Numerical Optimizer

SIMPLE 外部接続マニュアル V19

株式会社NTTデータ数理システム

2017年1月

MATHEMATICAL SYSTEMS INCARNATION

目　次

第 1 章 はじめに

このドキュメントでは，簡単なデモンストレーションを通じて，Numerical Optimizer[1]と外部プログラムの連結方法を解説します．Windows 版では VisualC++の IDE[2]を利用した例を用います．外部プログラムとの連結方法には大きく分けて，

（ア）実行形式を作成して別プロセスで起動する（第 2 章）

（イ）ライブラリ `solveLP`，`solveQP` をコールする（第 3 章）

（ウ）SIMPLE のモデル記述を手続きの中に記述して利用する（第 4 章）

（エ）SIMPLE のモデル記述からクラスを生成して利用する（第 5 章）

という四つの方法があります．このドキュメントではこれらについて順に解説します．各方法の簡単な説明は，次の通りです．

（ア）の方法は Numerical Optimizer を外部プログラムとして別プロセスで起動する方法です．プロセス起動やデータのやり取りにオーバーヘッドがありますが，実装・デバッグは最も容易です．

（ア）以外の（イ）（ウ）（エ）は，いずれも Numerical Optimizer をライブラリとしてリンクし，その機能を利用する方法です．

（イ）は LP，MILP，QP に属する問題を SIMPLE を介さずに専用の関数を使って解く方法です．高速ではありますが，ユーザ側でデータを標準形に変形する必要がございます．

（ウ）（エ）はいずれも SIMPLE を介する方法です．（ウ）は（エ）よりは簡便ですが，Numerical Optimizer のライブラリがリンクされているプログラム全体で同時に一つの問題しか扱うことができません．（エ）は手順がやや複雑になりますが，複数の問題を一つの外部プログラムで操作することが可能です．ただし，一部の UNIX/Linux 環境では（エ）の方法をサポートしておりません．

Numerical Optimizer の対応コンパイラに関するご案内

BC++・VC++：

最新のNumerical Optimizerは，以下のコンパイラに関して正式対応いたしておりません．

- Borland C++ 5.5（V12より非対応）
- Microsoft Visual C++ 6.0（V16より非対応）
- Microsoft Visual C++.net（V16より非対応）
- Microsoft Visual Studio 2005（V17より非対応）
- Microsoft Visual Studio 2008（V19より非対応）

なお，対応コンパイラに関する情報につきましては，http://www.msi.co.jp/nuopt/products/spec_table.html をご覧ください．

[1]「Numerical Optimizer」につきまして，V15 までに関しましては適宜「NUOPT」と読み替えをお願いいたします．

[2]Microsoft Visual Studio 2010 に基づいて説明します．

- **VC++**

  フォルダ：

  （Numerical Optimizer のインストール場所）\samples\app

  の下に以下の zip ファイルがあります.

  - app_VS2010.zip（VS2010 用）
  - app_VS2012.zip（VS2012 用）
  - app_VS2013.zip（VS2013 用）
  - app_VS2015.zip（VS2015 用）

  なお，デフォルトの Numerical Optimizer のインストール場所は 32bit 版 Windows であれば

  C:\Program Files\Mathematical Systems Inc\NUOPT

  64bit 版 Windows であれば

  C:\Program Files (x86)\Mathematical Systems Inc\NUOPT

  となっています.

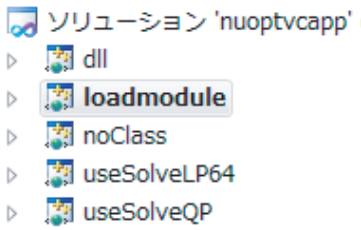
  まず，ご利用になるコンパイラに対応した zip ファイルを解凍します．なお，OS の設定によってはサンプルの場所に書き込み権限が無いため，適切なフォルダにコピーしてからサンプルの解凍・実行をする必要があります.

  解凍を行なうと，VC++のソリューションである

  nuoptvcapp.sln

  があります.

  このソリューション内には以下の 5 つのプロジェクトがあります.



  これらのプロジェクトが 3 章以降で説明する実行例に対応しています．本マニュアルで例として用いるのは簡単な整数計画問題である次のナップサック問題です.

変数	$x_i \in \{0,1\} \quad (i \in S)$
目的関数（最大化）	$\sum_{i \in S} c_i x_i,$
制約条件	$\sum_{i \in S} a_i x_i \le b$

  S の要素数だけ 0 − 1 変数があり，線形制約が一本，線形の目的関数を最大化するという問題です．この問題を設定するのに必要なデータは，

目的関数の係数 c_i，制約式の係数 a_i，制約式の右辺値 b

となります．この問題を解くというアプリケーションを上記の様々な方法で行います．

1.1 サポートプラットフォーム

このマニュアルに記述されている内容は Windows 版，UNIX/Linux 版共通です．

Windows 版で Numerical Optimizer のインストール時に 64bit コンパイラを選択した場合は，Win32 プロジェクト設定を64ビットプロジェクト構成に変更する必要があります．変更の方法については，「7.1 Microsoft Visual Studio プロジェクトの設定」を参照してください．また，Numerical Optimizer のインストール時に選択したコンパイラと外部接続で利用するコンパイラが異なっていますと，正常に動作しません．必ず同じコンパイラを用いてください．

UNIX/Linux 版について「SIMPLE のモデル記述からクラスを生成して利用する方法」は環境によってはご利用になれない場合がございます．詳細につきましては Numerical Optimizer サポート（nuopt-support@msi.co.jp）までご連絡ください．

1.2 実行環境に含まれるもの

外部プログラムの連結を行うための実行環境には次が含まれます．一部環境によって必要なファイルが異なります．

- インクルードファイル（共通して必要）：

`userapp/include/*.h`	Numerical Optimizer のインクルードファイル

- VC++版ライブラリとサンプル（VC++でのリンクにのみ必要）：

`userapp/lib/libnuopt_M*.lib`	Numerical Optimizer ライブラリ（VC++版）
`userapp/lib/libnuopt_ipm_MT_m.lib`	Numerical Optimizer 大規模線形計画問題用内点法ライブラリ（VC++版）
`SAMPLES/app/`	VC++版用サンプルディレクトリ

- UNIX/Linux 版ライブラリとサンプル（UNIX/Linux 環境でのリンクにのみ必要）：

`libnuopt/libnuopt.a`	Numerical Optimizer ライブラリ（UNIX/Linux 版）
`libnuopt/libnuopt64ipm.a`	Numerical Optimizer 大規模線形計画問題用内点法ライブラリ（UNIX/Linux 版）
`userapp/`	UNIX/Linux 版用サンプルディレクトリ

このライブラリから生成したロードモジュールは Numerical Optimizer がインストールされている環境でのみ動作し，動作条件は Numerical Optimizer に準じます．

1.3 サンプルディレクトリの内容

外部ライブラリの利用サンプルディレクトリです．以下がその一覧です．

- **VC++**

nuoptvcapp.sln	VC++のソリューション
useSolveQP.cpp	solveLP，solveQP を用いるサンプルメイン
useSolveQP/qp312.txt	useSolveQP.cpp 用データ 1
useSolveQP/qp312I.txt	useSolveQP.cpp 用データ 2
useSolveQP/knapsack.txt	useSolveQP.cpp 用データ 3
useSolveQP/hs23.txt	useSolveQP.cpp 用データ 4
useSolveQP/useSolveQP.vcxproj	プロジェクトファイル
useSolveLP64.cc	solveLP64 を用いるサンプルメイン
useSolveLP64/oil.txt	useSolveLP64.cc 用データ
useSolveLP64/useSolveLP64.vcxproj	プロジェクトファイル
dll/dll.vcxproj	プロジェクトファイル
dll/dllsamples.xls	dll を呼び出すサンプル Excel ブック
loadmodule/loadmodule.vcxproj	プロジェクトファイル
noClass/noClass.vcxproj	プロジェクトファイル
noClass.cpp	モデルの記述と実行を分離しないサンプル
knapsack.smp	クラスインターフェーステストモデル
knapsack.cc	genClass により自動生成されたクラスの実装
knapsack.h	genClass により自動生成されたヘッダー
knapsackControl.cc	knapsack のコントロール雛形
main.cpp	knapsackSolve のメイン
driver.cpp	knapsackSolve のドライバ
dllmain.cpp	dll のメイン

- **UNIX**

makefile	サンプルの makefile
useSolveQP.cc	solveLP，solveQP を用いるサンプルメイン
qp312.txt	useSolveQP.cc 用データ 1
knapsack.txt	useSolveQP.cc 用データ 2
useSolveLP64.cc	solveLP64 を用いるサンプルメイン
oil.txt	useSolveLP64.cc 用データ
useClass.cc	クラスを用いるサンプルメイン
QP.smp	クラスインターフェーステストモデル 1
qp312.dat	QP.smp 用データ
knapsack.smp	クラスインターフェーステストモデル 2

useSimple.cc	SIMPLE のモデル記述を手続きの中に直接記述する場合のサンプルメイン
MIP.cc	SIMPLE のモデル記述を手続きの中に直接記述する場合の問題記述の例

1.4 SIMPLE の行列・ベクトルクラスオブジェクトをご利用になる場合の注意点

外部接続機能をご利用になる際，SIMPLE のモデル記述において対称行列 SymmetricMatrix ／行列 Matrix ／ベクトル Vector などの SIMPLE のクラスオブジェクトを使用する場合には原則として

```
#include "simpleMatrix.h"
```

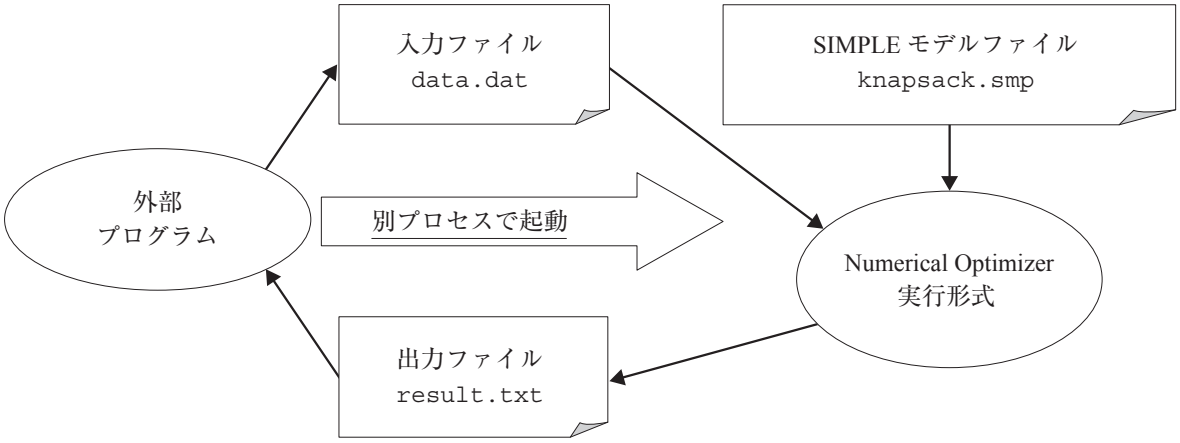
もしくは

```
using namespace SimpleMatrix;
```

の記述が必要となります．ただし，Windows 環境において「5.1.2 genClass のコール」を実行された場合には記述する必要はありません．

第2章 実行形式を作成して別プロセスで起動する例

Numerical Optimizer を外部プログラムと連結する方法として，まずは Numerical Optimizer 部分を実行形式として作成し，外部プログラムから別プロセスで起動する例を紹介します．データの入出力はファイルを介して行います．この方法は入出力ファイル処理やプロセス起動のオーバーヘッドがありますが，Numerical Optimizer と外部プログラムが独立しているので，問題が起こったときに原因の切り分けが容易で，開発しやすいというメリットがあります．特に上記のオーバーヘッドが問題になる場合を除いては，本方法を使用することをお勧めします．



2.1 実行形式の作成

まずは SIMPLE のモデルファイル `knapsack.smp` から実行形式を作成します．ここでは，

目的関数の係数： $c = (6 \quad 8 \quad 4 \quad 3 \quad 4)$
制約式の係数： $a = (4 \quad 2 \quad 3 \quad 6 \quad 7)$

という 5 変数のナップサック問題を解きます．以下にコマンドプロンプト上で実行形式を作成する例を示しますが，詳細については Numerical Optimizer/SIMPLE マニュアルをご参照ください．

knapsack.smp

```
//
//   ナップサック問題
//
Set S;
Element i(set=S);
```

```
IntegerVariable x(index=i,type=binary); // 整数変数
Parameter c(index=i);
Parameter a(index=i);
Parameter 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

solve(); // 求解

// 結果のファイル出力
FILE* fout = fopen("result.txt","w");
if(fout){
  simple_fprintf(fout, "%f\n", obj);
  simple_fprintf(fout, "%d\n", x[i]);
  fclose(fout);
}else{
  fprintf(stderr, "file open error!\n");
}
```

- **VC++**

```
prompt> mknuopt.bat knapsack.smp
```

- **UNIX**

```
prompt% mknuopt knapsack.smp
```

2.2 外部プログラムの例

Numerical Optimizer 部分は別プロセスとして起動するので，外部プログラムを実装するプログラミング言語としては別プロセスを起動することができるものであれば何でも構いません．ここでは C 言語と Java を用いた外部プログラムの例（Windows 版）を示します．

ここでは，モデルの b（制約式の上限）に相当するデータを 1 から 30 まで動かして繰り返し解くという手続きを作成します．

外部プログラム例（C 言語，Windows 版）

exeLoopC.c

```
//
// Numerical Optimizer 実行形式を別プロセスとして
// 実行する例（C 言語）
//
#include <stdio.h>
#include <stdlib.h>

void write(int n, double* a, double b, double* c, const char* input);
void read(int n, int* x, double* obj, const char* output);

int main(void)
{
  int i, n = 5;
  double a[] = {4,2,3,6,7};
  double b;
  double c[] = {6,8,4,3,4};
  int x[5];
  double obj[1];
  int ret;
  for(b = 1; b <= 30; b++){
    write(n, a, b, c, "data.dat"); // 入力データの作成
    ret = system("knapsack.exe data.dat"); // 別プロセスとして起動
    read(n, x, obj, "result.txt"); // 出力データの読み込み
    printf("ret = %d, b = %f, obj = %f, x =", ret, b, *obj);
    for(i = 0; i < n; i++)
      printf(" %d", x[i]);
    printf("\n");
  }
  return 0;
}

// 入力データの作成関数
void write(int n, double* a, double b, double* c, const char* input)
{
  int i;
  FILE* fp = fopen(input, "w");
  fprintf(fp, "a=\n");
  for(i = 0; i < n; i++)
```

```
    fprintf(fp, "[%d] %f\n", i+1, a[i]);
  fprintf(fp, ";\nb = %f;\nc=\n", b);
  for(i = 0; i < n; i++)
    fprintf(fp, "[%d] %f\n", i+1, c[i]);
  fprintf(fp, ";\n");
  fclose(fp);
}

// 出力データの読込関数
void read(int n, int* x, double* obj, const char* output)
{
  int i;
  char buf[256];
  FILE* fp = fopen(output, "r");
  fgets(buf, 256, fp);
  sscanf(buf, "%lf", obj);
  i = 0;
  while(fgets(buf, 256, fp) != NULL){
    sscanf(buf, "%d", x+i);
    i++;
  }
  fclose(fp);
}
```

外部プログラム例（Java，Windows 版）

exeLoopJ.java

```
//
// Numerical Optimizer 実行形式を別プロセスとして実行する例（Java）
//
import java.io.*;

class exeLoopJ{
  public static void main(String[] args){
    int n = 5;
    double[] a = {4,2,3,6,7};
    double[] c = {6,8,4,3,4};
    int[] x = new int[n];
    double[] obj = new double[1];
```

```
  for(double b = 1; b <= 30; b++){
    int ret = 1;
    try{
      write(n, a, b, c, "data.dat"); // 入力作成
      ret = execute("knapsack.exe","data.dat"); // 別プロセス起動
      read(n, x, obj, "result.txt"); // 出力読み込み
    }catch(IOException e){
      e.printStackTrace();
    }
    System.out.print("ret = "+ret+", b = "+b+", obj = "+obj[0]+", x =");
    for(int i = 0; i < n; i++)
      System.out.print(" "+x[i]);
    System.out.print("\n");
  }
}
// 入力作成メソッド
static void write(int n, double[] a, double b, double[] c, String input)
    throws IOException{
  FileWriter fw = new FileWriter(input);
  fw.write("a =\n");
  for(int i = 0; i < n; i++)
    fw.write("["+(i+1)+"] "+a[i]+"\n");
  fw.write(";\nb = "+b+";\nc =\n");
  for(int i = 0; i < n; i++)
    fw.write("["+(i+1)+"] "+c[i]+"\n");
  fw.write(";\n");
  fw.close();
}
// 別プロセス起動メソッド
static int execute(String exe, String input) throws IOException{
  ProcessBuilder pb = new ProcessBuilder(exe, input);
  Process p = pb.start();
  InputStream is = p.getInputStream();
  while(is.read() > 0);
  try{
    p.waitFor();
  }catch(java.lang.InterruptedException e){
    e.printStackTrace();
```

```
    }
    return(p.exitValue());
  }
  // 出力読込メソッド
  static void read(int n, int[] x, double[] obj, String output) throws IOException{
    FileReader fr = new FileReader(output);
    BufferedReader br = new BufferedReader(fr);
    obj[0] = Float.valueOf(br.readLine());
    for(int i = 0; i < n; i++)
      x[i] = Integer.valueOf(br.readLine());
    br.close();
    fr.close();
  }
}
```

C 言語，Java どちらの例においても write 関数（メソッド）を定義して，Numerical Optimizer への入力データをファイルとして出力しています．Numerical Optimizer への入力ファイルのフォーマットについては，Numerical Optimizer/SIMPLE マニュアルをご参照ください．

C 言語の例では system 関数を使って，Java の例では ProcessBuilder を使った execute メソッドを定義して Numerical Optimizer 実行形式を別プロセスとして起動しています．この際，write 関数（メソッド）で作った入力データファイル data.dat を引数として与えています．

Numerical Optimizer 実行形式は求解結果（x と obj）を result.txt に出力します．C 言語，Java どちらの例においても read 関数（メソッド）を定義して，Numerical Optimizer の求解結果を読み込んでいます．

第3章 ライブラリ solveLP，solveQP

solveLP，solveQP はそれぞれ（混合整数）線形計画問題，（混合整数）二次計画問題・二次制約付き二次計画問題を対象とする C++のライブラリです[3]．問題は次の形式に定式化されているものとして，変数や制約式の上下限や係数行列を C++の配列として直接引数として取ります．入力の際にモデリング言語は用いません．

- （混合整数）線形計画問題（solveLP が対応）

$$
\begin{array}{lll}
\text{最小化・最大化} & \sum_j c_j \cdot x_j & j = 1, \cdots, m \\
\text{条件} & cu_i \geq \sum_j A_{i,j} \cdot x_j \geq cl_i & i = 1, \cdots, n \\
 & & j = 1, \cdots, m \\
 & bu_j \geq x_j \geq bl_j & j = 1, \cdots, m \\
 & (x_j \in Z) & (j \in I)
\end{array}
$$

- （混合整数）二次計画問題・二次制約付き問題（solveQP が対応）

$$
\begin{array}{lll}
\text{最小化・最大化} & \sum_j c_j \cdot x_j + \frac{1}{2} \sum_{j,k} Q_{j,k} \cdot x_j \cdot x_k & j = 1, \cdots, m \\
 & & k = 1, \cdots, m \\
 & & i = 1, \cdots, n \\
\text{条件} & cu_i \geq \sum_j A_{i,j} \cdot x_j + \frac{1}{2} \sum_{j,k} Q^i_{j,k} \cdot x_j \cdot x_k \geq cl_i & j = 1, \cdots, m \\
 & & k = 1, \cdots, m \\
 & bu_j \geq x_j \geq bl_j & j = 1, \cdots, m \\
 & (x_j \in Z) & (j \in I)
\end{array}
$$

3.1 呼び出し形式

以下がライブラリ関数 solveLP，solveQP の呼び出し形式です．"=0"は C++の記法のデフォルト引数（省略が可能な引数）を示しています．solveLP は整数変数が存在しないとき，ivtype 以降を，solveQP は制約式の二次の部分が存在しないとき，nQCelem 以降をそれぞれ一括して省略することができます．呼び出し形式はヘッダーファイル

```
nuoIf.h
```

に含まれています．このヘッダファイルは Numerical Optimizer ライブラリの include ファイルの保管

[3] solveQP は混合整数の二次制約付きの問題は扱うことができません．

場所

(Windows 版)：

（Numerical Optimizer のインストール場所）\userapp\include

(UNIX 版)：

（Numerical Optimizer のインストール場所）/userapp/include

にあります．なお，nuoIf.h には NAMESPACE_NUOPT integer という型が書かれていますが，int 型だと考えていただいて問題ありません．

- solveLP

solveLP の呼び出し形式

```
nuoptResult*
solveLP
(
  nuoptParam* options
  ,int n,int m
  ,int minimize
  ,double* x0
  ,double* bL,double* bU,int* ibL,int* ibU
  ,double* cL,double* cU,int* icL,int* icU
  ,double* objL
  ,int nAelem,int* irowA,int* jcolA,double* a
// 以下は整数変数が存在しない場合一括して省略可能
  ,int* ivtype = 0
  ,int* pri  = 0
  ,int* dir = 0
  ,int* until = 0
  ,double* upc = 0
  ,double* dpc = 0
);
```

- solveQP

solveQP の呼び出し形式

```
nuoptResult*
solveQP
(
  nuoptParam* options
```

```
  ,int n,int m
  ,int minimize
  ,double* x0
  ,double* bL,double* bU,int* ibL,int* ibU
  ,double* cL,double* cU,int* icL,int* icU
  ,double* objL
  ,int nAelem,int* irowA,int* jcolA,double* a
  ,int nQelem,int* irowQ,int* jcolQ,double* q
// 以下は制約式の二次の部分が存在しない場合一括して省略可能
  ,int nQCelem = 0,int* ifunQC = 0,int* irowQC = 0,int* jcolQC = 0
  ,double* qc = 0
// 以下は整数変数が存在しない場合一括して省略可能
  ,int* ivtype = 0
  ,int* pri = 0
  ,int* dir = 0
  ,int* until = 0
  ,double* upc = 0
  ,double* dpc = 0
);
```

solveQP は solveLP の機能を含んでおり，solveQP で二次の項の指定をすべて消去すると，solveLP と同一の機能となります.

3.2 ルーチン仕様

solveLP，solveQP の引数の意味は共通しています．以降で各引数の解説を行います.

3.2.1 問題全体にかかわるもの（solveLP，solveQP 共通）

nuoptParam*	options	Numerical Optimizer の求解制御パラメータ（0：デフォルト）
int	n	変数の総数
int	m	制約式の総数（等式，不等式含む）
int	minimize	最小化かどうかのフラグ（零以外：最小化，0：最大化）
double*	x0	変数の初期値（長さ：n）
double*	bL	変数の下限ベクトル（長さ：n）
double*	bU	変数の上限ベクトル（長さ：n）
int*	ibL	変数の下限の有無（長さ：n，非零：あり，0：なし）
int*	ibU	変数の上限の有無（長さ：n，非零：あり，0：なし）
double*	cL	制約式の下限ベクトル（長さ：m）

double*	cU	制約式の上限ベクトル（長さ：m）
int*	icL	制約式の下限の有無（長さ：m，非零：あり，0：なし）
int*	icU	制約式の上限の有無（長さ：m，非零：あり，0：なし）

ここで紹介する引数は solveLP，solveQP に共通で，同じ意味を持ちます．最初の引数は Numerical Optimizer の求解制御パラメータを与えるものですが，0 を渡すことができます．その場合にはパラメータとしてデフォルトの設定を用いるという意味になります．通常はデフォルトで問題ありませんが，パラメータの指定を行う場合には，

```
nuoptParam myparam;
myparam.method = "asqp"; // 求解メソッドの指定
solveQP(&myparam,...);
```

のようにして，myparam のメンバ（個別のパラメータに相当する）に値を設定して，solveQP にアドレスを渡します．詳細は「6.7Numerical Optimizer オプション」を参照ください．

変数の上下限が存在しない場合には，対応する ibL または ibU のコンポーネントを 0 にします．例えば n = 3 の場合で

$$0 \le x_1$$

$$x_2 \le 1$$

$$2 \le x_3 \le 3$$

という上下限を表現する場合には次のように設定します．

配列の添字	bL	bU	ibL	ibU
0	0	任意	0 以外	0
1	任意	1	0	0 以外
2	2	3	0 以外	0 以外

C++では配列の添字は 0 はじまりですので，1 番目の変数は添字 0 の場所に対応します．配列の「任意」と書かれた場所は無視されます．「0 以外」という場所は 0 以外の任意の値です．制約式についても全く同様です．

変数の固定，および等式制約は上限と下限を一致させることによって表現します．ibL，ibU，icL，icU の場所に NULL ポインタを渡すことが許されています．その場合，すべて 0 が格納された配列を渡すのと同じ意味になります．

初期値に対応する x0 に NULL ポインタを渡すことができます．その場合には初期値はデフォルトのものを用いるという意味になります．

3.2.2　線形部分にかかわるもの（solveLP，solveQP 共通）

double*	objL	目的関数の線形部分（長さ：n）
int	nAelem	制約式の係数行列の非零要素数
int*	irowA	非零要素の行番号（長さ：nAelem）
int*	jcolA	非零要素の列番号（長さ：nAelem）
double*	a	非零要素の値（長さ：nAelem）

ここで扱う引数も solveLP，solveQP で同じ意味を持ちます．objL は目的関数の線形部分

$$\sum_j c_j \cdot x_j$$

の c_j に対応します．

制約式の係数行列は非零要素のみ与えます．同じ非零要素を二つ以上与えた場合には，それらの和が取られます．非零要素の場所は 1 始まりの番号で与えます．例えば nAelem=4 のとき，

配列の添字	irowA	jcolA	a
0	1	2	2.3
1	**1**	**3**	1.8
2	2	1	0.5
3	**1**	**3**	0.2

のように与えると

$$A = \begin{bmatrix} 0 & 2.3 & 2.0 \\ 0.5 & 0 & 0 \end{bmatrix}$$

を与えたことになります（A_{13} については二つ与えられているので和（1.8+0.2=2.0）が取られます）．非零要素の格納順番は任意です．

3.2.3 混合整数計画問題にかかわるもの（solveLP，solveQP 共通：一括省略可）

int*	ivtype	変数の種別（長さ：n，0:連続，1：整数，2：バイナリ）
int*	pri	変数の分枝優先度（長さ：n，0 より小さければ無視，小さい値のものほど優先）
int*	dir	変数の優先される分枝方向（長さ：n，正の値：押し上げ，負の値：押し下げ，0：指定なし）
int*	until	NULL ポインタを設定します
double*	upc	変数の押し上げ擬コスト（長さ：n，負なら無視，正の値）
double*	dpc	変数の押し下げ擬コスト（長さ：n，負なら無視，正の値）

ここに挙げた引数で整数変数の指定を行います．ivtype が変数の種別を与えるベクトルで，次の意味を持ちます．

0：連続変数

1：一般の整数変数
2：バイナリ変数（0 または 1）

ivtype 自体を NULL ポインタにするとすべての変数が連続変数であるという指定と同じ意味となります．until は Numerical Optimizer V15 より廃止されましたので，NULL ポインタにします．

pri，upc，dpc，dir はモデリング言語 SIMPLE を用いた場合に IntegerVariable のメンバとして与えられるものと同じで以下の意味を持ちます．

- 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のみ，Numerical Optimizer 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（分枝優先方向）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 Numerical Optimizer がどちらを優先するかを決定します．

pri，dir，upc，dpc はすべて NULL ポインタとすることができます．その場合にはこれらについて指定をしないのと同じ意味となります．整数変数が存在しない場合には，ivtype から以降の引数をすべて省略することができます（一括省略）．

3.2.4　目的関数の二次の部分にかかわるもの（solveQP のみ）

int	nQelem	目的関数のヘッセ行列の非零要素数
int*	irowQ	目的関数のヘッセ行列の行番号（長さ：nQelem）
int*	jcolQ	目的関数のヘッセ行列の列番号（長さ：nQelem）
double*	q	目的関数のヘッセ行列の非零要素の値（長さ：nQelem）

ここに挙げた引数で目的関数の二次の部分の係数行列（ヘッセ行列）

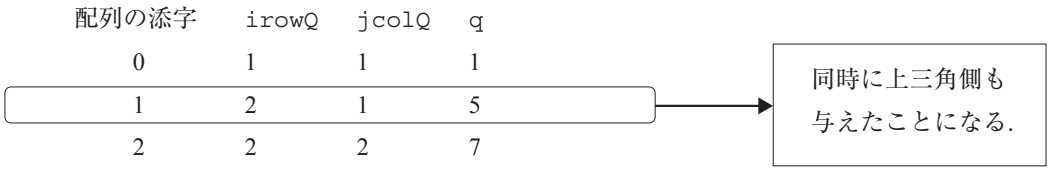
$$\frac{1}{2}\sum_{j,k} Q_{j,k}\cdot x_j \cdot x_k$$

の $Q_{i,j}$ を与えます．制約式の係数行列と同様に非零要素のみを与えます．指定の仕方は制約式の係数行列と同様です．非零要素の場所は 1 始まりの番号で与えます．しかし，目的関数のヘッセ行列は対称行列であることを前提としていますので，下三角部分の非零要素を与えると同時に上三角部分も与えたことになる（その逆も同じ）という点に注意してください．

例えば

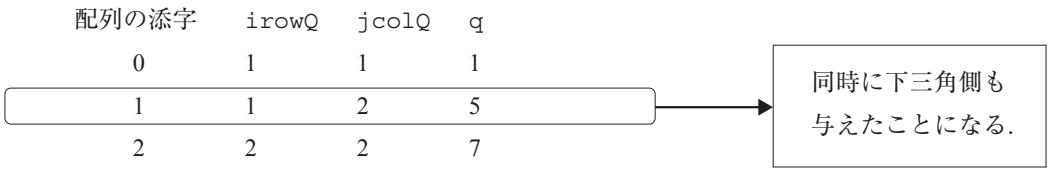
$$Q = \begin{bmatrix} 1 & 5 \\ 5 & 7 \end{bmatrix}$$

というヘッセ行列を定義するためには nQelem = 3 として

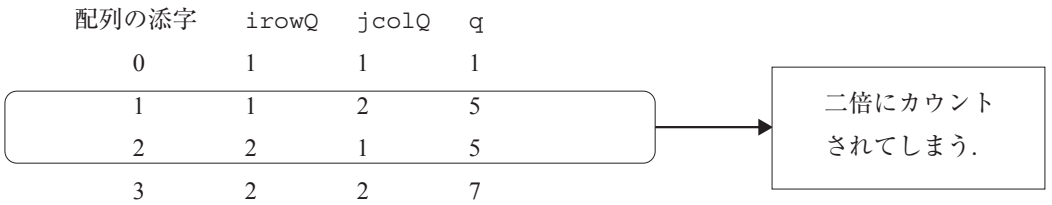


のように与えます.

あるいは以下のようにしても同じ意味です.



同じ非零要素を二つ与えるとその和が取られます．従って nQelem=4 として



とすると，上の下三角部分の非零要素を与えると同時に上三角部分も与えたことになるという原則（その逆も同じ）によって

$$Q = \begin{bmatrix} 1 & 10 \\ 10 & 7 \end{bmatrix}$$

を定義したことになりますのでご注意ください．また，非零要素の格納順番は任意です.

nQelem=0 とすると，目的関数に二次の部分が存在しないものと解釈されます．その場合には irowQ, jcolQ，q はすべて NULL ポインタとすることができます.

3.2.5 制約式の二次の部分にかかわるもの（solveQP のみ，一括省略可）

int	nQCelem	制約式のヘッセ行列の非零要素数
int*	ifunQC	制約式のヘッセ行列の非零要素が属する制約式番号（長さ：nQCelem）
int*	irowQC	制約式のヘッセ行列の行番号（長さ：nQCelem）
int*	jcolQC	制約式のヘッセ行列の列番号（長さ：nQCelem）
double*	qc	制約式のヘッセ行列の非零要素の値（長さ：nQCelem）

これらの引数で制約の二次部分

$$\frac{1}{2}\sum_{j,k} Q^i_{j,k} \cdot x_j \cdot x_k \quad (i \text{ は制約式の番号})$$

の係数行列（ヘッセ行列）の群

$$Q^i_{j,k}$$

を与えます．非零要素と，その非零要素が属する制約式の番号のみを与えます．

例えば，制約式 1，2 の二次の部分がそれぞれ

$$Q^1 = \begin{bmatrix} 4 & 3 \\ 3 & 6 \end{bmatrix},\ Q^2 = \begin{bmatrix} 8 & 2 \\ 2 & 3 \end{bmatrix}$$

である場合には `nQCelem = 6` として

配列の添字	ifunQC	irowQC	jcolQC	qc
0	1	1	1	4
1	1	1	2	3
2	1	2	2	6
3	2	1	1	8
4	2	1	2	2
5	2	2	2	3

と設定します．制約式は 1 始まりの番号で指定します．また，行列の非零要素の場所も 1 始まりの番号で指定します．

制約式のヘッセ行列も対称であることを前提としているので，目的関数のヘッセ行列と同じく下三角部分の非零要素を与えると同時に上三角部分も与えたことになる（その逆も同じ）という原則が適用されます．また，目的関数の二次部分のヘッセ行列と同じく，同一の非零要素が二つ以上与えられると和が取られます．

非零要素の格納順番は任意です．

`nQCelem=0` とすると，制約式に二次の部分が存在しないものと解釈されます．その場合には `ifunQC`，`irowQC`，`jcolQC`，`qc` はすべて NULL ポインタとすることができます．

制約式の二次の部分が存在しない場合には，`nQCelem` から以降の引数をすべて省略することができます．

3.2.6　出力とエラーメッセージ

`solveLP`，`solveQP` の戻り値として，`nuoptResult` というオブジェクトのポインタが返ります．このオブジェクトのメンバに実行結果についての情報が格納されています．利用可能なメンバは以下の通りです．このメンバの利用の詳細は次項で解説するサンプルコード `useSolveQP.cpp(.cc)` をご覧ください．このオブジェクトは利用するコード内で開放（`delete`）する必要があります．

```
int errorCode();          エラーコード
char* errorMessage();     エラーメッセージ
```

`double optValue();`	最適値
`double VarVal(int i);`	i 番目の変数の値
`double VarDual(int i);`	i 番目の変数の dual 値
`double FuncVal(int i);`	i 番目の制約の値
`double FuncDual(int i);`	i 番目の制約の dual 値

上記で i は変数，制約式のインデクスですが，0 始まりであることにご注意ください（最初の変数/制約式が 0 番目）.

エラーコードとエラーメッセージは Numerical Optimizer 本体のものと同じです.「Numerical Optimizer/SIMPLE マニュアル」の付録 A をご参照ください．また，solveLP，solveQP 特有のエラーとして，引数の矛盾や範囲オーバーがありますが，それは 151 から 165 のコード番号に対応します[4]．意味は以下のとおりです.

コード	意味
151	n が負
152	m が負
153	変数の下限（bL）が上限（bU）よりも大きい
154	制約式の下限（cL）が上限（cU）よりも大きい
155	nAelem が負
156	irowA のコンポーネントの範囲が違反
157	jcolA のコンポーネントの範囲が違反
158	nQelem が負
159	irowQ のコンポーネントの範囲が違反
160	jcolQ のコンポーネントの範囲が違反
161	nQCelem が負
162	ifunQC のコンポーネントの範囲が違反
163	irowQC のコンポーネントの範囲が違反
164	jcolQC のコンポーネントの範囲が違反
165	ivtype の範囲が違反

エラーメッセージは

```
(NUOPT 156) irowA[0] = 9 should be in [1,3]
```

のように，実際のデータに即したより詳しい情報を含んでいます.

3.3 実行サンプル

開発環境と同梱されているサンプル useSolveQP.cpp(.cc) は solveLP，solveQP を利用するサンプ

[4] Numerical Optimizer V18 からコード番号が変更となりました.

ルです．線形計画問題なら，solveLP を，二次計画問題なら，solveQP を呼びます．

- **VC++**

> **注意：**
> 以降では，サンプルを用いた実行について書かれています．読み進む前に，まず，お使いのコンパイラを確認してください．サンプルは
>
> %NUOPT%\samples\app
>
> （%NUOPT%は Numerical Optimizer のインストール場所，たとえば C:\Program Files\Mathematical Systems Inc\nuopt）
> にある
> - app_VS2010.zip（VS2010 用）
> - app_VS2012.zip（VS2012 用）
> - app_VS2013.zip（VS2013 用）
> - app_VS2015.zip（VS2015 用）
>
> のうち，お使いのコンパイラに整合するものを展開してご利用ください．なお，マシンの設定によっては zip ファイルのある場所に書き込み権限が無いため展開できないことがございます．この場合は書き込み権限があるフォルダに展開してください．また，コンパイラとプロジェクトが整合していない場合，リンクエラーやコンパイルエラーが生じてしまいます．

ソリューション nuoptvcapp のプロジェクト

useSolveQP

がこのインタフェースを用いて LP，QP を解くプログラム例です．

- **Windows・UNIX 共通**

useSolveQP.cpp(.cc) は solveLP，solveQP を利用するサンプルで，1 つの main 関数のみから成ります．ロードモジュールの引数名で与えられたファイルから問題のデータを読み込み，それを solveLP，solveQP に渡します．二次の項がなく，線形計画問題ならば solveLP を，あれば solveQP を呼びます．以下はそのプログラムの抜粋です．

useSolveQP.cpp(.cc)

```
//
//  solveQP の利用例
//
#include "nuoIf.h" // 必須.
int main(int argc, char** argv)
{
```

```
  int n;
  int m;
  ifstream inputFile(argv[1]);  // ロードモジュールの引数をファイル名とする
  inputFile >> n >> m ;

     (中略)

  nuoptResult* qpres = 0;
  nuoptParam myParam;
  if ( nQelem || nQCelem ) { // 2 次の係数があるなら solveQP をコール
    qpres = solveQP(&myParam
    ,n,m
    ,minimize
    ,x0
    ,bL,bU,ibL,ibU
    ,cL,cU,icL,icU
    ,objL
    ,nAelem,irowA,jcolA,a
    ,nQelem,irowQ,jcolQ,q
    ,nQCelem,ifunQC,irowQC,jcolQC,qc
    ,ivtype,pri,dir,until,upc,dpc
  );
  } else {// 2 次の係数がないのなら solveLP をコール
    qpres = solveLP(&myParam
    ,n,m
    ,minimize
    ,x0
    ,bL,bU,ibL,ibU
    ,cL,cU,icL,icU
    ,objL
    ,nAelem,irowA,jcolA,a
    ,ivtype,pri,dir,until,upc,dpc
  );
}

  if ( qpres->errorCode() ) {
    printf("error in solveQP code = %d, message = %s\n"
    ,qpres->errorCode(),qpres->errorMessage());
```

```
  fflush(stdout);
  exit(1);
  } else {
    printf("optimalValue = %17.10e\n",qpres->optValue());
    printf("X:\n");
    for ( i = 0 ; i < n ; ++i ) {
    printf("[%3d] %10.3e ",i+1,qpres->VarVal(i));
    if ( (i+1) % 4 == 0 ) {
      printf("\n");
    }
  }
  printf("\n");
  printf("F:\n");
  for ( i = 0 ; i < m ; ++i ) {
    printf("[%3d] %10.3e ",i+1,qpres->FuncVal(i));
    if ( (i+1) % 4 == 0 ) {
      printf("\n");
    }
  }
    printf("\n");
  }

  delete qpres;
  delete [] bL;
  delete [] bU;
  delete [] ibL;
  delete [] ibU;

   （後略）
}
```

次のようなナップサック問題を考えます.

変数	$x_i \in \{0,1\} \quad (i \in S)$
目的関数（最大化）	$\sum_{i \in S} c_i x_i,$
制約条件	$\sum_{i \in S} a_i x_i \leq b$
目的関数の係数：	$c = (42 \quad 12 \quad 45 \quad 5 \quad 2 \quad 61 \quad 89 \quad 32 \quad 47 \quad 18)$
制約式の係数：	$a = (39 \quad 13 \quad 68 \quad 15 \quad 10 \quad 20 \quad 31 \quad 15 \quad 41 \quad 16)$
制約式の右辺：	$b = 121$

これを solveLP で解くには，一般の線形計画問題：

最小化・最大化	$\sum_j c_j \cdot x_j$	$j = 1, \cdots, m$
条件	$cu_i \geq \sum_j A_{i,j} \cdot x_j \geq cl_i$	$i = 1, \cdots, n$ $j = 1, \cdots, m$
	$bu_j \geq x_j \geq bl_j$	$j = 1, \cdots, m$
	$(x_j \in Z$	$j \in I)$

の形にこの問題を表現する，すなわち

$c = (42 \quad 12 \quad 45 \quad 5 \quad 2 \quad 61 \quad 89 \quad 32 \quad 47 \quad 18)$

$Q = 0$

$cu = (121)$

$cl = (-\infty)$

$A = (39 \quad 13 \quad 68 \quad 15 \quad 10 \quad 20 \quad 31 \quad 15 \quad 41 \quad 16)$

$x_j \in \{0,1\}$（バイナリ変数）

とすればよいことがわかります．useSolveQP.cpp(.cc) の入力ファイルとしてこのデータを表現したのがファイル

(VC++版)：

（Numerical Optimizer のインストール場所）\samples\app にある zip ファイルを解凍した中にある useSolveQP\knapsack.txt

(UNIX 版)：

（Numerical Optimizer のインストール場所）/userapp/knapsack.txt

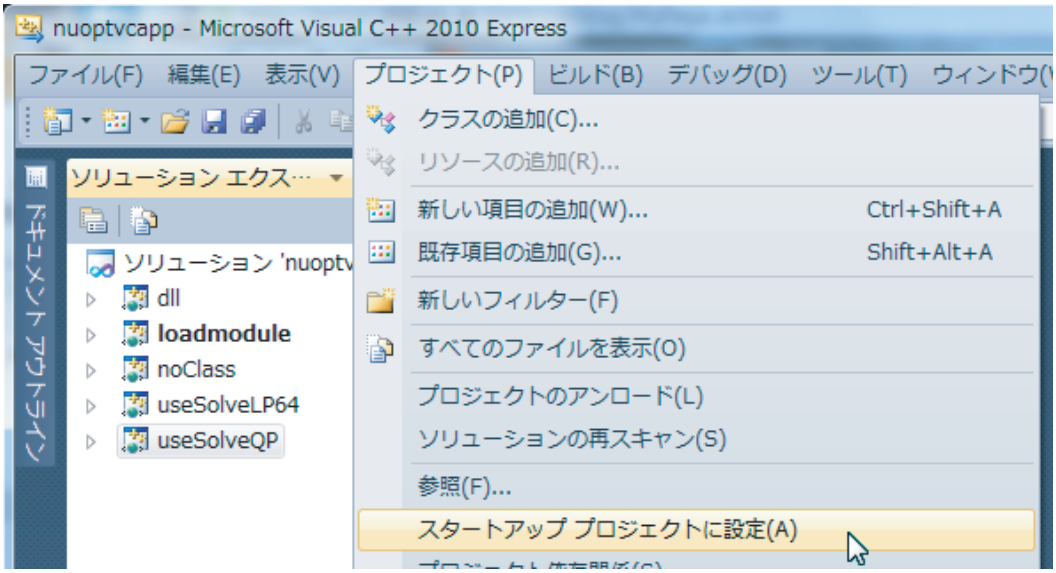
です．

3.4 実行例

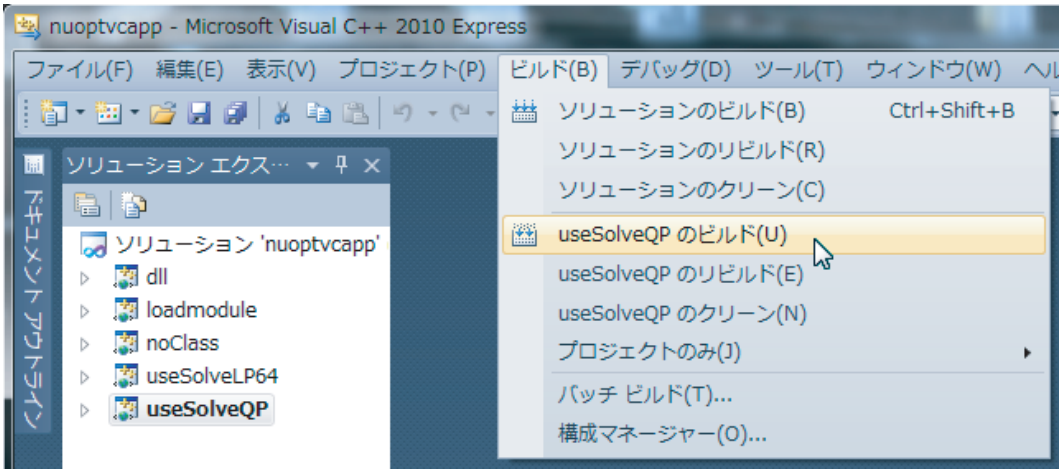
さて，サンプルコード useSolveQP の実行モジュールを作成して実行してみましょう．

- **VC++**

  プロジェクト「useSolveQP」を選択後，VC++のメニューの「プロジェクト」から「スタートアッププロジェクトに設定」を選び，プロジェクトを選択します．なお useSolveQP の構成は「Release」に設定してあります．

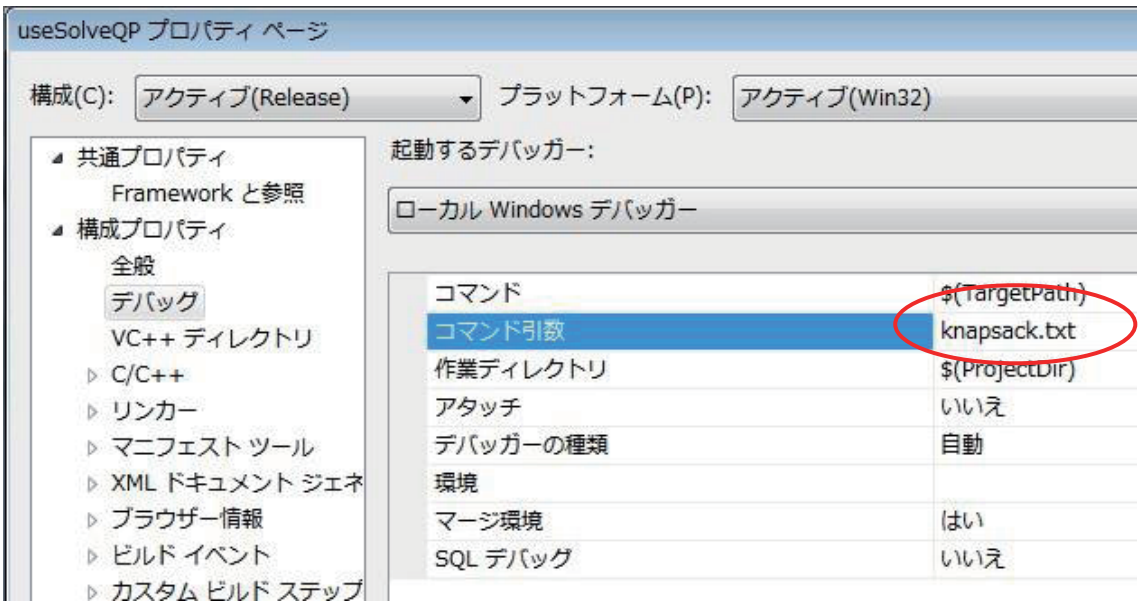


続いて「ビルド」メニューから「useSolveQP のビルド」:



を選択すると useSolveQP.cpp がコンパイル，Numerical Optimizer のライブラリとリンクされて，実行モジュールの useSolveQP.exe が作成されます[5]．引数として knapsack.txt を与えるように「プロジェクト」→「プロパティ」→「構成プロパティ」→「デバッグ」の項目で，「コマンド引数」に「knapsack.txt」と入力します．

[5]「ビルド」メニューが表示されていない場合は「ツール」→「設定」にある「上級者用の設定」をクリックしてください（Visual C++ 2010 Express Edition の場合）．



つづいて「デバッグ」メニューから

と実行すると，実行経過表示ウィンドウが現れ，その中に以下のような出力が得られます．solveLP がコールされて出力が成されます．

```
[About Numerical Optimizer]
MSI Numerical Optimizer x.x.x  (NLP/LP/IP/SDP module)
         <with META-HEURISTICS engine "wcsp"/"rcpsp">
         <with GLOBAL-OPTIMIZATION add-on "global">
         <with DERIVATIVE-FREE-OPTIMIZATION add-on "DFO">
        , Copyright (C) 1991 NTT DATA Mathematical Systems Inc.

[Problem and Algorithm]
PROBLEM_NAME                                      anon.LP
NUMBER_OF_VARIABLES                                    10
(#INTEGER/DISCRETE)                                    10
NUMBER_OF_FUNCTIONS                                     2
PROBLEM_TYPE                                  MAXIMIZATION
METHOD                                             SIMPLEX
```

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

#1                  lo:          -0                  time:  0.0s:mem(MB)=3/6:ava
il(MB)=1991/1991
                                                     llen:0 #prob:0 #piv:0

=== begin wcsp ===
# (hard/soft) = 0/111
# iteration = 1000
# time =  0.00 (s), succ = 1
=== end wcsp ===
#2                  lo:         242                  time:  0.1s:mem(MB)=2/6:ava
il(MB)=1991/1991
                                                     llen:0 #prob:0 #piv:0

----- breakdown -----
BB        :     0.0(sec), call=1, succ=2
wcsp      :     0.0(sec), call=1, succ=1
_____ breakdown _____

<iteration end>

[Result]
STATUS                                              OPTIMAL
VALUE_OF_OBJECTIVE                                      242
SIMPLEX_PIVOT_COUNT                                       0
RESIDUAL                                                  1
ELAPSED_TIME(sec.)                                     0.07
SOLUTION_FILE                                     solver.sol
optimalValue = 2.4200000000e+002
X:
[  1] 1.000e+000 [  2] 0.000e+000 [  3] 0.000e+000 [  4] 0.000e+000
[  5] 0.000e+000 [  6] 1.000e+000 [  7] 1.000e+000 [  8] 1.000e+000
[  9] 0.000e+000 [ 10] 1.000e+000
F:
```

```
[  1] 1.210e+002
続行するには何かキーを押してください . . .
```

上に挙げた内容は Numerical Optimizer からの求解に関する出力です．問題の変数の数（NUMBER_OF_VARIABLES）や目的関数の値（VALUE_OF_OBJECTIVE）を知ることができます．Numerical Optimizer の標準出力の内容については「Numerical Optimizer/SIMPLE マニュアル」（別冊）をご覧ください．

Numerical Optimizer の出力を抑制するには，

```
nuoptParam myParam; // 宣言
```

として，パラメータを宣言したのち，

```
myParam.outputMode = "silent";
```

とします．また，デフォルトで出力される Numerical Optimizer の求解レポートである解ファイル solver.sol の出力を抑制するには

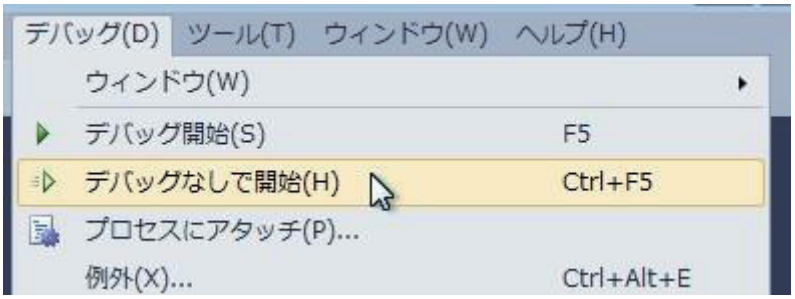
```
myParam.outfilename = "_NULL_";
```

と設定し，solveQP() の最初の引数として渡します．useSolveQP.cpp(.cc) の以下のコメントを取ると，その設定になり，Numerical Optimizer からの出力が抑制されて，useSolveQP が出力する表示のみになります．

出力を抑制する例

```
       ..
       nuoptParam myParam;
    // Numerical Optimizer の出力と解ファイルの出力を抑制する．
    myParam.outputMode = "silent";
    myParam.outfilename = "_NULL_";
if ( nQelem || nQCelem ) { // 2 次の係数があるのなら solveQP
     qpres = solveQP(&myParam
             ,n,m
     ..
```

再びビルドメニューから



とすると，コンパイルが自動的に実行されます．

- **UNIX**

```
Prompt% make useSolveQP
```

とすると，useSolveQP.cc がコンパイル，Numerical Optimizer のライブラリとリンクされて useSolveQP が作成されます．引数として knapsack.txt を与えてロードモジュールを実行します．次は，ナップサック問題に対応する knapsack.txt をこの設定で実行したときの出力です．

```
Prompt% ./useSolveQP knapsack.txt
...
optimalValue = 2.4200000000e+002
X:
[  1] 1.000e+000 [  2] 0.000e+000 [  3] 0.000e+000 [  4] 0.000e+000
[  5] 0.000e+000 [  6] 1.000e+000 [  7] 1.000e+000 [  8] 1.000e+000
[  9] 0.000e+000 [ 10] 1.000e+000
F:
[  1] 1.210e+002
prompt%
```

- **Windows・UNIX 共通**

knapsack.txt と同じ場所に別の useSolveQP.cpp(.cc) の入力ファイル

(VC++版)：

（Numerical Optimizer のインストール場所)\samples\app にある zip ファイルを解凍した中にある useSolveQP\qp312.txt

(UNIX 版)：

（Numerical Optimizer のインストール場所)/userapp/qp312.txt

があります．このデータは二次計画問題で

$$
\begin{array}{lll}
\text{最小化・最大化} & \displaystyle\sum_j c_j \cdot x_j + \frac{1}{2}\sum_{(j,k)} Q_{j,k} \cdot x_j \cdot x_k & j = 1,\cdots,m \\
 & & k = 1,\cdots,m \\
 & & i = 1,\cdots,n \\
\text{条件} & cu_i \geq \displaystyle\sum_j A_{i,j} \cdot x_j + \frac{1}{2}\sum_{j,k} Q^i_{j,k} \cdot x_j \cdot x_k \geq cl_i & j = 1,\cdots,m \\
 & & k = 1,\cdots,m \\
 & bu_j \geq x_j \geq bl_j & j = 1,\cdots,m \\
 & (x_j \in Z & j \in I)
\end{array}
$$

という定式で，次のように設定したものに対応します．2 変数，3 制約の問題で，制約式には二次の項はありません．

$$c = (-3 \quad 1)$$
$$Q = \begin{pmatrix} 11 & 0 \\ 0 & 22 \end{pmatrix},\ Q^i = 0$$
$$cu = (1000 \quad 1000 \quad 1000)$$
$$cl = (-1 \quad -2 \quad 2)$$
$$A = \begin{pmatrix} -1 & 0.1 \\ -0.2 & -1 \\ 2 & 1 \end{pmatrix}$$
$$bu = (1 \quad 2)$$
$$bl = (0 \quad 0)$$

(VC++版)：

VC++の GUI で「プロジェクト」→「設定」→「デバッグ」タブで,「プログラムの引数」を

```
qp312.txt
```

として再び実行してみてください.

(UNIX 版)：

```
prompt% ./useSolveQP qp312.txt
```

とします.

今度は useSolveQP が solveQP をコールして，次のような解が出力されます.

```
optimalValue = 2.3181818248e+000
X:
[  1] 9.394e-001 [  2] 1.212e-001
F:
[  1] -9.273e-001 [  2] -3.091e-001 [  3] 2.000e+000
```

次のデータ (Windows 版のみ)：

(Numerical Optimizer のインストール場所)\samples\app にある zip ファイルを解凍した中にある useSolveQP\qp312I.txt

はこの二次計画問題の変数を連続変数ではなく，整数変数という指定をつけたもので，二次の整数計画問題となります．これを入力として実行すると

```
optimalValue = 2.5000000000e+000
X:
[  1] 1.000e+000 [  2] 0.000e+000
F:
[  1] -1.000e+000 [  2] -2.000e-001 [  3] 2.000e+000
```

のような解が出力されます．
さらに，次のデータ (Windows 版のみ)：

(Numerical Optimizer のインストール場所)\samples\app にある zip ファイルを解凍した中にある useSolveQP\hs23.txt

は制約条件の中に二次式が含まれる問題となります．なお，このデータは [1] の No.23 を記述したものであり，次のような設定に対応します．

$$c = (0 \quad 0)$$

$$Q = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix},$$

$$Q^1 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix},\ Q^2 = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix},\ Q^3 = \begin{pmatrix} 18 & 0 \\ 0 & 2 \end{pmatrix},\ Q^4 = \begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix},\ Q^5 = \begin{pmatrix} 0 & 0 \\ 0 & 2 \end{pmatrix}$$

$$cu = (\infty \quad \infty \quad \infty \quad \infty \quad \infty)$$

$$cl = (1 \quad 1 \quad 9 \quad 0 \quad 0)$$

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & -1 \\ -1 & 0 \end{pmatrix}$$

$$bu = (50 \quad 50)$$

$$bl = (-50 \quad -50)$$

これを入力として実行すると

```
optimalValue = 2.0000000009e+000
X:
[  1] 1.000e+000 [  2] 1.000e+000
F:
[  1] 2.000e+000 [  2] 2.000e+000 [  3] 1.000e+001 [  4] 2.133e-010
[  5] 2.133e-010
```

のような解が出力されます．

3.5 大規模な線形計画問題を内点法で解くためのライブラリ solveLP64

制約式の係数行列の非零要素数が 32bit 整数型で表せないような巨大な線形計画問題を内点法を用いて求解する場合は solveLP64 を使用します．呼び出し形式は

```
solveLP64.h
```

に含まれています．このヘッダファイルは Numerical Optimizer ライブラリの include ファイルの保管

場所

(Windows 版)：

(Numerical Optimizer のインストール場所)\userapp\include

(UNIX 版)：

(Numerical Optimizer のインストール場所)/userapp/include

にあります.

- solveLP64

solveLP64 の呼び出し形式

```
nuoptResult* solveLP64
(
 nuoptParam* options // Numerical Optimizer の入力パラメータ
,int64_t n // 変数の総数
,int64_t m // 制約式総数
,int64_t minimize // 1 で最小化，0 で最大化
,double* x0 // 変数の初期値
,double* bL // 変数の下限ベクトル
,double* bU // 変数の上限ベクトル
,int64_t* ibL // 変数の下限の有無
,int64_t* ibU // 変数の上限の有無
,double* cL // 制約式の下限ベクトル
,double* cU // 制約式の上限ベクトル
,int64_t* icL // 制約式の下限の有無
,int64_t* icU // 制約式の上限の有無
,double* objL // 目的関数の線形部分
,int64_t nAelem // 制約式の係数行列の非零要素数
,int64_t* irowA // 非零要素の行番号
,int64_t* jcolA // 非零要素の列番号
,double* a // 非零要素の値
);
```

本 solveLP64 は Windows では VS2010 以降の 64bit コンパイラ，Unix では gcc4.8 以降から使用できます.

第4章 SIMPLEモデルとドライバを分離しない例

本章で紹介するのはSIMPLEによって記述したモデルをプログラムの一部として呼び出す方法です．この方法は次章のクラスを生成する方法と比べて簡便ですがNumerical Optimizerのライブラリが，リンクされているプログラム全体で同時に一つの問題のみしか定義できないという制限があります．

- **VC++**

> **注意：**
> 以降では，サンプルを用いた実行について書かれています．読み進む前に，まず，お使いのコンパイラを確認してください．サンプルは
>
> `%NUOPT%\samples\app`
>
> （`%NUOPT%`は Numerical Optimizer のインストール場所，たとえば `C:\Program Files\Mathematical Systems Inc\nuopt`）
> にある
> - `app_VS2010.zip`（VS2010用）
> - `app_VS2012.zip`（VS2012用）
> - `app_VS2013.zip`（VS2013用）
> - `app_VS2015.zip`（VS2015用）
>
> のうち，お使いのコンパイラに整合するものを展開してご利用ください．なお，マシンの設定によってはzipファイルのある場所に書き込み権限が無いため展開できないことがございます．この場合は書き込み権限があるフォルダに展開してください．また，コンパイラとプロジェクトが整合していない場合，リンクエラーやコンパイルエラーが生じてしまいます．

ソリューション `nuoptvcapp` のプロジェクト

`noClass`

が本方法に対応するプログラム例です．

- **UNIX**

この例に対応するのが，ファイル

（Numerical Optimizer のインストール場所）`/userapp`

/useSimple.cc

/MIP.cc

です.

VC++版ライブラリをお使いの方は

4.1 VC++版ライブラリ（SIMPLE モデルとドライバを分離しない例）へ

UNIX 版ライブラリをお使いの方は

4.2 UNIX 版ライブラリ（SIMPLE モデルとドライバを分離しない例）へ

お進みください.

4.1 VC++版ライブラリ（SIMPLE モデルとドライバを分離しない例）

4.1.1 モデル兼ドライバの記述（VC++）

このプロジェクトでは noClass.cpp というソースファイルにモデルとモデルを利用する手続きが両方書き込まれています．noClass.cpp では knapsackSolve という名前の手続きを定義しています．この手続きは入力として，ナップサック問題のモデルの

```
Parameter a(index=i);
Parameter b;
Parameter c(index=i);
```

に相当するデータ（aarg，barg，carg）を与えると

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

に相当する変数値（xarg）と目的関数（farg）を返します.

noClass.cpp

```
#include "simple.h"
//
//   ナップサック問題（モデルの記述と実行を分離しない）
//
int knapsackSolve(int narg,double* aarg,double barg,double* carg
       ,double* xarg,double* farg)
{
  //
  // システムの初期化.
  //
```

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

  IntegerVariable x(name="決定ベクトル",index=i,type=binary); // 整数変数
  Parameter c(name="価値",index=i);
  Parameter a(name="重量",index=i);
  Parameter b(name="許容重量");
  Objective obj(name="総価値",type=maximize);

  // 引数からデータを設定
  a.readD(narg,aarg);
  b = barg;
  c.readD(narg,carg);

  sum(a[i]*x[i],i) <= b;     // 制約条件
  obj = sum(c[i]*x[i],i);    // 目的関数

  // Numerical Optimizer からの最適化結果ファイル.sol の
  // 名前の設定
  //    options.outfilename = "nuoptout";
  // .sol のファイル出力を抑制する場合には次のように書く
  options.outfilename = "_NULL_";
  // 出力を抑制する
  options.outputMode = "silent";

  // 最適化の実行
  solve();

  // x の内容を C++の配列にダンプ

  int lenx;
  int* indx;
  double* valx;
  x[i].val.dump(lenx,indx,valx);

  if ( lenx != narg) {
```

```
      return 99; // x[i] の実際の長さが narg と異なる（データに矛盾あり）
    }

    // 引数に設定
    int it;
    for ( it = 0 ; it < lenx ; ++it ) {
      xarg[indx[it]-1] = valx[it];
      // indx[it] は 1,2, ... narg なので，1 を引く
    }

    // 目的関数値の設定
    *farg = result.optValue;

    // indx,valx は dump 内部で独自に allocate されるので free しておく．
    delete [] indx;
    delete [] valx;
  }
  // Numerical Optimizer のエラーコードを返す．
  return result.errorCode;
}
```

このインタフェースを用いる場合，モデルはプログラム全体で 1 つだけ定義することが可能です．変数，制約式は定義した場所によらずにそのモデルに追加されてゆくことになります．SimpleInitialize() のコールより後と，SimpleClearBuffer() のコールより前は中括弧{ }でくくる必要があります．SIMPLE のオブジェクト（Set や Element）の宣言は，この中括弧{ }の中で行います．SimpleInitialize() をコールせずに SIMPLE のオブジェクトを宣言して利用する，或いはこの中括弧{}の外で SIMPLE のオブジェクトを宣言すると実行時エラーとなります．

4.1.2　モデル兼ドライバのコール（VC++）

次は knapsackSolve を C++で呼び出しているメインルーチンを記述します．ここでは，第 2 章と同様にナップサック問題

$$\text{目的関数の係数：}\quad c = (6 \quad 8 \quad 4 \quad 3 \quad 4)$$
$$\text{制約式の係数：}\quad a = (4 \quad 2 \quad 3 \quad 6 \quad 7)$$

の容量制約の上限値を 1 から 30 の間で動かすような問題を考えます．このルーチンもプロジェクト noClass に追加されています．

main.cpp

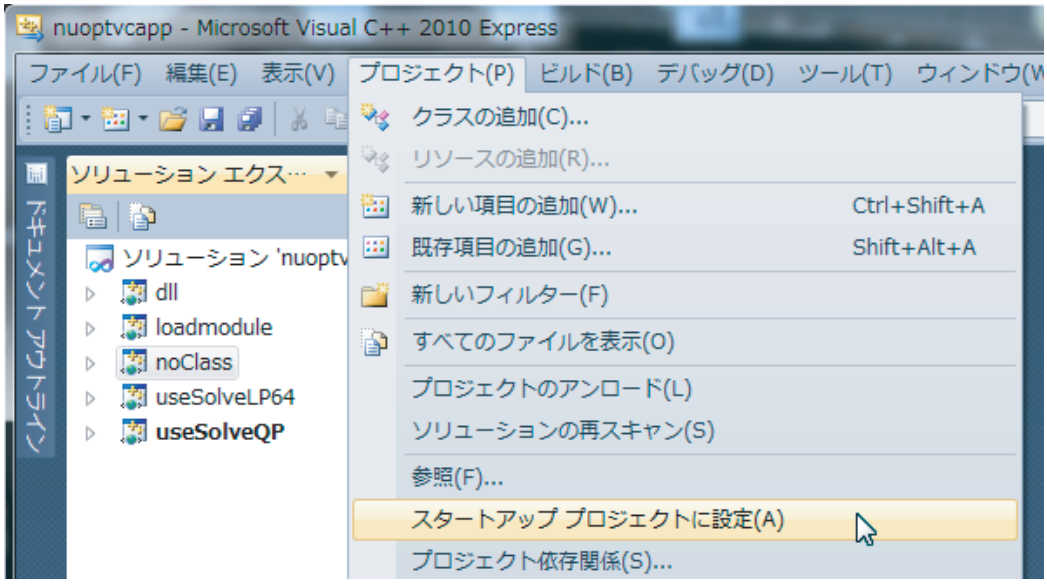
```
#include <stdio.h>

// main() の中でコールする最適化手続き（最適化ライブラリ）
int knapsackSolve(int narg,double* aarg,double barg,double* carg
        ,double* xarg,double* farg);
void main()
{
// 問題のデータ
  int narg = 5;
  double aarg[5] = {4,2,3,6,7};
  double carg[5] = {6,8,4,3,4};
  double xarg[5];
  double barg = 20;
  double farg;

  int errCode;

  // barg を変化させながら knapsackSolve を繰り返しコールする
  int ib;
  for ( ib = 1 ; ib <= 30; ++ib ) {
    barg = (double) ib;
    errCode = knapsackSolve(narg,aarg,barg,carg,xarg,&farg);
    printf("errCode=%d b=%g obj=%g ",errCode,barg,farg);
    printf(" x= ");
    int i;
    for ( i = 0 ; i < narg ; ++i ) {
      printf("%1.0f ",xarg[i]);
    }
    printf("\n");
  }
}
```

これを前項の knapsackSolve() と Numerical Optimizer のライブラリとリンクするとロードモジュールが作成できます．VC++のメニューの「プロジェクト」から「スタートアップ プロジェクトに設定」を選び，次のようにこのプロジェクトを選択します．



つづいて VC++GUI の実行を選択：

次のような出力が得られます.

```
errCode=0 b=1 obj=0  x= 0 0 0 0 0
errCode=0 b=2 obj=8  x= 0 1 0 0 0
errCode=0 b=3 obj=8  x= 0 1 0 0 0
errCode=0 b=4 obj=8  x= 0 1 0 0 0
errCode=0 b=5 obj=12  x= 0 1 1 0 0
errCode=0 b=6 obj=14  x= 1 1 0 0 0
     （中略）
errCode=0 b=26 obj=25  x= 1 1 1 1 1
errCode=0 b=27 obj=25  x= 1 1 1 1 1
errCode=0 b=28 obj=25  x= 1 1 1 1 1
errCode=0 b=29 obj=25  x= 1 1 1 1 1
errCode=0 b=30 obj=25  x= 1 1 1 1 1
続行するには何かキーを押してください . . .
```

4.2 UNIX 版ライブラリ（SIMPLE モデルとドライバを分離しない例）

4.2.1 モデル兼ドライバの記述（UNIX）

次は，SIMPLE の記述を利用して問題を入力，求解，解の出力を行う手続き（サンプルディレクトリ内の MIP.cc）です．

MIP.cc

```
#include "simple.h"


//
// ナップサック問題
//
// 引数ならびと関数名は任意

int MIPsolve(int narg,double* aarg,double barg,double* carg
        ,double* xarg,double* farg)
{
  //
  // システムの初期化（これまで定義したシステムをクリア，以降 SIMPLE の記述）
  //
  SimpleInitialize();

    Set S;
    Element i(set=S);

    IntegerVariable x(name="決定ベクトル"
                 ,index=i,type=binary); // 整数変数
    Parameter c(name="価値",index=i);
    Parameter a(name="重量",index=i);
    Parameter b(name="許容重量");
    Objective obj(name="総価値",type=maximize);

    // 引数からデータを設定
    // (Numerical Optimizer/SIMPLE マニュアルを参照)
    a.readD(narg,aarg);
    b = barg;
    c.readD(narg,carg);
```

```
0 <= x[i] <= 1;

sum(a[i]*x[i],i) <= b;      // 制約条件
obj = sum(c[i]*x[i],i);     // 目的関数

// Numerical Optimizer からの最適化結果ファイル.sol の
// 名前の設定
//    options.outfilename = "nuoptout";
// .sol のファイル出力を抑制する場合には次のように書く
options.outfilename = "_NULL_";
// 出力を抑制する
options.outputMode = "silent";

// 最適化の実行
solve();

// x の内容を C++の配列にダンプ
// (Numerical Optimizer/SIMPLE マニュアルを参照)
int lenx;
int* indx;
double* valx;
x[i].val.dump(lenx,indx,valx);

if ( lenx != narg) {
  return 99; // x[i] の実際の長さが narg と異なる(データに矛盾あり)
}

// 引数に設定
int it;
for ( it = 0 ; it < lenx ; ++it ) {
  xarg[indx[it]-1] = valx[it];
    // indx[it] は 1,2, ... narg なので, 1 を引く
}

// 目的関数値の設定
*farg = result.optValue;

// indx,valx は dump 内部で独自に allocate されるので free しておく.
```

```
    delete [] indx;
    delete [] valx;

    // Numerical Optimizer のエラーコードを返す.
    return result.errorCode;
}
```

この手続き（MIPsolve）は呼ばれるたびに，新しいデータをもとにシステムを作成し直すことを前提としておりますので，手続きが始まるとき定義したモデルの内容をクリアする必要があります．そうするにはここで行っているように最初に SimpleInitialize() をコールします．

4.2.2 モデル兼ドライバのコール（UNIX）

次は SIMPLE の記述を含む手続き MIPsolve をコールするメイン関数です．

useSimple.cc

```
#include <stdio.h>
#include "simple.h"

// 宣言 (必須)
extern "C" {
  extern void secini();
}
// SIMPLE で書かれた最適化手続き
int MIPsolve(int narg,double* aarg,double barg,double* carg
        ,double* xarg,double* farg);

int readData(FILE* fp,char* filename); // データを読む

int main()
{
  //
  // 以下は必須 (初期化)
  //
  secini();
  SimpleInitialize();



  int narg = 5;
```

```
  double aarg[5] = {4,2,3,6,7};
  double carg[5] = {6,8,4,3,4};
  double xarg[5];
  double barg = 20;
  double farg;

  int errCode;

  //
  // barg を変化させながら Numerical Optimizer を
  // 繰り返しコールする
  //
  int ib;
  for ( ib = 1 ; ib <= 30; ++ib ) {
    barg = (double) ib;
    //
    // システムの定義と求解.
    //
    errCode = MIPsolve(narg,aarg,barg,carg,xarg,&farg);
    printf("errCode=%d b=%g obj=%g ",errCode,barg,farg);
    printf(" x= ");
    int i;
    for ( i = 0 ; i < narg ; ++i ) {
      printf("%1.0f ",xarg[i]);
    }
    printf("\n");
  }

  //
  // SIMPLE が保持している static バッファのクリア
  //
  SimpleClearBuffer();

  return 0;
}
```

SIMPLE を記述した手続きを呼ぶ場合には，secini() と SimpleInitialize() を呼ぶ必要があります．ここでは，データファイルを読み込まずに C++の配列にデータを設定して MIPsolve を直接呼んでいますが，以下のようにしてデータファイルを読み込んでおくこともできます．

```
char* filename = "c:\\temp\\data.dat";
FILE* fp = fopen(filename,"r"); // ファイルを開く
readData(fp, filename); // データ
fclose(fp);
```

こうすると，データの名前と内容がバッファに蓄えられます．蓄えられたデータは SIMPLE オブジェクトで同じ名前（name）を持つオブジェクトと対応し，宣言時に読み込まれます．

例えばデータファイル内に

```
param = 2;
```

という記述があると，以降

```
Parameter a(name="param");
```

という宣言によって a に 2 が代入されます．同じものが複数回現れた場合には，そのすべてにデータの内容が代入されます．例えば，

```
Variable v(name="param");
```

という宣言があれば，v にも 2 が代入されます．最後の SimpleClearBuffer() は SIMPLE が保持しているスタティックバッファのクリアを行う命令です．このコールを行った後は，コールを行う前に定義したいかなる SIMPLE オブジェクトの内容も参照することはできなくなる代わりに，プロセスの占有メモリ領域を減らすことができます．

4.2.3 実行形式の作成と最適化の実行（UNIX）

さて，このサンプルコード useSimple.cc，Mip.cc からロードモジュールを作成して実行してみましょう．

```
prompt% make useSimple
prompt% ./useSimple
```

と打ちます．実行させると，

```
errCode=0 b=1 obj=0  x= 0 0 0 0 0
errCode=0 b=2 obj=8  x= 0 1 0 0 0
errCode=0 b=3 obj=8  x= 0 1 0 0 0
errCode=0 b=4 obj=8  x= 0 1 0 0 0
errCode=0 b=5 obj=12  x= 0 1 1 0 0
errCode=0 b=6 obj=14  x= 1 1 0 0 0
errCode=0 b=7 obj=14  x= 1 1 0 0 0
errCode=0 b=8 obj=14  x= 1 1 0 0 0
```

```
errCode=0 b=9 obj=18  x= 1 1 1 0 0
errCode=0 b=10 obj=18  x= 1 1 1 0 0
errCode=0 b=11 obj=18  x= 1 1 1 0 0
errCode=0 b=12 obj=18  x= 1 1 1 0 0
errCode=0 b=13 obj=18  x= 1 1 0 0 1
errCode=0 b=14 obj=18  x= 1 1 1 0 0
errCode=0 b=15 obj=21  x= 1 1 1 1 0
errCode=0 b=16 obj=22  x= 1 1 1 0 1
errCode=0 b=17 obj=22  x= 1 1 1 0 1
errCode=0 b=18 obj=22  x= 1 1 1 0 1
errCode=0 b=19 obj=22  x= 1 1 1 0 1
errCode=0 b=20 obj=22  x= 1 1 1 0 1
errCode=0 b=21 obj=22  x= 1 1 1 0 1
errCode=0 b=22 obj=25  x= 1 1 1 1 1
errCode=0 b=23 obj=25  x= 1 1 1 1 1
errCode=0 b=24 obj=25  x= 1 1 1 1 1
errCode=0 b=25 obj=25  x= 1 1 1 1 1
errCode=0 b=26 obj=25  x= 1 1 1 1 1
errCode=0 b=27 obj=25  x= 1 1 1 1 1
errCode=0 b=28 obj=25  x= 1 1 1 1 1
errCode=0 b=29 obj=25  x= 1 1 1 1 1
errCode=0 b=30 obj=25  x= 1 1 1 1 1
```

という出力が得られます．各行は異なるデータに対応するナップサック問題の解に対応しています．

第5章 SIMPLEモデル記述から クラスを生成して利用する例

本章で紹介するのもSIMPLEによって記述したモデルをプログラムの一部として呼び出す方法です[6]. まず数理計画モデルをモデリング言語SIMPLEで記述します．次にその記述から数理計画モデルに対応するC++のクラスを生成します．ユーザプログラムではそのC++のクラスのオブジェクトを宣言することによって

- データの読み込み
- 最適化の実行
- 変数の初期値の設定
- 最適解や関連する量の表示

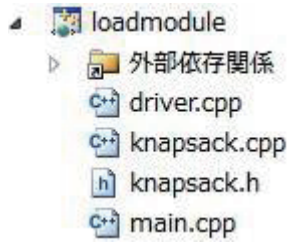
などの操作を実現します．クラスオブジェクト同士は独立しているので，複数の問題をひとつの外部プログラムで操作するなどの操作が可能になります．

- **VC++**

  ソリューション nuoptvcapp のプロジェクト

  loadmodule

  がこのインタフェースを用いてナップサック問題を解くプログラム例です．



- **UNIX**

  UNIX版では，SIMPLEのモデル記述からC++のクラスを生成して，二次計画問題，ナップサック問題のモデルに対して操作を行う手続き（ドライバ）を作成する例を紹介します．

  ファイル

  （Numerical Optimizer のインストール場所）

  /userapp/useClass.cc

  /QP.smp

[6] このインタフェースは，「1.1 サポートプラットフォーム」にあるように，一部のUNIX/Linux版のプラットフォームではサポートされておりません．

/knapsack.smp

がこのインタフェースを用いて二次計画問題，ナップサック問題を解くプログラム例です．

VC++版ライブラリをお使いの方は

5.1 VC++版ライブラリ（SIMPLE モデル記述からクラスを生成して利用する例）へ

UNIX 版ライブラリをお使いの方は

5.2 UNIX 版ライブラリ（SIMPLE モデル記述からクラスを生成して利用する例）へ

お進みください．

5.1 VC++版ライブラリ（SIMPLE モデル記述からクラスを生成して利用する例）

5.1.1 モデル（VC++）

次に示すプログラムはこのアプリケーションの核心となるモデル記述です．通常のモデルと異なるのは，パラメータの宣言部分に required というキーワードがあることですが，これはこの指定のあったデータを C++の配列から入力することを示しています．その他は通常のモデル記述と同じです．この例ではこのモデルが次のような knapsack.smp というファイルに記述されたものとします．

knapsack.smp

```
//
//   ナップサック問題
//
    Set S;
    Element i(set=S);
    IntegerVariable x(index=i,type=binary); // 整数変数
    Parameter c(index=i,required);
    Parameter a(index=i,required);
    Parameter b(required);
    Objective obj(type=maximize);
    VariableParameter d(index=i);

    obj = sum(c[i]*x[i],i);     // 目的関数
    sum(a[i]*x[i],i) <= b;      // 制約条件
```

このファイルは

(Numerical Optimizer のインストール場所)\samples\app

にある zip ファイルに含まれています．

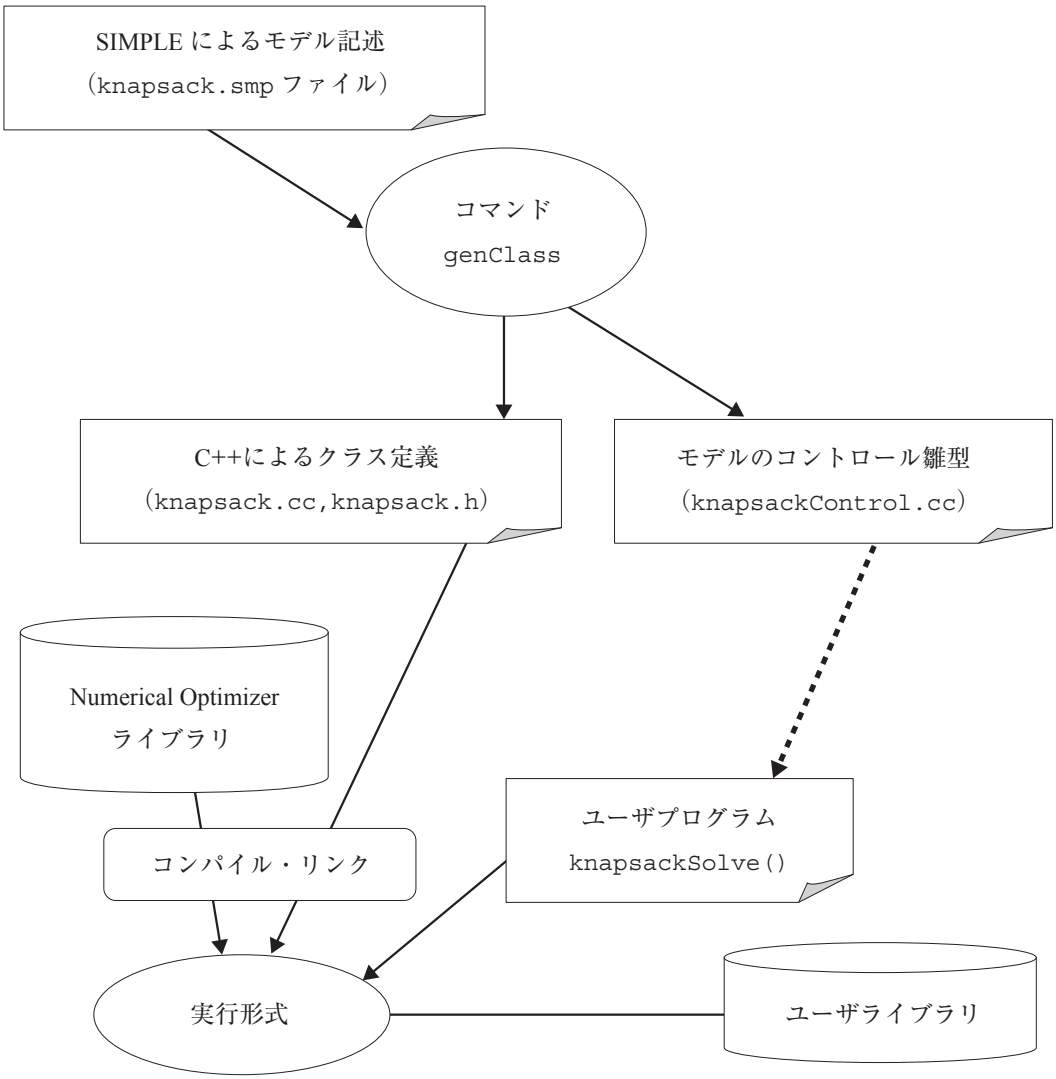
5.1.2 genClass のコール（VC++）

前節のモデルはこのままでは C++で利用できる形ではありませんが．genClass というコマンドを用いて変換すると，対応するクラス宣言（knapsack.h）とクラスの実装（knapsack.cc）が自動生成されます．

次のページに関連するファイルの関係を図示します．ユーザはまず，モデリング言語 SIMPLE を用いてモデル記述を行い，genClass によってクラスの実装[7]を自動生成し，ユーザプログラムからそれを使います．モデルを利用する場合には，こうして生成されたクラス宣言（同 knapsack.h）をインクルードして，クラスオブジェクトを宣言します．このクラスオブジェクトのメンバを通じて，ユーザは自分の C++プログラム（同 knapsackSolve()）からこのモデルにデータや解を入出力（print(), dump()）する操作が可能になります．

実行形式の生成の際には，先ほど自動生成されたクラス実装をコンパイル，リンクします．こうして生成された SIMPLE のクラスは何個でも，また何回でも利用することができます．ただし SIMPLE のクラスを利用するに先立って，ユーザのプログラムから処理前に SimpleInitialize()，処理後に SimpleClearBuffer() なる関数を呼び出す必要があります（これらは SIMPLE が独自に利用する記憶領域の確保と解放に必要となります）．

[7] このクラスの実装の内容をユーザは直接意識する必要はありません．



genClass の起動は DOS ウインドウのプロンプトから

```
> genClass knapsack.smp
```

とします．genClass は Numerical Optimizer が提供するユーティリティーで，この実行のためには別途設定が必要となります．詳細については Numerical Optimizer/SIMPLE マニュアルを参照してください．以下が knapsack.smp に対する genClass の実行例です．

```
C:\Program Files\Mathematical Systems Inc\NUOPT\SAMPLES\app>genClass knapsack.smp
genClass.bat Ver *.*.* for MSI Numerical Optimizer
Copyright (C) 1991 NTT Data Mathematical Systems Inc.
Build with Microsoft Visual Studio 2010 Express on 64bit
```

この操作でこの数理計画モデルに対応するクラス定義と実装が genClass を起動したフォルダに作成されます．

knapsack.h	knapsack.smp に対応するクラスの定義
knapsack.cc	同クラスの実装

knapsackControl.cc　利用方法サンプル

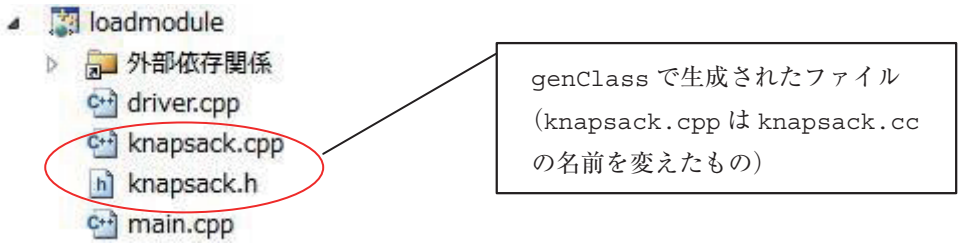
クラスの定義（knapsack.h）は，この数理計画問題に対して操作を行うコントロール手続き（C++のコード）の中で操作を行う前に必ずインクルードする必要があります．クラスの実装（knapsack.cc）の内容は通常ユーザが意識する必要はありませんが，実行形式を作成する際にユーザのプログラムにコンパイル・リンクする必要があります．

ソリューション nuoptvcapp のプロジェクト

loadmodule

がこのインタフェースを用いてナップサック問題を解くプログラム例ですが，genClass で生成された実装（knapsack.cc）とクラスの定義（knapsack.h）を追加しています．

ここで一点注意があります．生成された knapsack.cc はプロジェクトに追加する前に knapsack.cpp と拡張子を変更してください．こうすることにより，VC++の GUI の規約により，このファイルが C++のコードであると認識されるようになります．



knapsackControl.cc は knapsack.smp で定義したクラスの利用方法のサンプルが記載されています．このコードは一度最適化実行を行うというものです．入力オブジェクトがあるクラスの場合には受け渡し用オブジェクトの宣言が書かれていますので，クラスの利用のコード作成時には参考になります．

knapsackControl.cc

```
#include "knapsack.h"
void simpleControl()
{
    Set S;
    Element i( set = S );
    Parameter c( name = "c",  index = i );
    Parameter a( name = "a",  index = i );
    Parameter b(name = "b");
    System_knapsack s1(c, a, b);
}
```

5.1.3　ドライバ（VC++）

ではこのモデルを解く汎用の C++プログラムを準備しましょう．それはプロジェクトに追加されている

driver.cpp

というファイルで，knapsackSolve という名前の手続きです．このルーチンのインタフェースは入力として，ナップサック問題のモデルの

```
Parameter a(index=i);
Parameter b;
Parameter c(index=i);
```

に相当するデータ（aarg，barg，carg）を与えると

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

に相当する変数値（xarg）と目的関数（farg）を返すというものです．ドライバのこのインタフェースは問題に特化して調整できるので，より柔軟なコード作成が可能になります．

driver.cpp

```
#include "knapsack.h"   // インクルード宣言
//
//  ナップサック問題の求解
//
//
int knapsackSolve(int narg // 問題のサイズ
       ,double* aarg  // 係数 a
       ,double barg   // 係数 b
       ,double* carg  // 係数 c
       ,double* xarg  // 解ベクトル ( x )
       ,double* farg  // 目的関数
       )
{
  // 初期化
  SimpleInitialize();
  { // SimpleInitialize() のコールの後は{を付ける
    // 値転送用のオブジェクト
    Set S;
    Element i(set=S);
    Parameter b;
```

```
    Parameter c(index=i),a(index=i);

    // C 配列を初期化用のデータに与える.
    b = barg; // スカラはそのまま代入
    c.readD(narg,carg); // 配列から SIMPLE のオブジェクトへの設定
    a.readD(narg,aarg); // 配列から SIMPLE のオブジェクトへの設定

    //
    // knapsack 問題の求解
    //
    System_knapsack knap(c,a,b);

    int len;
    int* ind;
    double* knapx;
    knap.x.val.dump(len,ind,knapx);
    // knapsack 問題の x を C の配列である knapx に設定

    // 解を戻り配列に設定
    int it;
    for ( it = 0 ; it < len ; ++it ) {
      xarg[it] = knapx[it];
    }
    *farg = result.optValue;
    // 目的関数値を取る簡単な方法 (asDouble() でも可能)

    // 不要な領域の破壊
    delete [] ind;
    delete [] knapx;
  } // SimpleClearBuffer() のコールの前を}で閉じる.
    // 終了処理
    SimpleClearBuffer();
    return result.errorCode; // エラーコードを返す.
}
```

SimpleInitialize() と SimpleClearBuffer() は SIMPLE を利用した処理の最初と最後に必ず必要なコールです．実装上の理由により，SimpleInitialize のコールより後と，SimpleClearBuffer() のコールより前は{ }でくくる必要があります．SIMPLE のオブジェクト（Set や Element）の宣言は，この{ }の中で行います．SimpleInitialize をコールせずに SIMPLE のオブジェクトを宣言して利用

した場合，あるいはこの{}の外で SIMPLE のオブジェクトを宣言すると実行時エラーとなります．

手続きの中ほどで System_knapsack というクラスのオブジェクト knap を宣言していますが，これが knapsack.smp というモデル記述に対応するクラスのオブジェクトの宣言です．このプログラムの先頭の

```
#include "knapsack.h"
```

がこのモデル定義を含むファイルで，モデル knapsack に対応するクラスを利用する場合には必ずインクルードする必要があります．

一般に NAME.smp なるモデルには System_NAME というクラスが対応します．クラスの定義ファイル NAME.h には System_NAME の定義が書かれています．そのため，System_NAME を使用する際には必ず NAME.h のインクルードが必要になります．

クラス宣言の中で SIMPLE オブジェクトを required というキーワード付きで

```
// knapsack.smp の中
Parameter c(index=i,required);
Parameter a(index=i,required);
Parameter b(required);
```

のように宣言しているので，呼び出し側の手続きで同様に宣言した受け渡し用のオブジェクト

```
// driver.cpp の中
Set S;
Element i(set=S);
Parameter b;
Parameter c(index=i),a(index=i);
```

に

```
b = barg; // スカラはそのまま代入
c.readD(narg,cary); // 配列は readD を用いる．
a.readD(narg,aary); // 配列は readD を用いる．
```

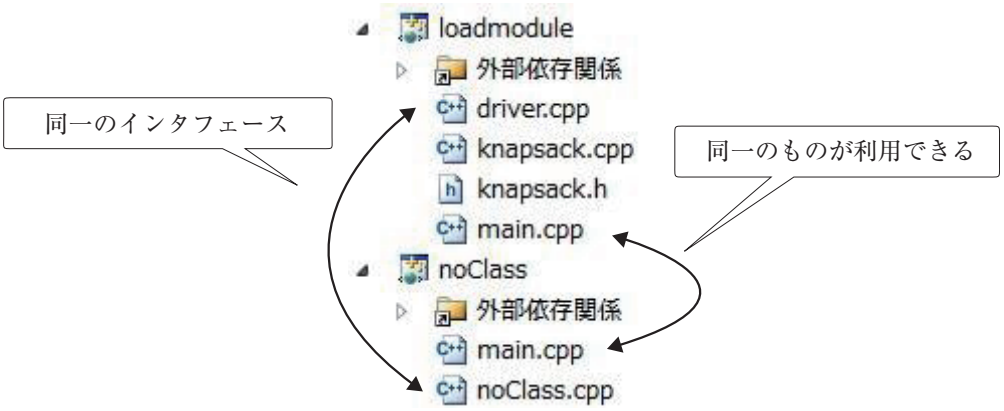
として値を設定，システムオブジェクト knap の宣言の際に

```
System_knapsack knap(c,a,b); // knapsack 問題の求解
```

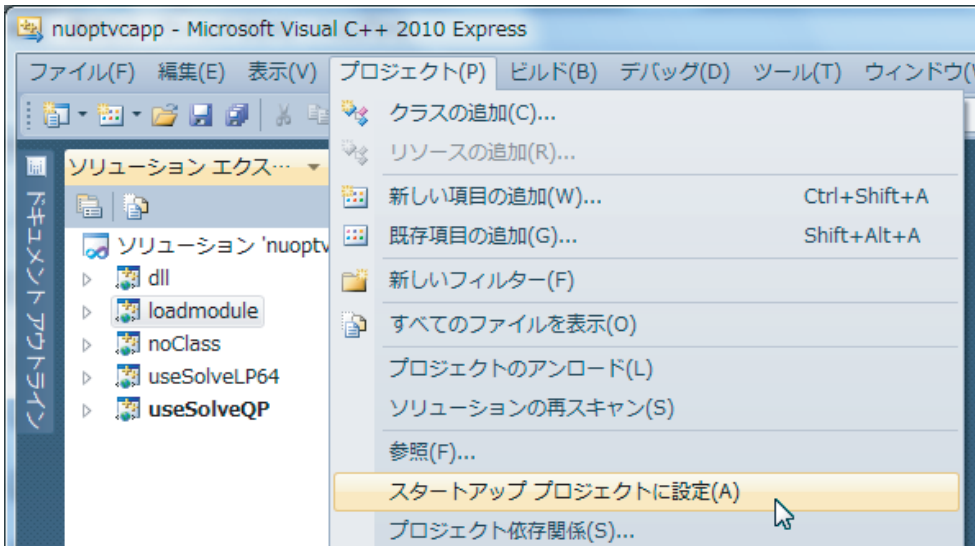
のように渡しています（この引数にはモデル中，required というキーワード付きで宣言したオブジェクトが出現順に並びます）．

5.1.4　C++関数からの呼び出し（VC++）

driver.cc で定義された knapsackSolve はプロジェクト noClass に含まれている noClass.cpp と同一の仕様ですので，プロジェクト noClass のメインルーチンをそのまま使うことができます．



VC++のメニューの「プロジェクト」から「スタートアップ プロジェクトに設定」を選び，次のようにこのプロジェクトを選択します．

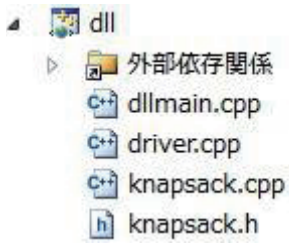


つづいて VC++GUI の実行を選択：

すると，プロジェクト noClass の出力と同一の出力が得られます．

5.1.5　VisualBasic からの呼び出し（VC++）

ライブラリ部分を DLL にすると一般に VisualBasic などのアプリケーションから呼ぶことができます．プロジェクト dll は同一のドライバルーチンを使って DLL を作成するプロジェクトです．



DLL の入り口となる knapsackSolve() をコールするルーチンを作成すると以下のようになります．このルーチンは入力を knapsackSolve に渡して一度だけ解くという操作を行います．

dllmain.cpp

```
#include <windows.h>

// nuoptmain() の中でコールする最適化手続き（最適化ライブラリ）
int knapsackSolve(int narg,double* aarg,double barg,double* carg
     ,double* xarg,double* farg);

// VB から呼ばれる DLL の入り口
#define EXPORT __declspec (dllexport)
#define FORVB _stdcall

extern "C" long EXPORT FORVB nuoptmain(
                        long narg // 問題サイズ
                        ,double* aarg // ベクトル a
                        ,double  barg // ベクトル b
                        ,double* carg // ベクトル c
                        ,double* xarg // ベクトル x(出力)
                        ,double* farg // 目的関数値（出力）
       )
{
  int errCode = knapsackSolve(narg,aarg,barg,carg,xarg,farg);
  return errCode ;
}
```

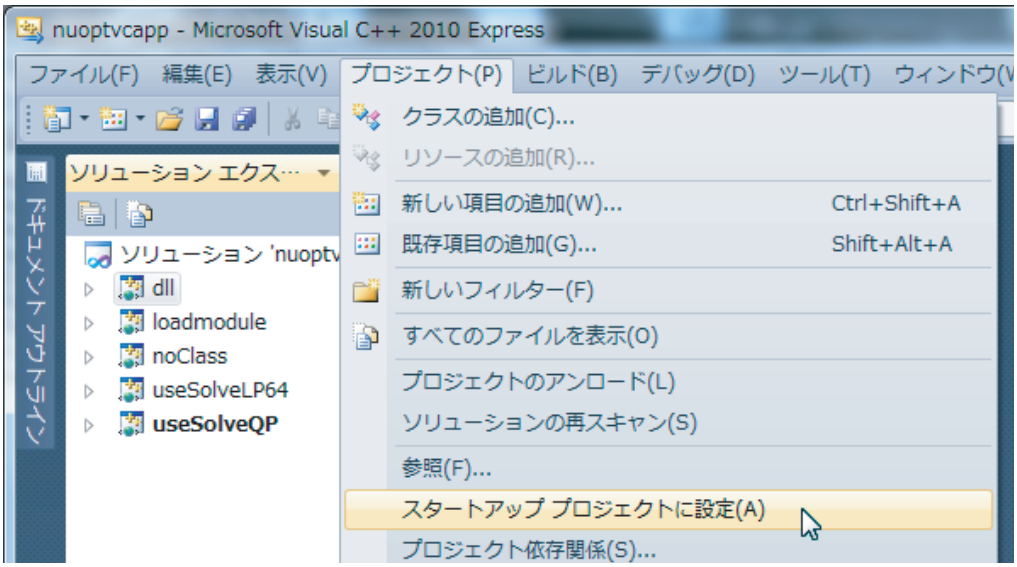
DLL のデバッグは

- 出力が得られない
- 異常が起こると起動アプリごと動作を停止する

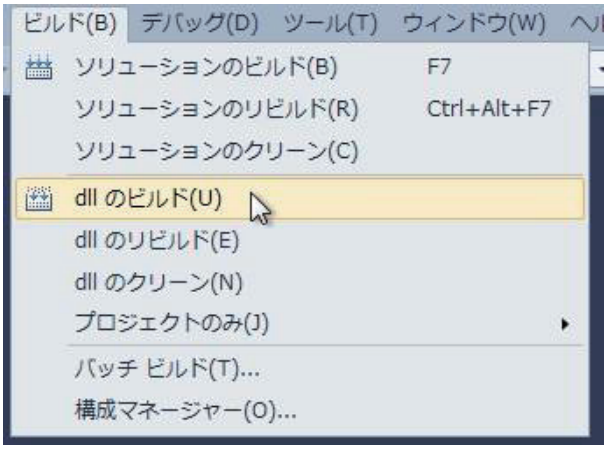
など，困難な点が多いので一度ダミーのメイン関数でコールするロードモジュールなどで動作を確認したのち，DLL を作成することをお勧めします．DLL のメインの宣言方法は通常の Windows アプリケーションと同様です．ここでは，生成した DLL を Excel の VBA からコールする例を示しますので

ここでは VBA から呼ばれることを前提とした宣言（_stdcall）となっています．まず，DLL を作成しましょう．

プロジェクト「dll」を選択後，VC++のメニューの「プロジェクト」から「スタートアップ プロジェクトに設定」を選びます．



次に，「プロジェクト」メニューから「dll のビルド」を選択すると，DLL が作成されます．



では，これをリンクして実行する VBA 側の設定を行います．この DLL を読み込んで実行するサンプルアプリケーションは

(Numerical Optimizer のインストール場所)\samples\app にある zip ファイルを解凍した中にある dll\dllsample.xls

です．「マクロを有効にする」で，このエクセルファイルを開き，VisualBasic エディタを開くと，Sheet1 には次のようなコードが書かれています．これはシートからデータを読み取って nuoptmain をコールするコードです．下線部は DLL の絶対パスです（先ほどアクティブな構成の設定でデバッグを選択するとこの場所に DLL が作成されます）．C++の int は VBA の Long 型に，double は VBA の Double 型にそれぞれ対応します．

Sheet1 に含まれるコード：

nuoptmain をコールする例

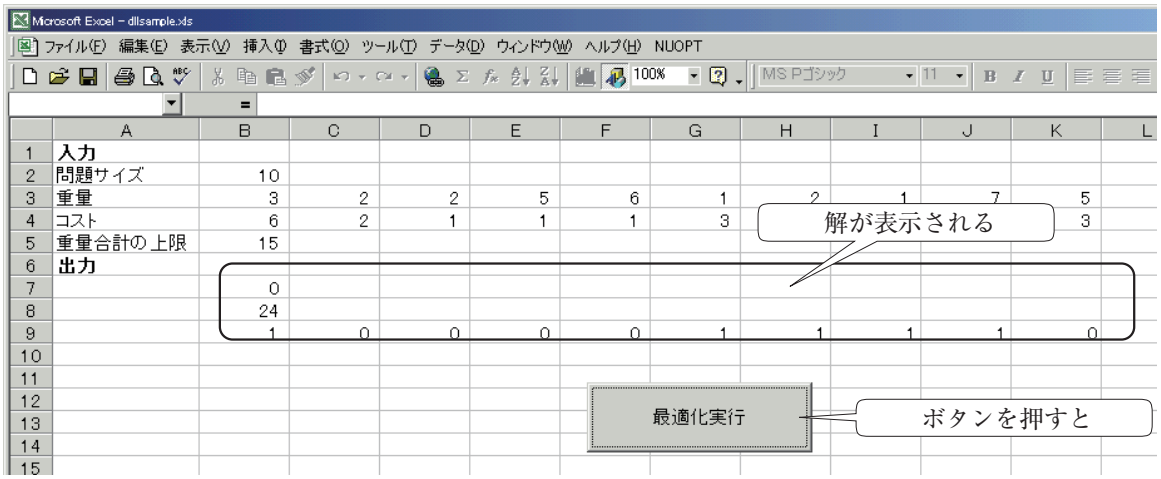
```
Private Sub execbutton_Click()
    ChDir ThisWorkbook.Path
    Dim a() As Double
    Dim c() As Double
    Dim x() As Double
    Dim n As Long
    Dim obj As Double
    n = Range("B2")
    ReDim a(n)
    ReDim c(n)
    ReDim x(n)
    For i = 1 To n
        a(i - 1) = Range("A3").Offset(0, i)
        c(i - 1) = Range("A4").Offset(0, i)
    Next i
    b = Range("B5")
    code = nuoptmain(n, a(0), b, c(0), x(0), obj)
    Range("B7") = code
    Range("B8") = obj
    For i = 1 To n
        Range("A9").Offset(0, i) = x(i - 1)
    Next i
End Sub
```

実行させるには，リンクを行う必要があります．サンプルの設定では Excel ブック dllsample.xls と同じフォルダにある Release フォルダ内に，dll.dll がある場合に正しく動作します．dll.dll の場所を変更する場合は，Module1 の下線部を調整します．

DLL の宣言：

```
Declare Function nuoptmain Lib "Release\dll.dll" Alias _
"_nuoptmain@28" (ByVal n As Long, ByRef a As Double, ByVal b As Double , ByRef c As Double, ByRef x As Double, ByRef f As Double) As Long
```

エクセルシートに戻ってボタンを押すと DLL の実行が行われます．以下はこのコードを使ったアプリケーションの実行例です．



5.1.6　GUI でドライバ（コントロールルーチン）や WINAPI を利用する（VC++）

smp モデルファイルに

```
%%%%% #include "C:\mysrc\myheader.h" %%%%%
```

という行を挟めばモデルとコントロールで共通にこのインクルードが行われます.

この機能を使うと，例えば次のように，WINAPI をモデル定義中に用いることができます．次のモデルは WINAPI（getdate）を用いて，時刻を表示するものです.

WINAPI を用いる例

```
//
// 簡単な LP（最適化の実行日付を表示する）
//
Variable x;
Variable y;
// 目的関数
Objective cost(type=minimize);
cost = 180*x + 160*y;
// 制約
6*x + y >= 12;
3*x + y >= 8;
4*x + 6*y >= 24;
// 変数の上下限
0 <= x <= 5;
0 <= y <= 5;


char* getdate(char*);
```

```
char chrTime[128];
printf("%s\n",getdate(chrTime));
fflush(stdout);

%%%%% #include <time.h> %%%%%

// C++の関数の定義
char* getdate(char* chrTime)
{
    _strdate(chrTime); // mm/dd/yy 形式で日付を文字列に格納
     return chrTime;
}
```

5.2 UNIX 版ライブラリ（SIMPLE モデル記述からクラスを生成して利用する例）

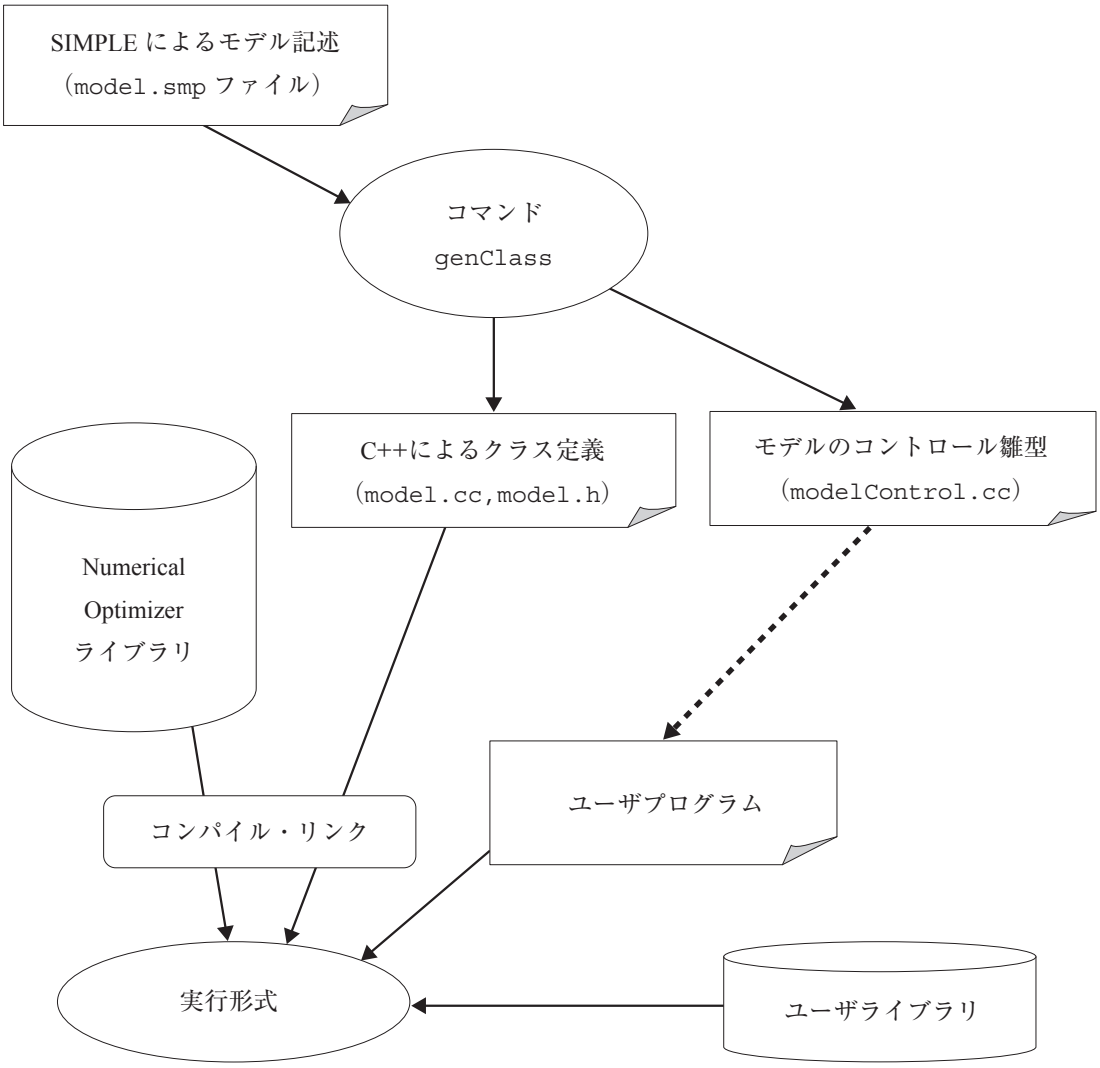
5.2.1 手順の概要（UNIX）

次のページに関連するファイルの関係を図示します．ユーザはまず，モデリング言語 SIMPLE を用いてモデル記述を行います．モデルが記述できたら，その内容を .smp という拡張子のファイル（例えば model.smp）に格納します．

次に genClass というコマンドを用いて model.smp から対応するクラス宣言（model.h）とクラスの実装（model.cc）を自動生成します．ユーザはこの内容を意識する必要はありません．

こうして生成されたクラス宣言（model.h）をインクルードすることによって，ユーザのプログラムからこのモデルに対してデータを入力したり，求解を指示したり（solve()），解を出力したり（cout<<, print()，dump()）する操作が可能になります．

実行形式の生成の際には，先ほど自動生成されたクラス実装をコンパイル，リンクします．こうして生成された SIMPLE のクラスは何個でも，また何回でも利用することができます．ただし SIMPLE のクラスを利用するに先立って，ユーザのプログラムから初期化関数を呼び出す必要があります（詳細は後述します）．



5.2.2 二次計画問題の例（UNIX）

サンプルディレクトリの中には次のような簡単な二次計画問題を記述したモデルファイル QP.smp があります.

1. モデルの記述（**UNIX**）

QP.smp

```
//
// 簡単な二次計画問題
//
    // 集合
    Set S;
    Element i(set=S);
    Set T;
```

```
Element j(set=T), k(set=T);
// パラメータ
Parameter c(name="c", index=j);
Parameter Q(name="Q", index=(j,k));
Parameter cu(name="cu", index=i);
Parameter cl(name="cl", index=i);
Parameter A(name="A", index=(i,j));
Parameter bu(name="bu", index=j);
Parameter bl(name="bl", index=j);
// 変数
Variable x(index=j);
// 最小化
Objective f(type=minimize);
f = sum(c[j] * x[j], j) + 1.0/2.0 *sum(Q[j,k]*x[j]*x[k], (j,k));
// 条件
Expression constr(index=i);
constr[i] = sum(A[i,j] * x[j], j);
cu[i] >= constr[i] >= cl[i];
bu[j] >= x[j] >= bl[j];
```

2. genClass の起動（**UNIX**）

さて，次はこのモデル記述 QP.smp からクラス定義と実装を記述したファイル（C++のコード）を生成します．そのためにはプロンプトから

```
prompt% genClass QP.smp
```

とします．genClass は Numerical Optimizer をインストールした環境でのみ実行可能なユーティリティーです．そうすると，この数理計画モデルに対応するクラス定義と実装が genClass を起動したディレクトリに作成されます．サンプルの makefile ではこの操作を生成規則に記述しています．

QP.h	QP.smp に対応するクラスの定義
QP.cc	同クラスの実装
QPControl.cc	コントロール手続きサンプル

クラスの定義（QP.h）は，この数理計画問題に対して操作を行うコントロール手続き（C++のコード）の中で操作を行う前に必ずインクルードする必要があります．クラスの実装（QP.cc）の内容は通常ユーザが意識する必要はありませんが，実行形式を作成する際にユーザのプログラムにコンパイル・リンクする必要があります．

3. モデルからの問題の生成，求解，出力（**UNIX**）

次にこの数理計画モデルに対する操作の例です．以下はサンプルプログラム useClass.cc からこ

の二次計画問題に関連する操作を行っている部分を抜き出したものです.

操作の例

```
#include "QP.h" //クラスの定義のインクルード（必須）

    …中略…

void simpleControl()
{
  int len,i,*ind;

  //
  //  QP の求解
  //
  System_QP qp; // QP.smp に対するクラスオブジェクト qp の宣言と最適化実行

  qp.f.val.print(); // QP.smp 中のオブジェクトの表示
  qp.x.val.print();
  qp.constr.val.print();

  double* x;
  qp.x.val.dump(len,ind,x); // QP の x を C++の配列にダンプ.

  printf("x(QP):\n");
  for ( i = 0 ; i < len ; ++i ) {
      printf("[%3d] %10.3e ",ind[i],x[i]);
      if ( (i+1) % 4 == 0 ) {
      printf("\n");
      }
  }
  printf("\n");

     …後略…
```

useClass.cc では simpleControl という手続きの中で二次計画問題に対する操作が記述されています．手続きの名前は任意です.

手続きの最初で System_QP というクラスのオブジェクト qp を宣言していますが，これが QP.smp というモデル記述に対応するクラスのオブジェクトの宣言です.

一般に

NAME.smp なるモデルには System_NAME というクラスが対応

します．クラスの定義ファイル NAME.h には System_NAME の定義が書かれています．そのため，System_NAME を使用する際には必ず NAME.h のインクルードが必要になります．
さて，宣言して作成された qp がその数理計画モデル（問題）そのものに対応します．以下ではこれを説明のため「システムオブジェクト」と呼びます．システムオブジェクトに対して

```
qp.show();
```

とすると，問題の中身が表示されます（showSystem() と同じ）．次にシステムオブジェクトと数理計画モデルの構成要素についてですが，一般的な原則として以下があります．

System_NAME のオブジェクト s について s.x は NAME.smp 中のオブジェクト x に対応する．

さらに，s.x として参照されたオブジェクトに対する

代入	=
表示	print()，cout，dump()
配列の設定	readD()

などの操作は通常の SIMPLE オブジェクトに対するのと全く同様に可能です．
例えば

```
qp.f.print();
qp.x.print();
qp.constr.print();
```

とすると，それぞれ QP.smp のモデル中の f，x，constr の値が表示されます．

デフォルト動作では宣言を行ったのみで自動的に最適化の実行が行われます．オプションの設定（options.noDefaultSolve = 1）によって，自動的に最適化の実行が行われるのを抑制することが可能ですが，その場合には

```
qp.solve();
```

として陽に求解を指示します．

サンプルルーチン useClass.cc では，QP.smp のデータ（Parameter）は与えていませんが，その場合には，SIMPLE のデータファイルの内容が自動的に設定されます．ただし，そうするにはメイン関数によってデータファイルを読みこんでバッファに蓄えておく必要があります．その方法については後述します．

求解のあと，

```
qp.x.val.dump(len,ind,x); // QP の x を C++の配列にダンプ.
```

と行うことによって，解を C++の配列に解を書き出しています．SIMPLE では一般に

```
(SIMPLE オブジェクト).val.dump
```

によってオブジェクトを配列に書き出すことができます.

4. モデルへのデータ入力（**UNIX**）

   モデルにデータを設定する方法は

   A) SIMPLE のデータファイルを用いる.

   B) C++の配列を直接設定する.

   という二つがあります.

5. データファイルの利用（**UNIX**）

   useClass.cc では，QP.smp のデータ（Parameter）の設定は 0 の方法で行うことを前提としています．この場合には，このこのデータからシステムオブジェクトを作成する前にメイン関数によってデータファイルを読みこんでバッファに蓄えておく必要があります.

   データを読み込むには

```
readData(FILE* fp, char* filename);
```

   なる手続きをコールします．fp は C の stdio.h で宣言されているファイル構造体で，読み込むデータファイルに対応します．データが読み込まれると，データの内容が共用バッファにプールされます．データ読み込みは繰り返すことができます．UseClass.cc の main() 手続きの中の次のコードは，呼び出し引数として与えられたファイルをすべて読み込んで共用バッファに蓄えます.

データ読み込みの例

```
//
// SIMPLE 形式で記述されたデータを読む
//
int i;
for (i = 1 ; i  < argc; i++) {
  char *file = argv[i];
  FILE* fp;
  if (strcmp(file, "stdin") == 0) fp = stdin;
  else fp = fopen(file, "r");
  if (!fp) {
    fprintf(stderr, "can not open %s\n", file);
    exit(1);
  }
  readData(fp, file);
}
```

useClass.cc は QP.smp に対応する例えば次のようなデータ：

qp312.dat

```
c = [1] -3 [2] 1;
Q =
[1,1] 11 [1,2] 0
[2,1] 0  [2,2] 22
;
cu = [1] 10 [2] 10 [3] 10;
cl = [1] -1 [2] -2 [3] 2;
A =
[1,1] -1   [1,2] 0.1
[2,1] -0.2 [2,2] -1
[3,1] 2    [3,2] 1
;
bu = [1] 1 [2] 2;
bl = [1] 0 [2] 0;
```

を読み込みますが，この時点で名前と値の対応がバッファに蓄えられます．続いて，

```
System_QP qp;
```

が実行された段階で，QP.smp の中の Parameter である a，Q，cu，cl，A，bu，bl にこの値が設定されます．バッファにプールされたデータとパラメータの対応付けは名前によって行われます．名前はコンストラクト時に name =で与えますが，name =によって名前を与えられていないオブジェクトには，そのオブジェクトの名前と同じ名前が自動的に与えられます．

例えば，通常

```
Parameter a(name="パラメータ");
```

に対しては

```
パラメータ = 2;
```

というデータファイルの記述が対応しますが，

```
Parameter a;
```

と書くと，これは

```
Parameter a(name="a");
```

としたことに相当して，データファイル中の

```
a = 2;
```

という記述に対応します.

データファイルから読み込んだデータは大域的に有効で，特定のモデルに結びついたものではないことにご注意ください.
例えば

```
Parameter a;
```

という記述を含む二つのモデル M1.smp，M2.smp があり，それから生成されたクラスの両方を利用するコード,

```
System_M1 m1;
System_M2 m2;
```

があるとします.
最初に

```
a = 2;
```

というデータファイルを読み込んだ状態でこのコードを実行すると，m1，m2 両方に a = 2 が設定されます.

5.2.3 ナップサック問題のモデル（C++の配列からのデータ設定）（UNIX）

useClass.cc ではもうひとつ knapsack.smp というモデルを利用していますが，このモデルのデータは C++の配列からデータを入力しています．その場合にはモデルの記述に変更が必要です．具体的には外部入力とするパラメータの宣言部分に required というキーワードを追加します.

knapsack.smp

```
//
// ナップサック問題
//

    Set S;
    Element i(set=S);
    IntegerVariable x(index=i,type=binary); // 整数変数
    Parameter c(index=i,required);
    Parameter a(index=i,required);
    Parameter b(required);
    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
```

次は useClass.cc の中で knapsack.smp に対するデータを設定している部分です．required をつけて宣言したオブジェクトを含む.smp に対応するクラス（ここでは System_knapsack）は，required をつけて宣言した Parameter を順に与えて宣言する必要があります．そうして与えられた Parameter が対応するオブジェクトの初期値になります．

データ設定の例

```
// knapsack.smp の初期化用のデータの宣言
Set S;
Parameter a(index=S);
Parameter c(index=S);
Parameter b;

// データを C++側から与える.
double cary[10] = { 42, 12, 45 , 5  , 2, 61, 89 , 32 , 47, 18};
double aary[10] = { 39, 13, 68 , 15 , 10 , 20 , 31 , 15 , 41 , 16};

// C++の配列を初期化用のデータに与える.
  b = 121; // スカラはそのまま代入
  c.readD(10,cary); // 配列は readD を用いる.
  a.readD(10,aary); // 配列は readD を用いる.

  System_knapsack knap(c,a,b); // knapsack 問題の求解

  double* knapx;
  knap.x.val.dump(len,ind,knapx); // knapsack 問題の x を knapx にダンプ

// 表示
  printf("x(knapsack):\n");
  for ( i = 0 ; i < len ; ++i ) {
      printf("[%3d] %10.3e ",ind[i],knapx[i]);
      if ( (i+1) % 4 == 0 ) {
       printf("\n");
      }
  }
  printf("\n");
```

```
  delete [] ind;
```

上の例では，まず，useClass.cc の中で c，a，b を宣言して，readD や代入によってデータの内容を設定，続いて knap の宣言に与えています．こうすることによってモデル中の c，a，b に useClass.cc の中の c，a，b の値が与えられます．useClass.cc のなかの c，a，b と knapsack.smp の中の c，a，b とはここでは同じ名前ですが，必ずしもそうである必要はありません．

5.2.4 初期設定と main 関数（UNIX）

前項のように SIMPLE のクラスをユーザプログラムから利用するには初期設定をおこなっておく必要があります．

次が simpleControl を呼び出している useClass.cc のメイン部分です．

useClass.cc のメイン部分

```
#include "simple.h" // 必要な初期設定
void simpleControl();

//
//  メイン関数サンプル
//
//
int main(int argc,char** argv)
{
  // SIMPLE 内部の初期化 (これはいつでも必須)
  SimpleInitialize();

  //
  // SIMPLE 形式で記述されたデータを読む
  //      （ファイル名が引数から与えられるものとする）
  //
  int i;
  for (i = 1 ; i  < argc; i++) {
    char *file = argv[i];
    FILE* fp;
    if (strcmp(file, "stdin") == 0) fp = stdin;
    else fp = fopen(file, "r");
    if (!fp) {
      fprintf(stderr, "can not open %s\n", file);
```

```
      exit(1);
    }
    readData(fp, file);
  }
  simpleControl(); // モデルの求解など

  // SIMPLE が利用するスタティックバッファのクリア
  SimpleClearBuffer();

  return 0;
}
```

このプログラムはデモ用に最低限の機能を果たすサンプルです．ユーザはこの内容にこだわることなく，自由に記述することができますが，以下に述べる注意を守る必要があります．

まず，メイン手続きで（すべての Numerical Optimizer 関連の処理を行う前に）

```
SimpleInitialize();
```

を必ず一度だけ呼ぶ必要があります．これを呼ぶためには，

```
#include "simple.h"
```

というヘッダファイルのインクルードが必要です．なお，文字列操作のために string.h のインクルードを行なう場合，#include <string.h>は#include "simple.h"より前に記述する必要があります．

実行形式を作成するには

使用するモデルファイルから生成されたクラスの実装（QP.cc に相当するもの）

Numerical Optimizer のライブラリ

をユーザプログラムにリンクする必要があります．

最後の SimpleClearBuffer() は SIMPLE が保持しているスタティックバッファのクリアを行う命令です．このコールを行った後は，コールを行う前に定義したいかなる SIMPLE オブジェクトの内容も参照することはできなくなる代わりに，プロセスの占有メモリ領域を減らすことができます．

次はメイクファイルの内容（一部）です．ライブラリはこのメイクファイルで“LIB”という変数に定義されています．

UseClass.cc では，QP.smp の他，knapsack.smp というモデルを使用しているので，これらをまず，genClass に入力，生成された QP.cc，knapsack.cc を useClass.cc とリンクしています．

```
QP.cc : QP.smp # *.smp からソースを生成する規則
     genClass $*.smp
```

```
knapsack.cc : knapsack.smp
   genClass $*.smp

useClass : QP.o knapsack.o useClass.o # 実行形式作成
   link -out:$@ QP.o knapsack.o useClass.o $(LIBS)
```

5.2.5 実行形式の作成と最適化の実行（UNIX）

さて，サンプルコード useClass のロードモジュールを作成して実行してみましょう．

```
prompt% make useClass
```

とすると，QP.cc，knapsack.cc，useClass.cc がコンパイル，リンクされて useClass が作成されます．引数として qp312.dat（QP.smp 用のサンプルデータ）を与えてロードモジュールを実行すると，以下のような出力が得られます．

```
%prompt ./useClass qp312.dat
[List of Data Files]
<reading data_file: qp312.dat>

[Expand Constraints and Objectives]
QP.smp:25:info: 展開中 目的関数 (1/3) name="f"
QP.smp:30:info: 展開中 制約式   (2/3) name=""
QP.smp:31:info: 展開中 制約式   (3/3) name=""

[About Numerical Optimizer]
MSI Numerical Optimizer x.x.x (NLP/LP/IP/SDP module), Copyright (C) 1991 NTT DATA
Mathematical Systems Inc.

[Problem and Algorithm]
PROBLEM_NAME                                          QP
NUMBER_OF_VARIABLES                                    2
NUMBER_OF_FUNCTIONS                                    4
PROBLEM_TYPE                                  MINIMIZATION
METHOD                                     TRUST_REGION_IPM

[Progress]
 // （中略）
```

```
[Result]
STATUS                                              OPTIMAL
VALUE_OF_OBJECTIVE                               2.318181825
ITERATION_COUNT                                           22
FUNC_EVAL_COUNT                                           26
FACTORIZATION_COUNT                                       37
RESIDUAL                                      5.297266072e-09
ELAPSED_TIME(sec.)                                      0.03
SOLUTION_FILE                                         QP.sol
f=2.31818
x[1]=0.939394
x[2]=0.121212
,[*] = [j]
constr[1]=-0.927273
constr[2]=-0.309091
constr[3]=2
,[*] = [i]
x[  1] =  9.394e-01 x[  2] =  1.212e-01
x(QP):
[  1]  9.394e-01 [  2]  1.212e-01
** QP is solved **

[Expand Constraints and Objectives]
knapsack.smp:16:info: 展開中 目的関数 (1/2) name="obj"
knapsack.smp:17:info: 展開中 制約式   (2/2) name=""

[About Numerical Optimizer]
MSI Numerical Optimizer x.x.x (NLP/LP/IP/SDP module), Copyright (C) 1991 NTT DATA
Mathematical Systems Inc.

[Problem and Algorithm]
PROBLEM_NAME                                        knapsack
NUMBER_OF_VARIABLES                                       10
(#INTEGER/DISCRETE)                                       10
NUMBER_OF_FUNCTIONS                                        2
PROBLEM_TYPE                                    MAXIMIZATION
METHOD                                                SIMPLEX
```

```
[Progress]
 //（中略）

[Result]
STATUS                                                  OPTIMAL
VALUE_OF_OBJECTIVE                                          242
SIMPLEX_PIVOT_COUNT                                           0
RESIDUAL                                                      1
ELAPSED_TIME(sec.)                                         0.01
SOLUTION_FILE                                            QP.sol
x(knapsack):
[  1]  1.000e+00 [  2]  0.000e+00 [  3]  0.000e+00 [  4]  0.000e+00
[  5]  0.000e+00 [  6]  1.000e+00 [  7]  1.000e+00 [  8]  1.000e+00
[  9]  0.000e+00 [ 10]  1.000e+00
** knapsack is solved **
```

上に挙げた内容は Numerical Optimizer からの求解に関するメッセージです.

```
System_QP qp;
... .
System_knapsack knap(c,a,b);
```

として生成された問題について順に求解が行われていることがわかります．問題の変数の数（NUMBER_OF_VARIABLES）や目的関数の値（VALUE_OF_OBJECTVE）を知ることができます．Numerical Optimizer の標準出力の内容については Numerical Optimizer/SIMPLE マニュアルをご覧ください.

5.2.6 その他の操作（UNIX）

これで簡単な操作について一通り解説しましたが，例えば

- Numerical Optimizer のパラメータを指定する.
- 最適化の初期値を設定する.
- 解をいろいろなフォーマットで印刷，ファイル出力する.

などが通常の SIMPLE の記述と同様に可能です．以下ではこれらについて具体的に解説します．Numerical Optimizer のパラメータを設定するには，通常の.smp の記法と同様に

```
options. パラメータ名 = ...;
```

というコードを挿入します．例えば Numerical Optimizer の標準出力を抑制するには

```
options.outputMode = "silent";
```

とします．また，デフォルト出力される Numerical Optimizer の求解レポート*.sol の出力を抑制する

には

```
options.outfilename = "_NULL_";
```

と設定します．例えば，useClass.cc で次のように書くと，

出力を抑制する例

```
void simpleControl()
{
  // Numerical Optimizer の標準出力を抑制,
  options.outputMode = "silent";
  // Numerical Optimizer の解ファイルの出力を抑制
  options.outfilename = "_NULL_";
  int len;
  int* ind;
  int i;
          ...
}
```

次のように Numerical Optimizer からの標準出力が抑制されます．

```
[List of Data Files]
<reading data_file: qp312.dat>
f=2.31818
x[1]=0.939394
x[2]=0.121212
,[*] = [j]
constr[1]=-0.927273
constr[2]=-0.309091
constr[3]=2
,[*] = [i]
x[  1] =  9.394e-01 x[  2] =  1.212e-01
x(QP):
[  1]  9.394e-01 [  2]  1.212e-01
** QP is solved **
x(knapsack):
[  1]  1.000e+00 [  2]  0.000e+00 [  3]  0.000e+00 [  4]  0.000e+00
[  5]  0.000e+00 [  6]  1.000e+00 [  7]  1.000e+00 [  8]  1.000e+00
[  9]  0.000e+00 [ 10]  1.000e+00
** knapsack is solved **
```

変数の初期値の設定はシステムオブジェクト内の変数に相当するオブジェクトに対して代入を行い

ます．すなわち useClass.cc の例では

```
System_QP qp;
qp.x[1] = 2.0; // x[1] の初期値の設定
```

のようにして行います．ただし，デフォルト動作では qp を宣言した段階で，初期値の設定を行う前に求解が行われてしまいますので，これを抑制し，陽に求解を指示するようにしましょう．次のようにします．

```
options.noDefaultSolve = 1; // デフォルトで求解を行わないようにする
System_QP qp;       // qp の宣言（求解は行わない）
qp.x[1] = 2.0;      // 初期値の設定
qp.solve();         // 求解を指示
```

変数のすべてのコンポーネントに同じ初期値を設定するなどの場合には，モデルの中の集合や要素オブジェクトを利用して

```
qp.x[qp.j] = 2.0;  // モデル中で x[j] = 2.0; とするのと同じ
```

あるいは

```
Element j(set=qp.T); // モデル内の T をまわる要素を宣言する.
qp.x[j] = 2.0;       // j の回る範囲について x の初期値を設定する.
```

などの SIMPLE 独特の記述方法が可能です．ここで，通常の int 型のループ変数を使って

```
int j;
for ( j = 1 ; j <= 3 ; ++j ) {
    qp.x[j] = 2.0;
}
```

のように書くこともできますが，特に大規模なモデルの場合（ループ長が 1000 以上になるような場合），には，Element をつかった表現にすることをお勧めします．これは SIMPLE の処理は Element の利用を前提として最適化されているためです．

simple_printf() を利用すると，オブジェクトの内容を書式を設定して出力することができます．例えば useClass.cc で求解を行ったのちに，

```
simple_printf("x[%3d] = %10.3e ",qp.j,qp.x[qp.j]);
```

と書くと

```
x[  1] = 9.394e-001 x[  2] = 1.212e-001
```

のような出力が得られます（同様の書式で simple_fprintf(fp,...); とすれば任意のファイルに結果を出力することができます）．

第6章 外部接続時に利用されるSIMPLEのツール

プロジェクト loadmodule の driver.cpp やプロジェクト noClass の noClass.cc（UNIX 版では useClass.cc や useSimple.cc）では SIMPLE のモデルを定義するだけではなく，値を設定したり解を取得したりしていますが，このような場合に必要なツールについて解説します．

6.1 モデルから作成されたオブジェクト（システムオブジェクト）の操作

driver.cpp（UNIX 版では useClass.cc）で行っているように，genClass で生成したクラス定義から

```
System_knapsack knap(c,a,b);
```

として生成されたオブジェクト（ここでは knap）がデータを与えられて作成したその数理計画モデル（この場合には knapsack 問題）そのものに対応します（以下ではこれを説明のため「システムオブジェクト」と呼びます）．ここで

```
knap.show();
```

とすると，問題の中身が表示されます（showSystem() と同じ）．

システムオブジェクトと数理計画モデルの構成要素については，一般的な原則として以下があります．

> System_NAME のオブジェクト s について s.x は NAME.smp 中のオブジェクト x に対応する．

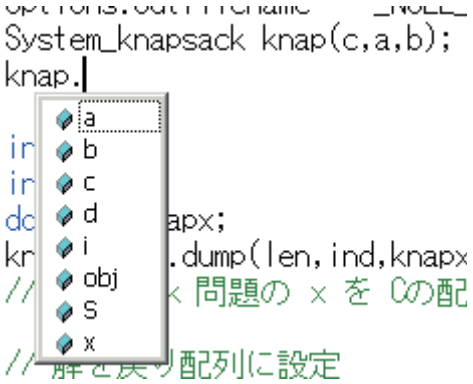
すなわち，

knapsack.smp

```
//
//   ナップサック問題
//
    Set S;
    Element i(set=S);
    IntegerVariable x(index=i,type=binary); // 整数変数
    Parameter c(index=i,required);
    Parameter a(index=i,required);
    Parameter b(required);
    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
```

と定義されたモデルから生成された knap について，VC++の GUI で“.”を打った後に以下のように，参照可能なメンバーが現れますが，それぞれはこのモデル中のオブジェクトに対応しています．



例えば

```
knap.x
```

はモデル中の変数（Variable x）に対応し，このオブジェクトに対する表示手続き：

```
print(),cout,simple_printf()
```

は通常の SIMPLE オブジェクトに対するのと全く同様に可能です．すなわち

```
knap.x.print();
simple_printf("objective = %d\n",knap.obj);
```

とすると，それぞれ knapsack.smp のモデル中の x，obj の値が表示されます．

```
simple_printf("x[%s] = %d\n",knap.i,knap.x[knap.i]);
```

は少々複雑ですが，“knap.i”はモデル中の Element i の意味となりますので，モデル中で

```
simple_printf("x[%s] = %d\n",i,x[i]);
```

としたのと同じ，変数と添字の書式付表示となります．

6.2 C/C++の配列の内容の設定

モデルを操作するルーチンが行う最初の操作は C/C++の配列として渡されたデータを SIMPLE のオブジェクトに設定するということです．スカラー値の場合には，double または int の値をそのまま

```
Parameter b;
b = barg; // barg は double の入力引数
```

と代入できますが，添字付けられた大量のオブジェクトを配列から一気に読み込むには readD という手続きを使います．readD は一般に C の配列から SIMPLE のオブジェクトを設定するためのツールで，本章のようなアプリケーション接続の際によく利用されます．呼び出しの一般形は次の通りです．

```
void readD(int len1,int len2,..,int lenM,double* data) const
```

引数の型:

len1,len2,...,lenM： 次元のサイズ

data： 実際のデータ

与えられる配列データ（data）を一般の多次元配列（添字は 1 から始まる整数値）に読みかえて解釈するので，行列など多次元の配列を定義することができます．次は実際の利用例です．

readD の使用例

```
// 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 次元配列データ
  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);
  Parameter p2(index=i2);   // 2 次元配列データ
  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);
  Parameter p3(index=i3);  // 3 次元配列データ
  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
```

6.3 求解

データの設定が済んだら数理計画問題を解きますが，loadmodule プロジェクトの driver.cpp（UNIX

版では useClass.cc）のように，genClass で生成したクラス定義から作ったモデルを解いている場合には宣言を

```
System_knapsack knap(c,a,b); // knapsack 問題の求解
```

と宣言を行えば自動的に求解が行われます．変数の初期値の設定を行ってから求解したい場合などは

```
options.noDefaultSolve = 1;
```

と設定してからオブジェクトの宣言を行い，

```
knap.x[i] = 1; // 初期値の設定
knap.solve(); // ここで求解
```

と solve() という手続きを呼びます．

noClass プロジェクトの noClass.cpp（UNIX 版では MIP.cc）の場合のようにモデル定義と操作を同時に行っている場合には

```
solve(); // 最適化の実行
```

とすれば，それ以前に定義したモデルに対する求解が行われます．

6.4 C の配列への書き出し

操作する手続きが次に行うのは，求解した結果を C++の配列の形で取得することです．システムオブジェクトに対して dump() という手続きを起動します．loadmodule プロジェクトの driver.cpp（UNIX 版では useClass.cc）では

```
int len;
char** ind;
double* knapx;
knap.x.val.dump(len,ind,knapx);
```

と行うことによって，C++の配列 knapx に解を書き出しています．一般に dump の引数は

```
dump(int len,char*& ind,double* data);
```

で，

len:　ind，data の総長さ
ind:　インデックス文字列の配列（長さ len）
data:　データ本体（長さ len）

です．data[0],data[1],... には添字 ind[0],ind[1],... がそれぞれ対応しています．データの並び順はインデックスをソートした際の自然な順番（数字の場合には昇順，文字列の場合には辞書順）となり，この場合，入力データは readD によって設定されているので，データ並びに対応する knapx に

は番号順に値が設定されることが保証されます．そのため，driver.cpp（UNIX 版では useClass.cc）ではインデックスを見ないで戻りの配列に値を設定しています．なお，ind，data に対応する領域は dump 内部で確保されるためコールした側で解放する必要があります．スカラー値（添字のないオブジェクト）は長さ 1 のオブジェクト（添字の値としてはヌル文字列が設定されます）として同様に dump で取得することができますが，便利な手続きとして double を返す asDouble() という手続きがあり，

```
double objval = obj.val.asDouble();
```

のように値を取ることができます．

6.5 計算結果に関する情報の取得

大域変数である result に直前の最適化の結果が設定されます．よく利用されるのは

```
int result.status  ... 最適化の終了時の状態
```

で，最適化のステータスコード（正常終了時には 0）を返します．

```
double result.optValue;
```

は最適化後の目的関数の値で，dump() による取得よりも簡便なので，ここでは目的関数値を戻すのに利用しています．

6.6 求解操作と代入

VariableParameter の値を変化させながら複数回最適化を行う場合，モデルに対応するシステムオブジェクト model について行う場合には

```
for ( int p = 1; p <= 5 ; ++p ) {
  model.p = p; // VariableParameter の設定
  model.solve();// 求解
}
```

のように記述を行います．（p がモデル中の VariableParameter の名前とします）．システムオブジェクトのメンバ（上記では p）に対しての代入は通常のモデル内での代入と同様に機能します．

noClass プロジェクトの noClass.cpp の場合のようにモデル定義と操作を同時に行っている場合には

```
for ( int pi = 1; pi <= 5 ; ++pi ) {
  p = pi; // VariableParameter の設定
  solve();// 求解
}
```

とします（VariableParameter p と同じ名前の int 変数 p を使うことはできませんので，pi としてい

ます).

6.7 Numerical Optimizer オプション

プログラム内で Numerical Optimizer の動作を制御するパラメータを設定することができます.

SIMPLE モデルに対してパラメータを設定するには，モデルやドライバ中に

```
options. パラメータ名 = 値;
```

と書きます．“options”は大域的に有効な nuoptParam のクラスオブジェクトです．設定した内容は次の Numerical Optimizer の起動時に反映されます．“options”は大域的に有効なので，以前の指定が残ることにご注意ください.

solveLP，solveQP を用いて問題を解く際に Numerical Optimizer のパラメータを設定するには

```
nuoptParam param;
```

のように Numerical Optimizer のパラメータ群を示す nuoptParam なるクラスのオブジェクトを定義して（名前は任意),

```
param. パラメータ名 = 値
```

として，パラメータを設定，その nuoptParam のオブジェクトを solveLP，solveQP の最初の引数として与えます．こうすると，その問題を解く際（solveLP，solveQP のコール）に対してここで設定したパラメータが与えられます．例えばモデルのアルゴリズムを単体法に指定する場合は,

```
nuoptParam param;
param.method = "simplex";
```

と書きます.

nuoptParam クラスで設定できるパラメータは，モデリング言語 SIMPLE の options で定義できるパラメータと同じです．設定できるパラメータの詳細については「Numerical Optimizer/SIMPLE マニュアル」の「パラメータ設定」をご参考ください.

第 7 章　VC++プロジェクトの設定

7.1　Microsoft Visual Studio プロジェクトの設定

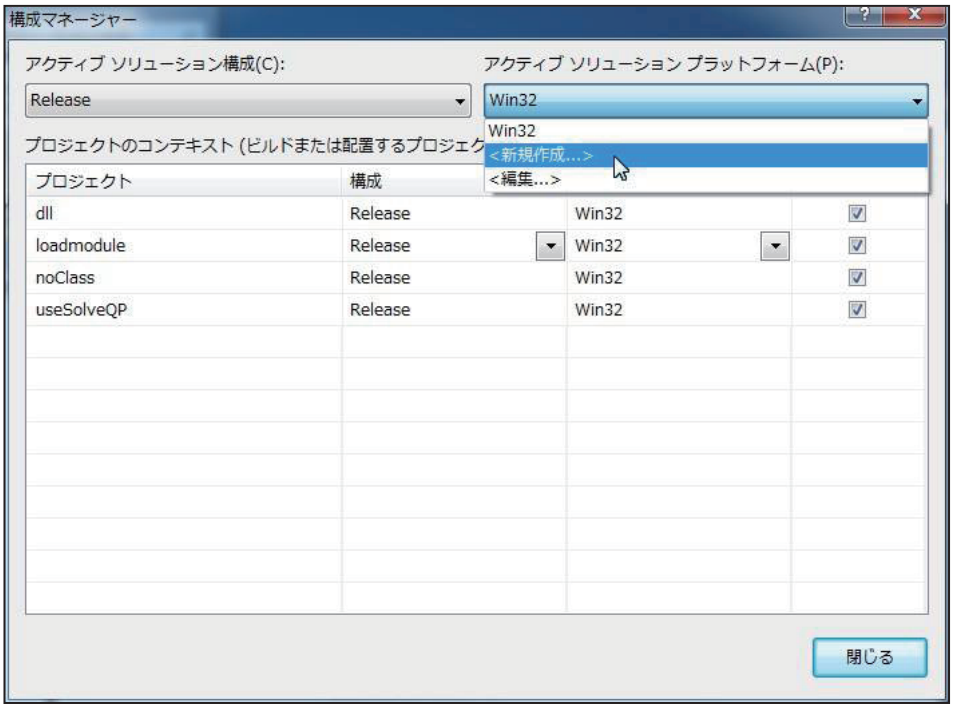
ソリューション

```
nuopvcapp
```

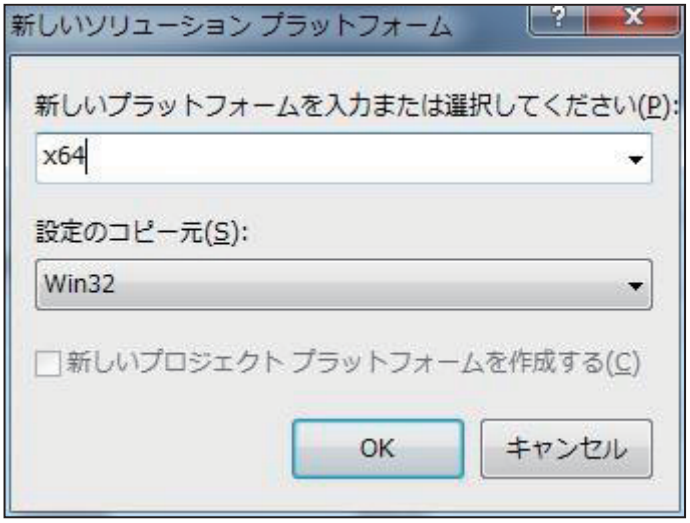
に追加されているプロジェクトでは設定済みですが，一般に Numerical Optimizer のライブラリとの連結を行ってライブラリ（.LIB），あるいは DLL を作成する場合には，プロジェクトの設定を行う必要があります．以下では，Visual Studio から Numerical Optimizer を利用する際に行う必要のある設定を紹介します．なお，ここでは Visual Studio 2010 の画面をもとに説明していますが，他のバージョンでも同様の設定を実施してください．

まず，Numerical Optimizer のインストール時に 64bit コンパイラを選択した場合は，Win32 プロジェクト設定を 64 ビットプロジェクト構成に変更する必要があります．以下この変更方法について説明します．最初に画面上部にあるコンボボックスをクリックし，「構成マネージャー」を選択します．

次にアクティブソリューションプラットフォーム欄をクリックし，「新規作成」を選択します．



［新しいプラットフォームを入力または選択してください］欄をクリックし，「x64」と入力後，[OK] ボタンをクリックします．



以上の設定で，ソリューション内のすべてのプロジェクトが 64 ビットプロジェクト構成になります．

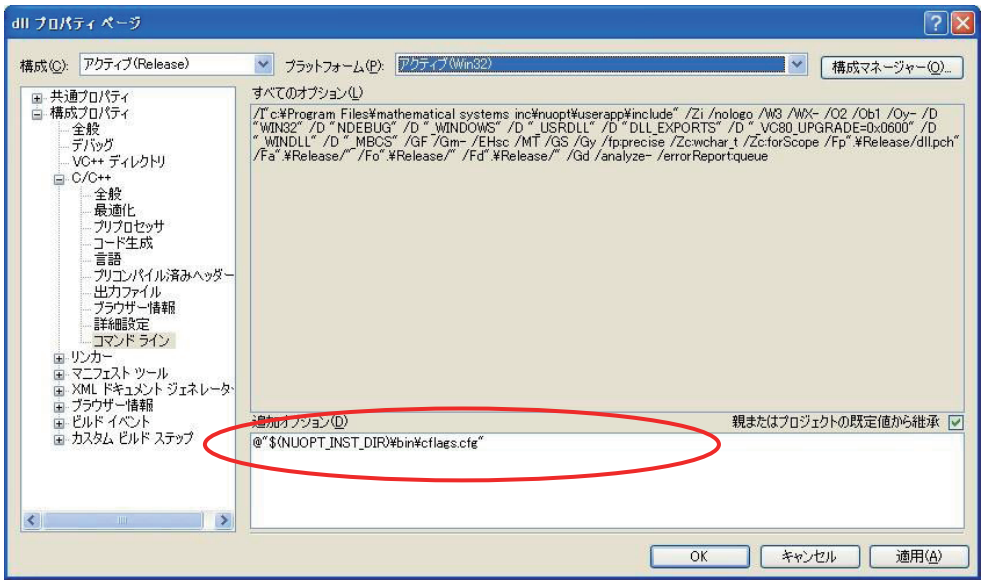
次にプロジェクトの構成を「Release」にする必要があります．下図のように，メニューバー下部にあるコンボボックスに「Release」と設定します．

次にビルドおよびリンクの設定を行います．これより以下の設定は全て［プロジェクト］→［プロパティ］→［構成プロパティ］から行います．まず「構成プロパティ」→「C/C++」→「コマンドライ

ン」と選択し，ウインドウの下部ある「追加のオプション」に

```
@"$(NUOPT_INST_DIR)\bin\cflags.cfg"
```

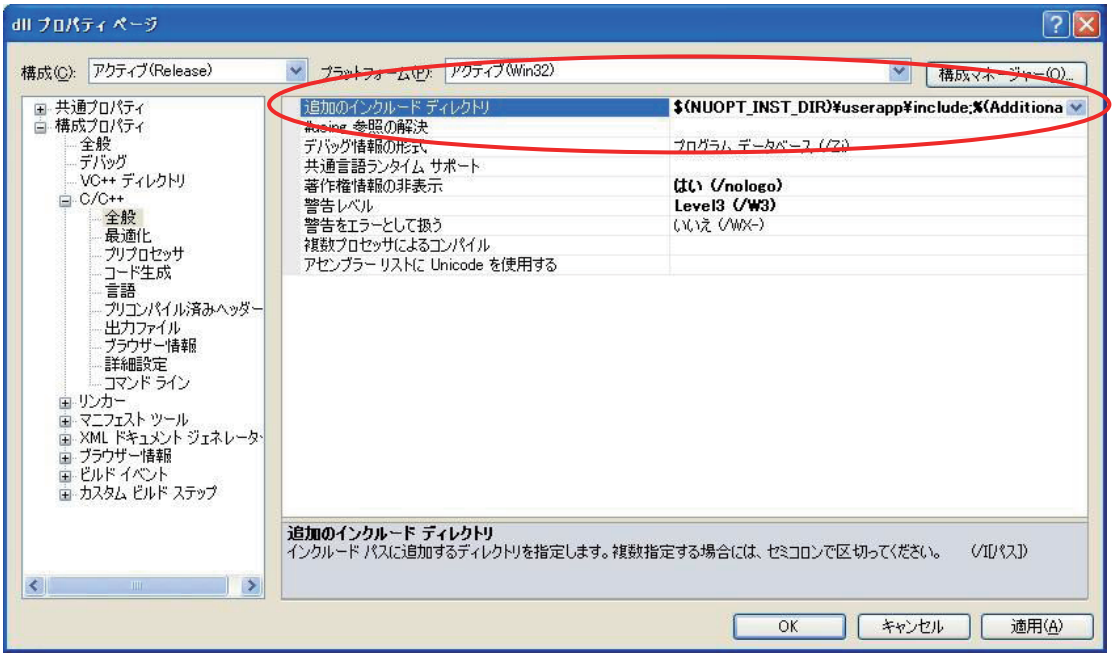
を追加します．NUOPT_INST_DIR は Numerical Optimizer のインストール場所を表す環境変数です．



「構成プロパティ」→「C/C++」→「全般」と選択し，ウインドウ内にある「追加のインクルードディレクトリ」に

```
$(NUOPT_INST_DIR)\userapp\include
```

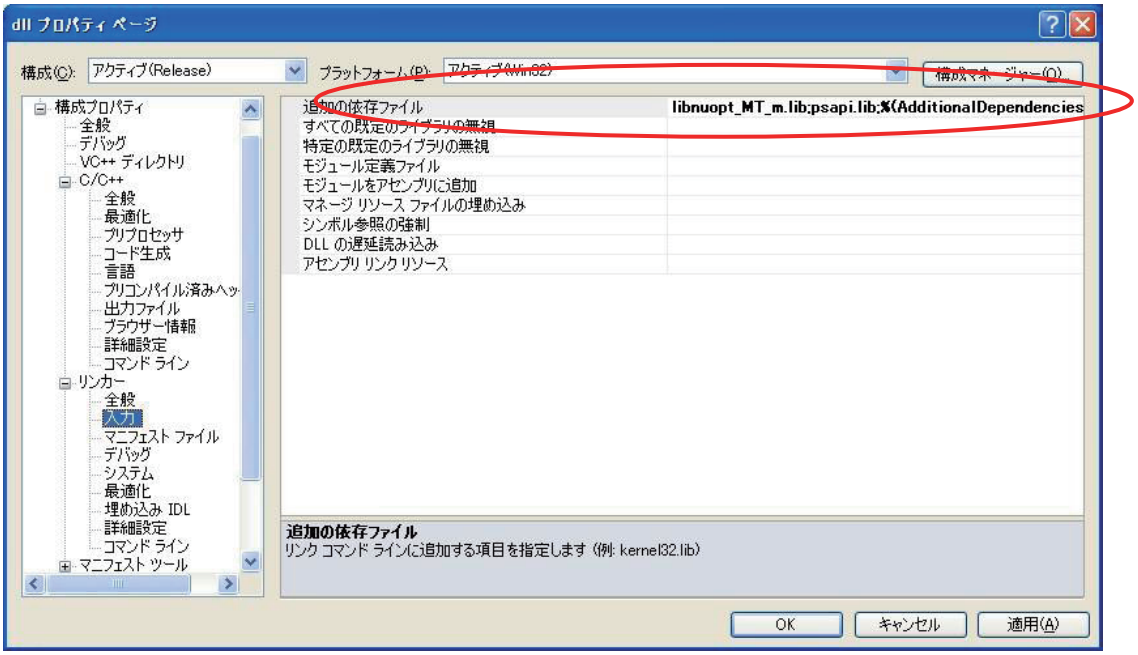
を追加します．なお，下線部は Numerical Optimizer のインストール場所に対応します．



「構成プロパティ」→「リンカ」→「入力」と選択し，ウインドウ内にある「追加の依存ファイル」に

```
libnuopt_MT_m.lib;psapi.lib;
```

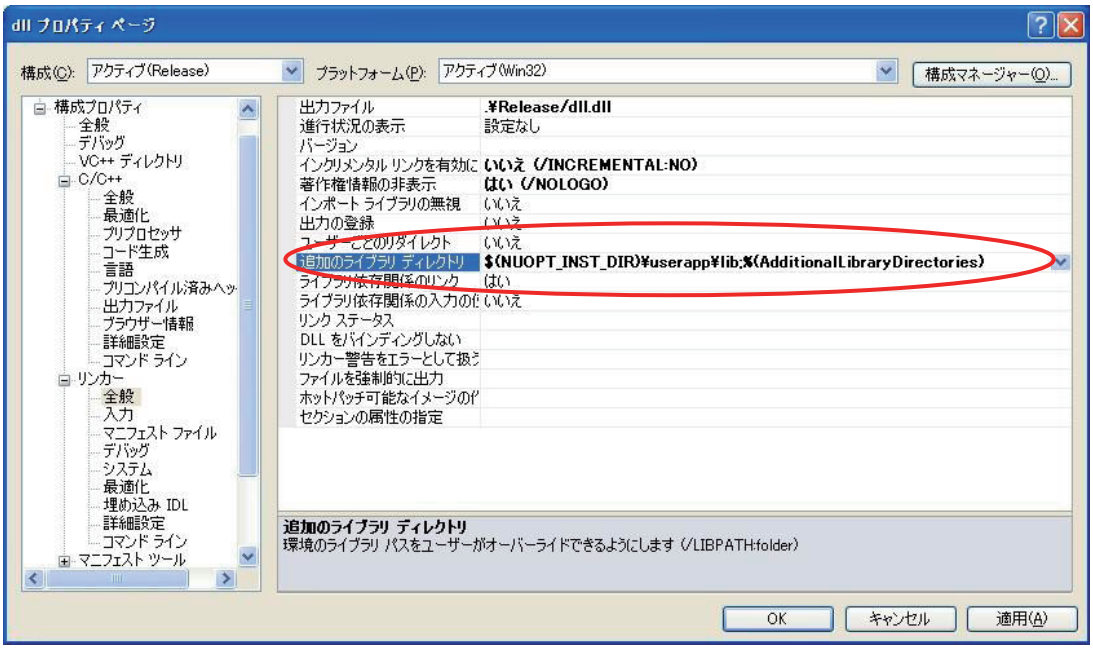
を追加します．なお本設定は Numerical Optimizer のライブラリに接続しているアプリケーションで使用するライブラリがデフォルトの「マルチスレッド」でコンパイルする場合です．詳細は以降にある「※注意 1」をご覧ください．



「構成プロパティ」→「リンカ」→「全般」と選択し，ウインドウ内にある「追加のライブラリディレクトリ」に

```
$(NUOPT_INST_DIR)\userapp\lib;
```

を追加します．上記の下線部は Numerical Optimizer のインストール場所に対応します．



※注意 1（アプリケーションで利用しているランタイムライブラリについて）：

上記は Numerical Optimizer のライブラリに接続しているアプリケーションで使用するランタイムライブラリが「マルチスレッド」でコンパイルされている場合です（この設定はプロジェクト→プロパティ→C/C++→コード生成で現れる「ランタイムライブラリ」の選択に対応します）．もし，Numerical Optimizer のライブラリと接続するコードがデフォルトと違って「マルチスレッド DLL」になっている場合には，3. で指定するライブラリの名前を

```
libnuopt_MD_m.lib
```

としてください．

また，大規模線形計画用内点法ライブラリ solveLP64（3.5）を使用する場合は 3. で指定するライブラリの名前を

```
libnuopt_ipm_MT_m.lib
```

としてください．

7.2 外部 CLAPACK(CBLAS) の使用方法

Numerical Optimizer V13 から外部 CLAPACK(CBLAS) を Numerical Optimizer ライブラリへリンクすることができるようになりました．これにより大規模な線形計画問題において内点法を用いる場合の求解速度が速くなる場合があります．例えば、Intel 社の MKL を使用することで，CPU が Core2 Quad 3GHz(Q9650) で求解時間が 1/3 となった問題がありました[8]．

本節では，外部 CLAPACK(CBLAS) を Numerical Optimizer ライブラリへリンクする方法について説明をします．基本的な設定方法はコンパイラのバージョンが異なっても同じであるため，ここでは Microsoft Visual Studio 2010 を例にします．

まずは，「7.1 Microsoft Visual Studio プロジェクトの設定」を行い，正しくソリューションがビルドできることを確認してください．

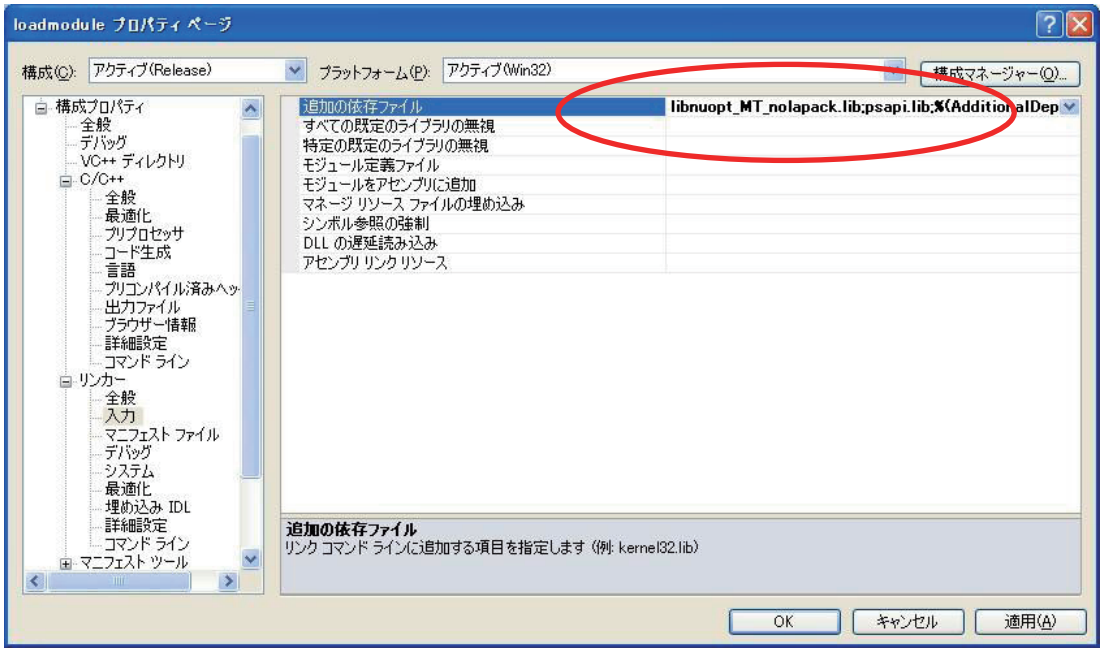
次に，Numerical Optimizer のライブラリを外部 CLAPACK とリンクするためのものに変更します．具体的には，プロジェクト→プロパティ→構成プロパティから，「構成プロパティ」→「リンカ」→「入力」と選択し，ウインドウ内にある「追加の依存ファイル」の

```
libnuopt_MT_m.lib;psapi.lib
```

を次のように変更します．

```
libnuopt_MT_nolapack.lib;psapi.lib
```

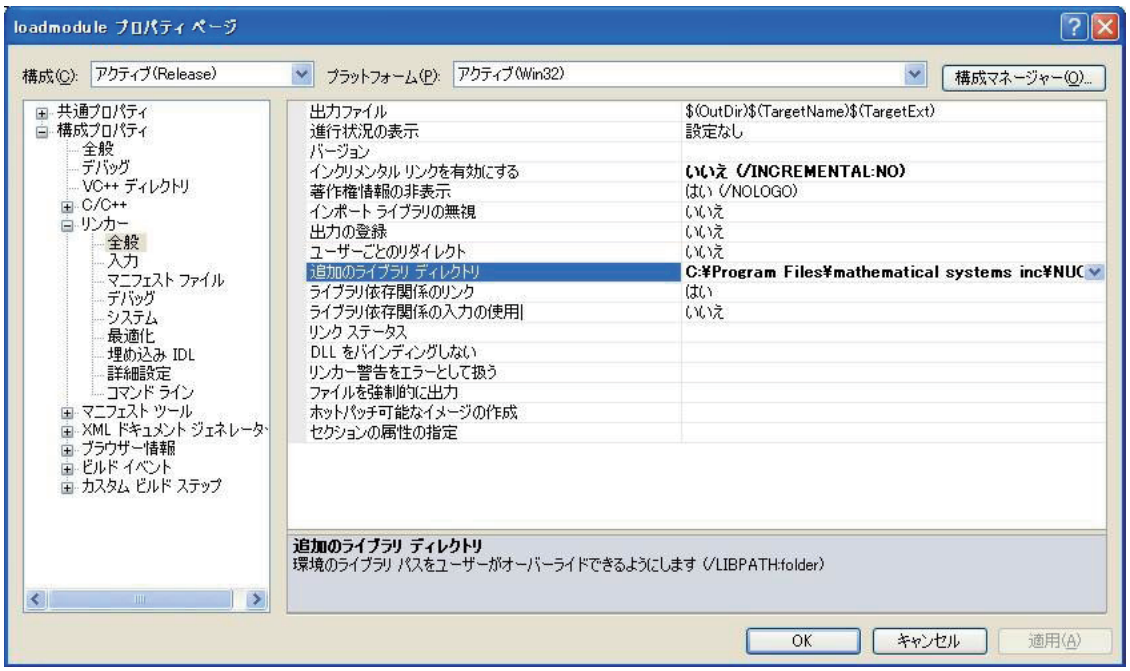
[8] これは結果の一例であり，全ての問題で必ず同程度速度が速くなることを保証できるわけではありません．しかしながら，MKL の場合はスレッド化されるため，Many Core の環境では多くの問題でより高パフォーマンスとなることが期待できます．



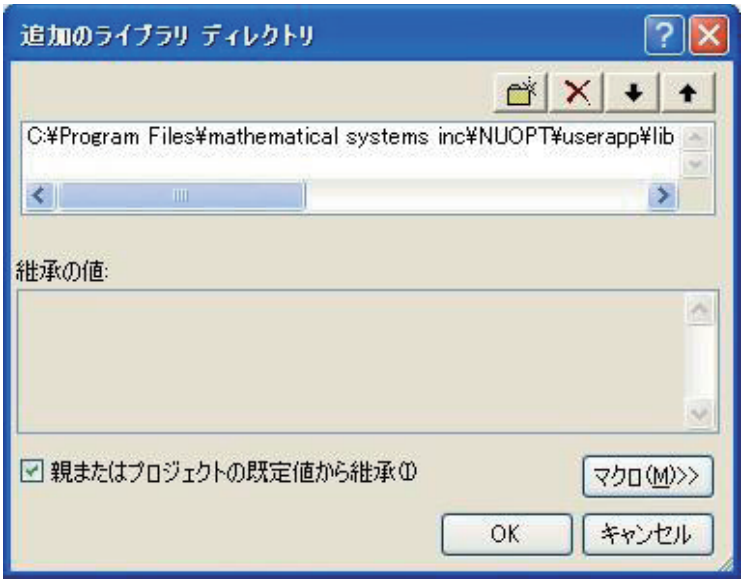
この後，ソリューションのビルドを行うと，以下のようなエラーメッセージが出力され正しくビルドが完了しません．これは外部 CLAPACK(CBLAS) の関数のリンクに失敗しているためです．そのため，以下で外部 CLAPACK(CBLAS) ライブラリの設定を行います．

```
4>リンクしています...
3>libnuopt_MT_nolapack.lib(Vt.obj) : error LNK2001: 外部シンボル"_idamax_"は未解決です。
3>libnuopt_MT_nolapack.lib(rhoupdsdp.obj) : error LNK2001: 外部シンボル"_idamax_"は未解決です。
3>libnuopt_MT_nolapack.lib(dgefa.obj) : error LNK2001: 外部シンボル"_idamax_"は未解決です。
...
```

先の「追加の依存ファイル」に外部 CLAPACK(CBLAS) のライブラリを入力してください．ライブラリ名は使用する外部 CLAPACK(CBLAS) により決まりますので，必要なら CLAPACK(CBLAS) のマニュアル等をご覧ください．次に，この (これらの) ライブラリがあるフォルダを指定します．「構成プロパティ」→「リンカ」→「全般」と選択し，ウインドウ内にある「追加のライブラリディレクトリ」の右側にあるボタンをクリックしてください．



ボタンをクリックすると次のようなダイアログが表示されます.



このダイアログから，先に指定した外部 CLAPACK(CBLAS) のあるフォルダを追加してください．ライブラリが複数ある場合は複数のフォルダを指定する必要があるかもしれません．適切なフォルダに関しては外部 CLAPACK(CBLAS) のマニュアル等をご覧ください.

以上で外部 CLAPACK(CBLAS) の設定になります．ご不明な点がございましたら

nuopt-support@msi.co.jp

までお問い合わせください.

7.3 分枝限定法の並列化の利用

Numerical Optimizer V14 から分枝限定法の並列化ができるようになりました．ここではこの機能を外部接続で利用するにあたって必要な手順を説明します．なお，分枝限定法の並列化に必要な動作環境については「Numerical Optimizer/SIMPLE マニュアル」における「分枝限定法・制約充足問題ソルバ wcsp の並列化」の章を参考にしてください．

まずは，「7.1 Microsoft Visual Studio プロジェクトの設定」を行い，正しくソリューションがビルドできることを確認してください．

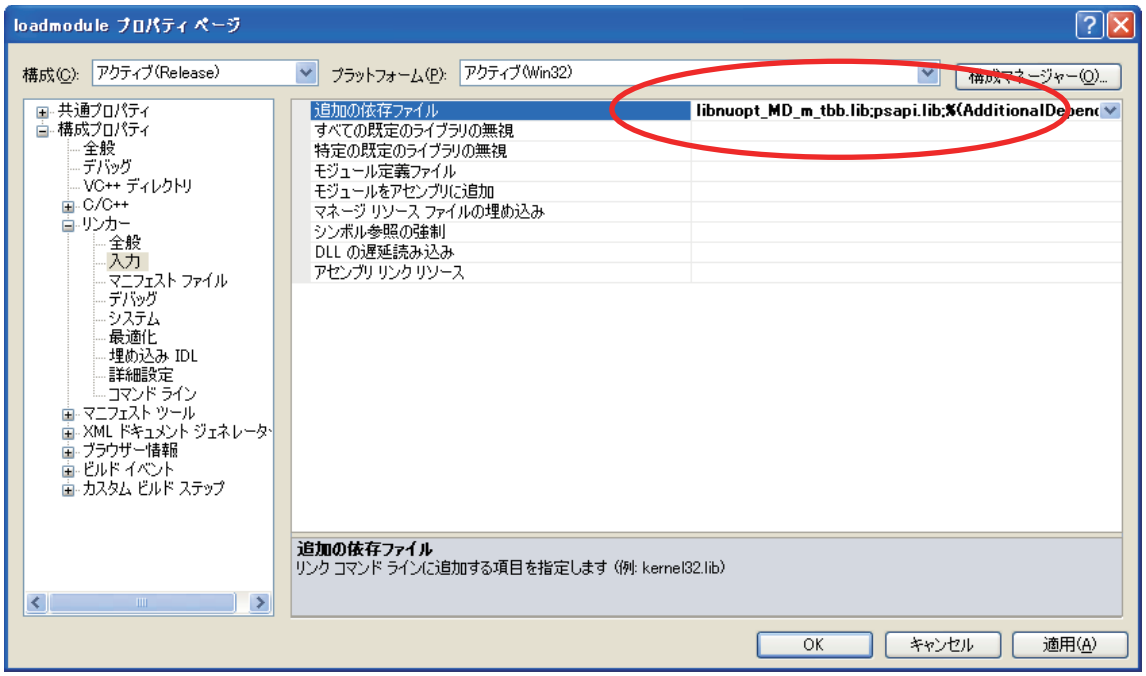
分枝限定法の並列化を利用する場合は，アプリケーションがリンクするランタイムライブラリが「マルチスレッド DLL(/MD)」である必要があります．この設定はプロジェクト→プロパティ→ C/C++→コード生成で現れる「ランタイムライブラリ」において「マルチスレッド DLL(/MD)」を選択することに対応します．

次に，Numerical Optimizer のライブラリを分枝限定法の並列化処理対応するライブラリに変更します．具体的には，プロジェクト→プロパティ→構成プロパティから，「構成プロパティ」→「リンカ」→「入力」と選択し，ウインドウ内にある「追加の依存ファイル」の

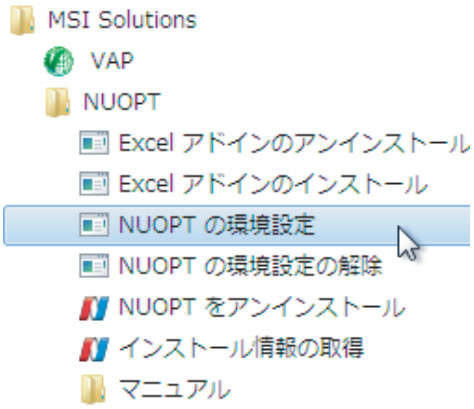
```
libnuopt_MT_m.lib;psapi.lib
```

を次のように変更します．

```
libnuopt_MD_m_tbb.lib;psapi.lib
```



最後に，作成されたアプリケーションの実行には Numerical Optimizer の配布する DLL を利用する必要があります．このため［すべてのプログラム］->[MSI Solutions]->[NUOPT]->[NUOPT の環境設定]を選択し実行に必要な環境を設定します．



以上で分枝限定法の並列化の設定になります．ご不明な点がございましたら

nuopt-support@msi.co.jp

までお問い合わせください．

参考文献

[1] W. Hock and K. Shittkowski, Test examples for nonlinear programming codes, Springer Verlag, 1981.

索　引

P

R

S

U

W

え

か

き

く

こ

し

せ

と

な

に

は

ふ

め

ら

れ