIMPLE で行っています。

SIMPLE 版ロードモジュールも MPS 版ロードモジュールと本体部分を共有しておりますので、

- ◆ 計算結果を記述したファイル(3 出力ファイルの解説参照)
- ◆ パラメータファイル(4 パラメータ設定参照)

については MPS ファイル用のインタフェースを用いる場合と共通しています。

2. SIMPLE 版ロードモジュール

この章では、SIMPLE 版ロードモジュール、すなわち SIMPLE による記述から mknuopt によって作成された SIMPLE 用のインタフェースを含むロードモジュールについて説明します。

Windows 版をお使いの方へ：

この章の説明は UNIX 版に特化しており、mknuopt の仕様も異なります。

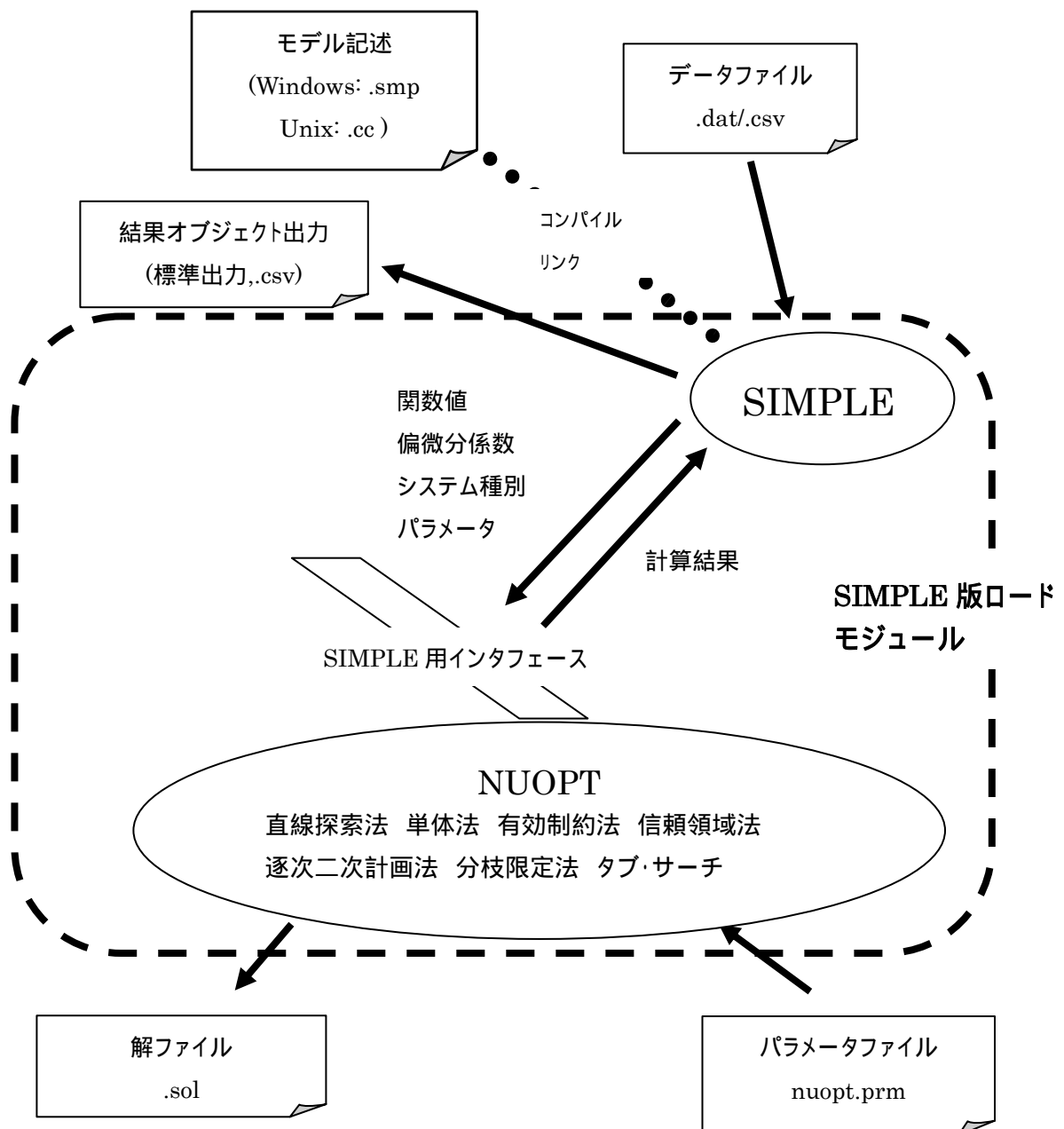
SIMPLE を用いた最適化の実行は NUOPT Windows 版マニュアルをご覧ください。

UNIX/LINUX 版における代表的な数理計画問題の SIMPLE による記述と求解例は **NUOPT/SIMPLE チュートリアル**にまとめてあります²⁹ので、そちらもあわせてご参照下さい。ここでは SIMPLE 版ロードモジュールの作成の方法や実行の詳細について例を挙げながら解説します。SIMPLE の文法の詳細に関しては**第一部モデリング言語 SIMPLE**をご覧ください。

²⁹ Windows 版では同様の内容が「NUOPT Windows 版マニュアル」に含まれていますので付属していません。

2.1 SIMPLE 版ロードモジュールの構成

以下の図が SIMPLE 版ロードモジュールの構成を示したものです。解きたい問題に特化した部分を含む SIMPLE 本体と NUOPT のライブラリ、その仲立ちをするインタフェースによって構成されています。



2.2 SIMPLE 版ロードモジュールの作成

SIMPLE 版ロードモジュールの作成には `mknuopt` を使用します。 `mknuopt` はモデリング言語 SIMPLE によるモデル記述 (C++ のソースファイル) をコンパイルして、上記 SIMPLE 版ロードモジュール (前ページ図点線の枠内) を作成するシェルスクリプトです。

例えば、(例題 6 二次計画問題)

$$\begin{array}{ll} \text{最大化} & f \equiv x + y \\ \text{条件} & (x-1)^2 + (y-1)^2 \leq \left(\frac{1}{2}\right)^2 \end{array}$$

(例題 6 二次計画問題)

を解く場合を例にとります。まずこれを SIMPLE にて記述した次のようなモデルファイル (C++ のソースファイル) を用意します。

```
#include "simple.h"
//
// 例題 6
//
void ufun()
{
    Variable x(name="x"), y(name="y"); // 変数 x, y の宣言
    Objective f(name="f", type=maximize); // 目的関数の宣言

    pow(x-1, 2) + pow(y-1, 2) <= pow(0.5, 2);
    // (x-1)^2 + (y-1)^2 <= (1/2)^2
}
```

このファイルを `ex2.cc` としましょうⁱ。SIMPLE にて記述された C++ のソースファイルを以下ではシステムコードと呼びます。

`mknuopt` はシステムコードから SIMPLE 版ロードモジュールを作成するシェルスクリプト (コマンド) で、次のように起動します。

```
prompt% mknuopt システムコードファイル名
```

この場合には `ex2.cc` を引数にして

```
prompt% mknuopt ex2.cc
```

とすると, `mknuopt` が起動して

```
mknuopt 1.1, Copyright (C) 1995-1997 Mathematical Systems Inc.
SYSTEM CODE: ex2.cc
Compiling ...
Compilation finished.
Linking ...
ex2 created.
prompt%
```

という標準出力が現れ, `mknuopt` を起動したディレクトリに, この問題に特化した `ex2` という名前の `SIMPLE` 版ロードモジュールが作成されます.

この時, システムコードファイル名の拡張子 `.cc` を省略して

```
prompt% mknuopt ex2
```

としても同じ意味になります³⁰.

作成されるロードモジュールの名前は `mknuopt` に与えたファイル名から次のように決定されます.

システムコードファイル名	ロードモジュール名	備考
<code>ex2.cc</code>	<code>ex2</code>	拡張子は除去
<code>fitting.1.cc</code>	<code>fitting.1</code>	最後の.以降のみ除去
<code>/local/lib/nuopt/minCost.cc</code>	<code>MinCost</code>	パス名は反映されない

表 7 ロードモジュール名

³⁰ 拡張子を省略したファイル名 (`name`) が与えられたとき, `mknuopt` はシステムコードを記述したファイルを `name.cc`, あるいは `name.C` と仮定してこの順に探します. 従って `name.cc`, `name.C` の両方が存在する際には, `name.cc` が選ばれることになりますのでご注意ください.

2.3 SIMPLE 版ロードモジュールの起動

前項の要領で作られた SIMPLE 版ロードモジュールは

```
prompt% ロードモジュール名
```

としてそのまま実行するとこの問題を解くことができます．前項の例では

```
prompt% ex2
```

とします³¹．

2.2SIMPLE 版ロードモジュールの作成で挙げた例題 2 はこのモデル定義のみで完結しておりますので不要ですが，モデルがデータを外部から読み込むことを前提としている際には

```
prompt% loadmodule-name datafileName1 atafileName2...datafileNameN
```

として，SIMPLE の定める形式のデータファイル³²名(複数可)をロードモジュール名の後に並べて与えます．

³¹このとおりに入力して実行できない場合(「コマンドがない」というエラーが発生する)ときには，
prompt% ./ex2

と入力してみてください．環境変数であるパス(PATH)にカレントディレクトリである"."が含まれていない場合には，実行の際に"./"を頭に付加する必要があります．UNIX シェル環境の初期設定などで，PATH 変数に"."を含めるとこのとおりの起動方法となります．

³²第一部の 4 データファイルに解説があります．

2.4 SIMPLE 版ロードモジュールの出力

2.4.1 標準出力

SIMPLE 版ロードモジュールを実行すると、問題の求解が行われ、計算の進行が以下の様に表示されます。

```
prompt% ex2
SIMPLE 2.9.6, Copyright (C) 1994-2003 Mathematical Systems Inc.
<system code file name: ex2.cc>
Expanding objective (1/2 ex2.cc:8 name="f")
Expanding constraint (2/2 ex2.cc:11)
NUOPT 6.0.5, Copyright (C) 1991-2003 Mathematical Systems Inc.
PROBLEM_NAME                      ex2
NUMBER_OF_VARIABLES                2
NUMBER_OF_FUNCTIONS                2
PROBLEM_TYPE                       MAXIMIZATION
METHOD                             TRUST_REGION
<preprocess begin>.....<preprocess end>
<iteration begin>
    res=1.1e+001 .... 1.5e-001 ... 2.8e-008
<iteration end>
STATUS                             OPTIMAL
VALUE_OF_OBJECTIVE                  2.707106781
ITERATION_COUNT                     9
FUNC_EVAL_COUNT                     12
FACTORIZATION_COUNT                 14
RESIDUAL                           2.811617561e-008
ELAPSED_TIME(sec.)                  0.17
SOLUTION_FILE                       ex2.sol
prompt%
```

```
<system code file name: ex2.cc >
```

は SIMPLE 版インタフェースがソースファイル ex2.cc にあるモデルを問題として NUOPT に渡していることを示します。ロードモジュールが

```
prompt% loadmodule-name datafileName1 datafileName2...datafileNameN
```

のように SIMPLE のデータファイルを引数として起動された場合には

```
< reading datafile datafileName1 >
< reading datafile datafileName2 >
...
< reading datafile datafileNameN >
```

と表示されます。

```
Expanding objective (1/2 ex2.cc:8 name="f")
Expanding constraint (2/2 ex2.cc:11)
```

は、モデルに記述されている数式の解釈（展開）をモデルの記述順に行っている旨のメッセージです³³。

以降の標準出力はどんな実行においても共通して出力される NUOPT の本体部からのメッセージです。出力の内容は起動する計算手法により異なります。詳細については 3.5 標準出力をご覧ください。

標準出力の終わりはロードモジュール名に拡張子 .sol を付けた名前のファイル

SOLUTION_FILE (解ファイル)	ex2.sol
-----------------------	---------

に計算結果が書き出されたことを示しています。ロードモジュール名に .sol を付けたものが解ファイルの名前になります(Windows 版の GUI では solfile.sol に固定されています)。解ファイルの名前をパラメータファイル nuopt.prm からの指定 (4.5 その他の設定) や SIMPLE 内部からの指定 (2.9.2SIMPLE 版ロードモジュールの解ファイル名の設定) によって変更することも可能です。

2.4.2 SIMPLE 版ロードモジュールの解ファイル

SIMPLE 版ロードモジュールは計算終了と共に、解ファイル(ロードモジュール名.sol)を起動されたディレクトリに出力します。これらにはアルゴリズム停止時における、変数や関数(目的関数及び制約式)、双対変数(シャドウプライス)の値が記されています。

ここでは、解ファイルの記述のうち、SIMPLE 版ロードモジュールに特化した部分について述べます。解ファイルのその他の内容はインタフェースに依存せず、MPS ファイル入力用ロー

³³大規模な問題においてはここに時間を要する場合がありますので、経過を示しています。

ドモジュール(5MPS ファイル入力用モジュール nuopt とも共通となっておりますので、3 出力ファイルの解説にてまとめて解説してあります。

SIMPLE 版ロードモジュールの出力した解ファイルの最初には

SYSTEM_CODE_FILE_NAME	ex2.cc
-----------------------	--------

とシステムコードのファイル名 (mknuopt に与えたファイル名) が表示されます。

また、ロードモジュールの起動時にデータファイルを与えた場合にはその名前が

DATA_FILE_NAMS	data1
----------------	-------

のように表示されます。データファイルが複数指定された場合には例えば

DATA_FILE_NAMES	dataCommon,dataSpec1
-----------------	----------------------

というように"," で区切られて表示されます。引き続き表示される

%%
%%
%%
%% RESULT OF NUOPT #1
%%
%%
%%
%%

は、この次の行以降の出力が最初 (" #1" は「1 回目」ということです) の実行に対応するものであることを示します³⁴。

解ファイルのこの行以降の書式は MPS ファイル入力ロードモジュールと共通です。続いて

PROBLEM_NAME	ex2
NUMBER_OF_VARIABLES	2
NUMBER_OF_FUNCTIONS	2
PROBLEM_TYPE	MAXIMIZATION
METHOD	TRUST_REGION
STATUS	OPTIMAL

³⁴ SIMPLE 版ロードモジュールでは 1 回の実行で複数回の NUOPT の実行を行うことができます。この出力は各実行の結果を区別するためのものです。

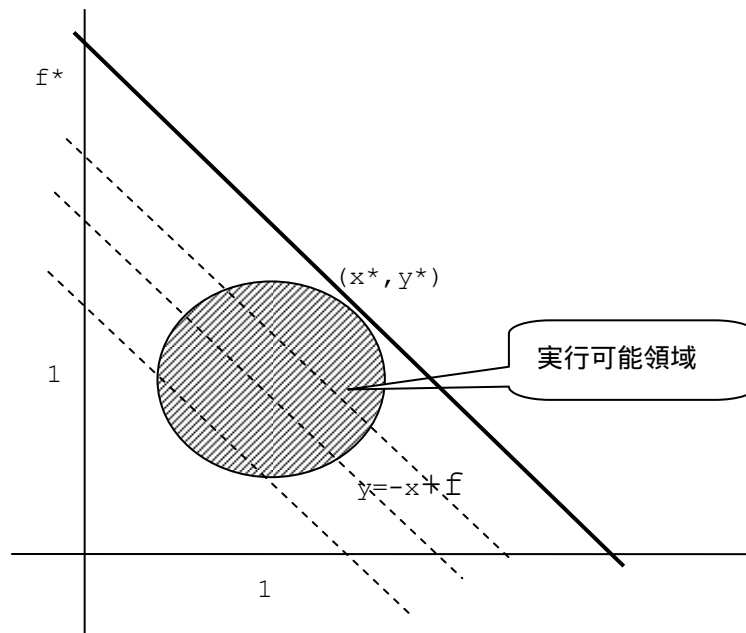
VALUE_OF_OBJECTIVE	2.707106778
ITERATION_COUNT	11
FUNC_EVAL_COUNT	13
FACTORIZATION_COUNT	17
RESIDUAL	1.184261328e-09
ELAPSED_TIME(sec.)	0.02

と，最適化に関するサマリ情報が出力されたのち

```
%%
%% VARIABLES
%%
      NAME      VALUE      STATUS [      BOUND TYPE      ]
V# 1      x      1.353553389  FREE [ -Inf  <=    x    <=  +Inf ]
V# 2      y      1.353553389  FREE [ -Inf  <=    y    <=  +Inf ]

%%
%% FUNCTIONS
%%
      NAME      VALUE      STATUS [  CONSTRAINT/OBJECTIVE TYPE ]
F# 1 OBJECTIVE  2.707106778  FREE [    OBJECTIVE (MAXIMIZE)    ]
F# 2              0.2499999976  UPPER [              <=    0.25 ]
```

のように，各変数，関数の値と上下限，双対変数に関する情報が出力され，ここから解として $(x^*, y^*, f^*) = (1.35, 1.35, 2.71)$ が得られていることがわかります．この問題の解は次の図によってより直観的に表されます．



解ファイル中の変数名は SIMPLE による問題記述 (ex2.cc) 中で Variable の宣言時に

```
Variable x(name="x"), y(name="y");
```

と name= の後に与えた文字列となります。目的関数名も同様に

```
Objective f(name="f", type=maximize); // 目的関数の宣言
```

と Objective の宣言時に与えられた文字列となります。制約式

```
pow(x-1, 2) + pow(y-1, 2) <= pow(0.5, 2);
```

には特に名前が与えられていないので、この名前に相当する部分は空欄となっています。制約式に名前を与えるには Constraint というオブジェクトを陽に

```
Constraint g(name="g"); // 名前 g を持つ制約式の宣言
g = pow(x-1, 2) + pow(y-1, 2) <= pow(0.5, 2);
```

と宣言して名前を与えてから本体をこのオブジェクトに代入します。詳細は第一部の 3 数理モデルを表現するクラス中の 3.3.7 制約式を参照して下さい。

2.5 SIMPLE における最適値の参照，最適化制御

SIMPLE は上記の様に単一の数理計画問題を記述するのみならず，

- ◆ 最適化の結果を表示する
- ◆ モデルを変化させて最適化の制御を行う

という機能があります．詳細は第一部の 5 システム制御に解説されていますが，ここでも前述した ex2.cc を元にした簡単な例題を通じてその機能を説明します．

2.5.1 SIMPLE による最適化の結果の表示

まず，2.2SIMPLE 版ロードモジュールの作成で挙げた（例題 6 で最適解における変数の値と目的関数の値を SIMPLE に表示させてみましょう．それには SIMPLE にて以下のように記述します．

```
#include "simple.h"
//
// 例題 6（変数と目的関数値の表示）
//
void ufun()
{
    Variable x(name="x"), y(name="y"); // 変数 x, y の宣言
    Objective f(name="f", type=maximize); // 目的関数の宣言

    f = x+y;
    pow(x-1,2)+pow(y-1,2) <= pow(0.5,2);
        // (x-1)^2 + (y-1)^2 <= (1/2)^2

    solve(); // 問題を解くことを明示的に指示

    // 変数値，目的関数値の表示
    cout << "x = " << x.val << "¥n";
    cout << "y = " << y.val << "¥n";
    cout << "f = " << f.val << "¥n";
}
```

solve() は SIMPLE から明示的に NUOPT を呼び出して最適化を行うことを指示する命令です．また，ここでは x, y という名で宣言されている SIMPLE のオブジェクトである変数 (Variable) はいくつかのメンバ (属性) を持っており，その一つが「現在の値」で val と名

付けられています。したがって `solve()` で最適化を行った後の

```
x.val    y.val
```

は、変数 `x, y` の現在値すなわち最適化後の値を参照していることになります。

この例では明示的にソルバを呼び出していますが、前項で説明した最初の例 (`ex2.cc`) でこれを書かずとも最適化が行われたのは、`ufun()` 中で一度も `solve()` を呼ばない場合には `ufun()` を解釈し終わったのちに自動的に一度だけ `solve()` を行う、と定めているためです。この例の場合には、`x, y` の値を表示する時点でそれらが最適化後の値となっている必要があるため、明示的に `solve()` を呼びました。さもないと最適化は `x, y` の値が表示された後に行われることになり、`x, y` の初期値 (`SIMPLE` の仕様としてデフォルトで 0 と定めています) が表示されてしまいます。

C++ の演算子 "`<<`" は右にあるものの内容を「出力ストリーム」にのせて最も左にある `cout` (標準出力) へと送るという意味がありますので、

```
cout << "x = " << x.val << "\n";
cout << "y = " << y.val << "\n";
```

という文で、`"x = "` や `"y = "` という文字列に引き続いて最適化後の `x, y` の値を表示することができます。目的関数である `f` にも現在の値を示す `val` というメンバがありますので同様に

```
cout << "f = " << f.val << "\n";
```

とすることによって、最適化における `f` (目的関数) の値を表示することができます。

現在の値を示す `val` 以外にも `SIMPLE` の各オブジェクトはその意味に応じてこの方法で参照できる多数のメンバ (属性) を持っています。表 8 がそれをまとめたものです。

`SIMPLE` の各オブジェクトのメンバ (属性) の出力方法の詳細については第一部の 5 システム制御中 5.3 オブジェクトの参照値の表示 (関数 `print, cout, simple_printf`) にあります。

メンバ	意味	参照可能なクラス
val	現在の値	Set, OrderedSet, CyclicSet, Sequence, Expression, Objective, Variable, IntegerVariable, DiscreteVariable, VariableParameter, Parameter, Constraint
init	解く前の値	Expression, Objective, Variable, IntegerVariable, Constraint
dual	双対変数値	Variable, IntegerVariable, Constraint
ub	上限	Variable, IntegerVariable, Constraint
lb	下限	Variable, IntegerVariable, Constraint

表 8 参照可能なクラスのメンバ

cout への出力を記述したファイルを ex2-1.cc とします .mknuopt にて作成したロードモジュールを実行すると最適化の実行が終わった後、標準出力に

```
x = 1.35355
y = 1.35355
f = 2.70711
```

と表示されます。

また、内点法の反復回数等、最適化計算自体のサマリ情報は SIMPLE では大域変数 result に格納され、次のようにして表示することができます。

```
// 変数の総数
cout << %nvars      = " << result.nvars      << "%n";
// 関数の総数
cout << %nfunc      = " << result.nfunc      << "%n";
// 内点法の反復回数
cout << "iters      = " << result.iters      << "%n";
// 関数の評価回数
cout << "fevals     = " << result.fevals     << "%n";
// 最適解
cout << "optValue   = " << result.optValue   << "%n";
// 反復の停止条件
cout << "tolerance  = " << result.tolerance  << "%n";
// 停止時における KKT 条件の残差
cout << "residual   = " << result.residual   << "%n";
// 所要時間
```

```
cout << "elapsedTime = " << result.elapsedTime << "\n";
// 終了時ステータス（成功時：0，失敗時：エラーコード）
cout << "errorCode = " << result.errorCode << "\n";
```

上記の ex2-1.cc の後にこれらの文を挿入して実行すると、次の様な出力が得られます。

```
nvars      = 2
nfunc      = 2
iters      = 11
fevals     = 13
optValue   = 2.70711
tolerance  = 1.49012e-06
residual   = 1.18426e-09
elapsedTime = 0
errorCode  = 0
```

表 9 は result に関して参照可能なメンバについての一覧です。

名称	型	意味
nvars	int	変数の数
nfunc	int	関数の数
iters	int	内点法の反復回数
fevals	int	関数評価回数
optValue	double	目的関数値
tolerance	double	内点法の収束判定値
residual	double	解における最適性条件の残差
elapsedTime	double	所要計算時間
errorMessage	char*	エラーメッセージ
errorCode	int	終了時ステータス (成功時：0，失敗時：エラーコード)

表 9 result に関して参照可能なメンバ

2.5.2 SIMPLE による最適化制御

solve() を複数回用いることによってモデルを変更しながら複数の最適化を連続して実行することができます。例えば 2.2SIMPLE 版ロードモジュールの作成で挙げた(例題 6 に、元の目的関数の他に最大化したい目的関数

$$f_2 \equiv x - y$$

がもう一つあるものとし，この状況に次の手順で対処するケースを考えましょう．

1. 最初の目的関数で最大化を行う（例題 6 を解く）．
2. f を最初の最適化で得られた最適値 f^* の 9 割以上に保つ様な以下の制約を加える

$$x + y \geq 0.9 \cdot f^*$$

3. 上記の制約のもとで f_2 に関する最適化を行う．

上記の手順を SIMPLE では次のように記述することができます．制約を付加している部分では最初の最適化の結果得られた目的関数の値 (`f.val`) を制約式の右辺値に用いています³⁵．新たに定義された式は元の問題に追加され，以前の制約式は明示的に除去（**第一部の 5 システム制御中 5.8 特定の制約式の削除/復帰 `deleteCo()`/`restoreCo()` 関数**）しない限り有効となります．2 度目の `solve()` の際，元の式を定義し直す必要はありません．2 度目の最適化の際には，目的関数が二つ定義されていることになっていますが，複数の目的関数が存在する場合には最も後に定義されたものが有効という規則（**第一部の 5.2 ソルバの起動 `solve()` 関数**）により，2 度目の最適化においては所望の f_2 に関する最適化が行われます³⁶．最後には 2 度目の最適化の結果得られた変数値と，最初の最適化に用いた目的関数値を表示させていま

```
#include "simple.h"
//
// 例題 6（再最適化を行うケース）
//
void ufun()
{
    Variable x(name="x"), y(name="y"); // 変数 x, y の宣言
    Objective f(name="f", type=maximize); // 目的関数の宣言

    f = x+y;
    pow(x-1,2)+pow(y-1,2) <= pow(0.5,2);
        // (x-1)^2 + (y-1)^2 <= (1/2)^2

    solve(); // 最初の最適化の実行

    x+y >= 0.9*f.val; // 制約の付加

    Objective f2(name="f2", type=maximize);
    f2 = x-y; // 新たな目的関数の定義
```

す．

³⁵ 第一部の 5.3 オブジェクトの参照値の表示 (関数 `print`, `cout`, `simple_printf`) で述べているように，`f.val` 等は SIMPLE における定数を示すオブジェクト `Parameter` と全く同様に扱うことができます．

³⁶ ここで `solve(f2)` と書いて f_2 の最適化を行うことを陽に指示することもできます．

```

solve(); // 2度目の最適化実行

// 変数値, 目的関数値の表示
cout << "x = " << x.val << "\n";
cout << "y = " << y.val << "\n";
cout << "f  = " << f.val << "\n";
cout << "f2 = " << f2.val << "\n";
}

```

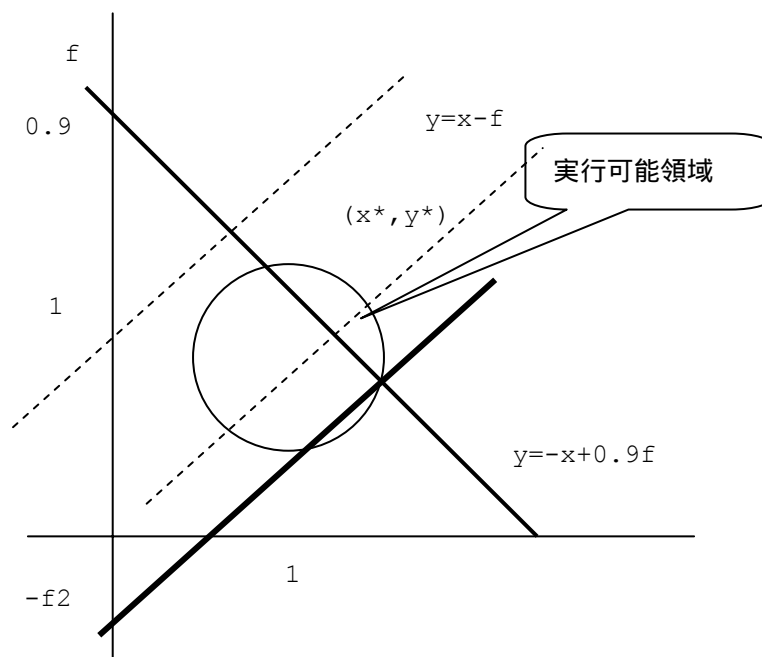
このファイル(ex2-2.cc とします) から mknuopt によって作成されたロードモジュールを実行すると、最適化が2度行われ、その後に

```

x = 1.49639
y = 0.940008
f  = 2.4364
f2 = 0.55638

```

という表示が得られます。2度目の最適化の結果、 f は元の最適値 2.71 から正確に 0.9 倍となり、 f_2 の最適解 0.56 が得られたことがわかります。この解は次の図によって直観的に表されます。



2.6 SIMPLE のデータファイルの利用

最適化問題の修正を行いたい場合には問題を記述したシステムコードを変更して再び `mknuopt` によりロードモジュールを作りなおすことになりますが、例え `ed` は問題中に現れる定数値のみしか変わらないのにいつもこの操作を行うのは面倒です。

`SIMPLE` はそのようなとき、問題記述を数値データに依存しない形で記述しておき、データを外部から与えるという方法をとることができます。

例えば、2.2 `SIMPLE` 版ロードモジュールの作成で挙げた（例題 6 において上図における円を定義していた

$$\text{条件} \quad (x-1)^2 + (y-1)^2 \leq \left(\frac{1}{2}\right)^2$$

を

$$\text{条件} \quad (x-2)^2 + (y-3)^2 \leq \left(\frac{5}{2}\right)^2$$

等書き換えるということが頻繁に発生する場合を想定してみます。`SIMPLE` ではパラメータ (x_0, y_0, r) を用いた形

最大化 $x + y$

条件 $(x - x_0)^2 + (y - y_0)^2 \leq r^2$

（例題 7 二次計画問題 2）

で問題を記述し、パラメータの具体的な値はデータファイルを通じて与えることができます。`SIMPLE` での記述は以下のようになります。

```
#include "simple.h"
//
// 例題 6a
//
void ufun()
{
    Variable x(name="x"), y(name="y"); // 変数 x, y の宣言
    Objective f(name="f", type=maximize); // 目的関数の宣言

    Parameter x0(name="x0"), y0(name="y0"), r(name="r"); // パラメータ
    f = x+y;
```

```

pow(x-x0,2)+pow(y-y0,2) <= pow(r,2);
// (x-x0)^2 + (y-y0)^2 <= r^2
}

```

と記述します。システムコード名を `ex21.cc` として, `mknuopt` を

```
prompt% mknuopt ex2a.cc
```

として起動すると,ロードモジュール `ex2a` が得られます。こうして,パラメータ (x_0, y_0, r) の値を `SIMPLE` の定める形式(第一部の 4 データファイル) にて以下の様に記述したものを用意します。

```
x0 = 2; y0 = 3; r = 2.5;
```

このデータファイルを(ファイル名を `ex2a.dat` とします)ロードモジュール `ex2a` に与えて

```
prompt% ex2a ex2a.dat
```

と実行すると

$$\text{条件 } (x-2)^2 + (y-3)^2 \leq \left(\frac{5}{2}\right)^2$$

に対応する解が得られます。このデータファイルを変更すれば,同一のロードモジュール `ex2a` にて (x_0, y_0, r) の異なった値に対する解を得ることができます。

`ex2a.cc` の様な具体的な値が入っていない状態の問題記述を「モデル」, データを導入してできた具体的な問題をモデルと区別して「システム」, モデルとデータを組み合わせてシステムを作る操作を「展開」と呼びます。

`SIMPLE` では上記に紹介した定数に相当するオブジェクトである `Parameter`, 集合に相当するオブジェクトである `Set` を利用して大規模になり得るシステムをも簡潔な「モデル」の形にて記述するための手段を提供しています。

2.6.1 目的関数に関して

`SIMPLE` による問題定義に複数の目的関数が存在する場合, 単に `solve()` とすると, 最後に代入されたものが目的関数として用いられます(第一部の 5.2 ソルバの起動 `solve()` 関数)。

```

#include "simple.h"
void ufun()
{
    Variable x;
    Objective f(type=minimize);
    Objective g(type=minimize);
    Objective h(type=minimize);

    -0.5 <= x <= 0.5;
    g = (x-1)*(x-1);
    f = (x+1)*(x+1);
    solve();    // f が目的関数となる

    h = x*x;
    solve();    // h が目的関数となる
}

```

目的関数に選ばれるのは最も後に代入されたもので、最も後に宣言されたものではないことに注意して下さい。上記では、 f は g よりも宣言は前ですが、後に代入されているので最初の `solve()` で目的関数となっているのは f です。

代入された順番に依らずに `solve()` において最適化する目的関数を指定したい場合には目的関数を陽に指定する `solve()` の呼び出し形式を用いて、例えば

```
solve(g); // g を目的関数とした最適化を行うことを陽に指定
```

等とします。

2.6.2 SIMPLE 版ロードモジュールの標準出力頻度の設定

特に指定を行わない場合の標準出力頻度は標準 (normal) ですが、繰り返し最適化を行う場合等で NUOPT 本体からの標準出力を抑制したい場合には `nuopt.prm` において

```
output:mode = silent
```

とするか、SIMPLE による問題記述中に

```
options.outputMode = "silent";  
solve(); // 標準出力を行わない
```

として下さい。ただし `nuopt.prm` でこの指定を行った場合には、`nuopt.prm` を読み込んだ旨のメッセージは抑制することができません。

2.7 整数計画問題に関するパラメータ設定

SIMPLE にて整数計画問題を記述するには通常の変数 (Variable) の代わりに IntegerVariable というオブジェクトを使用します。(第一部の 3.3.5 整数変数) 例えば次は NUOPT/SIMPLE チュートリアル(別冊)でも解説しているナップサック問題を記述した例です。

```
void ufun()
{
    Set S(name="S");
    Element i(set=S);
    IntegerVariable x(index=i,type=binary); // 整数(0/1) 変数
    Parameter c(name="c",index=i);
    Parameter a(name="a",index=i);
    Parameter b(name="b");
    Objective obj(type=maximize);

    obj = sum(c[i])*x[i],i);    // 目的関数
    sum(a[i]*x[i],i) <= b;    // 制約条件
}
```

NUOPT は整数計画問題を分枝限定法によって解きますが、その際分枝限定法アルゴリズムへとモデルを作成する側の情報を入力することで、パフォーマンスの向上が可能な場合がございます³⁷。SIMPLE の IntegerVariable は以下の意味を持つ分枝限定法のパラメータに対応するメンバを持っており、モデル定義側から設定が可能です。

パラメータ名	値	デフォルト値	意味
pri	1 以上の正の整数	擬コストに基づく推定	分枝優先順位
upc	正の実数	緩和問題からの推定値	押し上げ擬コスト
dpc	正の実数	緩和問題からの推定値	押し下げ擬コスト
dir	実数	0	分枝方向

表 10 分枝限定法のパラメータに対応するメンバ

例えば上の問題のモデル記述では

³⁷ Ver.8 の NUOPT に組み込まれている汎用の優先順位推定ルールは広汎な問題に対し、良い結果を与えることが実験的に検証されておりますので、まずはこの設定を行わないでお試しになることをお勧めします。

```

x[i].pri = 5; // すべての x に対して優先度 5 を設定
x[3].pri = 1; // x[3] に対してのみ優先度 1 を設定
Parameter upc(name="upc",index=i);
x[i].upc = upc[i]; // 押し上げ擬コストを upc の値に設定

```

等のように設定することができます。以下は上記に関する細則です。

- ♦ `pri`(分枝優先順位)の値は小さいものが先に分枝されることになります。分枝優先順位の与えられている変数と与えられていない変数が混在している場合には、分枝優先順位の与えられていない変数は最低の優先順位が与えられたと見なされます。優先順位として 0 以下の値は与えられても無視し、優先順が与えられないのと同じ扱いとします。
- ♦ `upc`, `dpc`(擬コスト)の値が大きい変数が先に分枝されます。擬コストは変数を押し上げ/下げるときの影響度を連続値で表現したもので、分枝の順番に影響します。与えられているものと与えられていないものが混在する場合、擬コストが与えられていない変数については擬コストがすべて零であるものとします。どの変数にも擬コストが全く与えられていない場合には、擬コストを緩和問題から推定した値に設定します。負の擬コストは無視し、与えられていないのと同じ扱いとします。
- ♦ `dir` は最初に分枝する方向に関してのパラメータで符号のみが問題になります。正なら押し上げを優先、負なら押し下げを優先、0 なら、擬コストからの推定を用いてどちらを優先するかを決定します。

2.8 制約充足問題ソルバのための重みおよび目標値設定

制約充足問題ソルバ (wcsp) にパラメータとして、制約式の「重み」また、目的関数の「目標値」を与えることができます。重みは正の実数で制約式毎に定義することができます。重みの大きな制約式は優先的に満たされるように解の探索が行われますので、重要度の高い制約式の重みを大きくしておくことで、より有用な解が得られることが期待できます。

目的関数の目標値は目的関数値の目安となる値です。制約充足問題ソルバ (wcsp) 内部では、目的関数 $f(x)$ と、目標値 μ が与えられた際に、

a) 最小化問題であれば

$$\mu - f(x) \geq 0$$

b) 最大化問題であれば

$$f(x) - \mu \geq 0$$

というソフト制約を定義して目的関数を扱っております。目標値 μ を実際の目的関数値よりも悪い値と設定すると、早期に a), b) が満たされていると判断されて所望の解が得られない可能性がありますのでご注意ください。目標値は実際の目的関数よりも良い値としておけば問題はありませんが、その値がどの程度実際の目的関数と離れているかによって制約充足問題ソルバ (wcsp) の出力は一般に変化します³⁸。

重みおよびターゲット値の設定は SIMPLE のモデル記述中に行うことができ、以下に紹介する方法で定義することができます。

1. `hardConstraint()`, `softConstraint(値)`

モデル中に

```
hardConstraint();
```

と記述すると、以降に定義した制約はすべてハード制約（重み の制約）とみなされます。

同様にモデル中に

```
softConstraint(値);
```

と記述すると、以降に定義した制約の重みは「値」となります。

「値」には整数値 (int) または実数 (double) あるいは `Parameter` オブジェクトを用いて -1 または非負の定数を与えます。-1 を与えると の意味となり、`hardConstraint()` と同一の意味となります。0 を与えると重みの指定を行わないのと同じ意味になり、後述の

³⁸制約充足問題ソルバは内部で重みの再調整を行うので、このようなことはそれほど多くは生じませんが、他にも満たすのが難しい制約式が多数存在する場合にはこのようなことが起こる可能性があります。

`options.defaultConstraintWeight` の設定に従います³⁹ .

2. `Objective(..., target=値)`

目的関数の定義において

```
Objective cost(type=minimize, target=値);
// 最小化される目的関数値の目標値
```

とすることによって、目標値 μ を与えます。「値」には整数値 (int) または実数 (double) あるいは `Parameter` オブジェクトを用いることができます。指定されない目的関数の目標値は `options.defaultObjectiveTarget` の指定に従います。

3. `options.defaultConstraintWeight, options.defaultObjectiveWeight`

```
options.defaultConstraintWeight = 値;
```

と記述することで、特に重みの指定がない制約式の重みを設定することができます。値は -1 または非負の数です。`solve()` の前であれば記述の場所は任意です。この設定は前項の `hardConstraint()`、`softConstraint()` (0 以外の値) のいずれかが記述された後に記述された制約式には無効です。デフォルト値は -1 (ハード制約) ですので、重みの設定を何も行わない制約式、`softConstraint(0)` と設定された制約式はすべてハード制約として定義されます。

```
options.defaultObjectiveWeight = 値;
```

と記述することで、目的関数に変形された制約 (上記の `a`), `b`) の重みを設定することができます。`solve()` の前であれば記述の場所は任意です。デフォルト値は 1 (重み 1 のソフト制約) です。制約充足問題ソルバー (wcsp) は内部で重みの再調整を行うので通常あまり変更する必要はありませんが、目的関数の値のオーダー (大きさ) が制約式とかけ離れている場合にはそれに合わせて調整すると良い場合があります⁴⁰。`options.defaultConstraintWeight`、`options.defaultObjectiveWeight` に 0 を設定すると、デフォルト (ハード制約) とみなされます。

4. `options.defaultObjectiveTarget`

³⁹ `defaultConstraintWeight` は設定しないと -1 なので、`softConstraint(0)` のみとすると、ハード制約という意味になります。ご注意ください。

⁴⁰ 例えば制約式の大きさが 1 程度で目的関数の大きさが 10000 程度である場合には、重みを 1/10000 としておくことで、目的関数に過度に傾斜することを避けます。

```
options.defaultObjectiveTarget = 値
```

と記述することで、目標値の指定のない目的関数の目標値を設定することができます。
`solve()`の前であれば記述の場所は任意です。デフォルト値は0です。

以下は上記設定の利用例です。

```
Constraint c1,c2,c3;
x + y <= 10;
    // 重み指定がないので, options.defaultConstraintWeight の設定に従う
softConstraint(10);
c1 = x - y <= 50; // 重みは 10
c2 = x*x + y*y >= 25; // 重みは 10
hardConstraint();
c3 = 2x+5y >= 10; // ハード制約として定義される
Objective f(type=minimize,target=10); // 目標値は 10 と設定
f = 2*x + 5*y;
options.defaultConstraintWeight = 5;
options.defaultObjectiveTarget = 20;
Objective g(type=maximize);
    //目標値の設定がないので options.defaultObjectiveTarget に従う
g = 3*y*y;
solve(f); // f を目的関数 (目標値 10) としての最適化実行 (g は無視)
solve(g); // g を目的関数 (目標値 20) としての最適化実行 (f は無視)
```

2.9 SIMPLE 版ロードモジュールの動作の制御

NUOPT 本体の動作に関して特別な指定を行って、動作をカスタマイズする方法として、SIMPLE 版ロードモジュールには次の 3 つの方法が用意されています。

1. パラメータの値を特定の文法で記述したパラメータファイル `nuopt.prm`⁴¹を作り、ロードモジュールを起動するディレクトリに置く。
 2. データ名 `nuopt.prm` にパラメータファイルの内容を設定する。
 3. SIMPLE のモデル記述において大域変数 `options` のメンバ(属性)を設定する。
- ただし、Windows 版 GUI を利用している場合には、2. と 3. のみが可能です。

1. は MPS ファイル用ロードモジュールに対する場合と同じ方法です。例えば以下の内容

```
begin
method: line          * 直線探索を用いる
output: mode = silent * 標準出力を抑制する
end
```

が書かれた `nuopt.prm` の存在するディレクトリにて、2.2SIMPLE 版ロードモジュールの作成で作った `ex2` を起動すると、

```
<reading parameter file: nuopt.prm >
begin
method:line
output:mode=silent
end
```

と標準出力に表示され、この内容に従って以降の最適化実行がカスタマイズされていることを示します。パラメータファイルで設定できるパラメータの意味やパラメータファイルの文法については 4 パラメータ設定で詳しく解説しています。ただし、この方法は起動ディレクトリという概念がない Windows 版の GUI では不可能です。

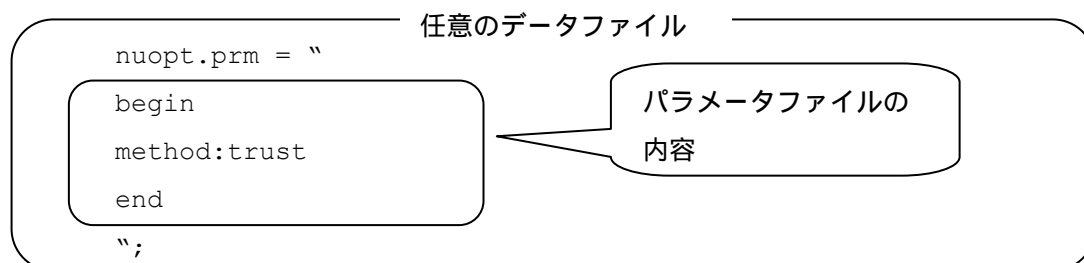
2. は SIMPLE のデータのアイテムの形でパラメータファイルの内容を与える方法です。特殊なデータ名として

`nuopt.prm`

というデータ名が定義されており、この内容(文字列)の設定は NUOPT のパラメータファイ

⁴¹この名前は固定されています。

ルの内容であると解釈されます．具体的にはデータファイル内の任意の場所に



のように書くと，この値の文字列と等価な NUOPT のパラメータファイルを与えたこととなります．Windows 版の GUI で nuopt.prm を使ってパラメータを設定するにはこの方法をとります．

3．はパラメータ設定を SIMPLE の問題記述中で行う方法です．モデルを記述した後 (solve() にてソルバを起動する前) に options というオブジェクト (この名前は決定されており，宣言の必要はありません) のメンバを，例えば

```

#include "simple.h"
//
// 例題 7
//
void ufun()
{
    Variable x(name="x"), y(name="y"); // 変数 x, y の宣言
    Objective f(name="f", type=maximize); // 目的関数の宣言

    f = x+y;
    pow(x-1,2)+pow(y-1,2) <= pow(0.5,2);
    // (x-1)^2 + (y-1)^2 <= (1/2)^2
    // 以下はソルバパラメータの設定
    options.method = "line"; // 最適化手法の定義
    options.outputMode = "silent"; // 標準出力の抑制
}
  
```

の様に設定することによって行います．options のメンバは NUOPT の実行を制御するパラメータの一部に対応しています．表 11 はその一覧です．options のメンバは nuopt.prm で設定できるパラメータ指定のすべてではありません．nuopt.prm で設定できるもののうち，変更頻度の高いものをこちらでも設定できるようにしてあります．そのため，nuopt.prm を用いる方が細かい設定ができます．

名称	選択	Default	意味
outputMode	"silent", "normal",	"normal" "	標準出力モード [output:mode = normal]
method	"auto", "lipm", "lepm", "line", "higher", "tipm", "tepm", "trust", "bfgs", "lbfgs", "simplex", "asqp", "lsqp", "tsqp", "wcsp", "rcpsp"	"auto"	求解アルゴリズム [method:auto] 詳細については NUOPT/SIMPLE マニュアル 第二部 4.2 最適化手法に関する設定 を参照のこと。
scaling	"off" "on"	"on"	スケーリングを行うか否か [scaling:on]
maxitn	int	150	内点法の反復回数の最大 [crit:maxitn = 150]
eps	double	自動設定	内点法の停止条件 (内点法専用) [crit:eps = 1.0e-8]
addToCutoff	double	0	足切り点設定用パラメータ (分枝限定法専用) [branch:addtocutoff = 0]
cutoff	double	未定義	足切り点 (分枝限定法専用) [branch:cutoff = 1.8]
p	int	10	探索深さ (分枝限定法専用) [branch:p = 10]

maxnod	int	-1 (無制限)	探索問題数上限 (分枝限定法専用) [branch:maxnod=100000]
maxtim	int	-1 (無制限)	計算時間上限 (秒) (分枝限定法と内点法全般) [branch:maxtim=3600]
maxmem	int	-10 (UNIX/ Linux:無制限 Windows :10Mb)	分枝限定法のメモリ利用量上限 (Mb), 残り利用可能メモリによる 制限 ⁴² (Windows 版のみ, 負値, Mb) [branch:maxmem=500]
gaptol	double	-1 (指定なし)	上下界ギャップの下限 (この値を下回ったら停止)
tolx	double	1.0e-8	主問題の実行不可能性判定値 (単体法のみ) [param:tolx=1.0e-8]
told	Double	1.0e-6	双対問題の実行不可能性判定値 (単体法のみ) [param:told=1.0e-6]
maxintsol	int	-1 (無制限)	整数解取得個数上限 (秒) [branch:maxintsol=3]
iisDetect	"off" "on"	"on" (行う)	実行不可能な行集合 (IIS) 探索 を行う/行わない
outfilename	char*	0 (未定義)	NUOPT の解ファイル名 "_NULL_"とすると出力を行わない [output:name=myout]
noDefaultSolve	int	0	solve() を陽に呼ばないと求解 を行わない
noDefaultSolout	int	0	Solout() を陽に呼ばないと解の 出力を行わない
outputParameter		0	Parameter, Set, Element, Exp
outputSet	int	0	ression の CSV ファイル出力を
outputElement		0	行うかどうか

⁴² maxmem に負の値を設定すると, システムの利用可能なメモリが -maxmem [Mb] を切ったら実行を停止します (Windows 版のみの機能). デフォルト値 -10 は UNIX 版では無制限という意味ですが, Windows 版では 10MByte を切ったら実行を停止するという意味です.

outputExpression		1	
multDataPolicy	int	0 (許さない)	同一のデータについてデータを重複して与えることを許すかどうか (1 に設定すると警告扱いとなる)
defaultConstraintWeight	double	-1	指定のない制約式の重み (デフォルトはハード制約)
defaultObjectiveWeight	double	1	目的関数を変形した制約式の重み (デフォルトは重み 1 のソフト制約)
defaultObjectiveTarget	double	0	目的関数の目標値 (デフォルトは 0)

表 11 NUOPT の実行を制御するパラメータ

表 11 で[]内は、同じパラメータの設定を `nuopt.prm` で行う場合の記述例です。意味に関する説明は[]内の記述ごとに 4 パラメータ設定に記載していますのでご覧下さい。

`options` を用いたパラメータ指定は直後の `solve()` で起動される以降の最適化すべてに有効ですので、複数回の最適化を行う場合にはご注意下さい。

```
void ufun()
{
    ....
    // パラメータの設定 (1)
    options.method = "simplex"; // simplex 法を指定
    options.outputMode = "silent"; // 標準出力を抑制
    solve(); // 単体法で、標準出力を抑制
    ....
    // パラメータの設定 (2)
    options.method = "trust"; // 信頼領域法を指定
    solve(); // 信頼領域法で、標準出力は抑制 (前の設定が残る)
    ....
}
```

パラメータ指定の優先順は

2. (データファイルによる指定)

1. (`nuopt.prm` による指定)

3. (モデル内 `options.` による指定)

の順に弱くなります。すなわち、あるパラメータに関して `nuopt.prm` による指定と `options` による指定が両方存在した場合には、**`nuopt.prm` による指定が優先**されます。

パラメータ指定が全くない場合、すべてのパラメータはデフォルト設定（特に指定を行わない場合の設定）となりますので、パラメータファイルや `options` による指定はデフォルト以外の設定が生じた場合にのみ必要になります。

デフォルト設定は通常の実行に際して問題がない様に選ばれておりますが、変更の可能性が高いものに関して次項以降でデフォルト設定と変更方法を解説します。

2.9.1 SIMPLE 版ロードモジュールの最適化手法の設定

最適化手法の指定を特に行わなかった場合、SIMPLE 版ロードモジュールは問題に対していずれの最適化手法を適用するかをその中に含まれる関数の非線形性や変数の整数性、含まれているクラスに応じて以下の様にして決定します。

関数がすべて線形で整数変数を含む 単体法 (`simplex`)

目的関数が非線形で整数変数を含む 有効制約法 (`asqp`)

関数がすべて線形

線形計画問題専用内点法 (`higher`)

上記以外

信頼領域法 (`tipm`)

DiscreteVariable,selection を含む 重み付き制約充足問題ソルバー (`wcsp`)

Activity,ResourceRequire

ResourceCapacity を含む 資源制約付スケジューリングエンジン (`rcpsp`)

最適化手法の選択について詳細は第二部 4.2.14.2.1 をご参照ください。

2.9.2 SIMPLE 版ロードモジュールの解ファイル名の設定

最適化終了後に出力される解ファイル名は設定を行わないとロードモジュール名となります。変更したい場合には `nuopt.prm` にて

```
begin
output: name = myout      * 解ファイルのルート名の指定
end                      (myout.sol を出力)
```

と設定して下さい。

SIMPLE 中から `options.outfilename` を

```
void ufun()
{
    options.outfilename = "myout";
    ...
    solve(); // myout.sol を出力
    ...
}
```

と与えても同じです⁴³。

SIMPLE 内部で値を参照するので、解ファイルが不要の際には `nuopt.prm` にて

```
begin
output: name = _NULL_      * 解ファイル出力を行わない
end
```

あるいはモデル記述中にて

```
options.outfilename = "_NULL_";
...
solve(); // 解ファイル出力を行わない。
```

として解ファイルルート名を"`_NULL_`"という特殊な名前に設定することによって、解ファイルの出力を抑制することができます。

2.10 資源制約付きスケジューリング問題のための重みの設定

資源制約付きスケジューリング問題ソルバ(rcpsp) には、資源の利用可能量、目的関数、一般の考慮制約に対し、重みを設定する事が出来ます。与えられる重みは正の整数値とします(実数の場合には小数点以下は切り捨てられます)。資源の利用可能量、目的関数の実際の利用、一般の考慮制約に関しては、それぞれ別冊「NUOPT__SIMPLE チュートリアル」の4章「4.2.3 クラス ResourceCapacity, 4.2.4 目的関数の設定, 4.6 一般の考慮制約の導入」をご覧ください。

資源の利用可能量への重みの設定

資源の利用可能量に関しては、クラスの宣言と同時に重みを設定します。

⁴³ただし、`outfilename` の指定は最初の `solve()` の直前に行ったものが有効になり、`solve()` を行った後に変更しても反映されません。

```
ResourceCapacity
```

```
cap(resource=Set(Element),timeStep=Set(Element),weight=Parameter);
```

weight に渡す Parameter は resource,timeStep に渡す Element で添え字付けする事が可能で、各資源毎、時間ステップ毎に重みを設定する事が出来ます。

目的関数への重みの設定

目的関数への重みの設定は、制約充足問題ソルバ(wcsp)の時と同様です。ただし、納期遅れ最小化を目的関数に設定した場合には、目的関数に重みを設定する事は出来ませんのでご注意ください。

一般の考慮制約への重みの設定

一般の考慮制約への重みの設定も制約充足問題ソルバ(wcsp)の時と同様ですが、設定されている目的関数によって、重みの与え方が異なります。最後の作業の終了時刻最小化時には、必ず重みを与えなければならず、hardConstraint() は使用出来ません。その為、ハード制約として扱いたい場合には、代わりに softConstraint(値) に十分大きな値を設定する必要があります。反対に、納期最小化を行った場合には、重みを設定する事は出来ず、必ずハード制約として扱われます。

3. 出力ファイルの解説

ここでは,

$$\begin{array}{ll}
 \text{最小化} & -3x_1 - 2x_2 - 4x_3 \\
 \text{条件} & x_1 + x_2 + 2x_3 \leq 4 \\
 & 2x_1 + 2x_3 \leq 5 \\
 & 2x_1 + x_2 + 3x_3 \leq 7 \\
 & x_1 \geq 0, \ x_2 \geq 0, \ x_3 \geq 0
 \end{array}$$

(例題 8 線形計画問題)

の求解結果として得られる解ファイル (UNIX/LINUX 版: ex1.sol, Windows 版: solfile.txt) を例として内容の解説を行います.

3.1 解ファイルのヘッダ部

解ファイルにおいて, 先頭から

```
%%
%% VARIABLES
%%
```

という表示よりも前の部分 (ヘッダ) には最適化実行に関する以下のサマリが表示されます.

例えば (例題 8 を線形計画問題専用の内点法 (デフォルト) で解いた際のヘッダ部分は

```

NUMBER_OF_VARIABLES                3
NUMBER_OF_FUNCTIONS                 4
PROBLEM_TYPE                        MINIMIZATION
METHOD                              HIGHER_ORDER
STATUS                              OPTIMAL
VALUE_OF_OBJECTIVE                   -10.5
ITERATION_COUNT                      6
FUNC_EVAL_COUNT                      9
FACTORIZATION_COUNT                  7
RESIDUAL                             1.354127444e-008
ELAPSED_TIME (sec.)                  0.01

```

となります.

サマリのタイトルと内容の一覧は以下の通りです。

タイトル	解説	備考
PROBLEM_NAME	問題名	SIMPLE 版:ロードモジュール名 MPS ファイル版:TITLE の内容
NUMBER_OF_VARIABLES	変数の総数	
NUMBER_OF_FUNCTIONS	関数 (目的関数を含む) の総数	
PROBLEM_TYPE	最小化 / 最大化のいずれか	
METHOD	適用した最適化手法	
ERROR_TYPE	発生したエラーの種類	エラー発生時のみ
STATUS	最適化が終了した状態	
VALUE_OF_OBJECTIVE	目的関数値	
ITERATION_COUNT	反復回数	内点法のみ
FUNC_EVAL_COUNT	関数の評価回数	内点法のみ
FACTORIZATION_COUNT	行列の分解回数	内点法のみ
RESIDUAL	最適性条件の充足度合	分枝限定法以外
SIMPLEX_PIVOT_COUNT	単体法の反復回数	単体法のみ
PARTIAL_PROBLEM_COUNT	部分問題数	分枝限定法のみ
DUAL_SIMPLEX_PIVOT_COUNT	双対単体法の反復回数	分枝限定法のみ
PENALTY	重みつき制約違反量	wcsp/rcpsp 適用時のみ
TERMINATE_REASON	計算終了理由	wcsp/rcpsp 適用時のみ
DETECTED_IIS_SIZE	検出された IIS に含まれる行の数	IIS 特定成功時のみ
(#IIS_RELATED_VAR)	IIS に含まれている行に含まれる変数の数	IIS 特定成功時のみ
INFEASIBILITY_OF_IIS	IIS 全体での実行不可能性	IIS 特定成功時のみ
NO_IIS_FOUND_BY	IIS 検出失敗の原因	IIS 特定失敗時のみ
(#NONLINEAR_CONSTR.)	IIS 検出失敗の原因の可能性のある非線形制約の数	IIS 特定失敗時のみ
NUMBER_OF_ACTIVITIES	アクティビティの総数	rcpsp のみ

NUMBER_OF_RESOURCES	資源の総数	rcpsp のみ
NUMBER_OF_PRECEDENCE	先行制約の総数	rcpsp のみ
NUMBER_OF_IMPRECEDENCE	直前先行制約の総数	rcpsp のみ
NUMBER_OF_GENERAL_CONSTRAINT	一般の考慮制約の総数	rcpsp のみ
NUMBER_OF_MODES	モードの総数	rcpsp のみ

表 12 サマリのタイトルと内容

ヘッダ部分は上から順番に，変数/関数⁴⁴の総数 (3/4)，この問題に適用された最適化手法 (HIGHER_ORDER)，最適化終了時の状態 (OPTIMAL)，目的関数値 (-10.5)，内点法の反復回数 (6)，関数の評価回数 (9)，内点法のステップ方向を求めるための行列の分解回数 (7)，解における最適性条件の残差 (約 1×10^{-8})，秒単位の計算時間 (0.01) を示しています。

STATUS が OPTIMAL とは，最適化が正常終了であることを示します。何らかの原因で異常終了した場合には，ここが例えば

STATUS	NON_OPTIMAL
ERROR_TYPE	<<NUOPT 14>> integrality is violated.

のようになり，次に現れる ERROR_TYPE という行に NUIPT のエラー番号とメッセージ (A.2.1NUOPT のエラー/警告) が表示されます。

(例題 8 に単体法を適用したときのヘッダ部は

NUMBER_OF_VARIABLES	3
NUMBER_OF_FUNCTIONS	4
PROBLEM_TYPE	MINIMIZATION
METHOD	SIMPLEX
STATUS	OPTIMAL
VALUE_OF_OBJECTIVE	-10.5
SIMPLEX_PIVOT_COUNT	3
RESIDUAL	0
ELAPSED_TIME (sec.)	0.00

となります。ITERATION_COUNT，FUNC_EVAL COUNT，FACTORIZATION_COUNT という行が表示されないかわりに SIMPLEX_PIVOT_COUNT という行に単体法の反復回数が表示されます。

⁴⁴ 目的関数と制約式を総称してこう呼びます。

(例題 8 の変数をすべて整数変数と指定した問題を分枝限定法によって解いた場合には

NUMBER_OF_VARIABLES	3
NUMBER_OF_FUNCTIONS	4
PROBLEM_TYPE	MINIMIZATION
METHOD	SIMPLEX
STATUS	OPTIMAL
VALUE_OF_OBJECTIVE	-10
SIMPLEX_PIVOT_COUNT	4
PARTIAL_PROBLEM_COUNT	3
DUAL_SIMPLEX_PIVOT_COUNT	6
ELAPSED_TIME(sec.)	0.017

と、PARTIAL_PROBLEM_COUNT という行に最適解に至るまでに調べた分枝限定法の部分問題の数 (3) と部分問題を解くために適用した双対単体法の反復回数の合計 (6) が出力されます。分枝限定法を行った場合に RESIDUAL が表示されないのは、整数解においては最適性条件 (連続変数を仮定) が充足しているとはいえないので、RESIDUAL の値が目的関数値の正しさを示す尺度とはならないためです。

3.2 解ファイルの変数値表示部

```
%%
%% VARIABLES
%%
```

で始まる部分には最適化アルゴリズム停止時における変数と上下限が記述されています。例題 1 を内点法にて解いた結果の解ファイル(ex1.sol)では

```
%%
%% VARIABLES
%%
      NAME      VALUE      STATUS [      BOUND TYPE      ]
V# 1  x1        2.499999998  FREE [      0      <=  x1      ]
V# 2  x2        1.499999999  FREE [      0      <=  x2      ]
V# 3  x3  1.223480816e-009  LOWER [      0      <=  x3      ]
```

の部分が、“変数値”、“変数値が上下限のいずれに付着しているかの状態”及び“変数の上下限”を示しています。“V# 1”から始まる行は x1 という名前の変数の値(2.5)と、それが下限にも上限にも等しくなっていないこと("FREE")、続いて x1 に課された制約の種類([]内)が示されています。

変数の状態を示す文字列の意味を以下に示します。

状態を示す文字列	状態
LOWER	下限制約に付着(下限制約が active)
UPPER	上限制約に付着(上限制約が active)
FREE	上下限, いずれにも付加していない
INFS	上下限を違反している

表 13 変数の状態を示す文字列の意味

表 13 で INFS は数値的な性質の悪い問題や最適化が途中で失敗した際に出力されます。これが含まれる場合には、エラーが出力されます。問題は正常に解けていません。

解ファイルにおいて行の最初のフィールドに"V#"という文字列があるのは変数の値の書かれている行のみに限られており、その行において変数値の書かれているのは必ず 4 番目のフィールドであることが保証されています。このことを利用して awk 等のユーティリティにより、

例えば

```
prompt% awk '$1 ~ /V#/ { print $4}' ex1.sol
2.4999999998
1.4999999999
1.223480816e-009
```

というように，変数値のみを取り出す等の操作が可能です．

また，資源制約付きスケジューリング問題ソルバ(rcpsp)を用いた場合には，変数の部分は，

```
%%
%% ACTIVITIES
%%
```

以下に相当します．

例えば，

```
%%
%% ACTIVITIES
%%
NAME          MODE          [          START/END TIME          ]
V# 1 sourceActivity  "DummyMode"  ACT [    0    <= sourceActivity <=    0    ]
V# 2   act[1]        "A_does"     ACT [    6    <=   act[1]   <=   12    ]
V# 3   act[2]        "C_does"     ACT [    0    <=   act[2]   <=   11    ]
V# 4   act[3]        "B_does"     ACT [    0    <=   act[3]   <=    8    ]
```

のような出力があった場合には，

MODE が アクティビティが処理されるモード，START/END TIME がアクティビティの開始時刻，終了時刻を表します．

3.3 解ファイルの関数値表示部

```
%%
%% FUNCTIONS
%%
```

で始まる部分には最適化アルゴリズム停止時における関数⁴⁵とその双対変数（シャドウプライス）の値が記述されています．例題 1 を内点法にて解いた結果の解ファイル(ex1.sol)では

```
%%
%% FUNCTIONS
%%
NAME      VALUE      STATUS [  CONSTRAINT/OBJECTIVE TYPE      ]
F# 1  f      -10.5    FREE [      OBJECTIVE (MINIMIZE)      ]
F# 2  g1      3.999999999 UPPER [      g1  <=      4      ]
F# 3  g2      4.999999998 UPPER [      g2  <=      5      ]
F# 4  g3      6.499999998 FREE [      g3  <=      7      ]
```

の部分が順番に“関数値”，“関数値が上下限のいずれに付加しているかの状態”及び“関数の上下限”を示しています．"F# 1"から始まる行には F という名前の関数（目的関数）の値（-10.5）と，それが最小化された("MINIMIZE")目的関数である旨が示されています．"F# 2"から始まる行 g1 という名前の制約式の値が 4 であり，それが上限に等しくなっていること("UPPER")，続いて g2 に課された制約の種類([] 内)が示されています．

関数(制約式)の状態を示す文字列は以下を示します．

状態を示す文字列	状態
LOWER	下限制約に付着(下限制約が active)
UPPER	上限制約に付着(上限制約が active)
FREE	上下限， いずれにも付加していない
INFS	上下限を違反している
TGIN	ソフト制約が制約を違反していない
TGOUT	ソフト制約が制約を違反し，ペナルティが発生している．

表 14 関数の状態を示す文字列

⁴⁵ 目的関数と制約式を総称してこう呼んでいます．

表 14 で INFS は数値的な性質の悪い問題や最適化が途中で失敗した際に出力されます。

解ファイルにおいて行の最初のフィールドに "F#" という文字列があるのは、関数の値の書かれている行のみに限られており、その行において関数値、双対変数の書かれているのは必ず 4 番目のフィールドであることが保証されています。このことを利用して awk 等のユーティリティにより、例えば解ファイルから

```
prompt% awk '$1 ~ /F#/ { print $4}' ex1.sol
-10.5
4
5
6.5
```

というように、関数値のみを取り出す等の操作が可能です。

3.4 解ファイルの上下限，制約と対応する双対変数の表示

解ファイルの

```
%%
%% BOUNDS
%%
```

から続く部分には変数の上下限と双対変数（シャドウプライス）が，また

```
%%
%% CONSTRAINTS
%%
```

から続く部分には制約式の上下限と双対変数（シャドウプライス）がそれぞれ表示されます．

例題 1 を内点法にて解いた結果の解ファイル (ex1.sol) では，例えば

```
%%
%% BOUNDS
%%
      [          BOUND TYPE          ]      DUAL VALUE
B# 1 [      0      <=   x1          ]      4.891837406e-010
B# 2 [      0      <=   x2          ]      8.151927281e-010
B# 3 [      0      <=   x3          ]      1.0000000001

%%
%% CONSTRAINTS
%%
      [  CONSTRAINT/OBJECTIVE TYPE  ]      DUAL VALUE
C# 1 [      OBJECTIVE (MINIMIZE)    ]              0
C# 2 [          g1 <=      4        ]      -1.999999998
C# 3 [          g2 <=      5        ]      -0.4999999986
C# 4 [          g3 <=      7        ]     -2.443858094e-009
```

のように，上下限の種類とそれに対応する双対変数の値が出力されます．双対変数は制約式や変数の上下限を単位あたり変化させたときの目的関数の変動を示しています．双対変数は別名，シャドウプライス，または `reduced cost` と呼ばれ，その正負や大きさから上下限のいず

れが `active` かを次のように判断することができます。

解ファイルの双対変数値	状態
正	下限制約に付着 (下限制約が <code>active</code>)
負	上限制約に付着 (上限制約が <code>active</code>)
零/零に近い値	上下限, いずれにも付加していない

表 15 双対変数値の状態

目的関数の双対変数値に対応する部分には零が出力されます。

3.5 標準出力

NUOPT を実行すると標準出力に計算の進行が表示されます .

```

prompt% nuopt ex1.mps          # MPS ファイル入力モジュール
NUOPT 6.0.0, Copyright (C) 1991-2001 Mathematical Systems Inc.
<reading MPS file: ex1.mps >
PROBLEM_NAME(TITLE)           example1
ROWS                           4
COLUMNS                       3
NONZEROS                       11
OBJECTIVE                      f
RHS                            b
NUMBER_OF_VARIABLES            3
NUMBER_OF_FUNCTIONS            4
PROBLEM_TYPE                   MINIMIZATION
METHOD                         HIGHER_ORDER
<preprocess begin>.....<preprocess end>
<iteration begin>
    res=2.4e+001 .... 2.7e-005  1.4e-008
<iteration end>
STATUS                         OPTIMAL
VALUE_OF_OBJECTIVE              -10.5
ITERATION_COUNT                 6
FUNC_EVAL_COUNT                 9
FACTORIZATION_COUNT             7
RESIDUAL                        1.354127444e-008
ELAPSED_TIME(sec.)              0.01
SOLUTION_FILE                   ex1.sol

```

上記は MPS ファイル入力モジュールの標準出力です .イタリックの部分はこのモジュールに特化した出力 (MPS ファイル中の行数 , 列数 , 目的関数名 , 右辺名) ですが , それ以降で中間部分を除いた :

```

NUMBER_OF_VARIABLES            3
NUMBER_OF_FUNCTIONS            4
PROBLEM_TYPE                   MINIMIZATION
METHOD                         HIGHER_ORDER

```

STATUS	OPTIMAL
VALUE_OF_OBJECTIVE	-10.5
ITERATION_COUNT	6
FUNC_EVAL_COUNT	9
FACTORIZATION_COUNT	7
RESIDUAL	1.354127444e-008
ELAPSED_TIME(sec.)	0.01

なる表示があります。これらはあらゆる NUOPT の実行において共通です。意味の詳細は 3.1 解ファイルのヘッダ部をご参照下さい。この問題では、変数/関数⁴⁶の数(3/4)、この問題に適用される最適化手法は線形計画問題用の内点法(HIGHER_ORDER)であることを表示しています。

中間部分の

```
<preprocess begin>.....<preprocess end>
<iteration begin>
  res=2.4e+001 .... 2.7e-005 1.4e-008
<iteration end>
```

は計算の進行を示している部分です。<preprocess begin>と<preprocess end>の間は収束計算に入る前の処理の進行を、<iteration begin>と<iteration end>の間は収束計算の進行を示しています。計算の進行中に表示される数字は最適性条件の残差で、この表示はそれが計算の進行とともに減少していく様子を示しています。この経過出力は問題が整数変数を含まない場合のデフォルト設定である内点法や逐次二次計画法の場合で、単体法と有効制約法では異なる経過表示となります。詳細は 3.5.1 単体法、有効制約法、クロスオーバーの場合の実行経過表示をご参照下さい。

標準出力の終わりは

SOLUTION FILE (解ファイル)	ex1.sol
-----------------------	---------

に計算結果が書き出されたことを示しています。解ファイルの名前の決定方法は環境やモジュールによって異なります。

⁴⁶ 目的関数と制約式を総称してこう呼びます。

3.5.1 単体法，有効制約法，クロスオーバーの場合の実行経過表示

線形計画法，二次計画法に対してはそれぞれ単体法（`options.method = "simplex"`）あるいは有効制約法（`options.method="asqp"`）による求解を指定することも可能です。その場合，実行経過表示は次のようになります。

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

ドットは単体法の反復の進行を示しています（1 つのドットにつき，数回の反復を示しています）。また，文字 1 はフェーズ（最適化実行可能解探索状態）からフェーズ（最適解探索状態）への遷移の様子を示しています。

線形計画法，二次計画法に対して内点法からのクロスオーバー（`options.cross="on"`）を指定した場合には

```
<iteration begin>
    res=2.4e+001 .... 2.7e-005  1.4e-008
<iteration end>
<iteration begin>
    1222
<iteration end>
```

のように，内点法の経過表示の後に単体法の経過表示が現れます。

3.5.2 制約充足問題ソルバー (wcsp) の実行経過表示

制約充足問題ソルバー (wcsp) の実行経過は，最良解におけるハード制約，ソフト制約のペナルティ値が逐次表示されます。

```

<iteration begin>
(hard/soft) penalty= 364/3090, time= 1.88(s)
(hard/soft) penalty= 91/3228, time= 1.89(s), iteration= 1
(hard/soft) penalty= 70/3247, time= 1.89(s), iteration= 2
(hard/soft) penalty= 52/3253, time= 1.89(s), iteration= 3
(hard/soft) penalty= 38/3215, time= 1.89(s), iteration= 4
(hard/soft) penalty= 19/3192, time= 1.89(s), iteration= 5
      ( 中略 )
(hard/soft) penalty= 0/1945, time= 2.58(s), iteration= 3199
(hard/soft) penalty= 0/1940, time= 2.90(s), iteration= 4938
(hard/soft) penalty= 0/1938, time= 3.15(s), iteration= 6271

```

```

# (hard/soft) penalty= 0/1938
# cpu time = 3.15/5.00(s)
# iteration = 6271/15166
<iteration end>

```

項目の意味は次の通りです .

表示	意味
(hard/soft)	発見された解に関する
penalty=値 1/値 2	値 1 : ハード制約違反量 , 値 2 : ソフト制約違反量
time=値 3 , iteration=値 4	値 3 : 経過時間 , 値 4 : 反復回数
#penalty=...	最良解のペナルティ値
#cpu time=...	最良解発見時の経過時間
#iteration=...	最良解発見時の反復回数
#status=...	反復停止の理由

表 16 制約充足問題ソルバーの場合の実行経過表示の意味

最初の項目で説明されている表示は最良解が更新される度に現れます . 解が更新されないと表示は停止しますが , 計算は行われています . メタ・ヒューリスティクスによる探索の一般的な性質として , 計算の後半には解の更新は鈍くなります .

3.5.3 分枝限定法の場合の実行経過表示

混合整数計画問題 (MIP) を解く場合は、通常、自動的に分枝限定法が起動されます。その場合の実行経過は次のようになり、求解の進行状況を確認できます。

```
<iteration begin>
      .1.2B
      up:      1e+50 lo:  8.6617e+05 time:  0.1s:mem(Mb)=0
                                llen:1 #prob:10 #piv:210
#1 up:  9.0859e+05 lo:  8.6655e+05 gap: 42036 time:  1.6s:mem(Mb)=2
                                llen:359 #prob:1009 #piv:4223
#2 up:  9.0746e+05 lo:  8.6655e+05 gap: 40901 time:  6.6s:mem(Mb)=4
                                llen:1002#prob:2652
                                #piv:6493
#3 up:  9.0713e+05 lo:  8.6655e+05 gap: 40574 time: 12.4s:mem(Mb)=8
                                llen:1876          #prob:4776
                                #piv:9105
#4 up:  9.0692e+05 lo:  8.6655e+05 gap: 40361 time: 17.9s:mem(Mb)=12
                                llen:2756#prob:6896#piv:11802
```

(中略)

```
#46 up:  8.8018e+05 lo:8.6655e+05 gap: 13622 time:172.5s:mem(Mb)=43
                                llen:2680#prob:58708iv:65012
#47 up:  8.7972e+05 lo:8.6655e+05 gap: 13161 time:174.4s:mem(Mb)=43
                                llen:2167#prob:59374 piv:65851
      up:  8.7972e+05 lo:  8.6655e+05 gap: 13161 time:180.1s:mem(Mb)=43
                                len:2169#prob:60990#piv:67529
#48 up:      8.7907e+05 lo:      8.6655e+05 gap:  12518
time:184.3s:mem(Mb)=43
                                llen:1470#prob:62102 piv:68561
#49 up:      8.79e+05 lo:      8.6655e+05 gap:  12446
time:190.4s:mem(Mb)=43
                                llen:1396#prob:64450#piv:71062
#50 up:      8.7843e+05 lo:      8.6655e+05 gap:  11876
time:192.2s:mem(Mb)=43
```

各行は、新しい暫定解が得られた場合、あるいは所要メモリが 50Mb 以上変動した場合、また、60 秒ごとに必ず出力されます。新しい暫定解が得られた場合は、行頭に "#" が表示されます。"#" に続く番号は暫定解の番号です。それ以外の場合には # の項目がありません。

他の項目の意味は次の通りです。

表示	意味
up	目的関数の上界値
lo	目的関数の下界値
gap	上下界ギャップ：(上界値 - 下界値)
time	経過時間（秒）
mem (Mb)	全使用メモリ / 実メモリ上にあるメモリ量
avail (Mb)	当該マシンで利用できるプロセスメモリ最大量 / 実メモリの残り最大量
llen	分枝限定法に対するリストの長さ
#prob	通算の子問題数
#pivot	通算のピボット数

表 17 分枝限定法の場合の実行経過表示の意味

実行環境によっては"mem"の表示の「実メモリにあるメモリ量」を示す部分 ,および"avail"項は表示されません .

この問題では合計 50 個の暫定解が求まり ,最後の暫定解の上下界ギャップは 11876 であったことがわかります .

3.5.4 資源制約付きスケジューリングソルバー (rcpsp)の実行経過表示

資源制約付きスケジューリングソルバー (rcpsp) は ,目的関数の設定によって 2 種類の表示があります .

最後の作業の完了時刻最小化時:

```
<iteration begin>
(soft) penalty= 18, time= 0.00(s),iteration= 0
(soft) penalty= 17, time= 0.00(s),iteration= 2
(soft) penalty= 16, time= 0.00(s),iteration= 3
(soft) penalty= 15, time= 0.00(s),iteration= 4
(soft) penalty= 14, time= 0.03(s),iteration= 6
(soft) penalty= 13, time= 0.05(s),iteration= 8
(soft) penalty= 12, time= 0.06(s),iteration= 15
....
```

項目の意味は次の通りです .

表示	意味
(soft)	発見された解に関する
penalty=値 1	値 1：ソフト制約違反量
time=値 2, iteration=値 3	値 2：経過時間, 値 3：反復回数

表 18 最後の作業の完了時刻最小化時の実行経過表示の意味

目的関数は内部では, ソフト制約として扱われていますので, 目的関数の値もソフト制約違反量にふくまれています。

納期遅れ最小化時:

```
<iteration begin>
(objective value) value= 18, time= 0.00(s), iteration= 0
(objective value) value= 10, time= 0.03(s), iteration= 1
(objective value) value= 4, time= 0.05(s), iteration= 8
...
```

項目の意味は次の通りです。

表示	意味
(objective value)	発見された解に関する
value=値 1	値 1：目的関数値 (総納期遅れ)
time=値 2, iteration=値 3	値 2：経過時間, 値 3：反復回数

表 19 納期遅れ最小化時の実行経過表示の意味

上記 2 つの表示は, 制約充足ソルバーの時と同様, 最良解が更新される度に現れます。その為, 解が更新されないと表示が停止する点においても制約充足ソルバーの時と同様です。

3.5.5 標準出力の抑制

NUOPT からの標準出力を消去するには, パラメータファイルで

```
begin
output:mode=silent    *標準出力を抑制する
end
```

のように指定するか, SIMPLE におけるモデル記述中において,

```
options.outputMode = "silent";
```

として下さい。パラメータファイルの記述方法の詳細につきましては 4 パラメータ設定を、標準出力の抑制の詳細につきましては 4.5 その他の設定をご参照下さい。

3.6 実行不可能性要因検出機能 (iisDetect) の適用例と出力

デフォルトの指定では, rcpsp/wcsp の起動時以外に実行不可能性が検出されると, iisDetect と呼ぶアルゴリズムが自動的に起動され, 実行不可能性の原因の探索を行い, その結果を解ファイルの出力に反映させます。ここでは, iisDetect 機能が起動した場合の出力結果について具体的に説明します。

例えば以下のモデル記述(モデルファイル名lp.smp とします)に書かれた線形計画問題は, 実行不可能(制約を満たす解なし)です。

```
Variable x,y,z;
Objective f(type=minimize);
f = x + y + z;

x >= y ;           // IIS
1 + z >= x ;       // IIS
y >= 2 + z;        // IIS
x + y + z >= 0;
```

よく見るとモデルで”IIS”のマークが付いた制約式群のどの一つを除去しても実行不可能性は解消しますが, すべてを満たす x,y,z は存在しません。また, マークされていない最後の制約式は実行不可能性とは無関係で, 除去する, しないにかかわらず, 問題は実行不可能であることもわかります。

実行不可能性を解消する際の調査のガイドラインとして, iisDetect 機能はこのように, 実行不可能性の原因となっている行の組 (Irreducible Infeasible Set: IIS と呼ばれます) を特定して出力するものです。一般に実行不可能な問題について IIS は複数存在しますが, このアルゴリズムは可能な限り小さなもの (含まれている行が少ない) ものを求めるようなヒューリスティクスが導入されています。

この問題を解かせたとき, 解ファイルのヘッダ部にはまず,

ERROR_TYPE	<<NUOPT 11>> infeasible.
DETECTED_IIS_SIZE	3 (追加)
(#IIS_RELATED_VAR)	3 (追加)
INFEASIBILITY_OF_IIS	1 (追加)

のように現れます。

タイトル	解説	備考
DETECTED_IIS_SIZE	検出された IIS に含まれる行の数	成功時のみ
(#IIS_RELATED_VAR)	IIS に含まれている行に含まれる変数の数	成功時のみ
INFEASIBILITY_OF_IIS	IIS 全体での実行不可能性	成功時のみ
NO_IIS_FOUND_BY	IIS 検出失敗の原因	失敗時のみ
(#NONLINEAR_CONSTR.)	IIS 検出失敗の原因の可能性のある非線形制約の数	失敗時のみ

ヘッダ部分に引き続き,

```

%%
%% IIS
%%
-----
#2      lp.smp:5      :      y - x
                                     <=      0      (      0)
-----
#3      lp.smp:6      :      -1 + x - z
                                     <=      0      (      0)
-----
#4      lp.smp:7 INFS :      2 + z - y
                                     <=      0      (      1)
-----

```

のように IIS に含まれる行の制約式の名前が出力されます。内部表現に従って移項されていますので、 x, y, z の順番はモデル記述とは異なります。()内は現在の変数値の設定(以降に出力されます)における、各、制約式の値です。INFS とマークがある行は現在の変数値の設定において、上下限を破っている行です。これを解消しようとすると、IIS の定義から、ここに現れている行のほかのいずれかに波及します。IIS 検出に成功した場合の変数の設定は IIS に含まれる行の違反の合計が最も小さくなるように行われます。その際の IIS に含まれる行の違反の合計が、ヘッダに現れる

INFEASIBILITY_OF_IIS

という値です。

以降の出力は通常と同一ですが、便宜のため IIS に関する変数と、関数のみが出力されます。

```
%%
%% VARIABLES          ( IIS に関連しない変数は現れない )
%%
      NAME      VALUE      STATUS [      BOUND TYPE      ]
V# 1   x                1  FREE [   -Inf   <=   x   <=   +Inf   ]
V# 2   y                1  FREE [   -Inf   <=   y   <=   +Inf   ]
V# 3   z                0  FREE [   -Inf   <=   z   <=   +Inf   ]

%%
%% FUNCTIONS          ( IIS に関連しない関数は現れない )
%%
      NAME      VALUE      STATUS [   CONSTRAINT/OBJECTIVE TYPE      ]
F# 2 lp.smp:5                0  UPPER [                lp.smp:5 <=   0      ]
F# 3 lp.smp:6                0  UPPER [                lp.smp:6 <=   0      ]
F# 4 lp.smp:7                1  INFS [                lp.smp:7 <=   0      ]
```

次に問題を等価ですが、IIS に含まれる最初の制約式の両辺を二乗し、非線形計画問題(このファイルを nlp.smp とします)に書き直してみましょう。

```
Variable x,y,z;
Objective f(type=minimize);
f = x + y + z;

x*x >= y*y ;    // IIS
1 + z >= x ;    // IIS
y >= 2 + z;     // IIS
x + y + z >= 0;
```

この場合、iisDetect のアルゴリズム上の制約から、IIS の正確な検出はできません。その場合には、ヘッダ部には IIS の検出が非線形性のために失敗したというメッセージが現れ、非線形な制約がいくつあるかを示します。

```
ERROR_TYPE          <<NUOPT 11>> infeasible.
NO_IIS_FOUND_BY          NON_LINEARLITY
(#NONLINEAR_CONSTR.)          1
```

引き続いての関数の表示の部分で非線形な関数には次のように*NLR*というマークが付きます。ここでマークされた非線形関数は IIS に入っている可能性があるということのみを示します。

```
%%
%% FUNCTIONS          ( IIS に入っているかもしれない非線形関数にはマークが付く )
%%
```

	NAME	VALUE	STATUS	[CONSTRAINT/OBJECTIVE	TYPE]
F# 1	f		2 FREE	[OBJECTIVE (MINIMIZE)]
F# 2	nlp.smp:5_*NLR*		4 INFS	[nlp.smp:5_*NLR* <=	0]
F# 3	nlp.smp:6		-1 FREE	[nlp.smp:6 <=	0]
F# 4	nlp.smp:7		0 UPPER	[nlp.smp:7 <=	0]
F# 5	nlp.smp:8		2 FREE	[0 <= nlp.smp:8]

4. パラメータ設定

この章では NUOPT の実行をカスタマイズするためのパラメータの内容について解説します。Windows 版 GUI を使っている場合に関するパラメータ設定方法の詳細は 2.9SIMPLE 版ロードモジュールの動作の制御をご参照下さい。

UNIX/LINUX 版では, `nuopt.prm` という名前のファイル (パラメータファイル) を作成してロードモジュールを起動するディレクトリに置くのが最も簡便です。SIMPLE 版ロードモジュールでは他に二通りの方法がありますが, それについては 2.9SIMPLE 版ロードモジュールの動作の制御" を参照ください。

4.1 パラメータファイル `nuopt.prm`

以下がパラメータファイルの例です。

```
begin
method: trust          * 最適化手法として信頼領域法を指定
scaling: on            * スケーリングを行うことの指定
crit: eps = 1.0e-8     * 内点法の停止条件を  $10^{-8}$  に設定
end
```

`begin` という文字列から始まる行 (`begin` 行) から `end` という文字列で始まる行 (`end` 行) までの間がパラメータ指定とみなされ, `end` 行以降の記述は無視されます。また, タブや空白, 空行, * から行末までもコメントとみなされて無視されます。

この章の以降の説明で

```
method: higher
```

という囲みは, `nuopt.prm` の `begin` と `end` の間の行の記述を示しています。この設定 (この場合には「線形計画問題専用内点法 (Higher Order Method) の適用」) を行うためには `nuopt.prm` に

```
begin
method: higher
end
```

と書けば良いことを示しています。

```
crit: eps = 1.0e-8
```

のように、囲みの中に “ ... = 値 ” と書かれている場合には、述べられているパラメータの設定（この場合には「内点法の停止条件の設定」）をするためには、`nuopt.prm` に

```
begin
crit: eps = 値
end
```

と書けば良いことを示しています。特に断らない限り、“=”の右の値はデフォルト値（設定を行わないときの値）です。値には整数や小数のほか、

```
9.836d-5   1.347D-4   3.4e-3   4.562384E-2
```

のような浮動小数点表記が許されています。

パラメータファイルが読み込まれた場合、標準出力にはパラメータファイルを読み込んだことを示すメッセージと内容が表示されます⁴⁷。

```
begin
maximize
method:trust
scaling:on
crit: eps = 1.0e-8
end
prompt% nuopt myprob.mps
<reading parameter file: nuopt.prm>
    begin
    maximize
    method:trust
    scaling:on
    crit:eps=1.0e-8
    end
<reading MPS file: myprob.mps >
...
```

⁴⁷ この場合の表示において、パラメータファイル中のスペースやコメントは現れません。

4.2 最適化手法に関する設定

この章では NUOPT に組み込まれている最適化手法とその設定方法について解説します．

次の表は NUOPT の備えている解法と適性を表したものです．まず解きたい問題の種別から適用可能な手法を選んで下さい．

	LP	MILP	MIQP	MINLP	CQP	CP	NLP	RCPSP
simplex			--	--	--	--	--	
asqp		--		--		--	--	--
higher		--	--	--	--	--	--	--
lipm/lepm/line		--	--	--			--	--
bfgs/lbfgs		--	--	--	--			--
tipm/tepm/trust		--	--	--	--			--
lsqp		--	--	--				--
tsqp		--	--	--				--
wcsp	--				--	--	--	--
global								
rcpsp	--	--	--	--	--	--	--	
iisDetect								--

表の内容は

最も適している

適用できるが非効率

-- 適用できない

を示します．wcsp は とありますが，0-1 整数変数と離散変数のみを含む問題に対して適用できます．連続変数や一般の整数変数を含む問題には適用できません．ご注意ください．

表の列見出しは問題の種別を表す文字列で，SIMPLE あるいはパラメータファイルからその

```
options.method = "tipm" (SIMPLE)          method:tipm (nuopt.prm)
```

手法を指定するときに用いる文字列です．ただし，最後の iisDetect は実行不可能性検出，という自動起動されるアルゴリズムなのでこのように指定はできません．詳細はこの節末尾をご覧ください．

◆ LP (Linear Programming:線形計画問題)

目的関数と制約式がすべて線形である問題で，整数変数を含まないものです．

- ◆ MILP (Mixed Integer Linear Programming:混合整数計画問題)
目的関数と制約式がすべて線形で、整数変数を含むものです。
- ◆ MIQP (Mixed Integer Quadratic Programming:混合整数二次計画問題)
制約式がすべて線形、目的関数が二次関数で、整数変数を含むものです。
- ◆ MINLP (Mixed Integer Nonlinear Programming:混合整数非線形計画問題)
制約式および目的関数が非線形で整数変数を含むものです。
- ◆ CQP (Convex Quadratic Programming:凸二次計画問題)
目的関数が凸な二次関数、制約式がすべて線形であるもの(ただし、目的関数の符号の変更で下に凸な目的関数の最小化に帰着できるもの)です。
- ◆ CP (Convex Programming:凸計画問題)
目的関数、制約式に非線形なものが含まれていますが、実行可能領域が凸で、目的関数の符号の変更で下に凸な目的関数の最小化に帰着できる問題です。
ここでは整数変数は含まないものを言います。
- ◆ NLP (Nonlinear Programming:非線形計画問題)
上記以外で、整数変数を含まない一般の非線形計画問題です。
- ◆ RCPSP (Resource Constrained Project Scheduling Problem:資源制約付きスケジューリング問題)
一定の資源制約の下で、決められた作業の開始・終了時刻を決定する問題です。一般の整数計画問題(MIP)として記述することも可能ですが、特殊な記法を行うと、タブーサーチによる、NUOPT に組み込まれた専用ソルバーrcpspにより高速に実行可能解を得ることができます。

以下、NUOPT が備えている解法の紹介を行います。行見出しは解法の名前、囲みは、各手法を起動する指定の方法を示しています。

- ◆ simplex :単体法(Simplex Method)
線形計画法の解法として古くから知られている方法です。大規模問題では内点法に速度的に劣りますが、可能基底解が求まり原理的に内点法/外点法よりも高精度です。

```
options.method = "simplex" (SIMPLE)    method:simplex (nuopt.prm)
```

整数変数を含む問題に対して指定すると、単体法を分枝限定法 (Branch and bound method) という枠組のなかで繰り返し行って、最適性の保証のある整数解を求めます。大規模問題において基底解が必要な場合には、4.2.2 クロスオーバーで説明するように "cross:on" と指定して内点法からのクロスオーバーを用いるのが有利です。

◆ asqp : 有効制約法 (Active Set Method)

単体法と同様、古典的な二次計画問題の厳密解法です。1 万変数以上の大規模問題では、一般に内点法 (直線探索法 (Line Search Method)) に劣りますが、

- ・ 変数に比べて制約式の数が非常に少ない (1/10 以下) 場合
- ・ 目的関数のヘッセ行列が密行列である場合

には内点法よりも高速かつ高精度です。また、整数計画法に対応しているので、整数変数が含まれている二次計画問題を解くことができます。4.2.2 クロスオーバーで説明するように "cross:on" と指定することで内点法からのクロスオーバーを用いることができるので、大規模問題に対して高精度な解を求めることができます。

```
options.method = "asqp" (SIMPLE)    method:asqp (nuopt.prm)
```

◆ higher : 線形計画問題専用内点法 (Higher Order Method)

線形計画法に特化した内点法で、大規模な線形計画問題の解法としては最も高速です。ただし単体法と違い、可能基底解は求められません。

```
options.method = "higher" (SIMPLE)    method:higher (nuopt.prm)
```

◆ lipm/lepm/line : 直線探索法 (Line Search Method)

一般の凸計画問題に適用可能な内点法・外点法です。問題が凸であることがわかっている場合には信頼領域法よりも高速です。

```
options.method = "lipm" (SIMPLE)    method:line (nuopt.prm)
options.method = "lepm" (SIMPLE)    method:lepm (nuopt.prm)
options.method = "line" (SIMPLE)    method:line (nuopt.prm)
```

幅広い範囲の問題に対して有効なのが内点法 (lipm)、外点法 (lepm) は問題に対して比較的良い初期値が得られている場合に有効であることが示されています。旧版の内点法 (line) は、以前のバージョンとの整合を取る場合にご利用ください (Ver.7 以前の内点法 line と Ver.8 の内点法 lipm では、メリット関数の定義が若干異なっておりますので、Ver.8 では微妙

に異なる結果を与える場合がございます)。

◆ bfgs/lbfgs : 準ニュートン法 (Line Search with BFGS)

```
options.method = "bfgs" (SIMPLE)    method:bfgs (nuopt.prm)
options.method = "lbfgs" (SIMPLE)   method:lbfgs (nuopt.prm)
```

準ニュートン法によって二階微係数を求める内点法です。“bfgs”はヘッセ行列の近似行列を密行列として保持しますので、小規模 (50 ~ 500 変数以下) な非線形計画問題に適しています。“lbfgs”は limited memory BFGS という手法を用いて、ヘッセ行列の近似行列の保持方法を変更しているので、大規模問題にも対応することができます。

◆ tipm/lepm/trust : 信頼領域法 (Trust Region Method)

```
options.method = "tipm" (SIMPLE)    method:tipm (nuopt.prm)
options.method = "tepm" (SIMPLE)    method:tepm (nuopt.prm)
options.method = "trust" (SIMPLE)   method:trust (nuopt.prm)
```

大規模なものを含む一般の非線形計画問題に適用可能な内点法・外点法です。幅広い範囲の問題に対して有効なのが内点法 (tipm)、外点法 (tepm) は問題に対して比較的良い初期値が得られている場合に有効であることが示されています。旧版の内点法 (trust) は、以前のバージョンとの整合を取る場合にご利用ください (Ver.7 以前の内点法 trust と Ver.8 の内点法 tipm では、メリット関数の定義が若干異なっておりますので、Ver.8 では微妙に異なる結果を与える場合がございます)。

◆ lsqp : 直線探索法に基づく逐次二次計画法 (Line Search SQP Method)

準ニュートン法によって二階微係数を求める逐次二次計画法です。小規模 (50 ~ 100 変数以下) な非線形計画問題に適しています

問題によっては内点法/外点法 (lipm/lepm/line) よりも安定的により精度の良い解を導くことができます。

```
options.method = "lsqp" (SIMPLE)    method:lsqp (nuopt.prm)
```

◆ tsqp : 信頼領域法に基づく逐次二次計画法 (Trust Region SQP Method)

二階微係数をそのまま用いる逐次二次計画法です。大規模なものを含む一般の非線形計画問題に適用可能な方法です。一般に内点法よりも低速ですが、問題によっては内点法よりも安定的に、より精度の良い解を導くことができます。

変数の数よりも制約式数が多い場合には内点法/外点法 (tipm/tepm/trust) よりも

高速な場合があります。

```
options.method = "tsqp" (SIMPLE)    method:tsqp (nuopt.prm)
```

♦ wcsp (タブー・サーチに基づく制約充足アルゴリズム)

京都大学「問題解決エンジン」グループの開発による制約充足問題のソルバーです。大規模な整数計画問題に対し、非常に高速に実行可能解を求めることができます。

ただし、整数変数のみを含む問題で、かつすべての変数の上限と下限がある問題に対してのみ有効です。

```
options.method = "wcsp" (SIMPLE)    method:wcsp (nuopt.prm)
```

♦ rcpsp (タブー・サーチに基づく資源制約付きスケジューリングアルゴリズム)

京都大学「問題解決エンジン」グループの開発による資源制約付きスケジューリング問題のソルバーです。資源制約の下、決められた作業の開始・終了時刻を決定する問題の実行可能解を高速に求めることができます。

rcpsp の記述にあたっては問題を SIMPLE の特殊なクラス：

(Activity, ResourceRequire, ResourceCapacity)

を用いて記述する必要があります⁴⁸。

```
options.method = "rcpsp" (SIMPLE)    method:rcpsp (nuopt.prm)
```

♦ global (凸緩和法に基づく大域的最適化アルゴリズム)

本アルゴリズムの実行に際しては NLP モジュール 及び NUOPT/Global が必要です。NUOPT/Global は有償アドオンであり、別途購入が必要となります。

一般の非線形計画問題または、整数変数を含む非線形計画問題に対し、大域的な最適解を求めることができます。NUOPT が備えている他の非線形計画法アルゴリズム(bfgs,trust)では、局所的な最適解を出力する可能性があるのに対し、この解法は大域的な最適解を与えます。ただし、凸緩和法に基づく分枝限定法を行いますので、計算機資源を多く消費します。数十変数程度の小規模でかつ複雑な問題に適しています。

本アルゴリズムは SIMPLE で記述される目的関数、制約で四則演算および以下の関数を用いて記述されたものをサポートします。

⁴⁸ 具体的には別冊「NUOPT/SIMPLE チュートリアル」をご参照ください。

```

+, -, *, /,
log, exp, pow, sqrt, cbrt
sin, cos, tan, sec, csc, cot
sinh, cosh, tanh, sech, csch, coth
fabs, ceil, floor
min, max, ifelse
asin, acos, atan, asec, acsc, acot
asinh, acosh, atanh, asech, acsch, acoth

```

```
options.method = "global" (SIMPLE)    method:global (nuopt.prm)
```

◆ `iisDetect` (実行不可能性要因行の特定アルゴリズム)

このアルゴリズムは `NUOPT` に与えられた問題が実行不可能と判定されたら、自動的に起動し、ユーザーに実行可能性の原因をレポートするためのものです。内部では単体法を繰り返し適用しています。

実行不可能な問題について、

$$\begin{aligned}
 x &\geq y \\
 1 + z &\geq x \\
 y &\geq 2 + z
 \end{aligned}$$

のように「どの一つを除去しても実行不可能性は解消するが、同時には満たすことができない制約式の組 (Irreducible Infeasible Set: IIS)」を発見します。この組の設定は一般に一意ではありませんが、このアルゴリズムは、できるだけ少ない行が含まれるような IIS を発見してレポートします。本アルゴリズムが正常に動作すれば、線形制約が原因で実行不可能になっている場合にはその原因の一つを正確に特定できますが、非線形制約や変数が整数であるという制約が原因である場合には、その限りではありません。その場合、非線形制約が原因で IIS の検出に失敗した、という旨のメッセージが出て、IIS には非線形制約が含まれていることのみが明らかになります。

`iisDetect` は

```
options.iisDetect = "on" / "off" (SIMPLE) param:iis=on/off (nuopt.prm)
```

と指定して起動の有無を選択します。デフォルトでは常に起動する ("on" となっています)。`iisDetect` は元の問題の一部の制約式を削減したものを複数回解きますので、処理に若干時間を所要する場合がございます。原因に関するレポートが不要であれば、"off" にしてご利用ください。

`iisDetect` の具体的な動作と出力例は 3.63.6 をご覧ください。

4.2.1 解法を選択

手法の指定をユーザーが陽に行わない場合には、.NUOPT は入力された問題の内容から自動的に手法を選択します⁴⁹。

関数がすべて線形で整数変数を含む	単体法 (simplex)
目的関数が非線形で整数変数を含む	有効制約法 (asqp)
関数がすべて線形	線形計画問題専用内点法 (higher)
DiscreteVariable, selection を含む	重み付き制約充足 (wcsp)
Activity, ResourceRequire	
ResourceCapacity を含む	資源制約付スケジューリング (rcsp)
上記以外	信頼領域法 (tipm)

ただ、次のような場合、個別に設定するとよりよい結果が得られる可能性があります。

整数変数が含まれている非線形問題の場合には、デフォルトは asqp です。制約式が線形ではない、あるいは目的関数が二次関数の場合にはエラーになります。もし、大域的最適化 global アドインをインストールされている場合には、

```
options.method = "global";    (SIMPLE)    method:global    (nuopt.prm)
```

として、お試しください。また、整数変数がすべて 0 - 1 変数であれば、重み付き制約充足ソルバー (wcsp) も有効ですので、

```
options.method = "wcsp";    (SIMPLE)    method:wcsp    (nuopt.prm)
```

としてお試しください。実行可能解を求めることができます。

整数変数が含まれない非線形計画問題の場合、デフォルトでは内点法による信頼領域法 (tipm) となりますが、問題が二次計画問題 (目的関数のみ二次関数) の場合、特に変数に比べて一般の制約式の数が少ない問題には有効制約法 (asqp)

```
options.method = "asqp";    (SIMPLE)    method:asqp    (nuopt.prm)
```

が有利な場合もあります。

同じ信頼領域法でも外点法 (tepm) や旧版の内点法 (trust)、逐次二次計画法 (tsqp) を用いることも以下のように指定すると可能です。

⁴⁹ ただし、MPS ファイル入力モジュール nuopt (Windows 版: nuoptmps) は、非線形な問題が与えられた場合の設定が異なります。詳細は 5.4.1nuopt の最適化アルゴリズムの設定を参照して下さい。

<code>options.method = "tepm";</code>	<code>(SIMPLE)</code>	<code>method:tepm</code>	<code>(nuopt.prm)</code>
<code>options.method = "trust";</code>	<code>(SIMPLE)</code>	<code>method:trust</code>	<code>(nuopt.prm)</code>
<code>options.method = "tsqp";</code>	<code>(SIMPLE)</code>	<code>method:tsqp</code>	<code>(nuopt.prm)</code>

外点法(`tepm`)は問題に対して比較的良好な初期値が得られている場合に有効であることが示されています。旧版の内点法(`trust`)は、以前のバージョンとの整合を取る場合にご利用ください(Ver.7以前の内点法 `trust` と Ver.8 の内点法 `tipm` では、メリット関数の定義が若干異なっておりますので、Ver.8 では微妙に異なる結果を与える場合がございます)。逐次二次計画法(`tsqp`)は若干時間を所要するケースもございますが、最も精度良く非線形最適化を行う方法です。

凸計画問題に対しては直線探索法を用いた方が一般に高速ですので、非線形ながら問題が凸であるとわかっている場合には(`SIMPLE` は凸であるかを自動判定できません)

<code>options.method = "lipm";</code>	<code>(SIMPLE)</code>	<code>method:lipm</code>	<code>(nuopt.prm)</code>
<code>options.method = "lepm";</code>	<code>(SIMPLE)</code>	<code>method:trust</code>	<code>(nuopt.prm)</code>
<code>options.method = "line";</code>	<code>(SIMPLE)</code>	<code>method:line</code>	<code>(nuopt.prm)</code>
<code>options.method = "lsqp";</code>	<code>(SIMPLE)</code>	<code>method:lsqp</code>	<code>(nuopt.prm)</code>

としていずれかの直線探索法を指定して下さい。通常お勧めできるのが(`lipm`)です。外点法(`lepm`)、旧版の直線探索法(`line`)、逐次二次計画法(`lsqp`)の特性は、上で説明している信頼領域法に基づくアルゴリズム(`tepm`, `trust`, `tsqp`)と同一です。

4.2.2 クロスオーバー

LP(線形計画問題)あるいはQP(二次計画問題)に対しては、内点法(`higher/lipm/lepm/line/bfgs/lbfgs/tipm/tepm/trust`)によって得られた解の情報をもとにして単体法を起動するクロスオーバーを行うことができます。大規模問題に関して精度の良い解を得るにはこの方法が最も有効です。

クロスオーバーを行うためには内点法の手法を設定した後に、

<code>options.cross = "on" (SIMPLE)</code>	<code>cross:on (nuopt.prm)</code>
--	-----------------------------------

と指定します。

4.3 最適化手法のパラメータ

ここでは特定の最適化手法を選んだ際のパラメータ値とその設定について解説します。

4.3.1 線形計画問題専用内点法(Higher Order Method) に有効なパラメータ

ここで解説するパラメータは以下の設定を行った場合に有効です。

```
options.method = "higher"(SIMPLE)    method:higher (nuopt.prm)
```

◆ スケーリング

大規模問題の線形制約を表す係数行列のオーダーを揃える操作を行う(スケーリングする)と求解プロセスの数値的安定性が向上する場合があります。このパラメータはそれを行うか否かを指定するものです。

```
options.scaling = "on"(SIMPLE)    scaling:on (nuopt.prm)
```

◆ 単体法へのクロスオーバー

この指定を行うと、内点法によって得られた解の情報をもとにして単体法を起動することができます。大規模問題に関して可能基底解を得るにはこの方法が最も有効です。

```
options.cross = "on" (SIMPLE)    cross:on (nuopt.prm)
```

◆ 内点法停止条件

停止条件として用いる最適性条件の残差です。最適性条件の残差がこの値以下になったときに、計算が収束したとみなして反復計算を終了します。

```
options.eps = 1.0e-8 (SIMPLE)    param:eps=1.0e-8 (nuopt.prm)
```

◆ 内点法の反復回数上限

停止条件として用いる反復計算の回数の上限です。反復回数がデフォルト値の150回を越えた場合は、これ以上の反復を続けても解が改善する可能性は低い場合が多いことが経験的に知られています。反復回数がこの回数を越えた場合には解が得られなかったとみなしてエラー(<<NUOPT 10>> IPM iteration limit exceeded.)を出力しま

す。

```
options.maxitn = 150 (SIMPLE)    crit:maxitn=150 (nuopt.prm)
```

◆ その他

以下は `nuopt.prm` のみから設定可能な内部パラメータで、特定の問題に対するチューニングを行う際に変更の可能性のあるものです。

```
linear:u = 1.0e-8      *列解法のピボット選択基準値
param:Mmu = 3.5        *バリアパラメータ減少率
param:Mtol = 2         *バリアパラメータ減少判定基準値
param:gamma = 0.9995   *ステップの境界への近さ
param:fccorr = 5       *ステップベクトルの修正コストパラメータ
crit:epsscl = 1.0e-4   *スケーリングの停止条件
```

4.3.2 直線探索法(Line Search Method) に有効なパラメータ

ここで解説するパラメータは以下の設定に対応します。

```
options.method = "lipm" (SIMPLE)    method:line (nuopt.prm)
options.method = "lepm" (SIMPLE)    method:lepm (nuopt.prm)
options.method = "line" (SIMPLE)    method:line (nuopt.prm)
```

値はこの手法を選んだ際のデフォルトです。

◆ スケーリング

大規模問題の線形制約を表す係数行列のオーダーを揃える操作を行う(スケーリングする)と数値的安定性が向上する場合があります。このパラメータはそれを行うか否かを指定するものです。

```
options.scaling = "on" (SIMPLE)    scaling:on (nuopt.prm)
```

◆ 内点法停止条件

停止条件として用いる最適性条件の残差です。最適性条件の残差がこの値以下になったときに、計算が収束したとみなし反復計算を終了します

デフォルト値 ε (約 $1.5\text{e-}6$) は非線形最適化用に次の式から定められています .

$$\varepsilon = \sqrt{\varepsilon_{mch}} \cdot 10^2$$

```
options.eps = 1.5e-6 (SIMPLE)    param:eps=1.5e-6 (nuopt.prm)
```

- ◆ 内点法の反復回数上限

停止条件として用いる反復計算の回数の上限です .反復回数がデフォルト値の 150 回を越えた場合は , これ以上の反復を続けても解が改善する可能性は低い場合が多いことが経験的に知られています . 反復回数がこの回数を越えた場合には解が得られなかったとみなしてエラー (<<NUOPT 10>> IPM iteration limit exceeded.) を出力します .

```
options.maxitn = 150 (SIMPLE)    crit:maxitn=150 (nuopt.prm)
```

- ◆ 外点法の実行不可能性ペナルティー (外点法 tepm/lepm 専用)

問題が実行不可能に非常に近い場合 , 外点法は不等式制約を満たさない解を与える可能性があります (<<NUOPT 55>> exterior solution obtained.) . そのような場合には , パラメータ `exrho` をデフォルトよりも大きく設定すると解消する可能性があります . 問題が実行可能な場合に , `exrho` が大きいと反復に時間がかかる場合があります . 内点法に比べて外点法のパフォーマンスが悪い場合には `exrho` をあえて下げて実行してみるのも一つの方法です .

```
options.exrho = 1.0e4 (SIMPLE)    param:exrho=1.0e4 (nuopt.prm)
```

- ◆ その他

以下は `nuopt.prm` からのみ設定可能な内部パラメータで , 特定の問題に対するチューニングを行う際に変更の可能性のあるものです .

linear:u = 1.0e-8	*行列解法のピボット選択基準値
crit:beta = 0.1	*直線探索のステップ減少率
param:gamma = 0.9995	*ステップの境界への近さ
param:Mmu = 1	*バリアパラメータ減少率
param:Mtol = 0.8	*バリアパラメータ減少判定基準値
param:Mzl = 10	*双対変数値の下限側マージン
param:Mzu = 10	*双対変数値の上限側マージン
crit:rhotol = 1e20	*ペナルティパラメータ上限
crit:epsscl = 1.0e-4	*スケーリンググループ停止条件

4.3.3 準ニュートン法(Line Search with BFGS)に有効なパラメータ

ここで解説するパラメータは以下の設定に対応します。

options.method = "bfgs" (SIMPLE)	method:bfgs (nuopt.prm)
options.method = "lbfgs" (SIMPLE)	method:lbfgs (nuopt.prm)

値はこの手法を選んだ際のデフォルトです。

◆ スケーリング

大規模問題の線形制約を表す係数行列のオーダーを揃える操作を行う(スケーリングする)と数値的安定性が向上する場合があります。このパラメータはそれを行うか否かを指定するものです。

options.scaling = "on" (SIMPLE)	scaling:on (nuopt.prm)
---------------------------------	------------------------

◆ 内点法停止条件

停止条件として用いる最適性条件の残差です。最適性条件の残差がこの値以下になったときに、計算が収束したとみなし反復計算を終了します。

デフォルト値 ε (約 $1.5e-6$) は非線形最適化用に次の式から定められています。

$$\varepsilon = \sqrt{\varepsilon_{mch}} \cdot 10^2$$

```
options.eps = 1.5e-6 (SIMPLE)    param:eps=1.5e-6 (nuopt.prm)
```

◆ 内点法の反復回数上限

停止条件として用いる反復計算の回数の上限です。反復回数がデフォルト値の 150 回を越えた場合は、これ以上の反復を続けても解が改善する可能性は低い場合が多いことが経験的に知られています。反復回数がこの回数を越えた場合には解が得られなかったとみなしてエラー (<<NUOPT 10>> IPM iteration limit exceeded.) を出力します。

```
options.maxitn = 150 (SIMPLE)    crit:maxitn=150 (nuopt.prm)
```

◆ その他

以下は `nuopt.prm` のみから指定可能な内部パラメータで、特定の問題に対するチューニングを行う際に変更の可能性のあるものです。

<code>linear:u = 1.0e-2</code>	*行列解法のピボット選択基準値
<code>crit:beta = 0.5</code>	*直線探索のステップ減少率
<code>param:gamma = 0.9995</code>	*ステップの境界への近さ
<code>param:Mmu = 200</code>	*バリアパラメータ減少率
<code>param:Mtol = 175</code>	*バリアパラメータ減少判定基準値

<code>param:Mz1 = 2.5</code>	*双対変数値の下限側マージン
<code>param:Mzu = 10</code>	*双対変数値の上限側マージン
<code>crit:rhotol = 1e20</code>	*ペナルティパラメータ上限
<code>crit:epsscl = 1.0e-4</code>	*スケーリンググループ停止条件

4.3.4 信頼領域法(Trust Region Method)に有効なパラメータ

ここで解説するパラメータは以下の設定に対応します。

```
options.method = "tipm" (SIMPLE)    method:tipm (nuopt.prm)
options.method = "tepm" (SIMPLE)    method:tepm (nuopt.prm)
options.method = "trust" (SIMPLE)    method:trust (nuopt.prm)
```

値はこの手法を選んだ際のデフォルトです。

- ◆ スケーリング

大規模問題の線形制約を表す係数行列のオーダーを揃える操作を行う(スケーリングする)と数値的安定性が向上する場合があります。このパラメータはそれを行うか否かを指定するものです。

```
options.scaling = "on" (SIMPLE)      scaling:on (nuopt.prm)
```

- ◆ 内点法停止条件

停止条件として用いる最適性条件の残差です。最適性条件の残差がこの値以下になったときに、計算が収束したとみなし反復計算を終了します。

デフォルト値 ε (約 $1.5e-6$) は非線形最適化用に次の式から定められています。

$$\varepsilon = \sqrt{\varepsilon_{mch}} \cdot 10^2$$

```
options.eps = 1.5e-6 (SIMPLE)      param:eps=1.5e-6 (nuopt.prm)
```

- ◆ 内点法の反復回数上限

停止条件として用いる反復計算の回数の上限です。反復回数がデフォルト値の 150 回を越えた場合は、これ以上の反復を続けても解が改善する可能性は低い場合が多いことが経験的に知られています。反復回数がこの回数を越えた場合には解が得られなかったとみなしてエラー (<<NUOPT 10>> IPM iteration limit exceeded.) を出力します。

```
options.maxitn = 150 (SIMPLE)      crit:maxitn = 150 (nuopt.prm)
```

- ◆ 外点法の実行不可能性ペナルティ (外点法 tepm/lepm 専用)

問題が実行不可能に非常に近い場合、外点法は不等式制約を満たさない解を与える可能性があります (<<NUOPT 55>> exterior solution obtained.)。そのような場合には、パラメータ `exrho` をデフォルトよりも大きく設定すると解消する可能性があります。問題が実行可能な場合に、`exrho` が大きいと反復に時間がかかる場合があります。内点法に比べて外点法のパフォーマンスが悪い場合には `exrho` をあえて下げて

実行してみるのも一つの方法です．

```
options.exrho = 1.0e4 (SIMPLE)    param:exrho=1.0e4 (nuopt.prm)
```

◆ その他

以下は `nuopt.prm` からのみ設定可能な内部パラメータで，特定の問題に対するチューニングを行う際に変更の可能性のあるものです．

<code>linear:u = 1.0e-2</code>	*行列解法のピボット選択基準値
<code>param:gamma = 0.99</code>	*ステップの境界への近さ
<code>param:Mmu = 60</code>	*バリアパラメータ減少率
<code>param:Mtol = 30</code>	*バリアパラメータ減少判定基準値
<code>param:Mzl = 10</code>	*双対変数値の下限側マージン
<code>param:Mzu = 10</code>	*双対変数値の上限側マージン
<code>param:delcs = 0.25</code>	*信頼領域縮小判定基準
<code>param:delrs = 2.0</code>	*信頼領域縮小率
<code>param:delcl = 0.75</code>	*信頼領域拡大判定基準
<code>param:delrl = 2.0</code>	*信頼領域拡大率
<code>param:betaq = 0.6</code>	*ニュートン方向評価基準値
<code>param:dnu = 0.1</code>	*ニュートン/最急降下方向混合比率ステップ
<code>crit:rhotol = 1e20</code>	*ペナルティパラメータ上限
<code>crit:epsscl = 1.0e-4</code>	*スケーリングループ停止条件

4.3.5 単体法(SimplexMethod)/有効制約法(ActiveSetMethod)に有効なパラメータ

ここで解説するパラメータは以下の設定に対応します．

```
options.method = "simplex" (SIMPLE)    method:simplex (nuopt.prm)
```

```
options.method = "asqp" (SIMPLE)    method:asqp (nuopt.prm)
```

この二つの手法はアルゴリズムが非常に似通っているため，同一のパラメータが有効です．値はこの手法を選んだ際のデフォルトです．

◆ スケーリング

大規模問題の線形制約を表す係数行列のオーダーを揃える操作を行う(スケーリングする)と数値的安定性が向上する場合があります．このパラメータはそれを行うか否かを指定するものです．

- ◆ 双対変数の非零性判定に関するトレランス

計算の停止条件に用いる双対変数（シャドウプライス）の零判定値です（この値以下の解の改善可能性を無視します）。

```
options.told = 1.0e-6 (SIMPLE)    simplex:told=1.0e-6 (nuopt.prm)
```

- ◆ 解ベクトルの各要素の可能性判定に関するトレランス

計算の停止条件に用いる変数の上下限違反判定値です。この値以下の変数値に関する上下限違反を無視します。

```
options.told = 1.0e-8 (SIMPLE)    simplex:told=1.0e-8 (nuopt.prm)
```

4.3.6 逐次二次計画法(line search SQP/trust region SQP)に有効なパラメータ

ここで解説するパラメータは以下の設定に対応します。

以下のパラメータは次の二つのアルゴリズムにおいて有効です。値はこの手法を選んだ際のデフォルトです。

```
options.method = "lsqp" (SIMPLE)    method:lsqp (nuopt.prm)
```

```
options.method = "tsqp" (SIMPLE)    method:tsqp (nuopt.prm)
```

- ◆ スケーリング

大規模問題の線形制約を表す係数行列のオーダーを揃える操作を行う（スケーリングする）と数値的安定性が向上する場合があります。このパラメータはそれを行うか否かを指定するものです。

```
options.scaling = "on" (SIMPLE)    scaling:on (nuopt.prm)
```

- ◆ 停止条件

停止条件として用いる最適性条件の残差です。最適性条件の残差がこの値以下になったときに、計算が収束したとみなし反復計算を終了します。

デフォルト値 ε （約 $1.5e-6$ ）は非線形最適化用に次の式から定められています。

$$\varepsilon = \sqrt{\varepsilon_{mch}} \cdot 10^2$$

```
options.eps = 1.5e-6 (SIMPLE)    param:eps=1.5e-6 (nuopt.prm)
```

◆ 反復回数上限

停止条件として用いる逐次二次計画法の反復計算の回数の上限です。反復回数がデフォルト値の 150 回を越えた場合は、これ以上の反復を続けても解が改善する可能性は低い場合が多いことが経験的に知られています。反復回数がこの回数を越えた場合には解が得られなかったとみなしてエラー (<<NUOPT 40>> SQP iteration limit exceeded.) を出力します。

```
options.maxitn = 150 (SIMPLE)    crit:maxitn=150 (nuopt.prm)
```

◆ その他

以下は `nuopt.prm` からのみ設定可能な内部パラメータで、特定の問題に対するチューニングを行う際に変更の可能性のあるものです。

```
crit:rhotol = 1.0e20 *実行不可能性判定値
sqp:dumppsp = off *部分問題に関する内部エラー発生時の出力有無 (デバッグ用)
sqp:diagval = 1.0 *BFGS 法の初期行列の対角要素値 (lsqp のみ)
sqp:omega = 0.2 * Powell の修正 BFGS 法におけるパラメータ (lsqp のみ)
```

4.3.7 制約充足アルゴリズム (wcsp/rcpsp) に有効なパラメータ

ここで解説するパラメータはタブー・サーチによる制約充足アルゴリズム (wcsp) または資源制約付きスケジューリングアルゴリズムにおいて有効です。値はこの手法を選んだ際のデフォルトです。

```
options.method = "wcsp" (SIMPLE)    method:wcsp (nuopt.prm)
options.method = "rcpsp" (SIMPLE)    method:rcpsp (nuopt.prm)
```

タブー・サーチによる制約充足アルゴリズム (wcsp/rcpsp) は制約をできるだけ充足する解をいずれか一つ求めるという手法で、厳密な最適解を求めるものではありません。そのため、他のアルゴリズムが採用している停止条件：

1. 最適性条件が一定精度で満たされること (連続変数のみの最適化問題)
2. 得られた解が最適解である保障が得られること (整数変数を含む最適化問題)

ではなく、

1. すべてのハード制約およびソフト制約を満たす解が求まった
(目的関数に関しては目標値を満たす解が求まった)
2. 反復回数が指定の上限を超えた
3. 計算時間が指定の上限を超えた
4. ユーザが停止を命じた

を停止条件としており、いずれのケースにおいても正常な終了とみなしています(制限時間により止まってもエラーとは解釈されません)。反復回数および、計算時間の上限は以下によって指定します。以下のデフォルト値において-1は無制限を示します

◆ 反復回数上限

```
options.maxitn = -1 (SIMPLE)      crit:maxitn=-1 (nuopt.prm)
```

◆ 計算時間上限

```
options.maxtim = -1 (SIMPLE)      crit:maxtim=-1 (nuopt.prm)
```

アルゴリズムに与えるパラメータとして、制約条件に対する「重み」と目的関数に対する「目標値」を設定することができます。その解説は 2.8 制約充足問題ソルバのための重みおよび目標値設定、2.10 資源制約付きスケジューリング問題のための重みの設定をご参照下さい。

4.3.8 整数変数計画、大域的最適化 (simple/asq/global) を行う際に有効なパラメータ

以下のパラメータは単体法 (simplex) / 有効制約法 (asqp) / 大域的最適化 (global) を、整数変数を含む問題について適用した際、あるいは大域的最適化 (global) を非線形計画問題に対して適用した場合にのみ有効です。

◆ 探索深さ

探索の深さを設定します。1 とすると深さ優先探索を行います。大きい値であるほど、発見的探索に近くなります。

p を大きめに設定すると実行可能解が見つかりにくく、所要メモリが大きくなりがちです。そのような場合には p=1 (深さ優先探索) という設定が有効な場合があります。

```
options.p = 10 (SIMPLE)      branch:p=10 (nuopt.prm)
```

◆ 足切り点設定用パラメータ Δ_f

- ◆ 整数計画を行っている際：

```
options.addToCutoff = 0 (SIMPLE)  branch:add = 0 (nuopt.prm)
```

- ◆ 非線形関数に関する大域的最適化を行っている際：

```
options.addToCutoff = 1.0e-5 (SIMPLE)
branch:add = 1.0e-5 (nuopt.prm)
```

NUOPT は分枝限定法の探索中，これまでで最良の目的関数値 f_{inc} を持つ実行可能解が求まると， f_{inc} を基に足切り点 f_{cut} を

$$f_{cut} = f_{inc} - \Delta_f \text{ (最小化問題)}$$

$$f_{cut} = f_{inc} + \Delta_f \text{ (最大化問題)}$$

と更新します．すなわち現在求まっている実行可能解から Δ_f 以上良い部分問題のみに探索を限ることになりますので， Δ_f を零以上の値に設定することにより，探索の絞り込みがよりきつくなり，計算の手間を減らすことができます．

ただし，その際には求まった解が最適であることは保証できません．その解から Δ_f 以上良い解がないことのみが保証されます．

- ◆ 足切り点

上記で説明した足切り点 f_{cut} の値そのものです．cutoff より悪い解しか与えないことが解った問題は探索の対象から外しますので，適切に設定すると計算の無駄を省くことができます．この値を良く（最小化問題の場合は小さく，最大化問題の場合は大きく）設定するほど，足切り条件は厳しく，探索の対象は狭くなり，計算の手間は減ります．ただし，厳しく設定しすぎると実行可能解がない(<<NUOPT16>> mip-infeasible) というエラーになりますので，注意が必要です．

```
options.cutoff = 設定なし (SIMPLE)
branch:cutoff = 設定なし (nuopt.prm)
```

◆ 探索問題数上限

探索問題数上限です。探索問題数がこの数を越えると、

```
<<NUOPT 17>> p.c.-max reached (with feasible.sol).
```

あるいは実行可能解が見つからない場合には、

```
<<NUOPT 19>> p.c.-max reached (no feasible.sol).
```

というエラーメッセージとともに現在までの最適解を出力して実行を終了します。(実行可能解が見つからない場合には LP (QP) 緩和解のみの出力となります。また、0 以下の値は設定していないのと同じ意味になります。)

```
options.maxnod = -1 (SIMPLE)    branch:maxnod=-1 (nuopt.prm)
```

◆ 計算時間上限

計算時間の上限です。計算開始から秒単位の計算時間が maxtim を越えると、

```
<<NUOPT 21>> B&B itr. timeout (with feasible.sol).
```

実行可能解が見つからない場合には

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

というエラーメッセージとともに現在までの最適解を出力して実行を終了します。(実行可能解が見つからない場合には LP (QP) 緩和解のみの出力となります。また、0 以下の値は設定していないのと同じ意味になります。)

計算時間には、前処理や最初の緩和解を求める時間が含まれます。

```
options.maxtim = -1 (SIMPLE)    crit:maxtim=-1 (nuopt.prm)
```

◆ 整数性個数上限

実行可能解の個数の上限です。計算開始が見つかった実行可能解が maxintsol を越えると、

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

というエラーメッセージとともに、現在までの最適解を出力して実行を終了します。0 以下は設定していない(無制限)と同じと見なされます。この指定を行えば、実行可能解を1つだけ求めて終了する、ということが可能になります。

```
options.maxintsol = -1 (SIMPLE)    branch:maxintsol=-1 (nuopt.prm)
```

◆ 最大メモリ量制限

プロセスモジュールが使用する最大メモリ量を Mb 単位で指定します。例えば 100 とすると 100Mb を上限とすることを意味し、100Mb を超えた場合に実行を停止します。

UNIX/LINUX 版では負の値を設定すると、制限なしという意味になります。

Windows 版では負の値を設定すると、システムで利用可能なメモリ量が残り $-\text{maxmem}$ 以下になったときに実行を停止します。

メモリ上限によって実行が停止した場合には

```
<<NUOPT 43>> B&B memory error (with feasible.sol.).
```

実行可能解が見つからない場合には

```
<<NUOPT 44>> B&B memory error (no feasible.sol.).
```

というエラーメッセージとともに現在までの最適解を出力して実行を終了します (実行可能解が見つからない場合には緩和解のみの出力となります)。

```
options.maxmem = -10 (SIMPLE)    crit:maxmem=-10
```

◆ 上下界値のギャップによる停止

上下界値のギャップが、指定した値を下回る場合に解の探索を停止します。

ただし、 $\text{gap} = (\text{上界値} - \text{下界値})$ です。gap は目的関数の実際の値に依存することにご注意下さい。以下のエラーが出力されて止まります。

```
<<NUOPT 45>> B&B gap reaches under the limit.
```

実行可能解が求まってはじめて上下界値のギャップは意味を持ちますので、このエラーで停止した場合には必ず実行可能解の出力が成されます。

```
options.gaptol = 設定なし (SIMPLE)
branch:gaptol=設定なし (nuopt.prm)
```

- ◆ 内点法の起動頻度の調整（大域的最適化 global 指定時のみ）

元の問題を凸緩和した問題（内部で自動的に生成されます）を解く際に，内点法を起動して暫定解を求める頻度を調整します．変数の上下限を限定して内点法を起動，出力された実行可能解（暫定解）の情報をを用いることにより，探索プロセス全体の削減が可能であることが確認されていますが，このパラメータはその内点法の起動の頻度を調整するためのものです．

```
options.ipmdepth = 7 (SIMPLE)    ( nuopt.prm からは設定不可)
```

0 または正の整数を与えます．探索木の深さがこの値以下の際に内点法を起動します

- 1 内点法は毎回起動します
- 2 以下の整数 内点法はまったく起動しません

実行可能解を早く得ようとするならば，一般に非線形性の強い（関数が直線で近似しにくい）問題の場合には，この値を大きくして，内点法の起動頻度を上げるのが有利と言えます．ただ，内点法はあくまで実行可能解（暫定解）を求めるために用いるもので，あまり大きくしすぎると，緩和問題の求解時間を圧迫し，解の存在範囲を詰めることができず，探索全体が非効率になる場合もございます．デフォルトは弊社での数値実験の結果ほぼ良いパフォーマンスが得られることが確認されています．

- ◆ デフォルトの上下限值の設定（大域的最適化 global 指定時のみ）

変数に上下限が与えられなかった場合に無意味に変数の分割を行うことを避けるため，変数にはデフォルトで一律 1.0e7 という上下限が与えられております．このパラメータはその値を調整するものです．

```
options.globalVarBound = 1.0e7 (SIMPLE)    (nuopt.prm からは設定不可)
```

一般に小さい値であるほど，探索は速く終了しますが，あくまで「安全弁」としての設定であり，個別の変数に対して意味のある上下限を与えるという方法にくらべて効果は副次的です．

4.4 MPS ファイルに関する設定

この章では問題を入力する MPS ファイルの付加情報の指定方法を解説します。

MPS ファイルから問題を入力する場合 (MPS ファイル入力用ロードモジュール `nuopt` 使用時) にのみ関係します。

◆ 最小化, 最大化の指定

MPS ファイルから読み込んだ問題を目的関数の最小化問題/最大化問題のいずれとして解くかを指定します。デフォルトは最小化です。

```
minimize (minimize/maximize のいずれか)
```

◆ 各種ラベル名

これらは MPS ファイル中に複数の RHS/BOUNDS/RANGE/目的関数行があるとき, 実際の計算で用いるものを指定します。

デフォルトでは最初に現れたものとなります。

```
mpsfile:rhs = 文字列 (RHS ラベル名)
mpsfile:bou = 文字列 (BOUNDS ラベル名)
mpsfile:ran = 文字列 (RANGE ラベル名)
mpsfile:obj = 文字列 (目的関数行ラベル名)
```

上記に関して該当するラベルを持つものが存在しない場合にはエラー:

```
<<MPS FILE 13>> Specified rhs: RHS データラベル not found
<<MPS FILE 11>> Specified bound: BOUND データラベル not found
<<MPS FILE 15>> Specified range data: RANGE データラベル not found.
<<MPS FILE 12>> Specified objective: 目的関数行名 not found
```

となります。

◆ MPS ファイルパス

MPS ファイルを読み込むディレクトリへのパスを指定します。

`nuopt` はコマンドラインから与えた名前の MPS ファイルを, 通常起動したディレクトリから読み込みますが, このパラメータはそれを変更するものです。

与えるディレクトリ名の末尾の / (スラッシュ) は不要ですが, 付けても正しく解釈されます。

```
mpsfile: path = パス
```

4.5 その他の設定

ここで説明する設定は最適化手法に依存しません。

◆ 出力モード

NUOPT が行う標準出力の頻度を設定します。通常は `normal` モードとなっています。
`verbose` モードは主として求解途中で不具合が起きた場合の動作の解析のために設けられており、アルゴリズムの実行を逐次追うのに適した内部出力が得られます。
`silent` モードとすると、標準出力を全く行いません。

```
output: mode = normal (silent/normal/verbose)
```

◆ 解ファイル名

解ファイル (*.sol) の名前のルート名 (拡張子を除いた部分) を指定します。
 ファイル名としてキーワード `_NULL_` を指定すると、解ファイルの出力を行いません。

```
output: name = ファイル名
```

指定を行わない場合、解ファイルのルート名は以下の様に定まります。

a) MPS ファイル入力用ロードモジュール

MPS ファイルのルート名に等しくなります。

MPS ファイル名	解ファイル名	備考
ex1	ex1.sol	
ex1.mps	ex1.sol	.以降が.sol に
ex1.4.mps	ex1.4.sol	最後の.以降のみが.sol に
/nuopt/samples/ex1.mps	ex1.sol	パス名は反映されない

b) SIMPLE 版ロードモジュール

問題を記述した C++ のソースファイルと同じ名前となります。ただし、Windows 版の場合には、C++ のソースファイルにかかわらず、`solver.sol` という名前になります。

システムコードファイル名	解ファイル名	備考
ex2.cc	ex2.sol	拡張子は除去
fitting.1.cc	fitting.1.sol	最後の.以降のみ除去
/local/lib/nuopt/minCost.cc	minCost.sol	パス名は反映されない

4.6 ソルバとのインタフェース

SIMPLE はソルバとのインタフェース用に、大域変数 `options`, `result` を設けており、次の意味で使われます。

- ◆ `options`

ソルバに与えるパラメータ群の格納場所

- ◆ `result`

直前に起動されたソルバからの実行結果に関する情報の格納場所

両者とも SIMPLE から設定/参照可能な複数のメンバを持っていますが、その詳細は連結するソルバに依存して変化します。

SIMPLE データ形式のデータファイルからパラメータを与えることも可能です。その方法については第一部の 4.1.4 データファイルを使った NUOPT のパラメータの定義をご参照下さい。

ソルバとして NUOPT を使用する場合は `options` のメンバは以下の表の通りで、それぞれが NUOPT のパラメータに対応します。

名称	選択	Default	意味
outputMode	"silent",	"normal"	標準出力モード
	"normal",	"	[output:mode = normal]
method	"auto",	"auto"	求解アルゴリズム [method:auto] 詳細については NUOPT/SIMPLE マニュアル 第二部 4.2 最適化手法に関する設定 を参照のこと。
	"lipm",		
	"lepm",		
	"line",		
	"higher",		
	"tipm",		
	"tepm",		
	"trust",		
	"bfgs",		
	"lbfgs",		
	"simplex",		
	"asqp",		
	"lsqp",		
	"tsqp"		
	"wcsp"		
	"rcpsp"		

scaling	"off" "on"	"on"	スケーリングを行うか否か [scaling:on]
maxitn	int	150	内点法の反復回数の最大 [crit:maxitn = 150]
eps	double	自動設定	内点法の停止条件 (内点法専用) [crit:eps = 1.0e-8]
addToCutoff	double	0	足切り点設定用パラメータ (分枝限定法専用) [branch:addtocutoff = 0]
cutoff	double	未定義	足切り点 (分枝限定法専用) [branch:cutoff = 1.8]
p	int	10	探索深さ (分枝限定法専用) [branch:p = 10]
maxnod	int	-1 (無制限)	探索問題数上限 (分枝限定法専用) [branch:maxnod=100000]
maxtim	int	-1 (無制限)	計算時間上限 (秒) (分枝限定法と内点法全般) [branch:maxtim=3600]
maxmem	int	-10 (UNIX/ Linux:無 制限 Windows :10Mb)	分枝限定法のメモリ利用量上限 (Mb), 残り利用可能メモリによる 制限 ⁵⁰ (Windows 版のみ, 負値, Mb) [branch:maxmem=500]
gaptol	double	-1 (指定なし)	上下界ギャップの下限 (この値を 下回ったら停止)
tolx	double	1.0e-8	主問題の実行不可能性判定値 (単 体法のみ) [param:tolx=1.0e-8]
told	Double	1.0e-6	双対問題の実行不可能性判定値 (単体法のみ)

⁵⁰ maxmem に負の値を設定すると, システムの利用可能なメモリが $-\text{maxmem} [\text{Mb}]$ を切ったら実行を停止します (Windows 版のみの機能). デフォルト値 -10 は UNIX 版では無制限という意味ですが, Windows 版では 10MByte を切ったら実行を停止するという意味です.

[param:told=1.0e-6]			
maxintsol	int	-1 (無制限)	整数解取得個数上限 (秒) [branch:maxintsol=3]
iisDetect	"off" "on"	"on" (行う)	実行不可能な行集合 (IIS) 探索を行う/行わない
outfilename	char*	0 (未定義)	NUOPT の解ファイル名 "_NULL_"とすると出力を行わない [output:name=myout]
noDefaultSolve	int	0	solve() を陽に呼ばないと求解を行わない
noDefaultSolout	int	0	Solout() を陽に呼ばないと解の出力を行わない
outputParameter	int	0	Parameter, Set, Element, Expression の CSV ファイル出力を行うかどうか
outputSet		0	
outputElement		0	
outputExpression		1	
multDataPolicy	int	0 (許さない)	同一のデータについてデータを重複して与えることを許すかどうか (1 に設定すると警告扱いとなる)
defaultConstraintWeight	double	-1	指定のない制約式の重み (デフォルトはハード制約)
defaultObjectiveWeight	double	1	目的関数を変形した制約式の重み (デフォルトは重み 1 のソフト制約)
defaultObjectiveTarget	double	0	目的関数の目標値 (デフォルトは 0)

表 20 options のメンバ

NUOPT 側からは, SIMPLE から options の上記のメンバを設定することと NUOPT のパラメータファイルからの指定をすることは等価となっており⁵¹, 上記の表の [] 内は NUOPT のパラメータファイルから同じ意味のパラメータに関して指定を行う場合の例を示しています. パラメータについての詳しい解説は 4 を参照して下さい.

⁵¹ ただし, 両方から同じパラメータを指定した場合には NUOPT のパラメータファイルからの指定が優先します.

また、表 21 は NUOPT による最適化実行の結果を示す `result` のメンバとその意味を示したものです。

名称	型	意味
<code>nvars</code>	<code>int</code>	変数の数
<code>nfunc</code>	<code>int</code>	関数の数
<code>iters</code>	<code>int</code>	内点法の反復回数
<code>fevals</code>	<code>int</code>	関数評価回数
<code>optValue</code>	<code>double</code>	目的関数値
<code>tolerance</code>	<code>double</code>	内点法の収束判定値
<code>residual</code>	<code>double</code>	解における最適性条件の残差
<code>elapsedTime</code>	<code>double</code>	所要計算時間
<code>errorMessage</code>	<code>char*</code>	エラーメッセージ
<code>errorCode</code>	<code>int</code>	エラーコード

表 21 `result` のメンバ

以下は NUOPT がソルバの場合の `options` と `result` の使用例です。

```

// モデル定義
Variable x1,x2;
Objective obj(type=minimize);

obj = log(1+pow(x1,2)) - x2;
pow(1+pow(x1,2),2) + pow(x2,2) == 4;
x1 = 2;
x2 = 2;
//
// ソルバ( NUOPT ) のパラメータ設定
//
// 標準出力を抑制
options.outputMode = "silent";
// 反復の停止条件を設定
options.eps = 1.0e-6;
// 反復の回数上限を設定
options.maxitn = 50;
// スケーリングを行うかどうかのスイッチを設定
options.scaling = "on";
// NUOPT の出力ファイル名設定
options.outfilename = "hock7";

// ソルバ( NUOPT ) の起動
solve();
// 実行結果に関する情報
// 変数の総数
cout << "nvars = " << result.nvars << "¥n";
// 関数の総数
cout << "nfunc = " << result.nfunc << "¥n";
// 内点法の反復回数
cout << "iters " << result.iters << "¥n";
// 関数の評価回数

```

```

cout << "fevals = " << result.fevals << "\n";
// 最適解
cout << "optValue = " << result.optValue << "\n";
// 反復の停止条件
cout << "tolerance = " << result.tolerance << "\n";
// 停止時における KKT 条件の残差
cout << "residual = " << result.residual << "\n";
// 所要時間
cout << "elapsedTime = " << result.elapsedTime << "\n";
// エラーコード
cout << "errorCode = " << result.errorCode << "\n";

```

実行結果は、次のようになります。

```

SIMPLE 2.9.6, Copyright (C) 1994-2003 Mathematical Systems Inc.
<system code file name: lp.cc>
nvars      = 2
nfunc      = 2
iters      = 7
fevals     = 18
optValue   = -1.73205
tolerance  = 1e-06
residual   = 1.87606e-08
elapsedTime = 0.0166667
errorCode  = 0

```

求解過程で異常が起きた場合には `result.errorCode` に零以外の値 (NUOPT のエラーコード) が返りますので

```

solve(); // 求解

if ( result.errorCode == 0 ) { // 判定
    // 正常終了
    SIMPLE 内で解の出力処理 ( print() や dump() のコール ) ;
} else {
    // 異常終了
    エラー処理;
}

```

などとすることによって、求解過程の実行結果に応じた処理を行うことができます。

4.7 基底ファイルとリスタート

アルゴリズムとして

- ◆ 単体法 (SIMPLEX_METHOD) `options.method = "simplex";`
- ◆ 有効制約法 (ACTIVE_SET_METHOD) `options.method = "asqp";`

のいずれかを利用している場合には、基底のセーブとリスタートが可能です⁵²。以下の手続きを利用します。

基底情報の入力先の設定：

```
void nuopt_setBasein(void* theBase);
```

基底情報の出力先の設定：

```
void void nuopt_setBaseout(void** theBase);
```

基底情報のファイルへのセーブ：

```
void nuopt_saveBaseInFile(FILE* fp,void* theBase);
```

基底情報のファイルからの読み込み：

```
void* nuopt_readBaseFromFile(FILE* fp);
```

基底情報の利用状況を示す文字列の取得：

```
const char* nuopt_restartDescChar();
```

基底情報は `void*` のオブジェクトをハンドルとして利用します。

次は典型的な利用例です。

```
VariableParameter rMin;
.... >= rMin; // rMin が制約の右辺に使われている。
options.method = "asqp"; // 有効制約法を指定
void* theBase = 0 ; // 基底情報のハンドラ
// rMin を 1 から 5 まで変化させて何度も解く
for ( rMin = 1; rMin.val <= 5; rMin = rMin.val + 1 ){
    nuopt_setBasein(theBase); // 基底情報を theBase から読み込む
    nuopt_setBaseout(&theBase); // 結果も theBase に書き出す
    solve(); // 求解 (二度目以降は基底情報を利用して解く)
    // 最適化終了時に基底情報を theBase に書き出す。
}
```

`nuopt_setBasein` , `nuopt_setBaseout` は NUOPT の起動 (`solve()`) の前に行ってお

⁵² Ver. 6 からの新機能です。

くことにご注意下さい。nuopt_setBasein では引数 0 が与えられると「利用できる基底情報はない」という意味になります。したがって、上記のループでは二度目以降に基底情報を利用した求解が行われます。nuopt_restartDescChar() は、例えば

```
C16R2L12U1A103
```

のような文字列を返し、基底情報作成時の状態と今回解こうとしている問題がどの程度隔たっているかを示しています。C,R,L,U,A という文字は

C: 目的関数の係数

R: 右辺の係数

L: 下限値

U: 上限値

A: 係数行列の非零要素値

を示しており、それぞれの文字の後の数字が何個変化したか(変化の度合い)を示しています。この文字列が

```
*NA* (N/M)
```

となった場合には、この basis ファイルを作成した問題と、今リスタートによって解こうとしている問題のサイズが異なるので、リスタートができなかったことを示します。

基底情報をディスク上にファイルとして残す場合、読み込む場合には

```
void nuopt_saveBaseInFile(FILE* fp,void* theBase);
void* nuopt_readBaseFromFile(FILE* fp);
```

を用います。fp はバイナリ形式で fopen されたファイル構造体のポインタである必要があります。

Windows 環境では

```
FILE* fp = fopen(ファイル名,"wb"); // 書き出し
FILE* fp = fopen(ファイル名,"rb"); // 読み込み
```

UNIX/LINUX 環境では

```
FILE* fp = fopen(ファイル名,"w"); // 書き出し
FILE* fp = fopen(ファイル名,"r"); // 読み込み
```

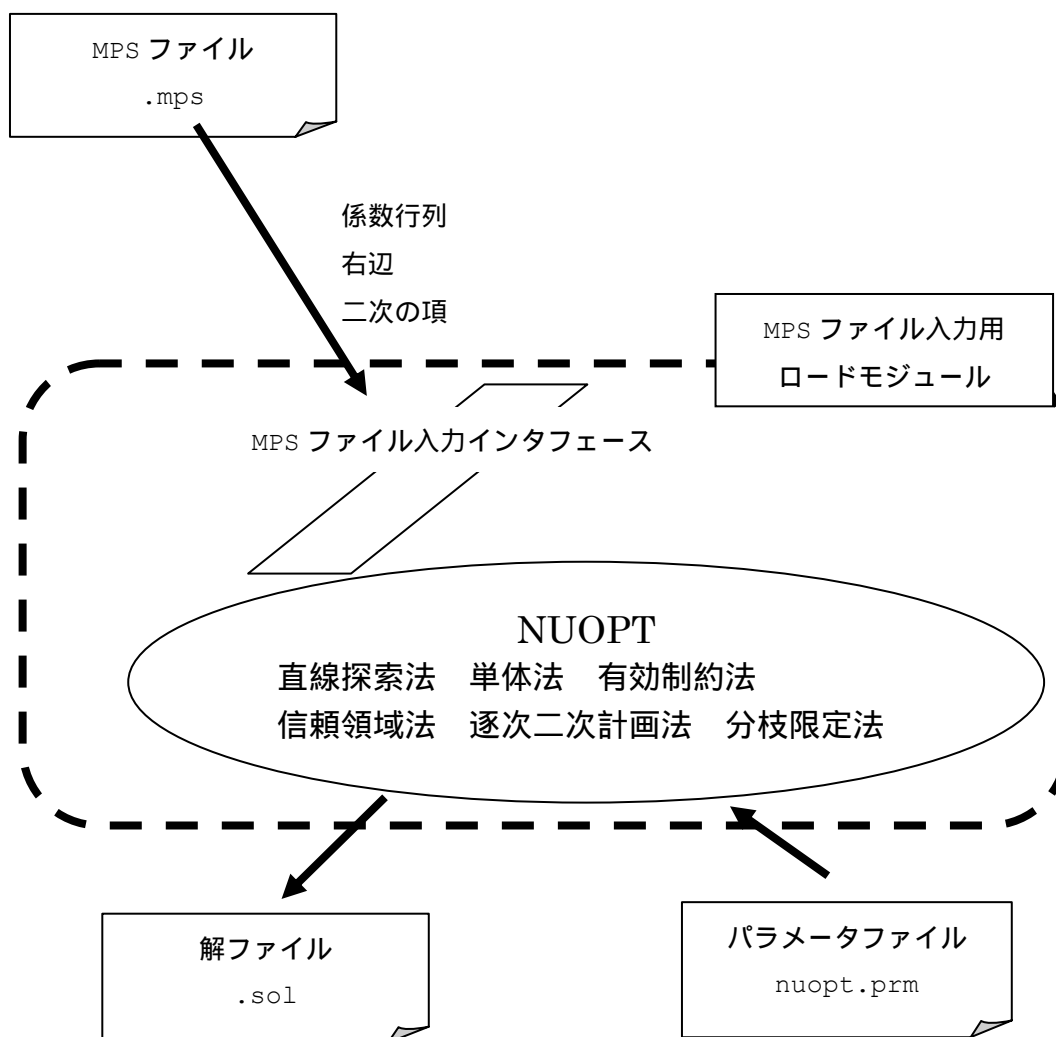
として作成します。

5. MPS ファイル入力用モジュール nuopt

この章では、最適化問題を記述した MPS ファイルを入力とするロードモジュール nuopt について説明します。MPS ファイル自身に関する解説については 5.6MPS ファイルを参照して下さい。Windows のコマンドライン (DOS プロンプト) から同一の使い方が可能です。以降の解説で「コマンドプロンプト」を「DOS プロンプト」と読み替えて下さい。

5.1 ロードモジュール nuopt の構成

以下の図がロードモジュール nuopt の構成を示したものです。NUOPT の本体部であるライブラリに MPS ファイルを読み込むインタフェース部を付加したものとなっています。



5.2 ロードモジュール `nuopt` の起動

`nuopt` はコマンドプロンプトから

```
prompt% nopt MPS ファイル名
```

あるいは MPS ファイル自体を標準入力から与えて

```
prompt% nopt < MPS ファイル (標準入力)
```

として起動します。標準入力から MPS ファイルを与える機能は、例えば

```
prompt% gzip largeprob.mps.Z | nopt
      # “gzip” は 非標準のコマンド
      # 圧縮した状態の MPS ファイルを読み込む
```

の様に使用すると便利な場合があります。

1.3 MPS ファイル用ロードモジュール `nuopt` の簡単な使い方で解説している簡単な線形計画問題の例を記述した MPS ファイル名を `ex1.mps` としましょう。このファイルを入力として `nuopt` を起動するには

```
prompt% nopt ex1.mps
```

または、

```
prompt% nopt < ex1.mps
```

とします。

5.3 ロードモジュール nuopt の出力

5.3.1 標準出力

nuopt を実行すると標準出力に計算の進行が表示されます。

```

prompt% nuopt ex1.mps
NUOPT 6.0.0, Copyright (C) 1991-2001 Mathematical Systems Inc.
<reading MPS file: ex1.mps >
PROBLEM_NAME (TITLE)                example1
ROWS                                4
COLUMNS                             3
NONZEROS                             11
OBJECTIVE                             f
RHS                                  b
NUMBER_OF_VARIABLES                  3
NUMBER_OF_FUNCTIONS                  4
PROBLEM_TYPE                         MINIMIZATION
METHOD                              HIGHER_ORDER
<preprocess begin>.....<preprocess end>
<iteration begin>
    res=2.4e+001 .... 2.7e-005  1.4e-008
<iteration end>
STATUS                              OPTIMAL
VALUE_OF_OBJECTIVE                   -10.5
ITERATION_COUNT                      6
FUNC_EVAL_COUNT                      9
FACTORIZATION_COUNT                  7
RESIDUAL                            1.354127444e-008
ELAPSED_TIME (sec.)                  0.01
SOLUTION_FILE                        ex1.sol

```

```

<reading MPS file: ex1.mps >
PROBLEM_NAME (TITLE)                                EXAMPLE1
ROWS                                                  4
COLUMNS                                              3
NONZEROS                                             11
OBJECTIVE                                             F
RHS                                                  B

```

は MPS ファイルを読み込むインタフェース部分からのメッセージで、MPS ファイルの NAME セクションにあるタイトル (EXAMPLE1)、ROWS セクションで指定された行の数 (4)、COLUMNS セクションで指定された変数の数 (3) と総非零要素数 (11)、目的関数の行の名前 (F) と右辺ラベル (B) を示しています。

```

NUMBER_OF_VARIABLES                                3
NUMBER_OF_FUNCTIONS                                4
PROBLEM_TYPE                                        MINIMIZATION
METHOD                                              HIGHER_ORDER

```

以降の標準出力は NUOPT の本体部からのメッセージで、すべての NUOPT の実行において共通しており、起動する計算手法により異なります。詳細については 3.5 標準出力をご覧ください。

5.3.2 nuopt の解ファイル

nuopt は計算終了と共に、解ファイル (.sol) を出力します。これらには最適化アルゴリズム停止時における、変数や関数 (目的関数及び制約式)、双対変数 (シャドウプライス) の値が記されています。解ファイルの拡張子は .sol ですが、拡張子を除いた部分 (ルート名) は入力した MPS ファイルのパス名から次の様に決定されます。

MPS ファイル名	解ファイル名	備考
ex1	ex1.sol	
ex1.mps	ex1.sol	.以降が .sol に
ex1.4.mps	ex1.4.sol	最後の .以降のみが .sol に
/nuopt/samples/ex1.mps	ex1.sol	パス名は反映されない

ここでは解ファイル (.sol) の記述の中で、MPS ファイル用ロードモジュールに特化した部分について主に述べます。解ファイルのその他の内容はインタフェースに依存せず、SIMPLE

版ロードモジュール(2SIMPLE 版ロードモジュール)の出力するものと共通となっておりますので、3 出力ファイルの解説でまとめて解説してあります。

nuopt の出力した解ファイルの最初には次の様に計算に用いられた MPS ファイルに関する情報が表示されます。この例では

MPS_FILE_NAME	ex1.mps
PROBLEM_NAME (TITLE)	EXAMPLE1
ROWS	4
COLUMNS	3
NONZEROS	11
OBJECTIVE	F
RHS	B

の部分に相当します。ここに書かれる内容とその意味は以下の通りです。

BOUNDS, RANGES という行は MPS ファイル中に BOUNDS, RANGES セクションが存在しない場合には出力されません。存在する場合、計算に使用されたラベル名が例えば以下のように出力されます。

BOUNDS	BND1 FR(2)
RANGE	RNG1

BOUNDS ラベル名の後(上記下線部)には設定された BOUNDS レコードの種類の内分けと数が

UP(21)/FX(3)	#	上限 21 個/固定 3 個
UP(131)/FX(16)/FR(14)	#	上限 131 個/固定 16 個/自由 14 個
UP(65)/LO(64)/FX(18)/FR(1)	#	上限 65 個/下限 64 個/固定 18 個/自由 1 個

のように表示されます。

NONZEROS_IN_HESSIAN, INIT_VALUE_GIVEN_COLUMNS も同様に、存在する場合にのみ

NONZEROS_IN_HESSIAN	4
INIT_VALUE_GIVEN_COLUMNS	2

のように表示されます。COLUMNS セクション中 MARKER 行(5.6.5COLUMNS セクション内 MARKER レコード)によって、整数変数の指定を行った場合には

COLUMNS	2 <u>INT (2)</u>
---------	------------------

と整数変数であると指定された数が表示されます。

解ファイルのこの行以降の書式はすべての NUOPT の実行において共通しています。

NUMBER_OF_VARIABLES	3
NUMBER_OF_FUNCTIONS	4
PROBLEM_TYPE	MINIMIZATION
METHOD	HIGHER_ORDER
STATUS	OPTIMAL
VALUE_OF_OBJECTIVE	-10.5
ITERATION_COUNT	6
FUNC_EVAL_COUNT	9
FACTORIZATION_COUNT	7
RESIDUAL	1.354127444e-008
ELAPSED_TIME (sec.)	0.01

と、最適化計算に関するサマリ情報に引き続いて

%%									
%% VARIABLES									
%%									
	NAME	VALUE	STATUS	[BOUND	TYPE]
V# 1	x1	2.499999998	FREE	[0	<=	x1]
V# 2	x2	1.499999999	FREE	[0	<=	x2]
V# 3	x3	1.223480816e-009	LOWER	[0	<=	x3]
%%									
%% FUNCTIONS									
%%									
	NAME	VALUE	STATUS	[CONSTRAINT/OBJECTIVE	TYPE]
F# 1	f	-10.5	FREE	[OBJECTIVE (MINIMIZE)]
F# 2	g1	3.999999999	UPPER	[g1 <=	4]
F# 3	g2	4.999999998	UPPER	[g2 <=	5]
F# 4	g3	6.499999998	FREE	[g3 <=	7]

のように、各変数、関数の値と上下限、双対変数（シャドウプライス）に関する情報が出力されます。関数名は MPS ファイルの ROWS セクション、変数名は COLUMNS セクションで与えたものに直接対応しています。詳しくは 3.2 解ファイルの変数値表示部をご覧ください。

5.4 nuopt の動作の制御

MPS ファイルに関する付加情報を与えたり, NUOPT 本体の解法アルゴリズムに関して特別な指定を行って, 動作をカスタマイズする際には, その指示を決められた文法で記述したパラメータファイル `nuopt.prm`⁵³を作り, `nuopt` を起動するディレクトリに置きます.

例えば以下の内容の `nuopt.prm` を, `nuopt` を起動するディレクトリに置きます.

```
begin
maximize                * 最大化問題とすることの指定
method: simplex          * 解法アルゴリズム(単体法)の指定
mpsfile: path = /local/lib/netlib/data
                        * MPS ファイルを読み込むディレクトリ
...
end
```

実行すると, `nuopt` の起動直後に

```
NUOPT 6.0.0, Copyright (C) 1991-2003 Mathematical Systems Inc.
<reading parameter file: nuopt.prm >
  begin
  maximize
  method:simplex
  mpsfile:path=/local/lib/netlib/data
  end
```

とパラメータファイルの内容⁵⁴が標準出力に表示され, この内容に従って以降の最適化実行がカスタマイズされていることを示します. パラメータファイルで設定できるパラメータの意味やパラメータファイルの文法については 4 パラメータ設定で詳しく解説しています.

パラメータファイルが存在しない場合, すべてのパラメータはデフォルト設定(特に指定を行わない場合の設定)となりますので, パラメータファイルはデフォルト以外の設定の必要が生じた場合にのみ必要となります.

デフォルト設定は通常の実行に際して問題がないように選ばれておりますが, 変更の可能性が高いものに関しては次項以降で意味と具体的な変更方法について解説します.

⁵³ この名前は固定されています.

⁵⁴ スペースやコメントのみからなる行は無視されます.

5.4.1 nuopt の最適化アルゴリズムの設定

最適化アルゴリズムの指定を特に行わなかった場合、ある問題に対していずれの最適化アルゴリズムを適用するかは MPS ファイルで指定された二次の項が現れている場所や整数変数の有無に応じて以下のように選択されます。

二次の項が存在せず整数変数を含む	単体法 (Simplex Method)
二次の項が存在して整数変数を含む	有効制約法 (Active Set Method)
等式制約に二次の項が現れている	信頼領域法 (Trust Region Method)
線形で整数変数を含まない	線形計画問題専用内点法 (Higher Order Method)
上記以外	直線探索法 (Line Search Method)

ただし凸性が保証できない二次計画問題の場合、上記は不適切な選択となる場合がありますので注意が必要です。不等式制約や目的関数に二次の項が現われている場合には凸な二次計画問題であると仮定して直線探索法を選びますが、二次の項が不等式制約や目的関数にしか現れていなくとも

下に凸な関数 ≥ 0

上に凸な関数 ≤ 0

なる制約や

maximize (下に凸な関数)

minimize (上に凸な関数)

なる目的関数を持つ問題の場合には凸計画問題ではありませんので、直線探索法でなく、信頼領域法を選択するのが適切です。等式制約に二次の項が現れた場合に信頼領域法が選択されるのは、等式制約に二次の項を含む問題が凸計画問題となり得ないことに対応しています。

最適化アルゴリズムを指定する場合には、パラメータファイル (nuopt.prm) に以下のように記述します。

```
begin
method:tipm    * 内点法による信頼領域法を指定
end
```

method 行で指定可能な手法や各手法の適用可能な範囲に関しては 4.2 最適化手法に関する設定を参照して下さい。

5.4.2 nuopt の最小化/最大化の設定

デフォルトは最小化です。最大化としたい場合にはパラメータファイル (nuopt.prm) に、

以下のような maximize 行を書く必要があります。

```
begin
maximize          * 最大化を行うことの指定
end
```

5.4.3 目的関数行/右辺ベクトル/BOUNDS データ/RANGE データの設定

MPS ファイル中に以下のものは複数存在できます。

- ◆ 目的関数行
- ◆ 右辺ベクトル (RHS セクション)
- ◆ BOUNDS データ
- ◆ RANGE データ

特に指定を行わない場合にはいずれもファイル中に最初に現れたものが計算に使用されます。複数指定したうちから特定のものを選択したい場合には、パラメータファイル (nuopt.prm) の mpsfile 行を用いて、例えば

```
begin
mpsfile: objective = COST
mpsfile: rhs = RHS3
mpsfile: bound = BOUND5
mpsfile: range = RNG1
end
```

とします。詳しくは 4.4 MPS ファイルに関する設定を参照して下さい。

5.4.4 解ファイル名の設定

出力ファイル名は特に指定しなかった場合、入力した MPS ファイル名を元に以下のように決定されます。

MPS ファイル名	解ファイル名	備考
ex1	ex1.sol	
ex1.mps	ex1.sol	.以降が .sol に
ex1.4.mps	ex1.4.sol	最後の .以降のみが .sol に

これを別の名前に設定するには `nuopt.prm` から次のように指定します .

```
begin
output: name = myout      * 解ファイルのルート名の指定
end                       ( 解ファイル名は myout.sol となる)
```

5.4.5 nuopt の標準出力モードの設定

指定を行わない場合の NUOPT の標準出力モードは標準 (`mode=normal`) ですが , `nuopt.prm` にて

```
output: mode = silent
```

と指定すると , NUOPT 本体部と MPS ファイル読み込み部の標準出力が , パラメータファイルを読み込んだ旨のメッセージと MPS ファイル読み込み部におけるエラーを除いてすべて抑制されます .

5.5 NUOPT のリスタート機能

Ver.6 から基底情報ファイル (basis ファイル) の出力と読み込み, その情報からのリスタートが可能になりました. nuopt の起動時に

```
-baseout basis ファイル名
-basein      basis ファイル名
```

というオプションを与えます. 以下具体的な起動例です.

basis ファイル名は任意です. basis ファイルはバイナリ形式で保存され, 内容を別の手段で見ることはできません.

1. basis ファイルの書き出し

次は prob.mps という問題を解いて, 基底を prob.bas という基底ファイルに出力します.

```
prompt% nuopt -baseout prob.bas prob.mps
```

nuopt のデフォルトは内点法ですが, -baseout が指定されると, 自動的にクロスオーバーを起動し, 基底情報を作成します.

2. basis ファイルの読み込み

基底を prob.bas という basis ファイルから基底を読み込み, prob2.mps という問題を解きます. basis ファイルが指定されたらいつでも単体法 (LP), 有効制約法 (QP) が起動します.

```
prompt% nuopt -basein prob.bas prob2.mps
```

3. basis ファイルの読み込みおよび書き出し

基底を prob.bas という基底ファイルから基底を読み込み, prob2.mps という問題を解き, その結果を prob2.bas という別の基底ファイルに出力します.

```
prompt% nuopt -basein prob.bas -baseout prob2.bas prob2.mps
```

次は基底を prob.bas という基底ファイルから基底を読み込み, prob2.mps という問題を解き, その結果で基底ファイル prob.bas を更新します.

```
prompt% nuopt -basein prob.bas -baseout prob.bas prob2.mps
```

一般に basis ファイルの読み込みによって最適化の実行が 2~5 倍加速されます。ただ、basis ファイルに含まれている基底情報が解こうとしている問題とそもそも異なる場合や、基底情報を作成した際の問題と基底情報を利用して解こうとしている最適化問題の性質が大きく変化している場合にはその限りではありません。

リスタート時には、例えば

```
BASE_USAGE_STATUS
```

```
C16R2L12U1A103
```

という標準出力があらわれます。これは basis ファイル作成時の状態と今回リスタートで解こうとしている問題がどの程度隔たっているかを示しています。C,R,L,U,A という文字は

C: 目的関数の係数

R: 右辺の係数

L: 下限値

U: 上限値

A: 係数行列の非零要素値

を示しており、それぞれの文字の後の数字が何個変化したか(変化の度合い)を示しています。

NA (N/M)

となった場合には、この basis ファイルを作成した問題と、今リスタートによって解こうとしている問題のサイズが異なるので、リスタートができなかったことを示します。

5.6 MPS ファイル

MPS ファイルは一般形の線形/二次計画問題を記述するためのものです⁵⁵。

一般の線形/二次計画問題は

$$\text{最小 / 最大化} \quad f(x)$$

$$\text{条 件} \quad c_{L_i} \leq g_i(x) \leq c_{U_i}, \quad i = 1, \Lambda, m$$

$$b_{L_j} \leq x_j \leq b_{U_j}, \quad j = 1, \Lambda, n$$

$$\text{初期値} \quad x_j = x_j^0 \quad j = 1, \Lambda, n$$

ここで $f(x)$, $g_i(x)$ は二次関数で

$$f(x) = c_1 x_1 + c_2 x_2 + \dots + c_n x_n + \frac{1}{2} x^t H_0 x$$

$$g_i(x) = a_{i1} x_1 + a_{i2} x_2 + \dots + a_{in} x_n + \frac{1}{2} x^t H_i x$$

と表されます。

MPS ファイルは MPS フォーマットと呼ばれる形式で、次の情報を記述したものです。

$$\text{目的関数, 制約式の線形部分の係数} \quad c_i, a_{ij}$$

$$\text{制約式の上下限} \quad c_{L_i}, c_{U_i}$$

$$\text{目的関数, 制約式の Hessian の要素} \quad H_0, H_i$$

$$\text{変数の上下限} \quad b_{L_j}, b_{U_j}$$

$$\text{変数の初期値} \quad x_i^0$$

次節以下では次のような二次計画問題を記述する場合を例として、MPS ファイルとそのフォーマットについて解説します。

⁵⁵ここでは制約式に2次のものが含まれる問題も二次計画問題と呼んでいます。

最小化	$4x_1 - x_3 + x_2^2$	
条件	$x_1 + x_4$	$= 4$
	$x_1 + 2x_2 - x_3 + x_1x_2$	≤ 10
	$x_2 + x_3$	≥ 2
	$x_1 \geq 3$	
	$4 \geq x_2 \geq 1$	
	$x_3, x_4 \geq 0$	
初期値	$x_1^0 = 4, x_2^0 = 2$	
	$x_3^0, x_4^0 = 0$	

(例題 9 二次計画問題)

次頁は上記の二次計画問題を記述した MPS ファイルの例 (sample.mps) です .

```

NAME          SAMPLE
ROWS
  E   R1
  L   R2
  G   R3
  N   C
COLUMNS
  X1      R1      1.      R2      1.
  X1      C       4.
  X2      R2      2.      R3      1.
  X3      R2     -1.      R3      1.
  X3      C      -1.
  X4      R1      1.
RHS
  B       R1      4.      R2      10.
  B       R3      2.
BOUNDS
  LO BND1  X1      3.
  LO BND1  X2      1.
  UP BND1  X2      4.
HESSIAN

```

X2	X2	2.
R2		
X1	X2	1.
INITIAL		
X1		4.
X2		2.
ENDATA		

5.6.1 書式

カードイメージのファイルで、一つの行がカラム位置によって6つの領域に分けられています。行をレコード、6つに分けられたそれぞれの領域をフィールドと呼びます（下図では[f1]、[f2]、...、[f6]として示されています）。見えない縦の線で仕切られた各行のフィールドを埋めていくイメージで内容を記述します。フィールドをはみ出したり、いくつかのフィールドにまたがってデータを書いたりすることは許されません。フィールドには左から順に番号がつけられており、そこに書くデータの型や種別も決められています。

ROWS					
E	R1				
L	R2				
G	R3				
N	C				
COLUMNS					
X1	R1	1.	R2	1.	
X1	C	4.			
X1	R1	1.	R2	1.	
[f1]	[f2]	[f3]	[f4]	[f5]	[f6]
(2 ~ 3)	(5 ~ 12)	(15 ~ 22)	(25 ~ 36)	(40 ~ 47)	(50 ~ 61)
[文字列]	[文字列]	[文字列]	[数値]	[文字列]	[数値]
(指示子)	(名前)	(名前)	(値)	(名前)	(値)

図 3 MPS ファイルのフィールド

文字列を書くフィールドではスペースが意味をもつことに注意して下さい。すなわち、

"X1"

" X1 "

" X1 "

等は、同一のものを示す文字列としては解釈されません。

フィールド番号とその位置，データの型と種別を以下にまとめます．

フィールド番号	カラム位置	データ型	(種別)
1	2 ~ 3	文字列	(指示子)
2	5 ~ 12	文字列	(名前)
3	15 ~ 22	文字列	(名前)
4	25 ~ 36	数値	(値)
5	40 ~ 47	文字列	(名前)
6	50 ~ 61	数値	(値)

表 22 MPS ファイルのフィールド

5.6.2 セクション

MPS ファイルはセクションと呼ばれるデータのまとまりに分けられます．セクションには以下の種類があります．

- 1 . NAME セクション
- 2 . ROWS セクション
- 3 . COLUMNS セクション
- 4 . RHS セクション
- 5 . RANGES セクション
- 6 . BOUNDS セクション
- 7 . HESSIAN セクション
- 8 . INITIAL セクション
- 9 . ENDDATA セクション

sample.mps に見られる NAME , ROWS 等 , 第一カラムから空白抜きで書かれているセクションの名前がセクションの始まりを示す指示子となります．

指示子はすべて小文字 `name` , `rows` 等としても同じ意味として解釈されます (ただし Name , Rows 等大文字と小文字を混在させたものはセクション名とは見なされません) ．

次のセクション名が現われるまでの記述が , そのセクションに属するデータと見なされます . セクション名とデータをまとめてセクションと呼びます . すなわち sample.mps の ROWS セクションは

ROWS	
E	R1
L	R2
G	R3
N	C

となります。

セクションの出現順は以上に挙げた順番でなければなりません。ただし RANGES , BOUNDS , HESSIAN , INITIAL セクションは省略が可能です (sample.mps では RANGES セクションが省かれています)。以下の節では各セクションの意味と書式について解説します。

5.6.3 NAME セクション

NAME	SAMPLE
------	--------

問題の表題を示すセクションです。データは問題の表題 (文字列) のみで、セクション名と同一レコード上のフィールド 3 に書きます。例ではこの問題の表題を SAMPLE としています。

5.6.4 ROWS セクション

ROWS	
E	R1
L	R2
G	R3
N	C

行 (制約式と目的関数) の名前と型を定義するセクションです。データは行の型を示す指示子と名前を記したレコード (群) です。フィールド 1 に指示子、フィールド 2 に行の名前を書きます。

次は行の型と指示子の対応です。セクション名の場合と同様に、指示子はすべて小文字としても有効です。

型	指示子
=	E
≤	L
≥	G
制約なし	N

表 23 行の型と指示子

MPS ファイルでは、目的関数と制約式は同等に扱われ、目的関数は制約のない特殊な行とし

て定義されます。目的関数を複数個記述しておき、実行時に選択することも可能です。その場合にはここで定義した行の名前を用います。その具体的な指定の方法については **4.4MPs ファイルに関する設定**を参照して下さい。

この例では、制約式の三つの行を順番に R1, R2, R3 とし、それぞれの制約の種類を定義しています。目的関数は C と名付けています。

同一の名前を持つ行が二つ以上現れた場合にはエラーになります。

5.6.5 COLUMNS セクション

COLUMNS				
X1	R1	1.	R2	1.
X1	C	4.		
X2	R2	2.	R3	1.
X3	R2	-1.	R3	1.
X3	C	-1.		
X4	R1	1.		

変数の名前とそれが各行(制約式/目的関数)の線形部分に現れている際の係数を記述するセクションです。データは、変数の名前とそれが現れている行(制約式/目的関数)の名前、並びにその係数の値を記したレコード(群)です。データを書くフィールドは次のように決められています。

フィールド 2	変数名
フィールド 3, 5	行名
フィールド 4, 6	係数の値

制約式と目的関数の線形部分をあわせて一つの行列と考えるなら、このセクションはその行列の非零要素を列毎に記述することになります(変数名はその行列の列名とも考えることができます)。(フィールド 3, 4), (フィールド 5, 6) でそれぞれ一つの非零要素の定義となりますから、1レコードで最大2つの非零要素の定義ができることになります。

例では、変数を X1, X2, X3, X4 と名付けています。変数 x1 (X1) は R1, R2 と名付けられた2つの制約式と目的関数の線形部分に現れていますが、その係数はデータの最初の2レコードを使用して定義されています。

同一の変数に関するデータは続けて書かれていなければなりません。連続しない異なる場所に現れた変数はたとえ同じ名前を持っていても同一の変数であるとみなされず、同じ名前の違う変数であると解釈されてしまいます。

この例では

COLUMNS					
X1	R1	1.	R2		1.
X2	R2	2.	R3		1.
X1	C	4.			

とすると正しく解釈されません．同一の変数がある制約式に 2 回以上現れた場合には，係数の値はその和であると解釈されます．

COLUMNS セクションの記述においてフィールド 5 とフィールド 6 は使用せず，例えば

COLUMNS					
X1	R1	1.			
X1	R2	1.			
X1	C	4.			
X2	R2	2.			
X2	R3	1.			
X3	R2	-1.			
X3	R3	1.			
X3	C	-1.			
X4	R1	1.			

としても有効です．

MARKER レコード

各変数が整数変数であることを指定するには COLUMNS セクションにおいて MARKER レコードという特殊なレコードを使用します．

フィールド 3 に 'MARKER' という文字列があるものを MARKER レコードと呼びず．フィールド 5 に 'INTORG' という文字列がある MARKER レコード，'INTEND' という文字列がある MARKER レコードは整数変数のグループの最初と終わりを示すものと解釈されます．

COLUMNS					
X1	R1	1.	R2		1.
X1	C	4.			
MARK0000	'MARKER'		'INTORG'		
X2	R2	2.	R3		1.
X3	R2	-1.	R3		1.
X3	C	-1.			

MARK0001	'MARKER'		'INTEND'
X4	R1	1.	

例えば上記のように書くと X2, X3 が整数変数であると解釈されます。MARKER レコードのフィールド 2 に書かれているのは MARKER の名前ですが、NUOPT の MPS ファイル解釈部はそれを無視します。

COLUMNS			
X1	R1	1.	R2 1.
X1	C	4.	
MARK0000	'MARKER'		'INTORG'
X2	R2	2.	R3 1.
MARK0001	'MARKER'		'INTEND'
MARK0000	'MARKER'		'INTORG'
X3	R2	-1.	R3 1.
X3	C	-1.	
MARK0001	'MARKER'		'INTEND'
X4	R1	1.	

'INTORG' と 'INTEND' で挟まれたグループを二つ以上書くことも可能で、上記のように書き換えることもできます。ただし、'INTORG' と 'INTEND' による囲みを閉じなかったり、ネストさせたりするとエラーになります。

'MARKER', 'INTORG', 'INTEND' はすべて小文字としても同じ意味となります。

5.6.6 RHS セクション

RHS			
B	R1	4.	R2 10.
B	R3	2.	

制約式の右辺の非零要素を記述するセクションです。

データは、右辺の識別用ラベルと非零要素の位置(右辺が非零である制約式の名前)、並びにその値を記したレコード(群)で、COLUMNS セクションと全く同じ形式を持っています(これは右辺が特殊な列と見なせることに対応しています)。COLUMNS セクションと同様に、データを書くフィールドは次のように決められています。

フィールド 2	ラベル
フィールド 3, 5	制約式名
フィールド 4, 6	値

RHS データは同一のファイルに一般に複数存在することができます。それを区別するのがフィールド 2 のラベルです。パラメータファイルから、特定の RHS データを指定する際にはこのラベルを使用します。ただし同一のラベルに属するデータは続けて記述されていることが前提とされており、たとえ同一のラベルを持っていても、連続しない異なる場所に現れたデータは別の RHS データであると解釈されてしまいます。

例では右辺に B というラベルをつけています。また、この問題の場合すべての制約式の右辺が非零なのでそれぞれ定義しています。目的関数と同様に複数の右辺を記述し、実行の際に選択することができます。ラベルは複数の右辺の中から特定の右辺を指定する際に使用されます。詳細は 4.4mps ファイルに関する設定を参照して下さい。複数の右辺を記述する場合に同一の右辺についてのデータは連続したレコードに書かれていなければならないこと、フィールド 5 とフィールド 6 は空欄としてもよいことは COLUMNS セクションの場合と同じです。

同一の制約式に二つ以上の値が指定された場合には右辺値はそれらの和であるものとして解釈されます。

5.6.7 RANGES セクション

制約式に上限と下限の両方を課したい場合に使用します。このセクションによって、ROWS セクションで定義した等式制約、上限制約、下限制約を上下限制約へと書き換えることができます。

この問題の 3 つの制約式

$$\begin{aligned} x_1 + x_4 &= 4 \quad (\text{R1}) \\ x_1 + 2x_2 - x_3 + x_1x_2 &\leq 10 \quad (\text{R2}) \\ x_2 + x_3 &\geq 2 \quad (\text{R3}) \end{aligned}$$

を RANGES セクションによってそれぞれ

$$\begin{aligned} 1 &\leq x_1 + x_4 \leq 4 \\ 6 &\leq x_1 + 2x_2 - x_3 + x_1x_2 \leq 10 \\ 2 &\leq x_2 + x_3 \leq 6 \end{aligned}$$

とする場合を例にとって説明します。そのためには sample.mps の RHS セクションと BOUNDS セクションとの間に以下の RANGES セクションを挿入します。

RANGES				
RNG	R1	-3.	R2	4.
RNG	R3	4.		

データは、`RANGE` データの識別用ラベルと上下限制約へと変更したい制約式の名前、並びに上下限制約の範囲値を記したレコード(群)で、`COLUMNS` セクションと全く同じ形式を持っています(これは `RANGE` データが特殊な列とみなせることに対応しています)。

データを書くフィールドも `COLUMNS` セクションと同様に次のように決められています。

フィールド 2 ラベル
 フィールド 3, 5 制約式名
 フィールド 4, 6 範囲値

`RANGE` データは一般に、同一のファイルに複数存在することができます。それを区別するのがフィールド 2 のラベルです。パラメータファイルから、特定の `RANGE` データを指定する際にはこのラベルを使用します。指定の方法の詳細については **4.4MPS ファイルに関する設定** を参照して下さい。

同一のラベルに属するデータは続けて記述されていることが前提とされており、たとえ同一のラベルを持っていたとしても、連続しない異なる場所に現れたデータは別の `RANGE` データであると解釈されてしまいます。

`COLUMNS` セクションと同じくフィールド 5 とフィールド 6 は空欄とすることもできます。

範囲値とは上下限の幅を指定する数値です。実際の上下限は、範囲値と制約式の右辺の値と、指定された制約式の元の型によって次のように決まります。

元の型	範囲値 r の符号	下限	上限
$L(\leq)$	$+/-$	$b - r $	b
$G(\geq)$	$+/-$	b	$b + r $
$E(=)$	$+$	b	$b + r $
$E(=)$	$-$	$b - r $	b

表 24 行の型と範囲値と最終的な上下限

表 24 中、 r はこのセクションで指定された範囲値、 b は `RHS` セクションで指定された各制約式の右辺の値です。

例で、2 行目 ($R2$) には、10 以下という上限制約のみがありました。それに 6 以上という下限を付加したいのですから、その幅は 4 です。そこで $R2$ には 4 という範囲値を指定しています。3 行目には 2 以上という下限制約がありましたから 4 という範囲値を指定すれば、 $6 (=2+4)$ という上限が加わります。等式制約の場合には範囲値の符号が意味を持ちます。

現在の右辺の値を上限とし、それよりも下側に下限を課したいのなら負の値、逆なら正の値

とします。この場合には 1 行目の等式制約の右辺 4 を上限として下限 1 を課したいのですから 3 という範囲値を指定しています。この RANGE データ全体は RNG というラベルがつけられており、複数の RANGE データが存在する場合、この RANGE データを特定するのに使用されます。

5.6.8 BOUNDS セクション

BOUNDS			
LO	BND1	X1	3.
LO	BND1	X2	1.
UP	BND1	X2	4.

変数についての上下限制約を記述するセクションです。

データは、制約の型、BOUND データの識別用ラベルと制約を課したい変数の名前、並びに制約値を記したレコード(群)です。制約値とは上下限を指定する数値で、上限制約、下限制約、固定を指定する際にのみ必要となります。

データを書くフィールドは以下のように決められています。

フィールド 1	制約型指示子
フィールド 2	ラベル
フィールド 3	変数名
フィールド 4	制約値

BOUND データ(のひとまとまり)は同一のファイルに複数存在することができ、それを区別するのがフィールド 2 のラベルです。パラメータファイルから、特定の BOUND データを指定する際にはこのラベルを使用します。指定の方法の詳細については 4.4MPs ファイルに関する設定を参照して下さい。

同一のラベルに属するデータは続けて記述されていることが前提とされており、たとえ同一のラベルを持っていても、連続しない異なる場所に現れたデータは別の BOUND データであると解釈されてしまいます。

変数に課すことのできる制約の型と指示子の対応を以下に示します。

型	指示子	意味
下限	LO	$b \leq x (\leq \infty)$
上限	UP	$(0 \leq)x \leq b$
固定	FX	$x = b$
自由	FR	$-\infty \leq x \leq \infty$
非正	MI	$-\infty \leq x (\leq 0)$
非負	PL	$(0 \leq)x \leq \infty$

表 25 BOUNDS セクションの指示子と意味

表 25 で x は制約を受ける変数, b は制約値です。セクション名の場合と同様に, 指示子はすべて小文字としても有効です。FX (自由制約), MI (非正制約), PL (非負制約) の場合には制約値を書く必要はありません。BOUNDS セクションで指定されていない変数は非負制約が課されているものとします。

LO, UP, MI, PL は上下限の一方のみを指定するもので, 他の指定と合せて行うことができます。例えば LO は下限制約のみ, UP は上限制約のみを指定するものですので, 同一の変数に同時に指定することができ, その変数には上下限が設定されたことになります。同様に MI と UP を同時に指定すること等が可能です (下限は $-\infty$, 上限は UP で指定された値になります)。

表中 ($\leq \infty$) 等はこれらの指定が他のものとあわせてでなく, 単独で現れたときに有効になる上下限です。例えばある変数に MI のみが指定されていると, 表から上限は 0 になることがわかります。

同一の制約を二度与えたり, LO と MI など意味上重複する意味の指定を同時に行った場合には以下のようなエラーになります。

```
<<MPS FILE 23>> Same bound spec. on column: 変数名
                    appeared more than once.
<<MPS FILE 24>> Column : 変数名 has bound specification FX and other.
<<MPS FILE 25>> Column : 変数名 has bound specification FR and other.
<<MPS FILE 26>> Column : 変数名 has bound specification LO and MI.
<<MPS FILE 27>> Column : 変数名 has bound specification UP and PL.
```

例では 3 以上という下限制約を変数 x_1 (X_1) に課しています。 x_3 (X_3) と x_4 (X_4) に課されているのは非負制約のみですから, BOUNDS セクションで特に記述する必要はありません。変数 x_2 (X_2) に下限 1 を課すレコードと上限 4 を課すレコードが連続して書かれていますが, これは上下限制約を記述したことになります。

BOUNDS への変数の出現順は, 原則として COLUMNS セクションで定義された順番でなければなりません。ただし, これに違反している実際のデータも多く見受けられるので, NUOPT の MPS ファイル解釈部はこれを見つけてもエラーとせず, 以下のような警告

```
<<MPS FILE 18>> Bound spec. on column : 変数名  should appear earlier.
<<MPS FILE 19>> 数字 column(s) appeared disorderly in BOUNDS section.
```

を出力するのみに止めています。同一の変数に関する指定が連続せずに二度以上現れた場合⁵⁶にも連続して現れたのと同じの処理を行います。

5.6.9 HESSIAN セクション

⁵⁶これは, BOUNDS への変数の出現順の規則に違反している場合にしか起こりません。

HESSIAN		
X2	X2	2.
R2		
X1	X2	1.

目的関数，制約式の二次の項を入力するセクションです⁵⁷．一般に，二次関数の二次の項は 2 階微係数を並べた行列 H (ヘッセ行列) を用いて

$$\frac{1}{2}x'Hx$$

と表されますので，`HESSIAN` セクションでは目的関数/制約式の二次の項に対応する行列 H の非零要素の場所と値とを記述することによって目的関数と制約式の二次の項を指定します．データは， H が属する行 (目的関数/制約式) の名前と H の非零要素の場所と値です．データを書くフィールドは以下のように決められています．

フィールド 2 行名/変数名 1
 フィールド 3 変数名 2
 フィールド 4 H の非零要素値

`HESSIAN` セクションに現れるレコードは

- 行名指定レコード (H の属する行名の指定を行う)

行名

- 非零要素指定レコード (H の非零要素の場所と値の指定)

変数名 1	変数名 2	値
-------	-------	---

のいずれかです．

行名指定レコードが現れると，次の行名指定レコードが現れるまでが，その行に属する二次の項の H の非零要素に関する記述であると解釈されます．目的関数行に属する H の非零要素を記述する場合には，行名指定レコードを省略することができます．すなわち，先行する行名指定レコードなしに，非零要素指定レコードがある場合，それらは目的関数行に属するものと解釈されます．

この例では，最初に目的関数 (C) に属する二次の項の H の非零要素が，続けて行名指定レコードを挟んで，次に 2 つ目の制約式 (R2) に属する二次の項の H の非零要素が，それぞれ指定

⁵⁷通常の MPS ファイルの書式には存在しません．

されています．ここでは目的関数に属する二次の項の指定にあたっての行名指定レコードが省略されています．

目的関数の行名指定を省略せずに

HESSIAN			
C			
X2	X2		2.
R2			
X1	X2		1.

と書いても有効です．

非零要素指定レコードの二つの変数名の組が、 H の非零要素の場所を示します． H は対称であることを仮定しており、上三角部分あるいは下三角部分の非零要素を与えれば、その対称な位置にも値が与えられたものと解釈します．同一の場所に二つ以上の非零要素を与えた場合には、実際の非零要素はその和であると解釈されます．

この例では、 $R2$ についての非零要素指定を

R2			
X2	X1		1.

として、下三角行列の要素を指定しても元と同じ意味となります．

R2			
X2	X1		1.
X1	X2		2.

と両方の順序で与えると、同一の場所に非零要素を重複して与えたことになるので和が取られ、

$H_{1,2}$ と $H_{2,1}$ はともに $3 (=1+2)$ であるものと解釈されます．

5.6.10 INITIAL セクション

INITIAL			
X1			4.
X2			2.

変数の初期値を与えるためのセクションです⁵⁸。データは初期値を指定したい変数の名前及び初期値です。データを書くフィールドは以下のように決められています。

フィールド 2 変数名

フィールド 4 初期値

省略した場合には初期値として 0 が採用されます。この例では、x3 (X3) と x4 (X4) の初期値は 0 ですので指定を省略しています。

同一の変数についての指定を 2 回以上書くと、**最初の指定**が有効になります。

5.6.11 ENDATA セクション

ENDATA

データの終わりを示すセクションです。このセクションはデータを必要としません。このセクション以降の記述は無視されます。

⁵⁸通常の MPS ファイルの書式には存在しません。

5.7 MPS ファイル出力

モデルを記述した後, `mpsout` という関数を呼び出すことによって, システムの内容を MPS ファイル形式にて出力することができます.

最も簡単には `ufun()` 内にシステムを記述した後, `solve()` を呼び出す要領で

```
mpsout(); // MPS ファイルの出力
```

と記述します. この `ufun()` から作成されたロードモジュールを, 通常と同様に実行するとロードモジュール名 `.mps` という名前の MPS ファイルが出力されます.

出力される MPS ファイル名を指定したい場合には次のように `.mps` を除いた部分を引数として与えて下さい.

```
mpsout("filename"); // filename.mps という MPS ファイルを出力
```

さらに `mpsout()` は `solve()` と同様 複数の Objective が定義されている場合に対応して, 引数として目的関数を与えることもできます. その場合には MPS ファイル名の指定を目的関数名の後に続けます.

```
// 複数の目的関数
Objective f(type=minimize);
Objective g(type=maximize);
...

mpsout(f, "fobj"); //目的関数が f の MPS ファイル(fobj.mps)を出力
mpsout(g, "gobj"); //目的関数が g の MPS ファイル(gobj.mps)を出力
```

5.7.1 MPS ファイル出力に関する制限

以下は MPS ファイル出力に関する制限事項です.

- ◆ 変数名, 関数名

MPS ファイルの変数名は文字数の制限がありますので, 変数名は一律 `x1, x2, ...`, 関数名は `f1, f2...` という名前に変更されます. これらの名前と SIMPLE 内部で付けた名前との対応は出力される MPS ファイルの先頭部分に, 次のように MPS ファイル書式のコメントの形式で記述されます.

```

VARIABLE NAME (MPSFILE - original)

X1          -   var[1]
X2          -   var[2]
            ...

FUNCTION NAME TABLE (MPSFILE - original)

F1          -   obj
F2          -   NONAME
            ...

```

- ◆ 特殊な整数変数

`IntegerVariable` のうち、以下の種類の整数変数を含むモデルは MPS ファイルに出力できません。

`type = partial` `until` と併用で、指定された限界 (`until`) まで整数値を取りそれ以上は任意の値を取る変数。
`= semicont 0` であるか、1 以上の変数。

- ◆ 一般の非線形計画問題

制約式と目的関数の次数が二次以下の問題の出力のみが保証されます (`SIMPLE` の現状の実装では `pow(x, 2)` など意味上二次の関数の一部を「非線形」と判定してしまうため、出力の抑制は行っておりません)。

- ◆ 最大化/最小化

最大化問題は**最小化問題に変換**されます (目的関数の符号が反対になります)。

- ◆ 問題規模の制限

変数、関数のいずれかが 9999999 個以上の問題は出力できません。

- ◆ 複数の Objective が定義されている場合

複数の Objective が定義されている場合にも、そのすべてが `N` (自由) 行として MPS ファイルに出力されます。そのため、この MPS ファイルを `NUOPT` に入力する際に適宜目的関数ラベルの指定を行うことによって MPS ファイルを出力し直さなくとも目的関数を切り換えることが可能です。

ただし複数出力された `N` 行のうち、`mpsout` で陽に指定されるかあるいは**第一部の 5.2 ソルバの起動 `solve()` 関数**で述べられているデフォルトルールで目的関数になるよう

に定められた Objective は最初に並べかえられています .NUOPT は特に指定がない場合には最初に現れた N 行を目的関数と解釈しますので ,NUOPT で目的関数ラベルの指定を行わない場合には ,SIMPLE で指定に従った目的関数が用いられることになります .

5.7.2 MPS ファイル出力時のエラー

以下は mpsout が出力するエラーメッセージです .

```
<<MPSOUT 1>> Memory error.
```

メモリーエラーが起きました .

```
<<MPSOUT 2>> too many variables(functions) to produce mps output
```

変数 (関数) の数が 9999999 以上となりました .

```
<<MPSOUT 3>> semi- or partially- continuous variables appeared
```

type=partial,semicont の整数変数 (IntegerVariable) が現れました .

```
<<MPSOUT 4>> 変数名 has infeasible bound.
```

変数名で示される変数の上下限が矛盾しています .

付録 A NUOPT/SIMPLE のエラーメッセージ

A.1 SIMPLE のエラーメッセージ

次表は SIMPLE の実行時のエラー一覧です。メッセージは英語 (UNIX 版) あるいは日本語 (Windows 版) で、下の表では両方が並べて記述されています。

エラー番号	エラーメッセージ	説明
1	<<SIMPLE 1>> Infeasible bound for variable XX . <<SIMPLE 1>> 変数 XX について矛盾した上下限が与えられました。	変数 XX に与えられた上下限が矛盾しています (モデル全体を通して上下限が結合された結果についてこのチェックが行われます)。
2	<<SIMPLE 2>> 関数 MINIMIZE/MAXIMIZE の中には、上下限を使用することができません。	
3	<<SIMPLE 3>> Infeasible bound for constraint XX . <<SIMPLE 3>> 制約 XX について矛盾した上下限が与えられました	制約式 XX に与えられた上下限が矛盾しています (モデル全体を通して上下限が結合された結果についてこのチェックが行われます)。
4	<<SIMPLE 4>> Warning: Length of a subscript exceeds 30. <<SIMPLE 4>> 警告: 添字の名前の長さが 30 文字を越えました。	警告: 添字として使われる文字列の文字数が 30 を越えています。 (30 文字目以後の文字は無視して処理します。)
5	<<SIMPLE 5>> Subscript has the wrong dimension. <<SIMPLE 5>> 添字の次元が合致しません。	使われたオブジェクトの添字の数が一致していません。(モデル中のオブジェクトの定義の際の添字の次元 (index=?) とそれが使われた時点 ([i,j] 等) での添字の次元が一致しているかどうかを確認して下さい。)

6	<p><<SIMPLE 6>> Illegal characters ("[" , "]" , "... " and so on)included in subscript.</p> <p><<SIMPLE 6>> 添字の中に、無効な文字 ("[" , "]" , "... " など) が含まれています。</p>	非合法の文字 ("[" , "]" , "... "など) が添字の現われるべき所に使われました。
10	<p><<SIMPLE 10>> Subscript expected on rhs assignment.</p> <p><<SIMPLE 10>> データファイル XX の YY 行目に記述エラーです。 ZZ とあるところには添字がなくてはなりません。</p>	データファイルにおいて、添字付きオブジェクトに代入される内容(等号の右側)に "["]" が現れていません。
11	<p><<SIMPLE 11>> Data file: Value expected on rhs of assignment.</p> <p><<SIMPLE 11>> データファイル XX の YY 行目に記述エラーです。 オブジェクト ZZ の右辺で WW とあるところにはデータがなくてはなりません。</p>	データファイルにおいて、添字付きオブジェクトに代入される内容(等号の右側)に値が現れていません。
12	<p><<SIMPLE 12>> ~<<SIMPLE 17>>Internal Error!</p> <p><<SIMPLE 12>> ~<<SIMPLE 17>> システム内部エラー!</p>	SIMPLE の内部エラー。 (nuopt-support@msi.co.jp へお知らせ下さい。)
19	<p><<SIMPLE 19>> Warning: No auto-assignment performed for constant set.</p> <p><<SIMPLE 19>> 警告: 自動代入によって定義されない集合があります。</p>	通常なら自動追加が行われる場合ですが、自動追加先が定数集合であるので行われません。(警告)(Setを定義する時、superSet に定数集合等を指定するところの警告メッセージが現れます。)
20	<p><<SIMPLE 20>> Data file: Can't match the pattern "from... to".</p> <p><<SIMPLE 20>> データ読み込み: "from ... to" 形式の文法エラーです。</p>	データファイルに from/to の省略記号 "... " が現れましたが、from/to がペアーになっていません。
22	<p><<SIMPLE 22>> Data file: Number of values not matched with the index dimension of set.</p> <p><<SIMPLE 22>> データ読み込み: データ中の添字の数とモデル中の集合の次元が合致しません。</p>	データファイル中で、あるオブジェクトに代入される内容の "["]" の中に現れる Element の数がそのオブジェクトの定義時の index の数と合っていない。
23	<p><<SIMPLE 23>> Expand from-to: "... " or ".. " appeared at the head or the tail of the stream.</p>	データを表す文字列の先頭または末尾に from/to の省略記号 "... "/".." が現われました。

	<<SIMPLE 23>> データ読み込み: "... " または "... " が定義の先頭か末尾に現われました.	
24	<<SIMPLE 24>> Attempt to find the maximum of an empty set. <<SIMPLE 24>> 空集合から最大要素を求めようとした.	空集合から最大要素を求めようとしています.
26	<<SIMPLE 26>> Only one-dimension data can be specified as interval number. <<SIMPLE 26>> 1次元のデータに限って, interval の指定は有効です.	範囲 Interval の定義(宣言)時に dim=1 以外を与えました.
30	<<SIMPLE 30>> Interval data not matched with the format: lower bound, upper bound. <<SIMPLE 30>> Interval の入力データ形式は " 下限, 上限" でなければなりません.	範囲 Interval の定義(宣言)時に下限, 上限順を与えていません.
33	<<SIMPLE 33>> A call has been made to an invalid function on Interval. <<SIMPLE 33>> Interval に対して(要素取り出しなど) 無効な関数呼び出しが行なわれました.	Interval に対して無効な関数を実行しようとした.
34	<<SIMPLE 34>> Warning: No auto-assignment performed for interval set. <<SIMPLE 34>> 警告: Interval に対しては自動代入は行なわれません.	通常なら自動追加が行われる場合ですが, 自動追加先が Interval なので行われません. (警告)
36	<<SIMPLE 36>> Sequence is empty. <<SIMPLE 36>> Sequence が空となっています.	列 Sequence が空です.
37	<<SIMPLE 37>> Only one-dimension data can be specified as Sequence. <<SIMPLE 37>> Sequence の指定は1次元のデータに限って有効です.	列 Sequence の定義(宣言)時に dim=1 以外を与えました.
39	<<SIMPLE 39>> Sequence data not matched with the format:	列 Sequence の定義(宣言)で, 最初の値, 列の最

	"first .. last, step". <<SIMPLE 39>> Sequence データ形式は " 開始要素 .. 終了要素, 増量" でなければなりません.	後の値, 増分 (default=1) のいずれかが抜けています.
40	<<SIMPLE 40>> Sequence: Illegal operation. <<SIMPLE 40>> Sequence に対する無効な演算が行なわれました.	列 Sequence に対して無効な演算を行おうとしました.
41	<<SIMPLE 41>> Sequence の指定は 1 次元のデータに限って有効です.	
46	<<SIMPLE 46>> Attempt to find the maximum of an empty sequence. <<SIMPLE 46>>空の Sequence から最大要素を求めようとしてしました.	空列から最大要素を求めようとしてしました.
47	<<SIMPLE 47>> Warning: No auto-assignment performed for sequence. <<SIMPLE 47>>警告: Sequence に対する自動代入は行なわれません.	通常なら自動追加が行われる場合ですが, 自動追加先が Sequence なので行われません. (警告)
49	<<SIMPLE 49>> A call has been made to an invalid function. <<SIMPLE 49>> 無効な関数呼び出しが行なわれました.	無効な関数呼び出しが行われました.
53	<<SIMPLE 53>> Operation between elements of different dimension. <<SIMPLE 53>> 要素が異なる次元を持つ集合同士で演算が行なわれました.	属している集合の次元が異なる Element の間に演算が行われました.
58	<<SIMPLE 58>> Set difference: s1-s2: s1 doesn't include s2. <<SIMPLE 58>> 集合引算: s1-s2 で, s2 は s1 に含まれていませんでした.	集合の差分 (s1-s2) 演算において, s2 は s1 の部分集合になっていません.
59	<<SIMPLE 59>> Fixed value (...) out of defined range (...). <<SIMPLE 59>>要素(...)に対して定義域外の値 (...) を代入しようとしてしました.	Element がその定義範囲外の値に固定されようとしています.
60	<<SIMPLE 60>> Illegal characters included in subscript. <<SIMPLE 60>> 添字に無効な文字が含まれています.	添字に無効な文字が含まれています.

62	<p><<SIMPLE 62>> Comparison between elements of different dimemsion.</p> <p><<SIMPLE 62>> 異なる次元を持つ要素の間に比較が行なわれました。</p>	属している集合の次元が異なる Element 同士間で比較が行われました。
64	<p><<SIMPLE 64>> Constraint: subscript not matched.</p> <p><<SIMPLE 64>> 制約式(Constraint): 添字が合致しません。</p>	制約式の添字エラーです。
67	<p><<SIMPLE 67>> Index error in reference of XX.</p> <p><<SIMPLE 67>> 参照オブジェクト XX の添字付けに誤りがあります。 [引き続き, 例えば次のようなメッセージが出力されます.] スカラなのに, 次元 XX の添字が付けられています。</p>	<p>代入の右辺や計算式の中にあるオブジェクト XX に正しい添字が与えられていません。</p> <p>XX が添字なしなのに添字付けられている</p> <p>XX に添字をつけるべきなのに添字付けられていない</p> <p>XX の添字の次元が違う</p> <p>というケースが該当します。具体的にどのケースかこれに続いてメッセージが出力されます。</p>
70	<p><<SIMPLE 70>> Unmatched or ambiguous element(s).</p> <p><<SIMPLE 70>> 合致しないか, 不定の要素がありました。</p>	添字の数が合っていません。(代入の左辺と右辺で現れる添字が違う場合等に生じます。)
74	<p><<SIMPLE 74>> Improper use of a dependent subscript.</p> <p><<SIMPLE 74>> 他の添字に依存する添字はその依存している添字と一緒に現われなければなりません。</p>	添字が他の添字に依存しているにも関わらず, 単独でしかも固定されないまま使われました。(例えば Element j(set=S[i]); として宣言された Element が i を伴わず, 固定されずに使われた場合に生じます。)
75	<p><<SIMPLE 75>> Index dimension error in assignment to object XX , scalar but with index.</p> <p><<SIMPLE 75>> 代入の対象オブジェクト XX の添字付けに誤りがあります。 スカラ(添字なしで宣言)ですが添字付けして値を設定しようとしています。</p>	代入の左辺にあるオブジェクト XX は添字なしで宣言されていますが添字を付けて値の設定や代入が成されました。

76	<p><<SIMPLE 76>> Index dimension error in assignment to object XX, Defined on index set N but try to assign a value with index of dimension M.</p> <p><<SIMPLE 76>> 代入の対象オブジェクト XX (定義集合の次元: N) の添字付けに誤りがあります. 添字の次元が M です.</p> <p>[あるいは]</p> <p>代入の対象オブジェクト XX の添字付けに誤りがあります. (Element の宣言の際に添字付きの集合の添字付けを忘れている可能性があります.)</p>	<p>代入の左辺にあるオブジェクト XX が宣言された集合の次元と違う次元の添字を付けて値の設定や代入が成されました.</p>
77	<p><<SIMPLE 77>> Index dimension error in assignment to object XX, Defined on index set with N but try to assign a scalar value.</p> <p><<SIMPLE 77>> 代入の対象オブジェクト XX (定義集合の次元: N) の添字付けに誤りがあります. 添字付けせずに値を設定しようとしています.</p>	<p>XX は N 個の添字を付けて定義されていますが, スカラのように添字付けせずに値を設定しようとしています.</p>
78	<p><<SIMPLE 78>> No element exists in [...] function.</p> <p><<SIMPLE 78>> "[...]" の中に添字がありません.</p>	<p>"[]" の中に, Element が現れていません.</p>
81	<p><<SIMPLE 81>> Subscript mismatch.</p> <p><<SIMPLE 81>> 添字の数が合致しません.</p>	<p>添字に関するエラーが検出されました.</p>
82	<p><<SIMPLE 82>> Subscript ... out of range.</p> <p><<SIMPLE 82>> ... の添字が定義域外である... となりました.</p>	<p>オブジェクトに付けられた添字の値が, その定義の際に (index=?) 設定された添字の値の範囲をはみ出しました.</p>
91	<p><<SIMPLE 91>> Operation between set of different dimension.</p> <p><<SIMPLE 91>> 異なる次元の要素を持つ集合の間に演算が行なわれました.</p>	<p>要素の次元 (dim) が異なる集合間で演算が行なわれた.</p>
92	<p><<SIMPLE 92>> Warning: No auto-assignment performed for ...</p> <p><<SIMPLE 92>> 警告: 演算結果から生成された以下の集合に対する自動代</p>	<p>通常なら自動追加が行われる場合ですが, 自動追加先が集合演算の結果 (和集合) であるので行われませ</p>

	入は行なわれません.	ん. (警告)
97	<<SIMPLE 97>> Warning: No auto-assignment performed for sets made by setof. <<SIMPLE 97>>警告: "setof" で作った集合に対する自動代入は行なわれません.	通常なら自動追加が行われる場合ですが, 自動追加先が setOf の結果であるので行われません. (警告)
98	<<SIMPLE 98>> from ... to has been defined before data file reading. <<SIMPLE 98>> "from ... to" がデータファイルを読み込む前に、定義されました.	
102	<<SIMPLE 102>> Set assignment: dimension of lhs. and rhs. conflicts. <<SIMPLE 102>>集合に対する代入で等号の右辺と左辺の集合の要素の次元が合致しません.	集合同士の代入(またはデータファイルからの読み込み)の場合, 右辺の集合の要素の次元と左辺の集合の要素の次元が合っていない.
103	<<SIMPLE 103>> Dimension not matched when using superSet. <<SIMPLE 103>>異なる次元を持つ集合が "superSet" として使用されました.	要素の次元の違う集合同士に包含関係を定義しようとした. (Set T(dim=2); Set S(dim=1,superSet=T); とした場合等.)
104	<<SIMPLE 104>> ... (1) can not be used as a superset of ... (2) <<SIMPLE 104>> ... (1) は... (2) の superSet として使用することができません.	"... (1)" は"... (2)" の親集合になれません.
105	<<SIMPLE 105>> Argument error: ... (1) of ... (2) <<SIMPLE 105>>引数エラー: ... (1) の... (2)	オブジェクト定義時の引数エラーです.
106	<<SIMPLE 106>> Argument arcs must be a 2-dimensional set. <<SIMPLE 106>>引数 "arcs" は 2 次元の集合でなければなりません.	グラフ定義時に, 属性 arcs が dim=2 の集合になっていません.

107	<<SIMPLE 107>> Argument nodes must be a 1-dimensional set. <<SIMPLE 107>>引数 "nodes" は 1 次元の集合でなければなりません.	グラフ定義時に, 属性 nodes が dim=1 の集合になっていません.
108	<<SIMPLE 108>> ERROR in check(): ... condition is not satisfied. <<SIMPLE 108>>関数 check() の中の条件 ... が満足されませんでした.	関数 check() の条件に違反しました.
110	<<SIMPLE 110>> Argument: invalid use of index. <<SIMPLE 110>> "index=" が誤った場所に使用されています.	オブジェクト定義時に属性引数 index が無効な場所に現れました.
111	<<SIMPLE 111>> Assignment: rhs includes free subscript. <<SIMPLE 111>>代入の右辺に決定できない添字がありました.	代入の右側に不定になる添字 (Element) が現れました.
113	<<SIMPLE 113>> Invalid assignment. <<SIMPLE 113>> 無効な代入が行なわれました.	その他の代入に関するエラーが検出されました.
114	<<SIMPLE 114>> Argument: Invalid use of set. <<SIMPLE 114>> "set=" が誤った場所に使用されています.	オブジェクト定義時に属性引数 set が無効に使われました.
116,117	<<SIMPLE 116>>,<<SIMPLE 117>> "set=" が誤った場所に使用されています.	
118	<<SIMPLE 118>> Set に対して無効な代入が行われました (マニュアルをご覧ください) .	
119	<<SIMPLE 119>> Assignment: "[" or "]" occured both in lhs and rhs of = when assigning a set by a string. <<SIMPLE 119>> オブジェクト[添字] = 文字列という代入文で, 文字列の中にも "[" または "]" が現れました	文字列を集合へと代入する際, 文字列の中身である集合の値と代入の左辺両方が添字付けされていました. (文字列による集合への代入を行う際には, 等号の両側に同時に "[" または "]" が現れることを禁止しています. 例えば, Set S; S[1]="[1] a 1 2 [1] 3"; はエラーになります.)
120	<<SIMPLE 120>> Operators "+=", "-=", "=", "/=", "++" and "--"	演算子 += -= = /= ++ -- を不適切なオブジェク

	<p>are not allowed here.</p> <p><<SIMPLE 120>> 演算子 "+=", "--=", "*=", "/=", "++",と"--" は使用することができません.</p>	トに適用しました.
121	<p><<SIMPLE 121>> Invalid constraint specification: (parameter <= expr => parameter is not allowed).</p> <p><<SIMPLE 121>> 有効でない制約式 ("parameter <= expr => parameter") が使用されました.</p>	SIMPLE が解釈できない制約式の形 (定数式<= 式>= 定数式, 定数式>= 式<= 定数式) が現れました.
122	<p><<SIMPLE 122>> Data file: = expected.</p> <p><<SIMPLE 122>> データファイル ... (1) の XX 行目に記述エラーです . オブジェクト名 ... (2) の右辺に等号 "=" がありません .</p>	<p>データファイル中 , = が現れるはずのところにその他の文字が現れました .</p> <p>(データファイルで a[1] = 3; a[2]= 5; を表現する場合には a= [1] 3 [2] 5 ; と記述します . 代入の左辺には [] 等の文字列は現れてはいけません .)</p>
123	<p><<SIMPLE 123>> Data file: Syntax error.</p> <p><<SIMPLE 123>> データファイル ... (1) の XX 行目に記述エラーです . オブジェクト ... (2) の定義の付近です . [あるいは] モデル中の文字列の記述エラーです . [引き続き , 次のメッセージが出力されます .] 文法エラーが起きました .</p>	データファイルの文法エラーです . (データファイルを定義するときに , データの区切の";" を忘れたりした場合に生じます .)
125	<p><<SIMPLE 125>> Remove/Restore Constraint: Constraint number out of range.</p> <p><<SIMPLE 125>> 制約式に対する Remove/Restore 関数で: 指定された制約式の番号が存在しません .</p>	制約式番号で削除/復帰させる制約式を指定する際にその番号が正しくありません .
127	<p><<SIMPLE 127>> String uncompleted (missing a " mark) .</p> <p><<SIMPLE 127>> 不完全な String (" が足りません) .</p>	値としての文字列の" がペアになっていません .

128	<<SIMPLE 128>> String is empty. <<SIMPLE 128>> データ文字列が空です.	値としての文字列が空です.
129	<<SIMPLE 129>> String contains space(s). <<SIMPLE 129>> データ文字列に空白が含まれています.	値としての文字列の名前に空が含まれています.
130	<<SIMPLE 130>> Invalid cast from character to int. <<SIMPLE 130>> 文字から数字へのキャストが行なわれました.	演算等で、文字を数字 <code>int</code> へキャストする必要が生じましたが、失敗しました.
131	<<SIMPLE 131>> Invalid cast from character to double. <<SIMPLE 131>> 文字から <code>double</code> へのキャストが行なわれました.	演算等で、文字を数字 <code>double</code> へキャストする必要が生じましたが、失敗しました.
135	<<SIMPLE 135>> Syntax error occurred within [...] function. <<SIMPLE 135>> データ書式の[...]の中に文法エラーが発生しました.	データファイルにおける"[]"の中の定義が文法エラーとなっています.
136	<<SIMPLE 136>> Syntax error occurred within <...> function. <<SIMPLE 136>> データ書式の<...>の中に文法エラーが発生しました.	データファイルにおける"< >"の中の定義が文法エラーとなっています.
140	<<SIMPLE 140>> Fixed value out of range for variable. <<SIMPLE 140>> Variable が上下限の範囲外に固定しようとしてしました.	変数の現在の値がその変数の上下限の範囲外にあるので、変数を固定することができません.
142	<<SIMPLE 142>> Constraint object required in function call. <<SIMPLE 142>> 制約式がインスタンスでなければなりません.	制約式をシステムから削除 / 復帰する関数 (<code>deleteCo()</code> , <code>restoreCo()</code>) の引数として宣言されている <code>Constraint</code> のインスタンス以外が渡されました. (<code>deleteCo()/restoreCo()</code> に渡せるのは <code>Constraint</code> と宣言されたオブジェクトのみです. 例えば, <code>deleteCo(x[i]+y[i]>=6)</code> などは違法です.)
148	<<SIMPLE 148>> Argument: invalid use of superSet. <<SIMPLE 148>> "superSet=" が誤った場所に使用されています.	オブジェクト定義の属性引数 <code>superSet</code> が無効に使われました.
150	<<SIMPLE 150>> Set Assignment: lhs of assignment must be an	集合に対する代入の左辺として、事前に定義した集

	<p>set instance.</p> <p><<SIMPLE 150>> 演算結果などの代入できない集合が代入の左辺に現れています.</p>	<p>合のインスタンス以外が使われました. (集合に対する代入の左側に来ることができるのは, 宣言されたオブジェクトのみです. 例えば, ST = "1 2 3"—などは違法となります.)</p>
151	<p><<SIMPLE 151>> Set Auto-assignment failed.</p> <p><<SIMPLE 151>> 集合に対する自動代入が失敗しました.</p>	<p>集合に対する自動追加が失敗しました. (ある集合の自動追加を行わねば集合の包含関係が矛盾となることがわかりましたが, その集合は定数集合等である等の理由で自動追加ができない場合に生じます.)</p>
160	<p><<SIMPLE 160>> Empty data can't be casted to double.</p> <p><<SIMPLE 160>>空データから double へのキャストが行なわれました.</p>	<p>オブジェクトの参照可能な値を評価した結果空であるので, double へキャストすることができません.</p>
162	<p><<SIMPLE 162>> Invalid assignment to current, init or dual member.</p> <p><<SIMPLE 162>> 値(.val)、初期値(.init) または双対変数値(.dual) に対する代入が行なわれました (参照のみが可能です).</p>	<p>オブジェクトの参照可能な値 (y[i].val , x.init , z.dual など) に対する代入を行おうとしました. (これらは通常の Parameter と同じに扱われますが, 対して代入を行うことはできません.)</p>
163	<p><<SIMPLE 163>> No current value for empty constraint.</p> <p><<SIMPLE 163>>制約式が空なので制約式の値(.val) が存在しません.</p>	<p>宣言したのみで定義されていない制約式に対して現状値を調べようとしてしました.</p>
164	<p><<SIMPLE 164>> No init value for empty constraint.</p> <p><<SIMPLE 164>>空の制約式 ... に関連する値は 0 として出力されます.</p>	<p>宣言したのみで定義されていない制約式に対して初期値を調べようとしてしました.</p>
165	<p><<SIMPLE 165>> Dual value of constraint XX is assumed zero Constraint is empty or model is not solved.</p> <p><<SIMPLE 165>>制約式 XX の双対変数は 0 として出力されます. 一度も求解を行っていません.</p>	<p>求解していない状態では,Constraint の双対変数値は 0 として出力されます (警告です).</p>
167	<p><<SIMPLE 167>> Leftmost part of Constraint XX is used for</p>	<p>制約式 XX が複数回代入された場合や二つ以上に分</p>

	<p>output. Contains more than two constraints.</p> <p><<SIMPLE 167>> 制約式 xx については最も左の部分が出力されます . 二つ以上の制約式を含んでいます .</p>	<p>解される様な定義 ($co = x[i] \leq y[i] \leq z[i]$ 等) を行った場合 , その参照値の出力をすると最も最初の 代入の内容かもっとも左の式に対する参照値が出力 されます (8.3 オブジェクトに関連する値の参照を 参照して下さい .)</p>
168	<p><<SIMPLE 168>> Objective can only be assigned for once.</p> <p><<SIMPLE 168>> 目的関数 (Objective) に対する代入は一度のみ可能です .</p>	<p>目的関数への代入を複数回行おうとした . (Objective には1回の代入のみが許されています)</p>
169	<p><<SIMPLE 169>> Argument: type Error!</p> <p><<SIMPLE 169>> "type=" が誤って使われました .</p>	<p>オブジェクト定義時の属性引数 type の指定が無効 である .</p>
171	<p><<SIMPLE 171>> Invalid assignment to Objective:</p> <p><<SIMPLE 171>> 目的関数 (Objective) に対する無効な代入が行われまし た: "Objective = Expression" のみ有効です .</p>	<p>目的関数に対して変数を含まない式が代入された .</p>
172	<p><<SIMPLE 172>> Objective has not been assigned.</p> <p><<SIMPLE 172>> 目的関数 (Objective) に対する代入が行なわれていま せん .</p>	<p>モデルを解く (solve() 関数) に未定義の (代入が 行われていない) 目的関数を渡しました .</p>
173	<p><<SIMPLE 173>> dual member only exists in Constraints and Variables.</p> <p><<SIMPLE 173>> 双対変数値を制約式と変数以外について参照しようとし ました .</p>	<p>制約式と変数以外のオブジェクトの dual 値を照会 しました .</p>
174	<p><<SIMPLE 174>> Set data can not be transformed to Parameterby .val .</p> <p><<SIMPLE 174>> 集合の値 (.val) をパラメータ (Parameter) に変換し ようとした .</p>	<p>集合の現状値 val を定数 Parameter に変換し ようとした . (例外として集合の現状値のみは Parameter と等価に扱うことはできません . 表示 したりダンプするのみです .)</p>
177	<p><<SIMPLE 177>> No lower bound for empty constraint.</p>	<p>宣言したのみで定義されていない制約式に対して下</p>

	<<SIMPLE 177>> 制約式の下限は存在しません：制約式が空です。	限值を調べようとした。
181	<<SIMPLE 181>> Argument: from is required in defining a Sequence. <<SIMPLE 181>> Sequence を定義する時には, "from="指定が必須です。	列 Sequence の定義に属性引数 from が現れていません。
182	<<SIMPLE 182>> Argument: to is required in defining a Sequence. <<SIMPLE 182>> Sequence を定義する時には, "to="指定が必須です。	列 Sequence の定義に属性引数 to が現れていません。
183	<<SIMPLE 183>> Argument: Either left or oleft is required in defining an Interval. <<SIMPLE 183>> Interval を定義する時には, "left=" か"oleft=" が必須です。	範囲 Interval の定義に属性引数 left または oleft の指定が現れていません。
184	<<SIMPLE 184>> Argument: Either right or oright is required in defining an Interval. <<SIMPLE 184>> Interval を定義する時には, "right=" か"oright=" が必須です。	範囲 Interval の定義に属性引数 right または oright の指定が現れていません。
185	<<SIMPLE 185>> Argument: name is ignored in the Interval/Sequence definition. <<SIMPLE 185>> Interval/Sequence を定義する時には, "name=" は無効です。	範囲 Interval または列 Sequence を定義するとき, 属性引数 name が現れました。(警告です。無視して実行します。)
186	<<SIMPLE 186>> Argument: index is ignored in the Interval/Sequence definition. <<SIMPLE 186>> Interval/Sequence を定義する時に, "index=" は無効です。	範囲 Interval または列 Sequence の定義に属性引数 index が現れました。(警告です。無視して実行します。)

187	<p><<SIMPLE 187>> Argument: dim is ignored in the Interval/Sequence definition.</p> <p><<SIMPLE 187>> Interval/Sequence を定義する時に, "dim=" は無効です.</p>	範囲 Interval または列 Sequence の定義に属性引数 dim が現れました. (警告です. 無視して実行します.)
188	<p><<SIMPLE 188>> Argument: left, right, oleft, oright are ignored in</p> <p><<SIMPLE 188>> Sequence を定義する時には, "left=, right=, oleft=, oright="は無効です.</p>	列 Sequence を定義するとき, 属性引数 left などは無視されます.
189	<p><<SIMPLE 189>> Argument: from, to are ignored in the Interval definition.</p> <p><<SIMPLE 189>> Interval を定義する時に, "from=, to=" は無効です.</p>	範囲 Interval を定義するとき, 属性引数 from と to は無視されます.
190	<p><<SIMPLE 190>> Interval has no val member.</p> <p><<SIMPLE 190>> Interval の".val" を参照しようとしてしました.</p>	範囲 Interval に対して現状値 val を照会しました. (Interval に val を参照することはできません.)
191	<p><<SIMPLE 191>> No assignment is allowed to Sets having</p> <p><<SIMPLE 191>> 添字と superSet を同時に持つような集合に対する直接の代入はできません. (データファイル経由のみが許されます)</p>	添字付きで親集合を持つ集合に対する代入を行おうとしました. (実装の都合上, このケースでの直接の代入は禁止されており, 自動追加のみが許されます .6.1.8 集合同士の包含関係を参照して下さい.)
192	<p><<SIMPLE 192>> Fatal Error in solve() before solution process</p> <p><<SIMPLE 192>> NUOPT の重大エラー (求解処理の前)</p>	NUOPT が前処理でエラーを起こしました.
193,194	<p><<SIMPLE 193>>, <<SIMPLE 194>></p> <p>Error in solve(), Fatal Error in solve()</p> <p>NUOPT のエラー:XX, NUOPT の重大エラー:XX</p>	NUOPT が計算途中にエラーを起こしました (193:解出力あり, 194:解出力なし).
195	<p><<SIMPLE 195>> Index with SuperSet error.</p>	Superset と添字つき集合の添え字付けに矛盾があ

	<<SIMPLE 195>> Index と SuperSet の定義エラー.	ります .
196	<<SIMPLE 196>>Warning: Objective function is constant. <<SIMPLE 196>> 警告: 目的関数がコンスタントとなっています.	目的関数が定数であることがモデルの解釈によって明らかになりました .
197	<<SIMPLE 197>> Set of fixed element can not be a superset. <<SIMPLE 197>>固定された要素によって定義される集合を superSet として使用することはできません .	代入によって固定した要素を superSet として使おうとしました .
198	<<SIMPLE 198>>Wrong argument for slice function. <<SIMPLE 198>> 関数 slice の引数が正しくありません .	Slice 関数の引数は slice しようとしている集合の次元数以下である必要がありますが , それを超えています .
199	<<SIMPLE 199>> Character-value (..) appeared in constraint/objective definition. <<SIMPLE 199>> 文字列値(..) が制約式や目的関数の定義に現れています .	文字列 .. に対する演算が制約式や目的関数の定義中に現れました . Parameter の値で不適切な個所に文字列が現れている可能性があります .
200	<<SIMPLE 200>> 警告: simple_fprintf() は引数並び中の Set オブジェクトを無視しました	simple_printf は Set の出力を行いません .
202	<<SIMPLE 202>> {...} appears invalid position. <<SIMPLE 202>> {...} が不正な場所に現れています .	データファイルや文字列中で自然数の連続を示す...の現れる場所が不正です .
203	<<SIMPLE 203>>Insufficient # of Data after {...}..{...} expected <<SIMPLE 203>> {...}..{...} にひきつづくデータ個数が不正です . xx 個必要ですが yy 個あります .	データファイルの省略形”{...}”において , 期待されるデータの個数が異なります .
204	<<SIMPLE 204>>Try to unlock Set with noname. <<SIMPLE 204>> 名前なしの集合を unlock() しようとした .	集合演算の結果など , 陽に宣言されていない集合に対して unlock() を呼びました .
205	<<SIMPLE 205>>Warning: Any Set without name is already locked. <<SIMPLE 205>> 警告: 名前なしの集合は常に lock() された状態ですの	集合演算の結果など , 陽に宣言されていない集合に対して lock() を呼びました . もともと lock() されて

	で lock() のコールは不要です .	いるという仕様なので lock() のコールは不要です .
206	<<SIMPLE 206>> Locked Set XX cannot be assigned. <<SIMPLE 206>> lock() された集合 XX に代入が行われました .	集合 XX を lock() によってロックしているのに , 代入が行われようとしていました .
207	<<SIMPLE 207>> Auto-assignment mechanism try to add some element[s] to locked Set XX. In setting object YY. <<SIMPLE 207>> データ YY の設定の際に、自動代入で lock されている集合 XX に要素が追加されようとしていました .	集合 XX を lock() によってロックしている際に , 自動代入によって新しい要素が追加されようとしています . YY へのデータ設定の添字に問題があります .
208	<<SIMPLE 208>> Note: Skip auto-assignment to locked base Set in assignment to XX <<SIMPLE 208>> 警告: XX の添字集合(lock 済)への自動代入は抑制されました .	集合 XX を lock() によってロックしている際に , 自動代入によって新しい要素が追加されようとしています (設定によって自動代入がチェックのみのモードになっている場合の警告出力) .
209	<<SIMPLE 209>> Note: Skip assignment to locked superSet XX <<SIMPLE 209>> 警告: XX の superSet(lock 済) への代入は抑制されました .	集合 XX を lock() によってロックしているのに , 代入が行われようとしていました (設定によって自動代入がチェックのみのモードになっている場合の警告出力) .
211	<<SIMPLE 211>> User Termination(at Data Input) <<SIMPLE 211>> ユーザによる中断 (データ読み込み)	データの読み込みの際にユーザによる中止が命令されました .
212	<<SIMPLE 212>> User Termination(at Model Expansion) <<SIMPLE 212>> ユーザによる中断 (モデル展開)	式の展開の途中にユーザによる中止が命令されました .
213	<<SIMPLE 213>> Warning from solve():XX <<SIMPLE 213>> NUOPT より警告:XX	NUOPT より警告 XX が出ました .
214	<<SIMPLE 214>> Warning constraint# XX reduce to YY (always satisfied) <<SIMPLE 214>>制約式 XX は以下の式に等価です YY (常に満たされる) .	式 XX の展開の結果 , YY という形の常に満たされる制約式が現れました .

215	<<SIMPLE 215>> constraint# XX reduce to YY (never satisfied) <<SIMPLE 215>>制約式 XX は以下の式に等価です YY (常に満たされない)	制約式 (番号: XX) は「YY」に等価で、決して満たされないことがわかりました。216 と同時に現れます。
216	<<SIMPLE 216>> Trivial and Infeasible constraint appeared. <<SIMPLE 216>> 常に Infeasible な制約式が現れました。	式 XX の展開の結果、YY という形の明らかに満たすことのできない制約式が現れました (式の解釈の結果、問題が実行不可能であることがわかりました)。215 と同時に現れます。
217	<<SIMPLE 217>> Internal Error XX <<SIMPLE 217>> 内部エラー XX	内部エラー XX です (nuopt-support@msi.co.jp へお知らせ下さい)。
218	<<SIMPLE 218>> Error in CSV file XX. The header column have N fields but line M has P fields. <<SIMPLE 218>> = CSV ファイル XX のエラーです。ヘッダー行には N 個のフィールドがありますが、M 番目の行は p 個のフィールドがあります。	データファイルとして与えられた CSV ファイル XX のフィールド数エラーです。 CSV ファイルのフィールド数はすべての行で同一でなければなりません。
219	<<SIMPLE 219>> Error in CSV file XX. Only the header row and no data row exist. <<SIMPLE 219>>CSV ファイル XX のエラーです。ヘッダー行しかありません。	データファイルとして与えられた CSV ファイルにはヘッダー行しかありません。
220	<<SIMPLE 220>> Error in CSV file XX.No data row exist. <<SIMPLE 220>> CSV ファイル XX のエラーです。有効行がまったくありません。	データファイルとして与えられた CSV ファイルにはデータとして解釈できる部分がまったくありません。
221	<<SIMPLE 221>> Error in CSV file XX. Doublicate name YY (field # N, and M) in header row. <<SIMPLE 221>> CSV ファイル XX のエラーです。名前 YY がヘッダー行で重複しています。(フィールド番号 N と M に現れています)	データファイルとして与えられた CSV ファイルのヘッダーが示す行名前は重複してはいけません。
222	<<SIMPLE 222>> Error in CSV file XX. Invalid field string XX	データファイルとして与えられた CSV ファイルに違

	<p>at line# N .</p> <p><<SIMPLE 222>> CSV ファイル XX のエラーです . フィールドデータとして不適切な文字 XX が N 行目に現れています .</p>	<p>法な文字が混ざっています .</p>
223	<p><<SIMPLE 223>> Error in CSV file XX. Empty field at line# N, field# M.</p> <p><<SIMPLE 223>> CSV ファイル XX のエラーです . N 行目の, フィールド M が空です .</p>	<p>データファイルとして与えられた CSV ファイルに空のフィールドが混ざっています .</p>
224	<p><<SIMPLE 224>> In reading scalar YY from CSV file XX, found too many N lines or scalar.</p> <p><<SIMPLE 224>> CSV ファイル XX からスカラデータ YY を読もうとしましたが, このファイルには無駄な行があります合計 (N) 行です .</p>	<p>データファイルとして与えられた CSV ファイルからスカラを与えようとしている場合には, 行はヘッダ行以外は一行でなければなりません, それ以上の行があります .</p>
225	<p><<SIMPLE 225>> Try to read data YY (with N index), from CSV file XX but it has too few (M) preceding column(s).</p> <p><<SIMPLE 225>> データ YY (添字の数 N) を CSV ファイル XX から読もうとしていますが該当列の前に添字となるはずの列が (N) 列しかありません .</p>	<p>データファイルとして与えられた CSV ファイルの添え字の数が足りません (添え字の数は読み込もうとしているオブジェクトの定義から判定されますが, それから考えて足りません).</p>
226	<p><<SIMPLE 226>> Try to read data YY (with N index, [1 or 2]D format) and ZZ (with M index, [1 or 2] D format) from the same CSV file XX (This file looks ambiguous)</p> <p><<SIMPLE 226>> データ YY (添字の数 N, [1 or 2]D 書式) と ZZ (添字の数 M, [1 or 2]D 書式) が両方, 同じ CSV file XX から読みとれてしまいます (記述が曖昧です) .</p>	<p>データファイルとして与えられた CSV ファイルの記述が曖昧な場合のエラーです (1D 書式か 2D 書式か判然としません).</p>
227	<p><<SIMPLE 227>> Multiple data entry XX found in data YY [and ZZ] .</p> <p><<SIMPLE 227>> データ XX についての記述がファイル YY と ZZ に複数み</p>	<p>XX というオブジェクトの内容をデータファイル YY と ZZ において二回以上定義しました . 二回以上定義されているオブジェクトを発見するたびに現れます .</p>

	わかりました.	231 番のメッセージとともに現れます.
228	<p><<SIMPLE 228>> Field# NN of the first row (index) is empty. In reading data XX (with YY index, 2D format) from CSV file ZZ.</p> <p><<SIMPLE 228>> データ XX (添え字の数 YY) を CSV file ZZ から読み込もうとしましたが, 最初の行のフィールド NN (添え字として使われる) が空です.</p>	XX というオブジェクトを 2D 書式で呼んでいるときに, 最初の行には添え字の並びが来なければいけません, フィールド NN が空になっています.
229	<p><<SIMPLE 229>> Error from asDouble(): this object is not scalar.</p> <p><<SIMPLE 229>> asDouble() のコールを行ったオブジェクトはスカラではありません.</p>	スカラーでないオブジェクトに asDouble() (doubl 値への変換) のコールを行いました.
230	<p><<SIMPLE 230>> Error in datafile FF at line NN In the RHS of xx, dimension of element [YY] should be same as others.</p> <p><<SIMPLE 230>> データファイル FF の NN 行目の記述エラーです. オブジェクト XX の右辺に現れる添字 YY の次元が他と異なります.</p>	添字付きオブジェクト XX に対するデータの並びで, YY という添字記述のみ次元が異なります. a = [1] 3.0 [<u>2</u> 3] 4.0 [5] 5.0 ; (モデル内の定義文字列に発見された場合にもこのエラーが出力されます).
231	<p><<SIMPLE 231>> Multiple data definition found. This may cause severe performance deterioration.</p> <p><<SIMPLE 231>> 同一のデータ定義が二つ以上あり, 大規模データでは深刻なパフォーマンス下落を招く可能性があります.</p>	データの二重定義が一つでもあると現れます. options.multDataPolicy を 0 (デフォルト) ならばエラーになります. 0 以外に設定すると警告の意味となり, 実行は停止しません.
232	<p><<SIMPLE 232>> Proxy object is used before set Can be used only after declaration.</p> <p><<SIMPLE 232>> オブジェクトが宣言前に使われています. オブジェクトは宣言した後に利用しなければなりません.</p>	宣言する前のオブジェクト (Parameter, Variable など) を式の定義に利用しました.

233	<<SIMPLE 233>> Floating point arithmetic error. <<SIMPLE 233>> 浮動小数点例外が発生しました	モデルの展開,最適化計算の実行時に浮動小数点エラーが発生しました(発生箇所を伝える付加的なメッセージとともに表示されます).
234	<<SIMPLE 234>> 値 is not in the domain of DiscreteVariable. The domain: {DiscreteVariable の domain の集合} <<SIMPLE 234>> 値は DiscreteVariable の値として不適切です(以下のいずれかである必要があります):{ DiscreteVariable の domain の集合}	DiscreteVariable とその domain に含まれない値の間の条件式を定義した場合に出力されます. 例えば x の domain が {a,b,c} のとき, Boolean(x == "d").. のように書いた場合.
235	<<SIMPLE 235>> DiscreteVariable 変数名 's domain is invalid (should contain only positive integer or string). domain:{ DiscreteVariable の domain の集合} <<SIMPLE235>> DiscreteVariable 変数名の domain が不適切です. (文字列または非負の整数を要素とする集合である必要があります).domain の内容:{ DiscreteVariable の domain の集合}	DiscreteVariable の domain の値は非負の整数か文字列である必要があります.
236	<<SIMPLE 236>> DiscreteVariable 変 数 名 's domain has no element . <<SIMPLE 236>> DiscreteVariable 変数名の domain が空集合です.	
237	<<SIMPLE 237>> = is inappropriately used. to define equality constraint, use == instead of = . <<SIMPLE 237>> = が不適切に使われています.おそらく == の誤りです. 等式制約を定義するには=(代入)ではなく==を用います.	$x+y = z;$ のように等式制約の定義に=を誤って用いた場合に出力されます.
238	<<SIMPLE 238>> = is inappropriately used in definition of Set . <<SIMPLE 238>> 集合演算において = が不適切に使われています.	
239	<<SIMPLE 239>> DiscreteVariable 変数名 has no doomain.	

	<<SIMPLE 239>> DiscreteVariable 変数名 の domain が定義されていません .	
240	<<SIMPLE 240>> DiscreteVariable 変数名's domain should not be indexed. <<SIMPLE 240>> DiscreteVariable 変数名 の domain が添え字付けられています .	
241	<<SIMPLE 241>>Table/Parameter Table 名 が 4 つ 以上 の DiscreteVariable で添字付けられています .	
242	<<SIMPLE 242>> ResourceRequire “名前” の引数 duration は整数の集合でなければなりません .	
243	<<SIMPLE 243>> 制約 “名前” は rcpsp では扱う事は出来ません .	r cpsp では扱えない制約 alldiff, valgroun 等が定義された場合に出力されます .
244	<<SIMPLE 244>> rcpsp を適用するのに必要な ResourceRequire が定義されていません .	
245	<<SIMPLE 245>> rcpsp を適用するのに必要な ResourceCapacity が定義されていません	
246	<<SIMPLE 246>> 異なる ResourceCapacity の引数 “timeStep” に与えられている集合が同一ではありません .	
247	<<SIMPLE 247>> ResourceRequire で定義されている資源 “名前” が ResourceCapacity に定義されていません .	
248	<<SIMPLE 248>> rcpsp を適用するのに必要な Activity が定義されていません .	

249	<<SIMPLE 249>> 定義されていない資源 “ 名前 ” が直前先行制約で使われています.	
250	<<SIMPLE 250>> Boolean で指定しているモード "名前" は定義されていません.	
251	<<SIMPLE 251>> rcpsp では一般の制約式において Boolean 同士の積は記述できません.	
252	<<SIMPLE 252>>一般の制約式の Boolean と startTime, endTime, processTime の積の Activity が異なります	
253	<<SIMPLE 253>>先行制約において同じ Activity に対し先行関係が与えられています.	
255	<<SIMPLE 255>> 一般 の 制 約 式 に Activity は 用 い ら れ ま せ ん.startTime, endTime, processTime に対して記述して下さい	
258	<<SIMPLE 258>> ResourceRequire で設定されていないモード"名前"が Activity に与えられています.	
259	<<SIMPLE 259>> sourceActivity は Activity を定義した時のみ使用出来ます.	
260	<<SIMPLE 260>> sinkActivity は Activity を定義した時のみ使用出来ます	
261	<<SIMPLE 261>> Activity “名前” の index と引数 mode “名前” の index とが異なります.	
262	<<SIMPLE 262>> Activity "名前" の 引数に mode が設定されていません.	

263	<<SIMPLE 263>> Activity “名前”の引数 duedate “名前” の index が異なります.	
264	<<SIMPLE 264>>先行制約の 2 つの Activity 間で index が異なります.	
265	<<SIMPLE 265>>警告: 先行制約に重みを設定する事は出来ません. 常に hard 制約として扱われます	
266	<<SIMPLE 266>>警告: 直前先行制約に重みを設定する事は出来ません. 常に hard 制約として扱われます	
267	<<SIMPLE 267>>先行制約の時間指定に文字列が使われています	
268	<<SIMPLE 268>>直前先行制約の index と指定された資源の index が異なります.	
269	<<SIMPLE 269>>直前先行制約に資源が指定されていません	
270	<<SIMPLE 270>>先行制約の index と時間指定の index が異なります	
271	<<SIMPLE 271>>先行制約の添字に他の添字に依存するものがありますが他の添字と同時に使われていません.	
272	<<SIMPLE 272>> 1 つの直前先行制約に複数の資源が指定されています	
273	<<SIMPLE 273>> ResourceCapacity “名前” の引数に resource が与えられていません	

274	<<SIMPLE 274>> ResourceCapacity "名前" の引数に timeStep が与えられていません	
275	<<SIMPLE 275>> ResourceCapacity "名前" の index と 引数 weight "名前" の index が異なります	
276	<<SIMPLE 276>> ResourceRequire "名前" の引数に mode が与えられていません	
277	<<SIMPLE 277>> ResourceRequire "名前" の引数に resource が与えられていません	
278	<<SIMPLE 278>> ResourceRequire "名前" の引数に duration が与えられていません	
279	<<SIMPLE 279>> ResourceRequire "名前" の 引数 duration に負の値が用いられています	
280	<<SIMPLE 280>> ResourceRequire にデータが設定されていません	
281	<<SIMPLE 281>>警告: モード "名前" が ResourceRequire に設定されていません	予期せぬ動作の原因になります .
282	<<SIMPLE 282>>警告: モード "名前" が ResourceRequire に定義されていますが使われていません .	予期せぬ動作の原因になります .
283	<<SIMPLE 283>>警告: 資源 "名前" が全スケジュール期間において ResourceCapacity の値が 0 となっています	初期解の失敗の原因になります .
284	<<SIMPLE 284>> rcpsp は一般の制約式を hard 制約として扱えません (十分大きな重みを与える必要があります)	

285	<<SIMPLE 285>>一般の制約式に負の重みが与えられています	
286	<<SIMPLE 286>>警告：納期最小化では一般の制約式に重みを与える事は出来ません (全て hard 制約として扱われます)	
288	<<SIMPLE 288>>目的関数 (最後の作業の完了時刻最小化) の重みに負の値が与えられています	
289	<<SIMPLE 289>>目的関数 (最後の作業の完了時刻最小化) は hard 制約として扱う事は出来ません	
290	<<SIMPLE 290>>警告：納期最小化に対して重みを設定する事は出来ません	
291	<<SIMPLE 291>>再生資源制約の重みは 0 より大きい値のみ与えられます (-1 は hard 制約とみなされます)	
292	<<SIMPLE 292>>警告：納期最小化時は資源制約に重みを設定する事は出来ません . 全て hard 制約として扱われます	
293	<<SIMPLE 293>>納期最小化時の一般の制約式は異なる 2 つの Activity の先行関係か Boolean のみからなる制約以外の制約は扱えません	
294	<<SIMPLE 294>>警告：納期最小化時は等式制約を扱う事は出来ません . 不等式制約を連立させます	
295	<<SIMPLE 295>>納期最小化時は直前先行制約は定義出来ません	
296	<<SIMPLE 296>>納期最小化時は一般の制約式で Boolean と Activity を混合する事は出来ません	

297	<<SIMPLE 297>> Gantt の dump に与えられたファイル "ファイル名" をオープン出来ません.	
298	<<SIMPLE 298>> rcpsp のオブジェクト (Activity, ResourceRequire, ResourceCapacity) に添字が与えられていないものがあります.	
299	<<SIMPLE 299>>警告: ResourceCapacity で定義されている資源 "名前" が ResourceRequire で使われていません	
300	<<SIMPLE 300>> ResourceRequire "名前" の引数 ¥"timeStep¥" に与える集合は 1 次元でなければなりません.	
301	<<SIMPLE 301>> ResourceRequire "名前" の引数 "timeStep" に与える集合の要素は整数でなければなりません	
302	<<SIMPLE 302>> ResourceRequire "名前" の引数 "timeStep" に与える集合の要素は 0 始まりでなければなりません	
303	<<SIMPLE 303>> ResourceRequire "名前" の引数 "timeStep" に与える集合の要素は 1 刻みでなければなりません	
304	<<SIMPLE 304>>警告: ResourceCapacity "名前" 値が実数です. 整数に切り捨てます	
305	<<SIMPLE 305>>警告: ResourceCapacity "名前" に与えられた重みに実数のものがあります. 整数に切り捨てます	
306	<<SIMPLE 306>>警告: ResourceRequire "名前" に与えられた重みに実数のものがあります. 整数に切り捨てます	
307	<<SIMPLE 307>>警告: 一般の制約式 "名前" に与えられた重みに実数のものがあります. 整数に切り捨てます	

308	<<SIMPLE 308>>警告：目的関数（最後の作業の完了時刻最小化）に与えられた重みに実数のものがあります．整数に切り捨てます	
310	<<SIMPLE 310>> rcpsp において目的関数が複数定義されています	
311	<<SIMPLE 311>> rcpsp において Objective に type=maximize が与えられています	目的関数は最小化のみ扱えます．
312	<<SIMPLE 312>>警告：rcpps において目的関数も一般の制約式も定義されていません	
313	<<SIMPLE 313>> Activity "名前" の引数 mode に与えられた集合に空のものがあります	
314	<<SIMPLE 314>>直前先行制約において条件式の前に資源を指定する事は出来ません	
315	<<SIMPLE 315>> tardiness は Activity を定義した時のみ使用出来ます	
316	<<SIMPLE 316>> ResourceRequire "名前" の引数 default に負の値が与えられています	
317	<<SIMPLE 317>> completionTime, tardiness 以外は Objective に設定する事は出来ません	rcpsp を用いる場合のみ

A.2 NUOPT のエラー/警告メッセージ

NUOPT 本体の出力するエラー/警告メッセージは、

- ◆ NUOPT 本体部の計算で検出されたエラー/警告

<<NUOPT 番号>> エラー/警告メッセージ

- ◆ パラメータに関するエラー/警告

<<PARAMETER 番号>> エラーメッセージ

- ◆ MPS ファイル解釈に検出されたエラー/警告

<<MPS FILE 番号>> エラーメッセージ

の3種類です。エラーメッセージは、標準出力 (WindowsGUI では結果表示ウィンドウとメッセージウィンドウ) に出力されます。SIMPLE ロードモジュールの場合には

ex2.smp の 5 行目の実行中にエラーが起きました。 <<SIMPLE 193>> NUOPT のエラー: <<NUOPT 10>> IPM iteration limit exceeded..

などのように、SIMPLE のエラーメッセージの一部として表示されますが、これはモデリング言語 SIMPLE が NUOPT を起動している形を取っているためです。

A.2.1 NUOPT のエラー/警告

エラーに関する解説の最後に「解出力なし」とあるエラーの場合には、解ファイルへの変数値や関数値に関する出力は行われません。「解出力あり」とあるエラーの場合には最適性の保証がない解を一応は出力します。

エラー番号	エラーメッセージ	説明
1	<<NUOPT 1>> memory error in preprocessing.	前処理部で所要メモリが、使用可能な量をオーバーしました。「解出力なし」
2	<<NUOPT 2>> infeasible (linear constraints and variable bounds).	線形制約や変数の上下限制約のために問題が infeasible となっていることが前処理部で検出されました。「解出力なし」
3	<<NUOPT 3>> no variables in this model.	モデル(問題)に変数がありません。「解出力なし」
5	<<NUOPT 5>> infeasible integer variable bounds.	変数の上下限制約のために問題が infeasible となっていることが前処理部で検出されました。(整数変数が整数性に違反する様な上下限を課されている場合等に発生します。)[解出力なし]
6	<<NUOPT 6>> unbounded (linear constraints and variable bounds).	線形制約と変数の上下限から問題の最適解は有界とはならないことが前処理部で判定されました。「解出力なし」
7	<<NUOPT 7>> internal error. [内部ルーチン名]	内部エラーが発生しました。 (nuopt-support@msi.co.jp にご連絡下さい)。「解出力なし」
8	<<NUOPT 8>> memory error in optimization phase.	計算部において所要メモリが使用可能な量をオーバーしました。「解出力なし」
9	<<NUOPT 9>> step reduction limit exceeded.	直線探索アルゴリズムにおいて step reduction の失敗が起き、最適化実行が止まってしまいました(凸でない問題に直線探索アルゴリズムを適用した場合や、問題が infeasible である場合に起きます)。「解出力あり」

10	<<NUOPT 10>> IPM iteration limit exceeded.	内点法の反復回数が上限を越えました (上限は特に指定しない場合には 150 回です パラメータファイルからは criteria: maxitn=300 として設定することができます) . [解出力あり]
11	<<NUOPT 11>> infeasible.	問題が実行不可能であると判定されました . [解出力あり]
13	<<NUOPT 13>> unbounded.	問題の最適解が有界でないことが判定されました . [解出力あり]
14	<<NUOPT 14>> integrality is violated.	整数変数を含む問題に単体法以外を適用したため、整数変数が整数となっていない解が出力されています . [解出力あり]
15	<<NUOPT 15>> simplex misapplied to NLP.	非線形計画問題に単体法が適用されようとしています . [解出力なし]
16	<<NUOPT 16>> Infeasible MIP.	混合整数計画問題に整数解が存在しないことがわかりました (LP/QP 緩和解が出力されます) . [解出力あり]
17	<<NUOPT 17>> B & B node limit reached (feasible) .	分枝限定法において生成する部分問題数が上限を越えました . 最適解である保証はありませんが、整数解は得られています . (branch:maxnod を設定した場合) . [解出力あり]
18	<<NUOPT 18>> MIP iteration failed (feasible) .	分枝限定法のループが途中で数値的問題等によって失敗しました . 最適解である保証はありませんが、整数解は得られています . [解出力あり]
19	<<NUOPT 19>> B & B node limit reached (infeasible) .	17 番と同じですが、整数解が得られていません . LP (QP) 緩和解が出力されます . [解出力あり]
20	<<NUOPT 20>> MIP iter. failed (infeasible) .	18 番と同じですが、整数解が得られていません . LP (QP) 緩和解が出力されます . [解出力あり]
21	<<NUOPT 21>> B & B itr. timeout (feasible) .	整数計画法 を解いている場合の NUOPT の起動時間が上限を越えました . 最適である保証はありませんが、整数解は得られています (branch:maxnod を設定した場合) . [解出力あり]
22	<<NUOPT 22>> B & B itr. timeout (infeasible) .	21 番と同じですが、整数解が得られていません . LP/QP 緩和解が出

		力されます．[解出力あり]
25	<<NUOPT 25>> Can't open file in current directory [no .sol made]	解ファイル(.sol ファイル)のオープンに失敗したので解ファイルが出力されていません．(警告です．計算は行われます)．[解出力なし]
26	<<NUOPT 26>> LP/IP module cannot handle NLP.	LP/IP モジュールで非線形計画問題を解こうとしました．[解出力なし]
27	<<NUOPT 27>> SIMPLEX iteration limit exceeded.	単体法の反復回数の上限をオーバーしました．[解出力あり]
28	<<NUOPT 28>> higher-order method is only for LP.	非線形計画問題に線形計画法専用の内点法が適用されようとしています．[解出力なし]
29	<<NUOPT 29>> iteration diverged.	線形計画法専用の内点法が発散しました．[解出力あり]
30	<<NUOPT 30>> terminated by user.	ユーザの指示により計算の中断を行いました．[解出力あり]
31	<<NUOPT 31>> B&B terminated by user (feasible).	分枝限定法の演算中ユーザの指示により計算の中断を行いました．最適解である保証はありませんが，整数解は得られています．[解出力あり]
32	<<NUOPT 32>> B&B terminated by user (infeasible).	31 番と同じですが，整数解が得られていません．LP/QP 緩和解が出力されます．[解出力あり]
33	<<NUOPT 33>> Bound violated.	変数の上下限を有意に違反している解が出力されています．違反している個所は，解ファイル(*.sol)で"INFS"と表示されています．スケーリングの悪い(制約式や変数の値のオーダーの差が激しい)問題と思われます．内点法で解いている場合には停止条件を小さくして実行する必要があります．[解出力あり]
35,36	<<NUOPT 35>> Constraint violated. <<NUOPT 36>> Equality Constraint violated.	制約式の上下限を有意に違反している解が出力されています．違反している個所は，解ファイル(*.sol)で"INFS"と表示されています．スケーリングの悪い(制約式や変数の値のオーダーの差が激しい)問題と思われます．内点法で解いている場合には停止条件を小さくして

		実行する必要があります。[解出力あり]
37	<<NUOPT 37>> B&B terminated with given # of feas.sol.	options.maxintsol で指定した数だけの整数解が発見されたので分枝限定法を停止しました。[解出力あり]
38	<<NUOPT 38>> dual infeasible.	リスタート時に起動される双対単体法のプロセスで実行不可能性が検出されました。[解出力あり]
39	<<NUOPT 39>> IPM iteration timeout	options.maxtim で設定した時間に対して、内点法の反復がタイムアウトしました。[解出力あり]
46	<<NUOPT 46>> Continuous Variable 変数名 cannot be included in model for wcsp.	連続変数を含む問題を wcsp で解こうとしました。wcsp は連続変数を扱うことができません。[解出力なし]
47	<<NUOPT 47>> IntegerVariable 変数名 should be declared as binary.	0-1 整数変数以外の整数変数が表れています。wcsp は一般の整数変数を扱うことができません。[解出力なし]
48	<<NUOPT 48>> Variable 変数名 appear in two selection() .	二つ以上の selection にまたがって現れている 0-1 整数変数が表れました。[解出力なし]
49	<<NUOPT 49>> Variable 変数名 is fixed to infeasible value.	0-1 整数変数が 0, 1 以外の値に固定されました。[解出力なし]
50	<<NUOPT 50>> Both of two variables 変数名 1 and 変数名 2 cannot be 1.	変数名 1 と変数名 2 の両方は 単一の selection に現れているので 両方 1 になることができません。[解出力なし]
51	<<NUOPT 51>> wcsp is not available without SIMPLE.	wcsp は SIMPLE によって定義された問題に対してのみ有効です。[解出力なし]
52	<<NUOPT 52>> 数字 th selection() statement has no movable binary variables.	対応する変数がすべて固定されているか存在しない selection() 文が「数字」番目に現れました。[解出力なし]
53	<<NUOPT 53>> You cannot use any method but “wcsp” for model with DiscreteVariable(s).	DiscreteVariable を含む問題に wcsp 以外の方法は適用できません。[解出力なし]
54	<<NUOPT 54>> Constraint 制約式名's weight is 値	重みとしては-1 もしくは非負の数を与えねばなりません。[解出力なし]

	should be -1 or non-negative value.	し]
55	<<NUOPT 55>> exterior solution obtained.	外点法 (lepm/tepm) が不等式制約を違反する解を出力しました。パラメータ exrho (デフォルトは 1.0e4) を大きく設定すると消える可能性があります, そうしてもこのエラーが出る場合には, 問題が実行不可能である可能性があります。[解出力あり]
56	<<NUOPT 56>> Method "global" not installed.	有償のアドイン (global) がインストールされていない状態で, 最適化手法として "global" を指定しました。
57	<<NUOPT 57>> You cannot use any method but "rcpsp" for model with Activity."	Activity の定義があるにも関わらず, rcpsp 以外のメソッドを適用しようとしてしました。[解出力なし]
58	<<NUOPT 58>> You cannot use any method but "rcpsp" for model with ResourceRequire.	ResourceRequire の定義があるにも関わらず, rcpsp 以外のメソッドを適用しようとしてしました。[解出力なし]
59	<<NUOPT 59>> You cannot use any method but "rcpsp" for model with ResourceCapacity.	ResourceCapacity の定義があるにも関わらず, rcpsp 以外のメソッドを適用しようとしてしました。[解出力なし]
60	<<NUOPT 60>> You cannot use any method but "rcpsp" for model with tardiness and completionTime.	目的関数が tardiness/completionTime に設定されているのにも関わらず, rcpsp 以外のメソッドを適用しようとしてしました。[解出力なし]
61	<<NUOPT 61>> You cannot use any method but "rcpsp" for model with sourceActivity	sourceActivity の定義があるにも関わらず, rcpsp 以外のメソッドを適用しようとしてしました。[解出力なし]
62	<<NUOPT 62>> You cannot use any method but "rcpsp" for model with sinkActivity	sinkActivity の定義があるにも関わらず, rcpsp 以外のメソッドを適用しようとしてしました。[解出力なし]
63	<<NUOPT 63>> You cannot use Variable for "rcpsp"	rcpsp は Variable は用いる事が出来ません。[解出力なし]
64	<<NUOPT 64>> You cannot use IntegerVariable for "rcpsp"	rcpsp は IntegerVariable は用いる事が出来ません。[解出力なし]
65	<<NUOPT 65>> failed to genelating initial	rcpsp において初期解生成に失敗しました。[解出力なし]

	solution	
66	<<NUOPT 66>> can't find feasible solution.	Rcspsp において実行可能解を得る事が出来ませんでした。[解出力なし]

A.2.2 パラメータのエラー

nuopt.prm または options を用いた NUOPT のパラメータ設定のエラーです。

エラー番号	エラーメッセージ	説明
1	<<PARAMETER 1>> Syntax error in parameter file.	パラメータファイルの記述にエラーが検出されました。
2	<<PARAMETER 2>> Parameter file is empty.	パラメータファイルが全く空になっています。
3	<<PARAMETER 3>> Invalid 型名 value 値 for parameter パラメータ名.	パラメータ名に対して不適切な値が設定されています (パラメータファイル以外からパラメータ値を与える場合のみに発生します)。
4	<<PARAMETER 4>> Internal error [内部ルーチン名]	パラメータの解釈を行うルーチンにて内部エラーが発生しました (nuopt-support@msi.co.jp にご連絡下さい)。
5	<<PARAMETER 5>> Invalid option in nuopt options.	AMPL 対応ロードモジュールがパラメータを読み込む環境変数 nuopt_options に無効なパラメータ名が含まれていました。

パラメータファイルの記述にエラーが発生した場合 (エラー番号 1) にはパラメータファイル読み込み部から標準出力 (Windows では GUI のメッセージウインドウ) に補助的なメッセージが表示されますので、併せて参考にして下さい。

エラーのあるパラメータファイル：

```
begin
    maximize
    method:trust
    cliteria:eps=1.0e-8
```

標準出力 (WindowsGUI のメッセージボックス):

```
<reading parameter file: nuopt.prm >
begin
maximize
method:trust
cliteria:eps=1.0e-8

+error+ undefined command. check the command name.
      cliteria          ( 行名が違う)
+error+ last command must be end-command
      end                ( end で終わっていない)
---- input data error occurred ----
<< PARAMETER 1 >> Syntax error in parameter file.
```

A.2.3 MPS ファイルのエラー

エラー番号	エラーメッセージ	説明
-------	----------	----

1	<<MPS FILE 1>> Failed to open mps file: ファイル名.	MPS ファイルのオープンに失敗しました .
2	<<MPS FILE 2>> Undefined row name: 行名.	COLUMNS/RHS/RANGES セクションに未定義の行が現れました .
3	<<MPS FILE 3>> Internal error. [内部ルーチン名]	[内部ルーチン] で内部エラーが発生しました . (nuopt-support@msi.co.jp にご連絡下さい .)
4	<<MPS FILE 4>> Syntax error in セクション名 section.	「セクション名」の示すセクションで文法エラーが発生しました .
5	<<MPS FILE 5>> Too many 'INTORG' marker.	'INTORG' マーカー行と 'INTEND' マーカー行の対応が取れていません . ('INTORG' マーカー行が多すぎます .)
6	<<MPS FILE 6>> Too many 'INTEND' marker.	'INTORG' マーカー行と 'INTEND' マーカー行の対応が取れていません . ('INTEND' マーカー行が多すぎます .)
7	<<MPS FILE 7>> Unknown marker: フィールド 3 の内容	'INTORG', 'INTEND' 以外のマーカー行が現れました . (NUOPT の MPS ファイル解釈部は 'INTORG', 'INTEND' 以外のマーカー行を解釈しません .)
8	<<MPS FILE 8>> Undefined row: 行名 in HESSIAN section.	HESSIAN セクションの二次の項を付加する行名として , 定義されていないものが現れました .
9	<<MPS FILE 9>> Undefined column: 変数名 in HESSIAN section.	HESSIAN セクションの非零要素の場所を示す際の変数名として , 定義されていないものが現れました .
10	<<MPS FILE 10>> row: 行名 appeared more than once.	ROWS セクションに同一の行名が二度以上現れました .
11	<<MPS FILE 11>> Specified bound: BOUND データラベル not found.	パラメータファイルで指定された (mpsfile: bound = BOUND データラベル) ラベルを持つ BOUNDS データが MPS ファイルには存在しません .
12	<<MPS FILE 12>> Specified objective: 目的関数行名 not found.	パラメータファイルで指定された (mpsfile: objective = 目的関数行名) 名前を持つ目的関数行が MPS ファイルには存在しません .
13	<<MPS FILE 13>> Specified rhs: RHS データラベル not	パラメータファイルで指定された (mpsfile: rhs = RHS データラベル) ラベルを持つ RHS データが MPS ファイルには存在しません .

	found.	タラベル) ラベルを持つ RHS データが MPS ファイルには存在しません .
1 4	<<MPS FILE 14>> Range data: RANGE データラベル contains unsuitable row.	「RANGE データラベル」を持つ RANGE データが目的関数行に対しての指定を行っています .
1 5	<<MPS FILE 15>> Specified range data: RANGE データラベル not found.	パラメータファイルで指定された (mpsfile: range = RANGE データラベル) ラベルを持つ RANGE データが MPS ファイルには存在しません .
1 7	<<MPS FILE 17>> Undefined column name: 変数名 in INITIAL section.	INITIAL セクションに定義されていない変数名が現れました .
1 8	<<MPS FILE 18>> Bound spec. on column : 変数名 should appear earlier.	「変数名」に関する上下限設定の現れるのはより前でなければなりません . (警告です . 最初に発見されたものに対してのみこのメッセージが出力されます .)
1 9	<<MPS FILE 19>> 数字 column(s) appeared disorderly in BOUNDS section	BOUNDS section において「数字」個の変数の現れる順番が違法です . (警告です . エラー18 と組になって出力されます .)
2 0	<<MPS FILE 20>> 2-pass required for reading from stdin. Please try again.	MPS ファイルを標準入力から入力している際に , MPS ファイルの読み直しが必要となりました . (このメッセージが表示された際には読み込みに必要なメモリ所要量を設定し , ファイルに書き落としていますので , もう一度同じ MPS ファイルを入力すれば正常に動作します .)
2 1	<<MPS FILE 21>> Undefined column name: 変数名 in BOUNDS section.	BOUNDS section において定義されていない変数名が現れました .
2 2	<<MPS FILE 22>> Hessian is implicitly bound for nonexistent objective.	HESSIAN セクションでは暗黙のうちに (行名を指定せず) 目的関数に対して二次の項が設定されていますが , MPS ファイルには

		全く目的関数が存在しません .
2 3	<<MPS FILE 23>> Same bound spec. on column: 変数名 appeared more than once.	BOUNDS section で「変数名」に関して同一の制約指定が二度以上行われました .
2 4	<<MPS FILE 24>> Column : 変 数 名 has bound specification FX and other.	BOUNDS section で「変数名」に関して FX 制約と他の制約指定が同時に行われました .
2 5	<<MPS FILE 25>> Column : 変 数 名 has bound specification FR and other.	BOUNDS section で「変数名」に関して FR 制約と他の制約指定が同時に行われました .
2 6	<<MPS FILE 26>> Column : 変 数 名 has bound specification LO and MI.	BOUNDS section で「変数名」に関して LO 制約と MI 制約指定が同時に行われました .
2 7	<<MPS FILE 27>> Column : 変 数 名 has bound specification UP and PL.	BOUNDS section で「変数名」に関して UP 制約と PL 制約指定が同時に行われました .

付録 B NUOPT アルゴリズム概説

B.1 内点法

B.1.1 問題

次の最適化問題

$$\begin{array}{ll} \text{最小化} & f(x), \\ \text{条件} & g(x)=0, \end{array} \quad \begin{array}{l} x \in R^n, \\ x \geq 0 \end{array}$$

を考えます⁵⁹．ここで，変数 $x = (x_1, \dots, x_n)^t$ は n 次元ベクトルで，関数 f は目的関数です．また， $g: R^n \rightarrow R^m$ は m 次元のベクトル値関数です．この問題のラグランジュ関数を

$$L(x, y, z) = f(x) - y^t g(x) - z^t x$$

としたとき，Karush-Kuhn-Tucker (KKT) 条件（最適性の一次の必要条件）は次式で与えられます：

$$\begin{aligned} \nabla_x L(x, y, z) &= 0, \\ g(x) &= 0, \\ XZe &= 0, \quad x \geq 0, z \geq 0. \end{aligned}$$

ただし， $X = \text{diag}(x_1, \dots, x_n) \in R^n$, $Z = \text{diag}(z_1, \dots, z_n) \in R^n$, $e = (1, \dots, 1)^t \in R^n$ です．ここで，相補性条件 $XZe = 0$ を $XZe = \mu e$ ($\mu > 0$) で置き換えたものを修正 KKT 条件と呼びます．

NUOPT ではバリア - ペナルティ関数：

$$F(x, z) = f(x) - \mu \sum_{i=1}^n \log(x_i) + \rho \sum_{i=1}^m |g_i(x)| + \nu \left(x^T z - \mu \sum_{i=1}^n \log(x_i z_i) \right)$$

をメリット関数として採用します．ここで， $\mu > 0$ はバリアパラメータ， $\rho > 0$ はペナルティパラメータ， $\nu > 0$ は主双対項の度合いを表すパラメータです．

非線形最適化問題に対する内点法では，「修正 KKT 条件を満たす点を求めて，修正 KKT 条件を更新する」という作業を逐次行います．この際に，メリット関数 $F(x, z)$ の一次近似

$F_l(x, z)$ あるいは二次近似 $F_q(x, z)$ の変化量が重要な手がかりとなります⁶⁰．次項以降で示す

ように，修正 KKT 条件を求める方法は複数存在します（直線探索を利用する方法・信頼領域を利用する方法）．

この作業を通して，最終的に元の KKT 条件を満たす点を求めます．

⁵⁹ここでは，説明の便宜上このような形式を考えますが，NUOPT では制約関数に上下限が存在する場合，等式条件，変数に上下限が存在する場合などの一般形を扱うことができます．また，制約条件が存在しない問題も扱うことができます．

⁶⁰詳細は論文 [14] [15] [16] を参照．

B.1.2 直線探索を利用する方法

修正 KKT 条件に対するニュートン法は次のようになります。

$$\begin{bmatrix} G + X^{-1}Z & -A' \\ -A & 0 \end{bmatrix} \begin{bmatrix} \Delta x_N \\ \Delta y_N \end{bmatrix} = \begin{bmatrix} -r_L - X^{-1}r_C \\ r_E \end{bmatrix},$$

$$\Delta z_N = -X^{-1}Z\Delta x_N - X^{-1}r_C.$$

ここで, $(\Delta x_N, \Delta y_N, \Delta z_N)$ は各変数に対する探索方向ベクトルで, G はラグランジュ関数のヘッセ行列あるいはその近似行列です。このとき, ρ が十分大きく, G が半正定値行列である場合 $\Delta F_l(x, z, \Delta x_N, \Delta z_N) < 0$ である⁶¹ という性質が成り立ちます。

この性質を利用して, 直線探索を利用して大域的に収束するアルゴリズムを構築することができます。主双対変数に対するステップ幅 α は以下のように計算されます。まず, 次の三式から α の上限値 α_{\max} を求めます。

$$\alpha_{\max}^x = \min_i \left\{ \frac{-x_i}{(\Delta x_N)_i} \mid (\Delta x_N)_i < 0 \right\},$$

$$\alpha_{\max}^z = \min_i \left\{ \frac{-z_i}{(\Delta z_N)_i} \mid (\Delta z_N)_i < 0 \right\},$$

$$\alpha_{\max} = \min \{ \alpha_{\max}^x, \alpha_{\max}^z \}$$

次に,

$$\alpha = \bar{\alpha}\beta^l, \quad \bar{\alpha} = \min\{\gamma\alpha_{\max}, 1\}$$

と定義します。ここで, $\gamma \in (0, 1), \beta \in (0, 1)$ です。そして, l は

$$F(x + \bar{\alpha}\beta^l \Delta x_N, z + \bar{\alpha}\beta^l \Delta z_N) - F(x, z) \leq \varepsilon_0 \bar{\alpha}\beta^l \Delta F_l(x, z, \bar{\alpha}\beta^l \Delta x_N, \bar{\alpha}\beta^l \Delta z_N)$$

をみたす最小の正整数として定義されます⁶¹。 $\varepsilon_0 \in (0, 1)$ です。

このように, 各種パラメータを適切に設定すれば, メリット関数 $F(x, z)$ が確実に減少するような探索方向ベクトル $(\Delta x_N, \Delta y_N, \Delta z_N)$ 及びステップ幅 α を定める事ができます。ここからアルゴリズムの大域的収束性が保証されます。

ラグランジュ関数のヘッセ行列が非負定値となるのは

- ◆ 線形計画問題 (ヘッセ 行列は 0)
- ◆ 一般の凸計画問題

です。また, 一般の非線形計画問題ではヘッセ行列を準ニュートン法で近似することによって行列 G を常に正定値に保つことができます。従って, このような場合に直線探索を利用した手

⁶¹ この一連のステップ幅の設定ルールを Armijo's Rule と呼びます。

法を使用することができます．

4.2 最適化手法に関する設定での説明にある

- ◆ 直線探索法 (Line Search Method)
- ◆ 準ニュートン法 (Line Search with BFGS)

はここで解説されている手法です．

B.1.3 信頼領域を利用する方法

行列 G が非負定値でないとき直線探索法を利用することは困難です．そこで， G が不定であるときも利用できる方法として主双対変数に対する信頼領域法を採用します．このとき，探索方向ベクトル w ，サイズ $\delta > 0$ ，ステップ幅 α は次のような関係を満たします．

$$\|w\| \leq \delta,$$

$$\alpha \leq \min \left\{ \frac{\delta}{\|w\|}, \gamma \alpha_{\max} \right\}$$

α_{\max} の導出方法は「直線探索を利用する方法」同様です．信頼領域のサイズ調整は通常の方法で行います．

大域的収束性を得るために基準となる最急降下方向ベクトル $(\Delta x_{SD}, \Delta y_{SD}, \Delta z_{SD})$ を

$$\begin{bmatrix} D + X^{-1}Z & -A^t \\ -A & 0 \end{bmatrix} \begin{bmatrix} \Delta x_{SD} \\ \Delta y_{SD} \end{bmatrix} = \begin{bmatrix} -r_L - X^{-1}r_C \\ r_E \end{bmatrix},$$

$$\Delta z_{SD} = -X^{-1}Z\Delta x_{SD} - X^{-1}r_C,$$

によって定義します．ここで $D > 0$ は対角行列です． α^* を方向 Δ_{SD} に沿って信頼領域で与えら

れる区間内で，メリット関数変化量の二次近似 $\Delta F_q(x, z, \Delta x, \Delta z)$ を最小化するステップ幅とし

て定義します．

$$\alpha^* = \arg \min \left\{ \Delta F_q(x, z, \alpha \Delta x_{SD}, \alpha \Delta z_{SD}) \mid \|\alpha (\Delta x_{SD} + \Delta z_{SD})\| \leq \delta, \alpha \in [0, \bar{\alpha}] \right\}$$

$$\bar{\alpha} = \min \{1, \gamma \alpha_{\max}\}, \gamma \in (0, 1),$$

信頼領域のステップ幅 α は以下の条件をみたすように設定します．

$$\Delta F_q(x, z, \alpha \Delta x, \alpha \Delta z) \leq \frac{1}{2} \Delta F_q(x, z, \alpha^* \Delta x, \alpha^* \Delta z) < 0,$$

$$\|\Delta x_{Nk}\| \leq M \|\Delta x_{SDk}\|,$$

$$\|\Delta z_{Nk}\| \leq M \|\Delta z_{SDk}\|$$

「信頼領域を利用する方法」では探索方向ベクトル $(\Delta x, \Delta z)$ は

$$\begin{pmatrix} \Delta x \\ \Delta z \end{pmatrix} = \nu \begin{pmatrix} \Delta x_{SD} \\ \Delta z_{SD} \end{pmatrix} + (1 - \nu) \begin{pmatrix} \Delta x_N \\ \Delta z_N \end{pmatrix},$$

として計算されます．ここで，パラメータ $\nu \in [0, 1]$ は $\nu = 0, 0.1, 0.2, \dots, 0.9, 1.0$ の中で条件：

$$\Delta F_q(x, z, \alpha \Delta x, \alpha \Delta z) \leq \frac{1}{2} \Delta F_q(x, z, \alpha^* \Delta x, \alpha^* \Delta z) < 0$$

をみたす最小の数です．このように，「信頼領域を利用する方法」では探索方向ベクトルの設定に異なる二方向 $(\Delta x_{SD}, \Delta z_{SD})$ ， $(\Delta x_N, \Delta z_N)$ を利用します．この結果，大域的収束性が保証されます．

4.2 最適化手法に関する設定での説明にある

- ◆ 信頼領域法 (Trust Region Method)

がここで解説されている手法です．

B.1.4 線形計画問題専用内点法

問題が線形の場合，KKT 条件

$$\begin{aligned} \nabla_x L(x, y, z) &= 0, \\ g(x) &= 0, \\ XZe &= 0 \end{aligned} \tag{1}$$

の第 1 式，第 2 式は線形であり，非線形なのは第 3 式のみとなります．この方程式系に関するステップ幅 1 のニュートン法を適用すると線形な式の残差は原理的に零になり，非線形な第 3 式のみに

$$\Delta X_N \cdot \Delta Z_N e$$

なる形の残差が現れます (ΔX_N ， ΔZ_N はニュートン法のステップ方向を対角に並べた行列)．この式にこの残差が発生することを見越してニュートン法のステップ方向を修正する (高次方向 (Higher Order) の修正を加える) ことによって，より良いステップ方向を得ることができます．ニュートン法のステップ方向修正分を計算するためにはニュートン法のステップ方向の計算時に得られる副産物を有効に利用できるため，少ない計算コストでニュートン方向を改善することができます，計算を効率化することができます．

NUOPT にはこのことを利用し、さらに問題が線形であることに特化したチューニングを加えた手法が組み込まれています。

4.2 最適化手法に関する設定での説明にある

- ◆ 線形計画問題専用内点法 (Higher Order Method)

がここで解説されている手法です。

B.2 単体法・有効制約法

単体法は線形最適化問題

$$\begin{array}{ll} \text{最小化} & c^t x \quad x \in R^n \\ \text{条 件} & b_U \geq Ax \geq b_L, \quad b_U \geq x \geq b_L \end{array}$$

に対して，有効制約法は二次計画問題

$$\begin{array}{ll} \text{最小化} & \frac{1}{2} x^t Q x + c^t x \quad x \in R^n \\ \text{条 件} & b_U \geq Ax \geq b_L, \quad b_U \geq x \geq b_L \end{array}$$

に対して，それぞれ有効な方法です．一度可能基底解が得られれば，問題に対して小さな変更を行った際の解を比較的高速に求めることができるなど，内点法にはない特徴を備えています．

B.2.1 改訂単体法

NUOPT に実装されている単体法は大規模問題用の改訂単体法と呼ばれる手法です．

4.2 最適化手法に関する設定での説明にある

- ◆ 単体法 (Simplex Method)

がこの手法に相当します．単体法自身に関する解説は例えば[3]を参照して下さい．

B.2.2 有効制約法

NUOPT に実装されている有効制約法は改訂単体法に基づいています．4.2 最適化手法に関する設定での説明にある

- ◆ 有効制約法 (Active Set Method)

がこの手法に相当します．有効制約法に関する解説は例えば[13]を参照して下さい．この手法は5千変数以上の大規模問題では，一般に内点法（直線探索法 (Line Search Method)）に劣りますが，

変数に比べて制約式の数が非常に少ない（1/10以下）場合

目的関数のヘッセ行列が密行列である場合

には内点法よりも高速かつ高精度です．

B.2.3 分枝限定法

NUOPT は線形な整数・二次計画問題に対して，単体法・有効制約法にもとづく分枝限定法を用います．分枝限定法とは，組み合わせ的に膨大な数にのぼる可能解の中から最適解を見つけ出す（探索する）一般的な手法です．分枝限定法にて，整数計画問題を解く場合には整数性の条件を一部緩めた問題（緩和問題）を繰り返し解いて，その解から最適性を調べる範囲を限定しながら探索を行います．その際，単体法・有効制約法の上記2.の特徴を生かすと，線形な整数計画

法に対する分枝限定法を効率的に実現することができます。この手法自体については例えば[9]を参照して下さい。

整数計画問題は一般に組み合わせ問題であるため、大規模なものに関しては多くの計算時間を要しますが、問題を作成した側の知識に基いて、以下の様なパラメータを与えて探索を手助けすることが求解の効率化に有効である場合が多々あります。

- ◆ おおむねの最適解の値
- ◆ 変数の重要度のランク付け (分枝優先順)
- ◆ 特定の変数の目的関数値に対する貢献度 (擬コスト)

NUOPT は、2.7 整数計画問題に関するパラメータ設定に解説されているように、これらのパラメータをモデリング言語 SIMPLE から入力する手段を提供しています。また、最適性の保証はないものの実用上支障のない程度良い解を得るべく探索を途中で打ち切ったり、見切り値を設定するためのパラメータを設定することもできます。その方法の詳細については 4.3.8 整数変数計画、大域的最適化 (simple/asq/global) を行う際に有効なパラメータの各パラメータの解説をご覧ください。

B.3 逐次二次計画 (SQP) 法

逐次二次計画法では, 次のような等式・不等式制約付きの非線形最適化問題の解を求めることができます.⁶²

目的関数 $f(x) \rightarrow$ 最小化

$$\begin{aligned} \text{制約条件} \quad & g_j(x) = 0, j \in J_E \\ & g_j(x) \geq 0, j \in J_I \end{aligned}$$

SQP 法とは, 元の問題を現在の反復点において二次計画問題で近似し, その二次計画問題の解を探索方向としながら解を求める手法です. NUOPT では二通りの SQP 法を用いることができます. 以下, それらについて説明します.

B.3.1 準ニュートン法を用いる方法

本手法では, 元の問題を次のような二次計画問題で近似します.

$$\text{目的関数} \quad \frac{1}{2} \Delta x^T B_k \Delta x + \nabla f(x_k)^T \Delta x \rightarrow \text{最小化}$$

$$\begin{aligned} \text{制約条件} \quad & g_j(x_k) + \nabla g_j(x_k)^T \Delta x = 0, j \in J_E \\ & g_j(x_k) + \nabla g_j(x_k)^T \Delta x \geq 0, j \in J_I \end{aligned}$$

ここで B_k は元の問題のラグランジュ関数のヘッセ行列を準ニュートン法によって近似した行列です. この問題の解 Δx を探索方向とし, 直線探索を行って, 次の反復点を定める方法となります. 4.2 最適化手法に関する設定での説明にある

♦ lsqp (直線探索法に基づく逐次二次計画法 (Line Search SQP Method))

がこの手法に相当します.

B.3.2 信頼領域法を用いる方法

本手法では, 二つの二次計画問題を解くことで点列を生成していきます. まず, 一つ目の二次計画問題は次の通りです.

$$\text{目的関数} \quad \frac{1}{2} (\Delta x_k^{SD})^T D_k \Delta x_k^{SD} + \nabla f(x_k)^T \Delta x_k^{SD} \rightarrow \text{最小化}$$

$$\begin{aligned} \text{制約条件} \quad & g_j(x_k) + \nabla g_j(x_k)^T \Delta x_k^{SD} = 0, j \in J_E \\ & g_j(x_k) + \nabla g_j(x_k)^T \Delta x_k^{SD} \geq 0, j \in J_I \end{aligned}$$

ここで D_k とは, 要素が正の値であるような対角行列とします. この問題の解 Δx_k^{SD} は, 本手

⁶² 内点法の説明の箇所でも説明しましたが, ここに挙げた数理計画問題の定式化はアルゴリズム説明のために挙げた一例であり, NUOPT は変数の上下限制約などを含んだ寄り一般的な問題を扱うことができます.

法に大域的収束性を与える上で大きな役割を果たします。

この問題の解を求めたとき，効いている制約の集合を J_A^k とします．すなわち

$$J_A^k = \{j \in J_E \cup J_I \mid g_j(x_k) + \nabla g_j(x_k)^T \Delta x_k^{SD} = 0\}$$

となります．このとき，もう一つの二次計画問題は次のように定められます．

$$\text{目的関数} \quad \frac{1}{2} (\Delta x_k^N)^T G_k \Delta x_k^N + \nabla f(x_k)^T \Delta x_k^N \rightarrow \text{最小化}$$

$$\text{制約条件} \quad g_j(x_k) + \nabla g_j(x_k)^T \Delta x_k^N = 0, j \in J_A^k$$

ここで G_k は元の問題のラグランジュ関数のヘッセ行列です．この問題の解 Δx_N は，反復点が元の問題の解に近づいたときに速い収束をするための方向になっています．また，この問題の KKT 条件は，次のように線形方程式系として表すことができます．

$$\begin{pmatrix} G_k & -\nabla g_{J_A^k}(x_k) \\ \nabla g_{J_A^k}(x_k) & 0 \end{pmatrix} \begin{pmatrix} \Delta x_k^N \\ y_{k+1, J_A^k}^N \end{pmatrix} = \begin{pmatrix} -\nabla f(x_k) \\ -g_{J_A^k}(x_k) \end{pmatrix}$$

ここで， $y_{k+1, J_A^k}^N$ はこの問題の制約条件に対するラグランジュ乗数とします．一般に，問題規

模が同程度であれば，線形方程式系は二次計画問題に比べ速く解くことができるので，この問題に対する計算コストは，先程の Δx_{SD} を求める問題と比べて小さいものと考えられます．

本手法では， Δx_{SD} と Δx_N の凸結合

$$\Delta x_k(v_k) = v_k \Delta x_k^{SD} + (1 - v_k) \Delta x_k^N$$

を探索方向とし，定められた信頼領域の中を探索し，次の反復点を生成します．

4.2 最適化手法に関する設定での説明にある

- ◆ tsqp (信頼領域法に基づく逐次二次計画法 (Trust Region SQP Method))

がこの手法に相当します．

B.4 タブー・サーチによる制約充足問題解法

このアルゴリズムは離散変数を変数とした制約充足問題：

$$\text{変数} \quad x_j \in X_j, \quad j = 1, \dots, n$$

$$\text{条件} \quad c_i^U \geq g_i(x_1, \dots, x_j, \dots, x_n) \geq c_i^L, \quad i = 1, \dots, m$$

をメタヒューリスティクスの解法の一つであるタブー・サーチを用いて解くものです。ここで、 X_j は各変数の定義域に対応する有限集合です。

各制約の違反量の合計を最小化する問題に帰着して解きますので、制約式をすべて満たす解が存在しない場合でもできるだけ制約を満たす解を得ることができます。各制約の違反量の合計を計算する際、各制約の違反量には「重み」と呼ばれる正の定数を掛けて合算します。重みの大きな制約式は優先的に満たされるように解の探索が行われますのでこの重みを制御することにより、重要度の高い制約式の重みを大きくしておくことで、より有用な解が得られることが期待できます。制約には重みをとして設定することもでき、そのようにして何よりも優先して満たすべき制約を表現することができます（このように重みがに設定されたものをハード制約、正の有限の値に設定されたものをソフト制約と呼びます）。

制約の他に最大化、最小化すべき目的関数 $f(x_1, \dots, x_j, \dots, x_n)$ を定義することもできますが、その際には

$$\text{a) 最小化問題であれば} \quad \mu - f(x) \geq 0$$

$$\text{b) 最大化問題であれば} \quad f(x) - \mu \geq 0$$

というソフト制約を定義して目的関数を扱っております。

4 パラメータ設定中 4.2 最適化手法に関する設定での説明にある「・タブー・サーチに基づく制約充足アルゴリズム (wcsp)」がここで解説されている手法です。アルゴリズムは京都大学「問題解決エンジン」グループの開発によるもので、詳細については[11]、[12]をご参照下さい。

B.5 凸緩和法に基づく大域的最適化アルゴリズム

一般に、非線形最適化問題は、実行可能領域（制約条件を満たす領域）の中に複数の局所的最適解を持ちます。「局所的最適解」とは、「その近くでは最も良い解だが、実行可能領域全体では最も良いとは限らない解」のことです。これに対し、「大域的最適解」とは「実行可能領域全体で最も良いことが保証された解」となります⁶³。次は 1 変数の非線形な最小化問題（制約は変数の上下限のみ）における局所的最適解が複数存在する例です。

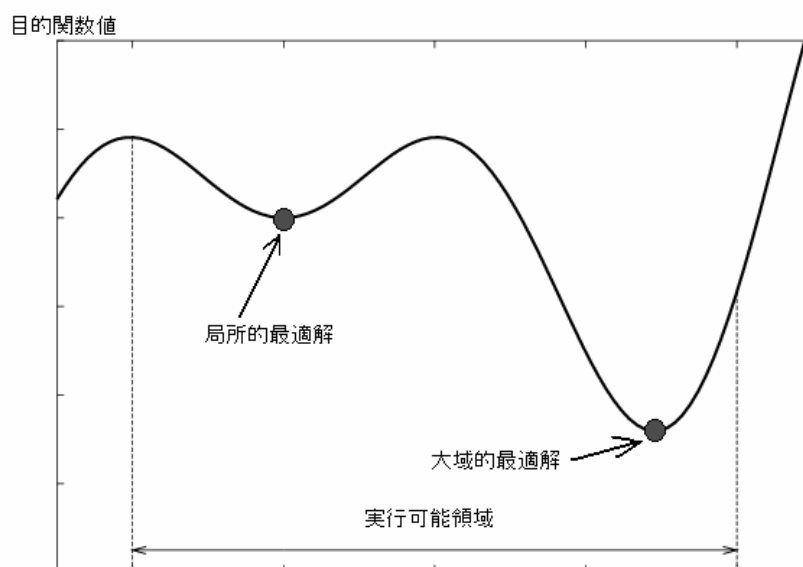


図 4 局所的最適解と大域的最適解

通常、非線形最適化問題を解く際に適用するアルゴリズム（例えば内点法、逐次二次計画法など）では、「局所的最適解の一つを見つける」ことが目的とされています。これは、非線形最適化問題において大域的最適解を見つけることが一般に困難であるからに他なりません。

しかし、大域的最適解を求めるための研究は比較的古くから行われており、そのための手法の一つとして、組み合わせ最適化問題でよく用いられる「分枝限定法」の考え方をを用いた手法が構成できます。

本手法は以下のようなステップを繰り返します。

- 1.（大域的最適解が存在する可能性のある）実行可能領域を分割し、分割された領域で「子問題」を生成して解く
2. 元の問題の実行可能解を何らかの方法で求める

⁶³ 大域的最適解も局所的最適解の一つと言えます。

3.1.2.の結果を用いて分割した各領域に大域的最適解が存在する可能性を判断し、存在しないと判断された領域を考慮から除外する

このアルゴリズムが機能するための前提条件として、各区間において生成された「子問題」が

『子問題の目的関数は、その実行可能領域において元の問題の目的関数を下から抑える』(*)

なる性質を持つことがあります。したがって、このアルゴリズムを実行するには、制約式や目的関数の解析的な性質を用いながら上記の性質を持つように、子問題を生成する必要があります。ただ、非線形関数として表された制約式や目的関数の形は非常に多様ですので、この子問題生成過程を自動化することは一般に困難です。しかし、定式の情報がモデリング言語を通じて、初等的な関数の合成（計算グラフ）として表現されていることを利用すれば、その過程を自動化、ユーザが特に意識することなく、大域的最適解の探索を行うことができます。

注意すべき点として、本手法は多数の子問題を繰り返し解くことになるので、計算時間ならびに多くのメモリを要する可能性が高いということです。本解法は組み合わせ最適化問題でよく用いられる「分枝限定法」に基づくものですが、組み合わせ問題の性質によっては分枝限定法が長い時間を要するのと同じように、本問題でも長い計算時間を要する可能性があります。

4 パラメータ設定中 4.2 最適化手法に関する設定での説明にある「・凸緩和法に基づく大域的最適化アルゴリズム(global)」がここで解説されている手法です。詳細については[7]、[17]をご参照下さい。

B.6 タブー・サーチによる資源制約スケジューリング問題解法

このアルゴリズムは以下の資源制約付きスケジューリング問題:

限られた資源の下、仕事の処理に用いる資源の分配、作業の開始時刻を決定する
をタブー探索を用いたリストの探索を用いて解くものです。

アルゴリズムは wesp と同じ京都大学「問題解決エンジン」グループによるもので、詳細については、[18]をご覧下さい。

B.7 外点法

内点法は変数が正の領域(内点)に反復点を取る方法ですが,問題を

$$f(x) + \rho \sum_i |x_i|_-, \quad x \in \mathbf{R}^n,$$

$$g(x) = 0$$

のように変形して(実際にはこの問題の滑らかな近似を考えます),非負制約を除去,この KKT 条件を解くと考えると,やはり直線探索・信頼領域法に基づく最適化アルゴリズムを構成することができ,大域的最適性を保証することができます. NUOPT に組み込まれている `lepm` は直線探索法に基づく外点法,`tepm` は信頼領域法に基づく外点法です.

```
options.exrho = 1.0e4 (SIMPLE)    param:exrho=1.0e4 (nuopt.prm)
```

実験の結果,内点法とほぼ同等のパフォーマンスを示しますが, ρ の大きさが相対的に小さいと非負制約を満たさない実行不可能な解に収束することはあり得ます(<<NUOPT 55>> exterior solution obtained.のエラーがそれを示します).逆に実行不可能な問題でも,等式制約を満たし,なるべく最適解に近い点を返すことが期待されます. ρ は次のパラメータで設定することができます.

内点法に比べてのメリットは,内点以外の点からも出発できるという点です.内点法では変数値を,非負制約を満たす内点に移動してから反復計算をスタートさせなければならないので,良い初期値が得られている場合でも,その操作により情報を失ってしまいます.外点法なら初期値を加工することなく,そのまま用いることができますので,良い初期値が得られている場合やリスタート時に比較的良好な性能を得ることが期待され,実験にてそれが検証されております.

付録 c 参考文献

- [1] J. E. Beasley(ed.), Advances in Linear and Integer Programming, Oxford University Press, 1996.
- [2] I. Bongartz, A. Conn, N. Gould and Ph. L. Toint, CUTE: Constrained and Unconstrained Testing Environment, Research Report RC 18860, IBM, T. J. Watson Research Center, Yorktown, U.S.A., 1993.
- [3] V・フバータル著/阪田省二郎・藤野和建訳,線形計画法(上/下),啓学出版,1983.
- [4] I. S. Duff and J. K. Reid, The Multifrontal solution of indefinite sparse symmetric linear systems, ACM Transaction on Mathematical

Software, Vol.9, No.3, 302-325, 1983.

[5] P. E. Gill, W. Murray, M.A.Saunders and M.H.Wright, A practical anti-cycling procedure for linearly constrained optimization, Mathematical Programming, 45:437-474, 1989.

[6] P. E. Gill, W. Murray, M. A. Saunders and M. H. Wright, Inertia-controlling methods for general quadratic programming, SIAM Review, 33:1-36, 1991.

[7] 逸見宣博, 山下浩, 計算グラフと分枝限定法を利用した大域的最適化 - 2, 2004 年日本オペレーションズ・リサーチ学会秋季研究発表会アブストラクト集, 42-43.

[8] W. Hock and K. Shittkowski, Test examples for nonlinear programming codes, Springer Verlag, 1981.

[9] 茨木俊秀・福島雅夫著, FORTRAN77 最適化プログラミング, 岩波書店, 1991.

[10] K. Nonobe and T. Ibaraki, A tabu search approach for the constraint satisfaction problem as a general problem solver, European Journal of Operational Research 106, 599-623, 1998.

[11] K. Nonobe and T. Ibaraki, An improved tabu search method for the weighted constraint satisfaction problem, INFOR 39, 131-151, 2001.

[12] 伊理正夫, 今野浩, 刀根薫監訳, 最適化ハンドブック, 朝倉書店, 1995.

[13] 矢部博, 八巻直一, 非線形計画法, 朝倉書店, 1999.

[14] H. Yamashita, A globally convergent primal-dual interior point method for constrained optimization, Technical Report, Mathematical Systems Inc., Tokyo, Japan, April 1992 (revised May 1992).

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

October 1993.

[16] H. Yamashita and H. Yabe, Superlinear and quadratic convergence of primal-dual interior point methods for constrained optimization, Technical Report, Mathematical Systems Institute Inc., Tokyo, Japan, June 1993.

[17] 山下浩, 逸見宣博, 計算グラフと分枝限定法を利用した大域的最適化, 2003 年日本オペレーションズ・リサーチ学会秋季研究発表会アブストラクト集, 284-285.

[18] K. Nonobe and T. Ibaraki, Formulation and tabu search algorithm for the resource constrained project scheduling problem, in: C.C. Ribeiro and P. Hansen (eds.): Essays and Surveys in Metaheuristics, Kluwer Academic Publishers, pp.557-588, 2002.

[19] 田辺隆人, 山下浩, 主双対外点法とそのパラメトリック最適化への応用, 2005 年日本オペレーションズ・リサーチ学会秋季研究発表会アブストラクト集 50-51.

1

1D 書式 82, 83

2

2D 書式 85

A

acos 31

acosh 31

acot 31

acoth 31

acsc 31

acsch 31

active 170, 172, 175

Active Set Method 191

Activity ...20, 22, 26, 27, 28, 30, 45, 52, 53, 54,
163, 193, 195, 271, 272, 273, 275, 276, 277,
283

add 68

alldiff 39

arc 74

arcs 17, 74

asec 31

asech 31

asin 31

asinh 31

atan 31

atan2 31

atanh 31

avail 181

awk 170, 173

B

begin 187

bfgs 192

binary 36

Boolean 36, 43

BOUNDS 225

Branch and bound method 191

by71

C

C++ 104, 109, 135, 144, 212

card 68

ceil 31, 48

char 33

char* 48

check 48

COLUMNS 224

Condition 41

Constraint 22, 28, 41, 142

contains 68, 73

Convex Programming 190

Convex Quadratic Programming 190

cos 31

cosh 31

cot 31

coth 31

cout 144

CP 190

CQP 190

csc 31

CSV 75, 81

CyclicSet 23, 54

D

defaultConstraintWeight	156
defaultObjectiveTarget.....	156
defaultObjectiveWeight.....	156
defaultval	28, 52, 53
deleteCo	91, 111, 147
dim	54, 57
dir	37, 153
DiscreteVariable	38
dom	38
double	33, 48
dpc	37, 153
dual	145
DUAL_SIMPLEX_PIVOT_COUNT....	167, 184
duedate.....	28, 52, 273
dump	104
duration	28, 52, 271, 274

E

elapsedTime	146, 216
Element.....	22, 48
elementAt.....	55, 70
end.....	187
Equation.....	22
ERROR TYPE.....	168
ERROR_TYPE	167
errorCode	146, 216
errorMessage	146, 216
exp	31
Expression	22, 25, 27

F

fabs	31
------------	----

FACTORIZATION_COUNT	167
fevals.....	146, 216
first	55, 70
fixVariable	118
floor.....	31, 48
fmod	31
for.....	43, 52
FREE.....	170
from.....	71
FUNC_EVAL_COUNT	167

G

gap	181
Graph.....	23, 74
GUI	21

H

hardConstraint.....	155
higher	191
Higher Order Method .	168, 177, 191, 228, 293, 298, 300
hypot.....	31

I

if 43	
ifelse.....	43
in 17, 74	
index	28
init	94, 101, 145
int.....	33, 43, 48
integer	36
IntegerVariable	22, 28, 35, 108, 153
Interval.....	23, 72

ITERATION_COUNT 167
 iters 146, 216

K

Karush-Kuhn-Tucker 条件 289

L

last 55, 70
 lb 94, 101
 left 72
 Line Search Method 191, 228, 291
 Line Search SQP Method 192, 296
 Line Search with BFGS 192, 291
 Linear Programming 189
 llen 181
 lo 181
 lock 64
 log 31
 log10 31
 LP 189
 lsqp 192, 296

M

max 35
 maximize 228, 229
 maxitn 206
 maxtim 206
 mem 181
 METHOD 167
 MILP 190
 min 35
 minimize 228
 MINIMIZE 172

MIP 190
 MIQP 190
 Mixed Integer 190
 Mixed Integer Linear Programming 190
 mknuopt 132, 133, 135, 136
 mode 28, 52, 272, 274, 277
 mpsfile 229
 mpsout 248
 MPS ファイル 127, 211, 221
 BOUNDS セクション 243
 COLUMNS セクション 238
 ENDATA セクション 247
 HESSIAN セクション 244
 INITIAL セクション 246
 NAME セクション 237
 RANGES セクション 241
 RHS セクション 240
 ROWS セクション 237
 右辺 240
 上下限制約 241, 243
 整数変数 239
 制約式 237
 二次の項 245
 のカラム位置 236
 の書式 235
 のセクション 236
 のフィールド 235
 パス 211
 変数 238
 目的関数 237

N

name 28
 NAME 224
 next 55, 70

nfunc 146, 216
 NLP 190
 node 74
 nodes 17, 74
 Nonlinear Programming 190
 normal 151, 230
 NUMBER_OF_FUNCTIONS 167
 NUMBER_OF_VARIABLES 167
 nuopt.prm 81, 131, 158, 163, 187, 227, 228
 nuopt_readBaseFromFile 219
 nuopt_restartDescChar 219
 nuopt_saveBaseInFile 219
 nuopt_setBasein 219
 nuopt_setBaseout 219
 nvars 146, 216

O

Objective 22, 27
 oleft 72
 OPTIMAL 168
 options 112, 158, 160, 163, 213
 optValue 146, 216
 OrderedSet 23, 54
 oright 72
 out 17, 74
 outfilename 163

P

Parameter 22, 28, 48
 partial 36
 PARTIAL_PROBLEM_COUNT .. 167, 169, 184
 position 55, 70
 pow 31
 prev 55, 70

pri 37, 153
 print 95
 PROBLEM_NAME 167
 PROBLEM_TYPE 167
 prod 33, 35

R

RANGES 225
 rcpsp .. 30, 52, 53, 160, 163, 164, 167, 168, 171,
 181, 183, 189, 190, 193, 195, 205, 213, 271,
 272, 274, 276, 277, 283
 readD 109
 readS 109
 remove 68
 residual 146, 216
 RESIDUAL 167, 169
 resource 28, 53, 165, 274
 ResourceCapacity 20, 22, 26, 28, 30, 52, 163,
 164, 193, 195, 271, 273, 274, 276, 283
 ResourceRequire20, 22, 26, 28, 30, 52, 163, 193,
 195, 271, 272, 274, 276, 277, 283
restoreCo 91, 111, 147
 result 145, 213
 right 72
 ROWS 224

S

sample.mps 234
 sec 31
 sech 31
 selection 35
 semicont 36
 Sequence 23
set 49

Set	23
setOf	70
showSystem	90, 91
silent	212
SIMPLE	133
simple_fprintf	100
simple_printf	98
Simplex Method	163, 190, 195, 228
SIMPLEX_PIVOT_COUNT	167, 168, 184
sin	31
sinh	31
slice	69
softConstraint	155
solfile.sol	124, 139, 166
solve	143, 144
solver.sol	212
SQP 法	296
sqrt	31
STATUS	167, 168
sum	33
superSet	58, 66

T

Table	38
tan	31
tanh	31
tardiness	277, 283
time	181
timeStep	28, 53, 165, 271, 274, 276
to 71	
tolerance	146, 216
trust	192
Trust Region Method	163, 192, 195, 228, 292
Trust Region SQP Method	192, 297
tsqp	192, 193, 297

type	29
------------	----

U

ub	94, 101, 145
ufun	144
unfixVariable	118
UNIX	123
until	30, 37
up	181
upc	37, 153

V

val	94, 101, 143, 145
VALUE_OF_OBJECTIVE	167
Variable	22, 27
VariableParameter	22, 28, 121

W

wcsp	205
weight	53, 165, 274, 282
while	52
Wild Card	79
Windows	123
Windows 版	212

あ

足切り点	206, 207
------------	----------

い

射像	59
依存	61

イテラ	52, 55, 70
違反	172
インスタンス	27
インデックス	15

う

上に凸	228
右辺ラベル	224

え

エラー	167
エラーコード	216
エラーメッセージ	146, 216
演算	58
演算結果	25

か

改行	77, 78
回数の上限	197
改訂単体法	294
解ファイル	139, 166, 224, 225
のルート名	212
名	212
解ファイル名	212
解法	189
下限	145
下限値	94, 101
可能基底解	190, 191, 196
可変定数	22, 28
完結	61
関数	167, 172
関数の数	146, 216
関数の上下限	172

関数名	248
緩和問題	294

き

記号

!	41
!=	40
#pivot	181
#prob	181
&	58
&&	41
*	58
.cc	136
.sol	139, 224
.val	147
	58
	41
<	41, 65
<<	96, 144
<=	40
==	40
>	41, 65
>=	40
擬コスト	153, 154, 295
基底	219

く

空行	187
空集合	55
空白	187
組み合わせ問題	295
クラスオブジェクト	22
グラフ	23, 74
構造	17

節点.....	17
節点.....	74
辺 17, 74	
クロスオーバー	196

け

計算時間	146, 216
計算時間上限	208
現在の値.....	94, 101, 144, 145

こ

高次方向	292
コメント	76, 187
混合整数計画問題	190
コンマ.....	50, 77, 78

さ

最後の作業の完了時刻	53, 181, 182, 275, 277
最小化	29, 167, 228
最小コスト経路.....	18
最小コスト経路問題	16, 118
最大化	29, 147, 167, 228
最短経路	120
最適化	141
アルゴリズム.....	228
手法.....	167,
の実行	92
最適性条件	168, 169, 177
最適性条件の残差	197, 198, 200, 202, 204
最適性の必要条件	289
差集合	58
サマリ情報.....	141, 145
残差.....	146, 168, 177, 216

参照値.....	94, 103
----------	---------

し

次元	57, 79
システム	90, 150
システムコード	135
システム制御.....	118
下に凸.....	228
実行可能領域.....	190
実行時エラー	32, 48, 52, 59, 62
自動設定	195
自動追加	62, 63
シャドウプライス	101, 139, 172, 174, 224, 226
周期的順序付き集合	54
集合	23, 54
修正 KKT 条件	289
収束判定値	146, 216
出力	95
出力ファイル名.....	229
出力モード.....	212
循環集合	23
順序集合	23
順序付き集合.....	52, 54
準ニュートン法	192, 200, 290
上下限.....	170, 174, 226
上限	145
条件式.....	15, 41, 69, 103
上限値.....	94, 101
条件付け	26, 40
初期値.....	31, 76, 94, 101
信頼領域法	163, 192, 195, 201, 228, 291, 292
信頼領域法に基づく逐次二次計画法.....	192, 297

す

数式	22, 26, 27
数値的な性質	173
数値計画法ライブラリ	21
数列	23, 71
スケーリング	197, 281
ステップ方向	168
スペース	77, 78

せ

正定値	290
正常終了	168
整数解	169
整数計画	206, 295
整数計画問題	36, 153
整数変数	22, 28, 163, 191, 195, 206, 225, 228
制約式	26, 28, 40, 111, 142
制約式の値	172
制約充足アルゴリズム	205
制約条件	22
積集合	58, 63, 67
線形計画問題	13, 189, 222, 290
線形計画問題専用内点法	163, 191, 195, 197, 228, 293, 298, 300
宣言の引数	27
先行制約	53, 168, 272, 273
選択	35

そ

双対単体法	167
双対変数	101, 139, 141, 172, 174, 224, 226
双対変数値	94, 101, 145
相補性条件	289

添字	15, 22, 48
添字集合	28, 56
添字集合並び	56
添字付け	26, 27
ソート	96
ソフト制約	155, 178, 206, 298

た

大域的収束性	291
大域変数	145, 158
タイトル	224
代入	31, 150
タブ	77, 78
タブ・サーチ	205
探索深さ	206
探索問題数	208
単体法	131, 163, 168, 190, 195, 203, 228, 294

ち

逐次代入	30
逐次二次計画法	204, 296, 297
直積	29, 50, 57, 58, 63, 67
直前先行制約	53, 168, 272, 273, 275,
直線探索	290
直線探索法	191, 196, 198, 291
直線探索法に基づく逐次二次計画法	192, 296

て

定義集合	49
への操作	60
定数	11, 22,
定数式	12, 26
データ	137, 149

データ構造.....	23
データファイル.....	18, 75, 139, 140, 149
デフォルト.....	163, 227
展開.....	50, 60, 90, 150

と

等式.....	11
等式制約	228
解く前の値	145
途中結果	11
凸228	
凸計画問題.....	190, 191, 196, 290

な

内点法	128, 168, 191
内部構造	125
ナップサック問題	153

に

二次計画問題	121, 135, 190, 191, 228
二次の項.....	228
二重引用符	76, 78
入力.....	57, 109
ニュートン法	290

ね

ネットワーク.....	17, 23
ネットワーク問題	74

の

納期遅れ.....	53, 165, 182
-----------	--------------

は

ハード制約.....	155, 178, 206, 298
配列	89, 104, 109
破壊	106
パラメータ	112
パラメータファイル	81, 131, 159, 215, 227, 228
の例	187
バリアパラメータ	289
バリア - ペナルティ関数	289
範囲指定並び.....	34
反復回数	146, 168, 216

ひ

非線形計画問題	190
非線形性	163
評価回数	146, 216
表示	21, 89, 94, 113, 144, 147
標準出力	112, 138, 144, 176, 223, 227, 230
標準入力	222

ふ

ファイル構造体.....	100
フィールド	170
複数の目的関数	150
不等式.....	11
部分問題	169
分解回数	168
分枝限定法	153
分枝方向	153
分枝優先順	295
分枝優先順位	153
分離	116

へ

ヘッセ行列	191, 290
ペナルティパラメータ	289
ベルヌーイ関数	47
変数	22, 26, 27
変数の数	146, 216
変数の上下限	170
変数名	248

ほ

包含関係	65, 66
方程式	14

ま

前処理	177
-----------	-----

め

メリット関数	289
--------------	-----

も

目的関数	10, 22, 27, 93, 167, 224
目的関数値	146, 216
目的関数名	142
文字列	31
モデル	21, 150

ゆ

有効制約法	191, 203, 294
-------------	---------------

優先順位	154
------------	-----

よ

要素の直積	50
要素を含む式	15
抑制	164

ら

ライブラリ	134
ラグランジュ関数	289
ラベル名	211

り

離散変数	38
リスタート	219
略記法	79

る

ループ	52, 70
-----------	--------

れ

連続範囲	23, 72
------------	--------

ろ

ロードモジュール	137
----------------	-----

わ

和集合	58, 63
-----------	--------
