USER'S GUIDE FOR TOMNET - .NET[1] - The Base Module

Per Strandberg[2] and Marcus M. Edvall[3]

February 27, 2007

[1]More information available at the TOMNET home page: http://tomopt.com/tomnet/. E-mail: **tomlab@tomopt.com**.
[2]Tomlab Optimization AB, Västerås Technology Park, Trefasgatan 4, SE-721 30 Västerås, Sweden, per.strandberg@tomopt.com.
[3]Tomlab Optimization Inc., 855 Beech St #121, San Diego, CA, USA, medvall@tomopt.com.

Contents

1 Introduction

TOMNET is a general purpose development environment for practical solution of optimization problems using .NET.

TOMNET has grown out of the need for advanced, robust and reliable tools to be used in the development of algorithms and software for the solution of many different types of applied optimization problems.

There are many good tools available in the area of numerical analysis, operations research and optimization, but because of the different languages and systems, as well as a lack of standardization, it is a time consuming and complicated task to use these tools. Often one has to rewrite the problem formulation, rewrite the function specifications, or build some new interface routine to make everything work. Therefore the first obvious and basic design principle in TOMNET is: *Define your problem once, run all available solvers.* The system takes care of all interface problems, whether related to the solver or due to different demands on the problem specification.

Sometimes it is not clear what solver is best for the particular type of problem and tests on different solvers can be quite useful. When developing new solutions tests on thousands of problems are necessary to fully assess the pros and cons of the setup. One might want to solve a practical problem very many times, with slightly different conditions for the run.

All these issues and many more are addressed with the TOMNET optimization environment. TOMNET gives easy access to a large set of standard test problems, optimization solvers and utilities. Furthermore, it is easy to define new problems within the problem classes, and solve them using any suitable solver objects.

1.1 Overview

Welcome to the TOMNET User's Guide. The Base Module includes a set of solvers for use in .NET and interface routines needed for all other packages.

TOMNET includes the following solvers:

- glbDirect - an unconstrained global solver suitable for problems with up to 100 variables. Only simple bounds on the decision variables are supported.
- glcDirect - a constrained global solver for problems with up to 100 variables. The solver accepts linear, integer and nonlinear constraints.

1.2 Contents of this manual

- Section 1 provides a basic overview of the TOMNET Base Module package.
- Section 2 details the system design.
- Section 3 defines the problem types and format for optimization as well as the solver suites available.
- Section 4 shows how to setup and define the problems properly.
- Section 4.2 provides information regarding .NET test cases for use with the solvers.
- Section 5 provides an overview of accessing the solvers.
- Section 5.1 describes how to set solver options from .NET.

- Section 6 gives detailed solver references for the Base Module.
- Section 7 contains information on how to use the problem classes, and how to use the tools for creating custom objective functions and constraints.
- Section 8 specifies how derivatives are obtained.
- Section 9 gives some additional notes.
- Section 10 contains all reference material for sparse matrix handling.

1.3 More information

Please visit the following links for more information and see the illustrative references at the end of this manual.

- http://tomopt.com/tomnet/
- http://tomopt.com/tomnet/products/base/

1.4 Prerequisites

In this manual we assume that the user is familiar with some basics about optimization and nonlinear programming, and with the .NET programming environment in general.

2 Overall Design

The scope of TOMNET is large and broad, hence there is a need for a well-designed system. It is natural to use the power of the .NET environment to make the system flexible and easy to use and maintain.

2.1 Input and Output

TOMNET solves the problem in two steps. First the problem is defined and stored in a .NET object. Any applicable solver can then be called from an instance of *Result* (a standardized result container).

2.2 Problem Types

TOMNET solves a number of different optimization problems. The currently defined types are listed in Table 1.

Table 1: The different types of optimization problems defined in TOMNET.

probType		Type of optimization problem
uc	1	Unconstrained optimization (incl. bound constraints).
qp	2	Quadratic programming.
con	3	Constrained nonlinear optimization.
ls	4	Nonlinear least squares problems (incl. bound constraints).
lls	5	Linear least squares problems.
cls	6	Constrained nonlinear least squares problems.
mip	7	Mixed-Integer programming.
lp	8	Linear programming.
glb	9	Box-bounded global optimization.
glc	10	Global mixed-integer nonlinear programming.
miqp	11	Constrained mixed-integer quadratic programming.
minlp	12	Constrained mixed-integer nonlinear optimization.

Each *probType* corresponds to a problem class (exceptions applies).

Table 2: Problem classes and problem types in TOMNET.

Class	Type	Ref.
`LinearProblem`	**lp**	7.1
`QuadraticProblem`	**qp**	7.2
`NonlinearProblem`	**uc, glb**	7.3
`ConProblem`	**uc, con, glb**	7.4
`ConLinProblem`	**con**	7.5
`LinearLeastSquaresProblem`	**lls**	7.6
`NonlinearLeastSquaresProblem`	**ls, cls**	7.7

There are a number of test problems included with TOMNET. The folder *testprob* contains a large set of test problems, primarily for benchmark testing and development. The problems are implemented in .NET and the user functions are coded in C. The folder *quickguide* contains smaller problems on the standard format. It is recommended to look at the quick guide problems when implementing custom problems.

The following table defines the test problems included with the TOMNET Base Module.

Table 3: Defined test problem sets in TOMNET.

probSet	**probType**	**Description of test problem set**
chsProb	3	Hock-Schittkowski constrained test problems.
clsProb	6	Constrained nonlinear least squares problems.
conProb	3	Constrained test problems.
glbProb	9	Box-bounded global optimization test problems.
glcProb	10	Global MINLP test problems.
lgoProb	9	Box-bounded global optimization test problems (LGO).
llsProb	5	Linear least squares problems.
lpProb	8	Linear programming problems.
mghProb	4	More, Garbow, Hillstrom nonlinear least squares problems.

Table 3: Defined test problem sets in TOMNET, continued.

probSet	probType	Description of test problem set
minlpProb	12	Mixed-integer nonlinear programming problems.
mipProb	7	Mixed-integer programming problems.
miqpProb	11	Mixed-integer quadratic programming problems.
qpProb	2	Quadratic programming test problems.
ucProb	1	Unconstrained test problems.
uhsProb	1	Hock-Schittkowski unconstrained test problems.

The problem may be accessed by creating instance using the class constructors or methods.

2.3 Solution Process

The process of optimization in TOMNET is quite straight forward, even to the beginner. It is efficient and easy to use the interactive system. In general, the following process may be followed:

- Identify the problem type to be solved.
- Select and use the constructor to create a problem instance.
- Create an appropriate solver instance.
- Modify solver options as needed.
- Call the method *Solve* of the TOMNET Solver class.

If solving one of the pre-defined problems:

- Create a problem instance. Select the problem set and number.
- Create a solver instance and modify its options if needed.
- Use *solve* to call the solver.

Depending on the type of problem, the user needs to supply the routines that calculate the objective function, constraint functions for constrained problems, and also if possible, derivatives. To simplify the coding process, TOMNET provides predefined interfaces to for the objective functions of a nonlinear problem.

For example, when working with a least squares problem, it is natural to code the function that computes the vector of residual functions $r_i(x_1, x_2, \ldots)$, since a dedicated least squares solver probably operates on the residual while a general nonlinear solver needs a scalar function, in this case $f(x) = \frac{1}{2} r^T(x) r(x)$.

If derivatives are not given and the selected solver requires this information, numerical differentiation can be turned on.

2.3.1 Examples in TOMNET

This section is to illustrate how to use some routines in TOMNET. For example one can try to solve one of the chsProb examples with SNOPT. The following code snippet, available in `TOMNET/usersguide/ChsProb8/` illustrates the solution process.

chsProb problem number 8 is solved. The problem is created with its constructor, subsequently solved by calling the solvers `Solve`-method. The outputs are the optimal solution x_k, the optimal functional value f_k and the time consumed to solve the problem, all stored in an instance of the `Result` class.

```
using System;
using TOMNET;

class ChsProb8
{
  static void Main(string[] args)
  {
    // 1 - Create Problem, Solver and Result.
    chsProbProblem prob = new chsProbProblem(8);
    SNOPT solver = new SNOPT();
    Result result;

    // 2 - Solve the problem.
    solver.Solve(prob, out result);

    // 3 - Write the optimum, objective value
    //     and time on the console.
    Console.Write("x_k  =");
    foreach (double d in result.x_k)
      Console.Write(" {0}", d);
    Console.WriteLine();

    Console.WriteLine("f_k  = {0}", result.f_k);
    Console.WriteLine("time = {0}", result.REALtime);
  }
}
```

It is also easy to solve a quadratic programming problem using for example SQOPT. The problem is created and solved in the same manner. Since this example uses the built-in class `QuadraticProblem` the syntax is a little different. The example as a whole is available in `TOMNET/usersguide/QpProb15/`

```
using System;
using TOMNET;

class QPProb15
{
  static void Main(string[] args)
  {
```

```
    // 1 - Create Problem, Solver and Result.
    QuadraticProblem prob = qpProb.getQpProb(15);
    SQOPT solver = new SQOPT();
    Result result;

    // 2 - Solve the problem.
    solver.Solve(prob, out result);

    // 3 - Write the objective value
    //     and time on the console.
    Console.WriteLine("f_k  = {0}", result.f_k);
    Console.WriteLine("time = {0}", result.REALtime);
  }
}
```

The qpProb test problem number 15 is created and solved. The output printing included here is the optimal objective value and the time consumed. The ouput but may be expanded to include further information by accessing the `Result` object.

3 Problem types and Solver Routines

Table 1 defines all problem types in TOMNET. Each formal problem definition below is accompanied by brief suggestions about suitable solvers. This is followed in Section 3.2 by a complete list of the available solver routines in TOMNET and the various available extensions, such as /SOL.

3.1 Definition of Problem Types

3.1.1 Unconstrained Optimization

The **unconstrained optimization** (**uc**) problem is defined as

$$\begin{array}{lll} \min\limits_{x} & f(x) & \\ s/t & x_L \leq x \leq x_U, & \end{array} \tag{1}$$

where $x, x_L, x_U \in \mathbb{R}^n$ and $f(x) \in \mathbb{R}$. For unbounded variables, the corresponding elements of x_L, x_U may be set to $\pm\infty$.

Recommended solvers: **TOMNET /NPSOL and TOMNET /SNOPT**.

3.1.2 Quadratic Programming

The **quadratic programming** (**qp**) problem is defined as

$$\begin{array}{lll} \min\limits_{x} & f(x) = \frac{1}{2}x^T F x + c^T x & \\ s/t & \begin{array}{l} x_L \leq x \leq x_U, \\ b_L \leq Ax \leq b_U \end{array} & \end{array} \tag{2}$$

where $c, x, x_L, x_U \in \mathbb{R}^n$, $F \in \mathbb{R}^{n\times n}$, $A \in \mathbb{R}^{m_1 \times n}$, and $b_L, b_U \in \mathbb{R}^{m_1}$.

Recommended solvers: **TOMNET /QPOPT, TOMNET /SNOPT and TOMNET /SQOPT**.

A positive semidefinite F-matrix gives a convex QP, otherwise the problem is nonconvex. Nonconvex quadratic programs are solved with a standard active-set method [17], implemented in for example routine *SNOPT*. It converges to a local minimum for indefinite quadratic programs.

MINOS, or the dense QP solver *QPOPT* may be used. In the TOMNET /SOL extension the *SQOPT* solver is suitable for both dense and large, sparse convex QP and *SNOPT* works fine for dense or sparse nonconvex QP.

For very large-scale problems, an interior point solver is recommended, such as TOMNET /CPLEX.

3.1.3 Constrained Nonlinear Optimization

The **constrained nonlinear optimization** problem (**con**) is defined as

$$\begin{array}{lll} \min\limits_{x} & f(x) & \\ s/t & \begin{array}{l} x_L \leq x \leq x_U, \\ b_L \leq Ax \leq b_U \\ c_L \leq c(x) \leq c_U \end{array} & \end{array} \tag{3}$$

where $x, x_L, x_U \in \mathbb{R}^n$, $f(x) \in \mathbb{R}$, $A \in \mathbb{R}^{m_1 \times n}$, $b_L, b_U \in \mathbb{R}^{m_1}$ and $c_L, c(x), c_U \in \mathbb{R}^{m_2}$.

Recommended solvers: **TOMNET /SNOPT, TOMNET /NPSOL and TOMNET /MINOS**.

TOMNET /MINOS solves constrained nonlinear programs. The TOMNET /SOL extension gives an additional set of general solvers for dense or sparse problems.

3.1.4 Box-Bounded Global Optimization

The **box-bounded global optimization** (**glb**) problem is defined as

$$\begin{array}{lll} \min\limits_{x} & f(x) & \\ s/t & -\infty < & x_L \leq x \leq x_U < \infty, \end{array} \tag{4}$$

where $x, x_L, x_U \in \mathbb{R}^n$, $f(x) \in \mathbb{R}$, i.e. problems of the form (1) that have finite simple bounds on all variables.

Recommended solver: **TOMNET /LGO**.

The TOMNET Base Module solver *glbDirect* implements the DIRECT algorithm [5], which is a modification of the standard Lipschitzian approach that eliminates the need to specify a Lipschitz constant. In *glbDirect* no derivative information is used.

3.1.5 Global Mixed-Integer Nonlinear Programming

The **global mixed-integer nonlinear programming** (**glc**) problem is defined as

$$\begin{array}{lllllll} \min\limits_{x} & f(x) & & & & & \\ & -\infty < & x_L & \leq & x & \leq & x_U < \infty \\ s/t & & b_L & \leq & Ax & \leq & b_U \\ & & c_L & \leq & c(x) & \leq & c_U, \quad x_j \in \mathbb{N} \quad \forall j \in \mathrm{I}, \end{array} \tag{5}$$

where $x, x_L, x_U \in \mathbb{R}^n$, $f(x) \in \mathbb{R}$, $A \in \mathbb{R}^{m_1 \times n}$, $b_L, b_U \in \mathbb{R}^{m_1}$ and $c_L, c(x), c_U \in \mathbb{R}^{m_2}$. The variables $x \in I$, the index subset of $1, ..., n$, are restricted to be integers.

Recommended solvers: **glcDirect**.

The TOMNET Base Module solver *glcDirect* implements an extended version of the DIRECT algorithm [16], that handles problems with both nonlinear and integer constraints.

3.1.6 Linear Programming

The **linear programming** (**lp**) problem is defined as

$$\begin{array}{lllll} \min\limits_{x} & f(x) = c^T x & & & \\ s/t & x_L & \leq & x & \leq & x_U, \\ & b_L & \leq & Ax & \leq & b_U \end{array} \tag{6}$$

where $c, x, x_L, x_U \in \mathbb{R}^n$, $A \in \mathbb{R}^{m_1 \times n}$, and $b_L, b_U \in \mathbb{R}^{m_1}$.

Recommended solvers: **TOMNET /CPLEX**.

The TOMNET Base Module solver *milpsolve* implements simplex algorithms for **lp** problems.

When a dual feasible point is known in (6) it is efficient to use the dual simplex algorithm implemented in the TOMNET Base Module solver *milpsolve*. *MINOS* is efficient for solving large, sparse LP problems. Dense problems are solved by *QPOPT*. The TOMNET /SOL extension gives the additional possibility of using *SQOPT* to solve large, sparse LP.

The recommended solver normally outperforms all other solvers.

3.1.7 Mixed-Integer Programming

The **mixed-integer programming problem** (**mip**) is defined as

$$\begin{array}{lll} \min\limits_{x} & f(x) = c^T x & \\ & & \\ s/t & \begin{array}{ccccc} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U,\ x_j \in \mathbb{N} \quad \forall j \in \mathrm{I} \end{array} & \end{array} \tag{7}$$

where $c, x, x_L, x_U \in \mathbb{R}^n$, $A \in \mathbb{R}^{m_1 \times n}$, and $b_L, b_U \in \mathbb{R}^{m_1}$. The variables $x \in I$, the index subset of $1, ..., n$ are restricted to be integers. Equality constraints are defined by setting the lower bound equal to the upper bound, i.e. for constraint i: $b_L(i) = b_U(i)$.

Recommended solver: **TOMNET /CPLEX**.

Mixed-integer programs can be solved using the TOMNET Base Module routine *milpsolve* that implements a standard branch-and-bound algorithm, see Nemhauser and Wolsey [19, chap. 8]. *milpsolve* also implements user defined priorities for variable selection, and different tree search strategies. The TOMNET /CPLEX extension gives access to the state-of-the-art LP, QP, MILP and MIQP solver CPLEX. For many MIP problems, it is necessary to use such a powerful solver, if the solution needs be obtained in any reasonable time frame.

3.1.8 Linear Least Squares Problems

The **linear least squares** (**lls**) problem is defined as

$$\begin{array}{lll} \min\limits_{x} & f(x) = \frac{1}{2}||Cx - d|| & \\ & & \\ s/t & \begin{array}{ccccc} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U \end{array} & \end{array} \tag{8}$$

where $x, x_L, x_U \in \mathbb{R}^n$, $d \in \mathbb{R}^M$, $C \in \mathbb{R}^{M \times n}$, $A \in \mathbb{R}^{m_1 \times n}$, $b_L, b_U \in \mathbb{R}^{m_1}$ and $c_L, c(x), c_U \in \mathbb{R}^{m_2}$.

Recommended solvers: **TOMNET /LSSOL or TOMNET /LSQR**.

LSQR solves **unconstrained** sparse **lls** problems. In the TOMNET /NPSOL or TOMNET /SOL extension the *LSSOL* solver is suitable for dense linear least squares problems.

3.1.9 Constrained Nonlinear Least Squares Problems

The **constrained nonlinear least squares** (**cls**) problem is defined as

$$\begin{array}{lrcccl} \min\limits_{x} & f(x) = \frac{1}{2} r(x)^T r(x) & & & & \\ & & & & & \\ & x_L & \leq & x & \leq & x_U, \\ s/t & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \end{array} \tag{9}$$

where $x, x_L, x_U \in \mathbb{R}^n$, $r(x) \in \mathbb{R}^M$, $A \in \mathbb{R}^{m_1 \times n}$, $b_L, b_U \in \mathbb{R}^{m_1}$ and $c_L, c(x), c_U \in \mathbb{R}^{m_2}$.

Recommended solvers: **TOMNET /NLSSOL**.

A fast and robust solver is *NLSSOL*, available in the TOMNET /NPSOL or the TOMNET /SOL extension toolbox.

3.2 Solver functionality

3.2.1 TOMNET Base Module

TOMNET includes a set of optimization solvers and routines that convert problems to a smooth format. Most of them were originally developed by the Applied Optimization and Modeling group (TOM) [14]. Since then they have been improved e.g. to handle TOMNET sparse arrays and been further developed. Table 4 lists the main set of TOM optimization solvers in all versions of TOMNET.

Table 4: The optimization solvers in TOMNET Base Module.

Function	Description	Section	Page
glbDirect	Box-bounded global optimization, using only function values. Fortran/C implementation.	6.1.1	20
glcDirect	Constrained global optimization, using objective function linear, integer and nonlinear constraints. Fortran/C implementation.	6.1.2	23

3.2.2 TOMNET /MINOS

Table 5 lists the solvers included in TOMNET /MINOS. All functionality of the SOL solvers are available and changeable in the TOMNET framework in .NET.

3.2.3 TOMNET /NPSOL

The add-on toolbox TOMNET /NPSOL is a sub package of TOMNET /SOL. The package includes the MINOS solvers as well as NPSOL, LSSOL, LSQR and NLSSOL. The TOMNET /NPSOL manual is available at `http://tomopt.com`.

Table 5: The SOL optimization solvers in TOMNET /MINOS.

Function	Description	Reference
MINOS 5.51	Sparse linear and nonlinear programming with linear and nonlinear constraints.	[18]
QPOPT 1.0-10	Non-convex quadratic programming with dense constraint matrix and sparse or dense quadratic matrix.	[9]

3.2.4 TOMNET /SNOPT

The add-on toolbox TOMNET /SNOPT is a sub package of TOMNET /SOL. The package includes the MINOS solvers as well as SNOPT and SQOPT. The TOMNET /SNOPT manual is available at `http://tomopt.com`.

3.2.5 TOMNET /SOL

The extension toolbox TOMNET /SOL gives access to the complete set of Fortran solvers developed by the Stanford Systems Optimization Laboratory (SOL). These solvers are listed in Table 5 and 6.

Table 6: The optimization solvers in the TOMNET /SOL toolbox.

Function	Description	Reference
NPSOL 5.02	Dense linear and nonlinear programming with linear and nonlinear constraints.	[13]
SNOPT 7.1-1	Large, sparse linear and nonlinear programming with linear and nonlinear constraints.	[12, 10]
SQOPT 7.1-1	Sparse convex quadratic programming.	[11]
NLSSOL 5.0-2	Constrained nonlinear least squares. NLSSOL is based on NPSOL. No reference except for general NPSOL reference.	[13]
LSSOL 1.05-4	Dense linear and quadratic programs (convex), and constrained linear least squares problems.	[8]

Note that solvers for a more general problem type may be used to solve the problem. In Table 7 an attempt has been made to classify these relations.

Table 7: The problem classes (*probType*) possible to solve with each type of solver (*solvType*) is marked with an x. When the solver is in theory possible to use, but from a practical point of view is probably not suitable, parenthesis are added (x). See table 1 for an explanation on problem types.

	solvType									
probType	**uc**	**qp**	**con**	**ls**	**lls**	**cls**	**mip**	**lp**	**glb**	**glc**
uc	x		x						x	(x)
qp		x	x							(x)
con			x							(x)
ls			x	x		x				(x)
lls		x	x	x	x	x				(x)
cls			x			x				(x)
mip							x			(x)
lp		x	x				x	x		(x)
glb			(x)						x	x
glc			(x)							x
exp	x		x	(x)		x				(x)

4 Defining problems in TOMNET

TOMNET is based on the principle of creating a problem object that defines the problem and includes all relevant information needed for the solution. One unified format is defined, the TOMNET format. The TOMNET format provides an intuitive way of setting up a problem object and of solving it using any suitable TOMNET solver.

In this section follows a more detailed description of the TOMNET format and problem sets available.

4.1 The TOMNET Format

The TOMNET format is a standardized way to setup problems and solve it using any of the TOMNET solvers. The principle is to put all information in a .NET class that is passed to the solver, which extracts the relevant information.

1. Define the problem (a `TOMNETProblem`) by calling the appropriate constructor.
2. Call the solver.
3. Evaluate the results.

Step 1 is equivalent to calling a constructor of one of the `TOMNETProblem` classes listed in table 2.

Step 2, the solver call, using the Solver's Solve method.

Step 3 could be a call to check `Result.ExitFlag`, or `Result.f_k` as shown in all the quick guide examples.

See the different examples that illustrates how to apply the TOMNET format: *lpQG.cs*, *nlpQG.cs* and more.

4.2 The TOMNET Test Problems

Several test problems are included with the distribution. The sets are available from the *TOMNETTestProb.dll* located in *tomnet/assembly*.

It is possible to initialize individual problem instances by using built-in get-functions or constructors. The methods demands the problem number as input. If the number is out of range an exception will be thrown.

As before, the problem is solved by using the solver method *solve*.

Note that when solving a sequence of similar problems, it is recommended to create one problem object, make a loop, and inside the loop do the minor changes to get individual problems that are solved without creating a new instance each time.

For example one might want to solve problem 10 in *conProb* one hundred times for different starting values in the interval $[1, 100]$.

```
using System;
using TOMNET;

class myTestProgram
{
  static void Main(string[] args)
  {
    TOMNETProblem Prob = new conProbProblem(10);
    Result result;
    NPSOL solver = new NPSOL();

    for (int i = 1; i < 100; i++)
    {
      Prob.x_0[0] = i ;
      solver.Solve(Prob, out result);
      Console.WriteLine("iteration {0}: f_k = {1}", i, result.f_k);
    }
  }
}
```

All the predefined test problem sets are available in the *TOMNETTestProb.dll* assembly. See also the different demonstration files in the *quickguide* directory.

See Section 2.3.1 for more information on how to access the test problems.

5 Using the TOMNET Solvers

The solvers are normally accessed by creating a TOMNETSolver object using the constructor of each solver class. After initial development work, call the solver directly by using the `Solve` method.

Built-in checks are executed to make sure that illegal problem statements, such as crossover for bounds, are not sent to the solver.

5.1 Setting Solver Options

Each applicable solver has a member called *Options*. The Options control the usage of print files, print levels, various convergence criteria and more.

All *Options* members are described in the documentation in the *tomnet/docs* folder.

The following snippet of code demonstrates an example of how to set the PrintFile option for a solver (in this case SNOPT):

```
SNOPT solver = new SNOPT();
string printfilename = "SnoptPrintFile.txt";
solver.Options.PrintFile = printfilename;
```

6 Solver Reference

Detailed descriptions of the TOMNET solvers, driver routines and some utilities are given in the following sections. Also see the help for each solver. All solvers except for the TOMNET Base Module are described in separate manuals.

6.1 TOMNET /BASE

For a description of solvers, see the help in the the User's Guide for the particular solver.

6.1.1 glbDirect

Purpose
Solve box-bounded global optimization problems.

glbDirect solves problems of the form

$$\begin{array}{ll} \min\limits_{x} & f(x) \\ s/t & x_L \quad \leq \quad x \quad \leq \quad x_U \end{array}$$

where $f \in \mathbb{R}$ and $x, x_L, x_U \in \mathbb{R}^n$.

glbDirect was originally a Fortran implementation.

Calling Syntax

```
glbDirect solver = new glbDirect();
solver.Solve(Prob, out result);
```

Description of Inputs

Prob Problem description. The following members are used:

x_L	Lower bounds for x, must be given to restrict the search space.
x_U	Upper bounds for x, must be given to restrict the search space.
Name	Name of the problem. Used for security if doing warm start.
f	The objective function $f(x)$ is used.

Description of Outputs

Result Object with result from optimization. The following fields are of special interest:

x_k	Optimal point.
f_k	Function value at optimum.
Iter	Number of iterations.

Result Object with result from optimization. The following fields are of special interest:, continued

FuncEv	Number function evaluations.
ExitText	Text string giving ExitFlag and Inform information.
ExitFlag	Exit code. 0 = Normal termination, max number of iterations /func.evals reached. 1 = Some bound, lower or upper is missing. 2 = Some bound is inf, must be finite. 4 = Numerical trouble determining optimal rectangle, empty set and cannot continue.
Inform	Inform code. 1 = Function value f is less than fGoal. 2 = Absolute function value f is less than fTol, only if fGoal = 0 or Relative error in function value f is less than fTol, i.e. $abs(f - fGoal)/abs(fGoal) <= fTol$. 3 = Maximum number of iterations done. 4 = Maximum number of function evaluations done. 91= Numerical trouble, did not find element in list. 92= Numerical trouble, No rectangle to work on. 99= Other error, see ExitFlag.

Description of Options

Options available for glbDirect

LogFile	File for log information.
PrintLevel	Print Level. 0 = Silent. 1 = Errors. 2 = Termination message and warm start info. 3 = Option summary.
MaxFunc	Maximal number of function evaluations, default 10000 (roughly).
Maxiter	Maximal number of iterations, default 200.
Parallel	Set to 1 in order to have glbDirect to call Prob.FUNCS.f with a matrix x of points to let the user function compute function values in parallel. Default: 0. Not currently active.
WarmStart	If true, > 0, *glbDirect* reads the output from the last run if it exists. *glbDirect* uses this warm start information to continue from the last run.
IterPrint	Print iteration log every ITERPRINT iteration. Set to 0 for no iteration log. PRILEV must be set to at least 1 to have iteration log to be printed.
FunTol	Relative accuracy for function value. Stop if $abs(f - FGOAL) <= abs(FGOAL) * FUNTOL$, if $FGOAL = 0$. Stop if $abs(f - FGOAL) <= FUNTOL$, if $FGOAL == 0$.

Options available for glbDirect, continued

VarTol	Convergence tolerance in x. All possible rectangles are less than this tolerance (scaled to (0,1)). See the output field MAXTRI.
GLWeight	Global/local weight parameter, default 1E-4.
FGoal	Goal for function value, if empty not used.

Description

The global optimization routine *glbDirect* is an implementation of the DIRECT algorithm presented in [5].The algorithm in *glbDirect* is a Fortran implementation of the Matlab algorithm in *glbSolve*. DIRECT is a modification of the standard Lipschitzian approach that eliminates the need to specify a Lipschitz constant. Since no such constant is used, there is no natural way of defining convergence (except when the optimal function value is known). Therefore *glbDirect* runs a predefined number of iterations and considers the best function value found as the optimal one. It is possible for the user to **restart** *glbDirect* with the final status of all parameters from the previous run, a so called *warm start*. Assume that a run has been made with *glbDirect* on a certain problem for 50 iterations. Then a run of e.g. 40 iterations more should give the same result as if the run had been using 90 iterations in the first place. To do a warm start of *glbDirect* an option *WARMSTART* should be set to one and warm start information defined. Then *glbDirect* is using output previously obtained to make the restart. The code also includes the subfunction *conhull* (embedded) which is an implementation of the algorithm GRAHAMHULL in [22, page 108] with the modifications proposed on page 109. *conhull* is used to identify all points lying on the convex hull defined by a set of points in the plane.

6.1.2 glcDirect

Purpose

Solve global mixed-integer nonlinear programming problems.

glcDirect solves problems of the form

$$\begin{array}{lccccc} \min\limits_{x} & f(x) & & & & \\ s/t & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \\ & & & x_i \text{ integer} & & i \in I \end{array}$$

where $x, x_L, x_U \in \mathbb{R}^n$, $c(x), c_L, c_U \in \mathbb{R}^{m_1}$, $A \in \mathbb{R}^{m_2 \times n}$ and $b_L, b_U \in \mathbb{R}^{m_2}$.

The variables $x \in I$, the index subset of $1, ..., n$ are restricted to be integers. Recommendation: Put the integers as the first variables. Put low range integers before large range integers. Linear constraints are specially treated. Equality constraints are added as penalties to the objective. Weights are computed automatically, assuming f(x) scaled to be roughly 1 at optimum. Otherwise scale f(x).

glcDirect is originally a Fortran implementation that has been embedded in .NET.

Calling Syntax

```
glcDirect solver = new glcDirect();
solver.Solve(Prob, out result);
```

Description of Inputs

Prob Problem description. The following members are used:

Name	Problem name. Used for safety when doing warm starts.
f	The objective function $f(x)$ is used.
c	The nonlinear constraints $c(x)$ is used.
A	Linear constraints matrix.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
c_L	Lower bounds on the nonlinear constraints.
c_U	Upper bounds on the nonlinear constraints.
x_L	Lower bounds for x, must be finite to restrict the search space.
x_U	Upper bounds for x, must be finite to restrict the search space.
IntVars	IntVars is assumed to be a set of indices. It is advised to number the integer values as the first variables, before the continuous. The tree search will then be executed more efficiently.

Description of Outputs

Result Object with result from optimization. The following fields are of special interest:

x_k	Optimal point.
f_k	Function value at optimum.
c_k	Nonlinear constraints values at *x_k*.
Iter	Number of iterations.
FuncEv	Number function evaluations.
ExitText	Text string giving ExitFlag and Inform information.
ExitFlag	0 = Normal termination, max number of iterations func.evals reached. 2 = Some upper bounds below lower bounds. 4 = Numerical trouble, and cannot continue. 7 = Reached maxFunc or maxIter, NOT feasible. 8 = Empty domain for integer variables. 10= Input errors.
Inform	1 = Function value f is less than fGoal. 2 = Absolute function value f is less than fTol, only if fGoal = 0 or Relative error in function value f is less than fTol, i.e. abs(f-fGoal)/abs(fGoal) <= fTol. 3 = Maximum number of iterations done. 4 = Maximum number of function evaluations done. 5 = Maximum number of function evaluations would most likely be too many in the next iteration, save warm start info, stop. 6 = Maximum number of function evaluations would most likely be too many in the next iteration, because $2 * sLen >=$ maxFDim - nFunc, save warm start info, stop. 7 = Space is dense. 8 = Either space is dense, or MIP is dense. 10= No progress in this run, return solution from previous one. 91= Infeasible. 92= No rectangles to work on. 93= sLen = 0, no feasible integer solution exists. 94= All variables are fixed. 95= There exist free constraints.

Description of Options

Options available for glcDirect

LogFile	File for log information.
PrintLevel	Print Level. This controls both regular printing from glcDirect and the amount of iteration log information to print.

Options available for glcDirect, continued

	0 = Silent. 1 = Warnings and errors printed. Iteration log on iterations improving function value. 2 = Iteration log on all iterations. 3 = Log for each function evaluation. 4 = Print list of parameter settings. See ITERPRINT for more information on iteration log printing.
WarmStart	If true, > 0, glcDirect reads the output from the last run if it exists. glcDirect uses this warm start information to continue from the last run.
MaxCpu	Maximum CPU Time (in seconds) to be used. Default 36000.
FCall	=0 (Default). If linear constraints cannot be feasible anywhere inside rectangle, skip f(x) and c(x) computation for middle point. =1 Always compute f(x) and c(x), even if linear constraints are not feasible anywhere in rectangle. Do not update rates of change for the constraints. =2 Always compute f(x) and c(x), even if linear constraints are not feasible anywhere in rectangle. Update rates of change constraints.
UseRoC	=1 (Default). Use original Rate of Change (RoC) for constraints to weight the constraint violations in selecting which rectangle divide. =0 Avoid RoC, giving equal weights to all constraint violations. Suggested if difficulty to find feasible points. For problems where linear constraints have been added among the nonlinear (NOT RECOMMENDED; AVOID!!!), then option useRoc=0 has been successful, whereas useRoC completely fails. =2 Avoid RoC for linear constraints, giving weight one to these constraint violations, whereas the nonlinear constraints use RoC. =3 Use RoC for nonlinear constraints, but linear constraints are not used to determine which rectangle to use.
Branch	=0 Divide rectangle by selecting the longest side, if ties use the lowest index. This is the Jones DIRECT paper strategy. =1 First branch the integer variables, selecting the variable with the least splits. If all integer variables are split, split on the continuous variables as in BRANCH=0. DEFAULT! Normally much more efficient than =0 for mixed-integer problems. =2 First branch the integer variables with 1,2 or 3 possible values, e.g [0,1],[0,2] variables, selecting the variable with least splits. Then branch the other integer variables, selecting the variable with the least splits. If all integer variables are split, split on the continuous variables as in BRANCH=0. =3 Like =2, but use priorities on the variables.
RecTie	When minimizing the measure to find which new rectangle to try to get feasible, there are often ties, several rectangles have the same minimum. RECTIE = 0 or 1 seems reasonable choices. Rectangles with low index are often larger then the rectangles with higher index. Selecting one of each type could help, but often =0 is fastest.

Options available for glcDirect, continued

=0 Use the rectangle with value a, with lowest index (original).
=1 (Default): Use 1 of the smallest and 1 of largest rectangles.
=2 Use the last rectangle with the same value a, not the 1st.
=3 Use one of the smallest rectangles with same value a.
=4 Use all rectangles with the same value a, not just the 1st.

EqConFac Weight factor for equality constraints when adding to objective function f(x) (Default value 10). The weight is computed as EQCONFAC/"right or left hand side constant value", e.g. if the constraint is Ax <= b, the weight is EQCONFAC/b If DIRECT just is pushing down the f(x) value instead of fulfilling the equality constraints, increase EQCONFAC.

AxFeas Set nonzero to make glcDirect skip f(x) evaluations, when the linear constraints are infeasible, and still no feasible point has been found. The default is 0. Value 1 demands $FCALL == 0$. This option could save some time if f(x) is a bit costly, however overall performance could on some problems be dramatically worse.

FEqual All points with function values within tolerance FEQUAL are considered to be global minima and returned. Default 1E-10.

LinWeight $RateOfChange = LINWEIGHT * ||a(i,:)||$ for linear constraints. Balance between linear and nonlinear constraints. Default 0.1. The higher value, the less influence from linear constraints.

Alpha Exponential forgetting factor in RoC computation, default 0.9.

AvIter How many values to use in startup of RoC computation before switching to exponential smoothing with forgetting factor alpha. Default 50.

FGoal Goal for function value, if empty not used.

MaxFunc Maximal number of function evaluations, default 10000 (roughly).

MaxIter Maximal number of iterations, default 10000.

IterPrint Print iteration log every ITERPRINT iteration. Set to 0 for no iteration log. PRILEV must be set to at least 1 to have iteration log to be printed.

FunTol Relative accuracy for function value. Stop if $abs(f - FGOAL) <= abs(FGOAL) * FUNTOL$, if $FGOAL = 0$. Stop if $abs(f - FGOAL) <= FUNTOL$, if $FGOAL == 0$. Default 1E-2.

VarTol Convergence tolerance in x. All possible rectangles are less than this tolerance (scaled to (0,1)). See the output field MAXTRI. Default 1E-11.

Options available for glcDirect, continued

GLWeight	Global/local weight parameter. Default 1E-4.
NLContol	Nonlinear constraint tolerance. Default 1E-5.
LContol	Linear constraint tolerance. Default 1E-7.
FIP	An upper bound on the optimal f(x) value. If empty, set as Inf.

Description

The routine *glcDirect* implements an extended version of DIRECT, see [16], that handles problems with both nonlinear and integer constraints. The algorithm in *glcDirect* is a Fortran implementation of the Matlab algorithm in *glcSolve*.

DIRECT is a modification of the standard Lipschitzian approach that eliminates the need to specify a Lipschitz constant. Since no such constant is used, there is no natural way of defining convergence (except when the optimal function value is known). Therefore *glcDirect* is run for a predefined number of function evaluations and considers the best function value found as the optimal one. It is possible for the user to **restart** *glcDirect* with the final status of all parameters from the previous run, a so called *warm start*. Assume that a run has been made with *glcDirect* on a certain problem for 500 function evaluations. Then a run of e.g. 200 function evaluations more should give the same result as if the run had been using 700 function evaluations in the first place. To do a warm start of *glcDirect* an option *WARMSTART* should be set to one.

DIRECT does not explicitly handle equality constraints. It works best when the integer variables describe an ordered quantity and is less effective when they are categorical.

Warnings

A significant portion of *glcDirect* is coded in Fortran format. If the solver is aborted, it may have allocated memory for the computations which is not returned. This may lead to unpredictable behavior if *glcDirect* is started again.

6.2 TOMNET /MINOS

Includes MINOS and QPOPT. This package is included in the following three as well.

6.3 TOMNET /NPSOL

The following solvers are included in this package: LSSOL, LSQR, NLSSOL and NPSOL. A separate manual is available from the TOMNET home page.

6.4 TOMNET /SNOPT

The following solvers are included in this package: SNOPT and SQOPT. A separate manual is available from the TOMNET home page.

6.5 TOMNET /SOL

All three packaged listed above (MINOS, NPSOL and SNOPT).

7 Problem Reference

The following sections contain detailed descriptions of all types of `TOMNETProblem` present in TOMNET.

The final parts describe four interfaces that are central when modeling any function or constraint that is not linear or quadratic:

1. `IFunction`: interface to Function (see 7.8).
2. `IDFunction`: interface to Differentiable Function (see 7.9).
3. `IConstraints`: interface to Constraints (see 7.10).
4. `IDConstraints`: interface to Differentiable Constraints (see 7.11).

These Interfaces function as a sort of templates that help the solver evaluate the objective function and constraints.

7.1 *class LinearProblem*

Description
One of the basic classes of TOMNET is `LinearProblem`. It is used for modeling `TOMNETProblem's` with a linear objective function and only linear constraints. See **linear programming** in 3.1.6.

Calling Syntax

```
LinearProblem Prob = new LinearProblem(c, A, b_L, b_U, x_L, x_U,
  x_0, Name, double.NegativeInfinity, null, null);
```

Constructor Inputs

double[] c	The vector c in the objective function.
object A	The linear constraints (a double[] or a Sparse).
double[] b_L	The lower bounds for the linear constraints.
double[] b_U	The upper bounds for the linear constraints.
double[] x_L	Lower bounds on x.
double[] x_U	Upper bounds on x.
double[] x_0	Starting point x.
String Name	The name of the problem.
double fLowBnd	A lower bound on the function value at optimum if known. If not known set to NegativeInfinity.
double[] f_opt	Optimal function value(s), if known (Stationary points). If not known set Null.
double[] x_opt	The x-values corresponding to the given f_opt, if known. If not known set Null.

7.2 *class QuadraticProblem*

Description
The class `QuadraticProblem` is used for modeling a `TOMNETProblem` with a quadratic objective function and only linear constraints. See **quadratic programming** in 3.1.2.

Calling Syntax

```
QuadraticProblem Prob = QuadraticProblem.qpAssign(F, c, A,
  b_L, b_U, x_L, x_U, x_0, Name);
```

Constructor Inputs

object F	The matrix F in the objective function (a double[] or a Sparse).
double[] c	The vector c in the objective function.
object A	The linear constraints (a double[] or a Sparse).
double[] b_L	The lower bounds for the linear constraints.
double[] b_U	The upper bounds for the linear constraints.
double[] x_L	Lower bounds on x.
double[] x_U	Upper bounds on x.
double[] x_0	Starting point x.
String Name	The name of the problem.

7.3 *class NonlinearProblem*

Description

A `NonlinearProblem` is used to model a problem with a general objective function and linear constraints. The user needs to implement a class that inherits from either the interface `IDFunction` (see 7.9) or `IFunction` (see 7.8) to model the objective function.

It is always better to use `IDFunction` and describe the derivatives analytically. If the derivatives are impossible or very hard to model numerical differentiation can be used by implementing an `IFunction`, this may however result in loss of robustness.

For an example of how to use the `IFunction` the reader is encouraged to see *nlpQG.cs* in *tomnet/quickguide* and compare `NonlinearProblem` with `ConProblem` in 7.4.

To create an unconstrained optimization problem (see **unconstrained optimization** 3.1.1) call the constructor with `Null`-parameters instead of `A`, `b_L` and `b_U`.

Calling Syntax

```
//
// User-defined objective function
//
public class myNLPFunc : IDFunction    { /* ... */ }

//
// NonlinearProblem needs an instance of myNLPFunc
//
myNLPFunc F = new myNLPFunc();
NonlinearProblem Prob = new NonlinearProblem(F, A, b_L, b_U,
  x_L, x_U, x_0, Name, double.NegativeInfinity, null, null);
```

Constructor Inputs

IFunction F	User-defined objective function (see 7.8 and 7.9).
object A	The linear constraints (a double[] or a Sparse).
double[] b_L	The lower bounds for the linear constraints.
double[] b_U	The upper bounds for the linear constraints.
double[] x_L	Lower bounds on x.
double[] x_U	Upper bounds on x.
double[] x_0	Starting point x.
String Name	The name of the problem.
double fLowBnd	A lower bound on the function value at optimum if known. If not known set to NegativeInfinity.

Constructor Inputs, continued

double[] f_opt	Optimal function value(s), if known (Stationary points). If not known set Null.
double[] x_opt	The x-values corresponding to the given f_opt, if known. If not known set Null.

7.4 *class ConProblem*

Description

One of the more general classes for modeling a problem in TOMNET is `ConProblem`. It requires coding of an objective function and nonlinear constraints by the user.

For an extensive example the reader is encouraged to see *nlpQG.cs* in *tomnet/quickguide*. Also see **constrained nonlinear programming** in 3.1.3.

The user needs to implement a class that inherits from either the interface `IFunction` (see 7.8) or `IDFunction` (see 7.9) to model the objective function. Also the nonlinear constraints must be modeled in a class that inherits from `IConstraints` (see 7.10) or `IDConstraints` (see 7.11).

Calling Syntax

```
//
// User-defined objective function class myConFunc
// and nonlinear constraints class myConCons.
//
public class myConFunc : IDFunction    { /* ... */ }
public class myConCons : IDConstraints { /* ... */ }

//
// ConProblem needs an instance of IDFunction and IDConstraints
//
myConFunc myF = new myConFunc();
myConCons myC = new myConCons();
TOMNETProblem Prob = new ConProblem(myF, myC, c_L, c_U, x_L, x_U,
  x_0, Name, double.NegativeInfinity, null, null);
```

Constructor Inputs

IFunction F	User-defined objective function (see 7.8 and 7.9).
IConstraints C	User-defined nonlinear constraints (see 7.10 and 7.11).
double[] c_L	The lower bounds for the nonlinear constraints.
double[] c_U	The upper bounds for the nonlinear constraints.
double[] x_L	Lower bounds on x.
double[] x_U	Upper bounds on x.
double[] x_0	Starting point x.
String Name	The name of the problem.
double fLowBnd	A lower bound on the function value at optimum if known. If not known set to NegativeInfinity.

Constructor Inputs, continued

double[] f_opt	Optimal function value(s), if known (Stationary points). If not known set Null.
double[] x_opt	The x-values corresponding to the given f_opt, if known. If not known set Null.

7.5 *class ConLinProblem*

Description

One of the more general classes for modeling a problem in TOMNET is `ConLinProblem`. It requires coding of the objective function and nonlinear constraints by the user. See 3.1.1 for a formal definition.

For a detailed example the reader is encouraged to see A.2.

The user needs to implement a class that inherits from either the interface `IFunction` (see 7.8) or `IDFunction` (see 7.9) to model the objective function. Also the nonlinear constraints must be modeled in a class that inherits from `IConstraints` (see 7.10) or `IDConstraints` (see 7.11).

Calling Syntax

```
//
// User-defined objective function class myConFunc
// and nonlinear constraints class myConCons.
//
public class myConFunc : IDFunction    { /* ... */ }
public class myConCons : IDConstraints { /* ... */ }

//
// ConLinProblem needs an myConFunc and a myConCons
//
myConFunc myF = new myConFunc();
myConCons myC = new myConCons();

ConLinProblem Prob = new ConLinProblem(myF, A, b_L, b_U,
  myC, c_L, c_U, x_L, x_U, x_0, Name, double.NegativeInfinity,
  null, null);
```

Constructor Inputs

IFunction F	User-defined objective function (see 7.8 and 7.9).
IConstraints A	Linear Constraints (a Sparse or a double[]).
double[] b_L	The lower bounds for the linear constraints.
double[] b_U	The upper bounds for the linear constraints.
IConstraints C	User-defined nonlinear constraints (see 7.10 and 7.11).
double[] c_L	The lower bounds for the non linear constraints.
double[] c_U	The upper bounds for the non linear constraints.
double[] x_L	Lower bounds on x.
double[] x_U	Upper bounds on x.
double[] x_0	Starting point x.

Constructor Inputs, continued

String Name	The name of the problem.
double fLowBnd	A lower bound on the function value at optimum if known. If not known set to NegativeInfinity.
double[] f_opt	Optimal function value(s), if known (Stationary points). If not known set Null.
double[] x_opt	The x-values corresponding to the given f_opt, if known. If not known set Null.

7.6 *class LinearLeastSquaresProblem*

Description

The `LinearLeastSquaresProblem` efficiently models all **linear least squares problem** (see 3.1.8 for definitions).

Calling Syntax

```
LinearLeastSquaresProblem Prob = new LinearLeastSquaresProblem(
  n, m, k, C, d, A, b_L, b_U, x_0, x_L, x_U);
```

Constructor Inputs

int n	Number of variables.
int m	Number of linear constraints.
int k	Number of residuals.
double[] C	C matrix of the objective function.
double[] y	The observations.
double[] A	The linear constraints.
double[] b_L	The lower bounds for the linear constraints.
double[] b_U	The upper bounds for the linear constraints.
double[] x_L	Lower bounds on x.
double[] x_U	Upper bounds on x.
double[] x_0	Starting point x.

7.7 *class NonlinearLeastSquaresProblem*

Description

In TOMNET `NonlinearLeastSquaresProblem` can be used to model a nonlinear least squares problem with nonlinear constraints. See **constrained nonlinear least squares problem** in 3.1.9 for definitions. Also see *nllsQG.cs* in *tomnet/quickguide* for a comprehensive example.

Calling Syntax

```
NonlinearLeastSquaresProblem Prob = new
  NonlinearLeastSquaresProblem(R, C, A, b_L, b_U, x_L, x_U, x_0,
  c_L, c_U, y, t, "NllsProb", double.NegativeInfinity, null,
  null);
```

Constructor Inputs

IFunction F	User-defined objective function (see 7.8 and 7.9).
IConstraints C	User-defined nonlinear constraints (see 7.10 and 7.11).
object A	The linear constraints.
double[] b_L	The lower bounds for the linear constraints.
double[] b_U	The upper bounds for the linear constraints.
double[] x_L	Lower bounds on x.
double[] x_U	Upper bounds on x.
double[] x_0	Starting point x.
double[] c_L	The lower bounds for the nonlinear constraints.
double[] c_U	The upper bounds for the nonlinear constraints.
double[] y	The observations.
double[] t	Array of time points of observations.
String Name	The name of the problem.
double fLowBnd	A lower bound on the function value at optimum if known. If not known set to NegativeInfinity.
double[] f_opt	Optimal function value(s), if known (Stationary points). If not known set Null.
double[] x_opt	The x-values corresponding to the given f_opt, if known. If not known set Null.

7.8 *Interface IFunction*

Description
An `IFunction` is used to model a problem with any scalar objective function. It is preferable to use `IDFunction` (see 7.9) and define the gradient of the objective function to gain robustness and speed.

Calling Syntax
To use an `IFunction` the method `Evaluate` must be implemented.

```
public class myObjective : IFunction
{
  public void Evaluate(double[] fx, double[] x)
  {
    // ...
  }
}
```

Methods to Implement	
void Evaluate(double[] fx, double[] x)	User-defined objective function.

Examples
See A.2 for an example of how to use `IFunction`, `IDFunction`, `IConstraints` and/or `IDConstraints`.

7.9 *Interface IDFunction*

Description
An `IDFunction` is used to model a any scalar objective function where the gradients are known.

`IDFunction` inherits from `IFunction` so any `IDFunction` is also an `IFunction`.

Calling Syntax
To use an `IDFunction` two methods needs to be implemented: `Evaluate` (for the function value) and `Grad` for the values of the gradient.

```
public class myObjective : IDFunction
{
  public void Evaluate(double[] fx, double[] x)
  {
    // ...
  }

  public void Grad(double[] gx, double[] x)
  {
    // ...
  }
}
```

Methods to Implement	
void Evaluate(double[] fx, double[] x)	User-defined objective function.
void Grad(double[] gx, double[] x)	User-defined gradient.

Examples
See A.2 for an example of how to use `IFunction`, `IDFunction`, `IConstraints` and/or `IDConstraints`.

7.10 *Interface IConstraints*

Description
An `IConstraints` is used to model a problem with any nonlinear constraints. It is preferable to use `IDConstraints` (see 7.11) and define the derivatives of the constraints to gain robustness and speed.

Calling Syntax
To use an `IConstraints` two `Evaluate` methods must be implemented.

```
public class myConstraints : IConstraints
{
  public void Evaluate(double[] cx, double[] x)
  {
    // ...
  }

  public double[] Evaluate(double[] x)
  {
    // ...
  }
}
```

Methods to Implement	
void Evaluate(double[] cx, double[] x)	Evalate the constraints.
double[] Evaluate(double[] x)	Evalate the constraints.

Examples
See A.2 for an example of how to use `IFunction`, `IDFunction`, `IConstraints` and/or `IDConstraints`.

7.11 *Interface IDConstraints*

Description
An `IDConstraints` is used to model a any constraints function where the gradients can be expressed analytically.

`IDConstraints` inherits from `IConstraints` so any `IDConstraints` is also an `IConstraints`.

Calling Syntax
To use an `IDConstraints` the methods `Evaluate` and `dc` must be implemented.

```
public class myConstraints : IDConstraints
{
  public  void Evaluate(double[] cx, double[] x)
  {
    // ...
  }

  public double[] Evaluate(double[] x)
  {
    // ...
  }

  public void dc(Jacobian dcx, double[] x)
  {
    // ...
  }
}
```

Methods to Implement

void Evaluate(double[] cx, double[] x)	Evaluate the constraints.
double[] Evaluate(double[] x)	Evaluate the constraints.
void dc(Jacobian dcx, double[] x)	Evaluate the derivatives of the constraints.

Examples
See A.2 for an example of how to use `IFunction`, `IDFunction`, `IConstraints` and/or `IDConstraints`.

8 Approximation of Derivatives

8.1 Numerical differenatiaion in using TOMNET.

This section about derivatives is particulary important from a practical point of view. It often seems to be the case that either it is nearly impossible or the user has difficulties in coding the derivatives.

Observe that the usage depends on which solver is used. Clearly, if a global solver, such as *glbDirect* is used, derivatives are not used at all and the following sections do not apply. Some solvers use only first order information. *SNOPT* for example requires this. If a solver requires second order information it is highly recommended that at least the first order information is given analytically or the accuracy requested is changed.

Prob.NumDiff and *Prob.ConsDiff* could be set to 1 to use numerical derivatives even when analytical are given (for comparison purposes).

Calling Syntax

```
// to restore later
int OriginalNumDiff = Prob.NumDiff;

// Force numerical differentiation
Prob.NumDiff = 1;
solver.Solve(Prob, out result);

// restore to original value
Prob.NumDiff = OriginalNumDiff;
```

Table 25 describes the differentiation options available in TOMNET.

Table 25: The differentiation options in TOMNET.

Problem Member	Value	Comments
Prob.NumDiff	1	TOMNET calculates the gradient. **Default if an `IFunction` is used**.
Prob.ConsDiff	1	TOMNET calculates the Jacobian of the constraints. **Default if an `IConstraints` is used**.

8.2 Numerical differentiation in using the SOL solvers.

The SOL solvers *MINOS*, *NPSOL*, *NLSSOL*, *SNOPT* and other solvers include numerical differentiation.

To enable the numerical differentiation an any SOL solver the option `DerivativeLevel` is used.

The following values are allowed:

- 3 All objective and constraint gradients are known (`default`).
- 2 All constraint gradients are known, but some or all components of the objective gradient are unknown.

- 1 The objective gradient is known, but some or all of the constraint gradients are unknown.
- 0 Some components of the objective gradient are unknown and some of the constraint gradients are unknown.

Calling Syntax
This example tells *NPSOL* to perform numerical differentiation of the objective function.

```
NPSOL solver = new NPSOL();
solver.Options.DerivativeLevel = 2;
```

9 Special Notes and Features

The TOMNET package for optimization in .NET is the most versatile implementation in industry. All solver suites feature state-of-the-art solvers underneath the surface and the calling syntax is completely standardized, i.e. the user only needs to define the problem once and can then run any applicable solver (for QP problems more than 8 options are available).

10 Sparse Matrix Handling

These sections outline the functionality available for sparse matrix handling included with TOMNET.

10.1 Create

The following functions are available to create sparse matrices.

10.1.1 Sparse Constructors

There are three Constructors for `Sparse`.

- `Sparse(Sparse S)`
- `Sparse(int m, int n, int nnz, int[] ir, int[] jc, double[] pr)`
- `Sparse(int m, int n, int nnz, int[] RowIdx, int[] ColIdx, double[] Values, int dummy)`

The first one creates a copy of an already created sparse - to avoid shallow copying. The other two constructors are for creating instances from three arrays and three integers. Since they both require the same types of inputs a dummy parameter is needed to separate the two.

Sparse(int m, int n, int nnz, int[] ir, int[] jc, double[] pr)

Description
Initiate a sparse matrix from three arrays and three integers. One array with matrix element values, and two arrays defining the pattern of the `Sparse`. The integers describe the number of rows, number of columns and number of nonzero elements in the `Sparse`.

Description of Inputs

int m	Number of rows in S.
int n	Number of columns in S.
int nnz	Number of elements that are not zero in S.
int[] ir	`ir` stores at position `i` the row-index of the value `pr[i]`.
int[] jc	The number `jc[i+1]` - `jc[i+1]` equals the number of elements in column `i` stores at position `i` the index of `pr`.
double[] pr	Values of the `Sparse`. The element `pr[i]` correponds to the `i`'th non-zero element (counting row-wise).

Calling Syntax

```
const int n = 5;
const int m = 3;
const int nnz = 7;

int[] ir = new int[nnz] { 0, 1, 0, 1, 2, 0, 1 };
int[] jc = new int[n + 1] { 0, 2, 2, 3, 5, 7 };
double[] pr = new double[nnz] { 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };
Sparse S = new Sparse(m, n, nnz, ir, jc, pr);
```

Sparse(int m, int n, int nnz, int[] RowIdx, int[] ColIdx, double[] Values, int dummy)

Description

Initiate a sparse matrix from three arrays and three integers. One array with matrix element values, and two arrays with column and row indices of the values. The integers describe the number of rows, number of columns and number of nonzero elements in the `Sparse`.

Description of Inputs

int m	An integer equal to the number of rows in A.
int n	An integer equal to the number of columns in A.
int nnz	An integer equal to the number of columns in A.
int[] RowIdx	Array row index. The array contains row index corresponding to the array of values.
int[] ColIdx	Array column index. The array contains column index corresponding to the array of values.
double[] Values	Array value. The array contains matrix element values.

Calling Syntax

```
const int n = 5;
const int m = 3;
const int nnz = 7;

int[] RowIdx = new int[nnz] { 0, 1, 0, 1, 2, 0, 1 };
int[] ColIdx = new int[nnz] { 0, 0, 2, 3, 3, 4, 4 };
double[] Values = new double[nnz] { 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };
Sparse S = new Sparse(m, n, nnz, RowIdx, ColIdx, Values, 0);
```

10.1.2 Sparse.Eye

There are four `Sparse.Eye` methods for initiating a `Sparse` with a diagonal, for example the Identity Matrix.

```
Eye(int m)

Eye(int m, double value)

Eye(int m, int n)

Eye(int m, int n, double value)
```

Description
Initiate a sparse identity matrix from one or two integers and optionally a value. The integers describe the number of rows and columns in the identity matrix and the optional value the diagonal entries (default 1).

Description of Inputs

int m	An integer equal to the number of rows in matrix I.
int n	An (optional) integer equal to the number of columns in matrix I.
double value	An (optional) value to use instead of 1.0 in the diagonal.

Description of Outputs

I	A sparse identity matrix with size described by the inputs.

Calling Syntax

```
Sparse I = Sparse.Eye(5);
// Sparse I = Sparse.Eye(5, 3.0);
// Sparse I = Sparse.Eye(5, 4);
// Sparse I = Sparse.Eye(5, 4, 3.0);
```

10.2 Modify

The following functions are available to merge and modify existing sparse matrices.

10.2.1 Sparse.Transpose

Purpose
The function returns the transpose of the input matrix.

$$B \quad = \quad A^T$$

where $B \in \mathbb{R}^{n \times m}$, $A \in \mathbb{R}^{m \times n}$.

Description of Inputs

Sparse A An instance of a `Sparse` matrix.

Description of Outputs

Sparse B `Sparse` B will be created as the transpose of the input matrix.

Description
The routine `Sparse.Transpose` uses optimal data handling and structures to return the transpose of a matrix.

Calling Syntax

```
Sparse B = A.Transpose();
```

10.2.2 Sparse.Concatenate

Description
Concatenates two sparse matrices either rowwise or columnwise. The `Sparse.Concatenate` uses optimal data handling to concatenate two sparse matrices eighter rowwise or columnwise.

$$C = \begin{matrix} \begin{bmatrix} A & B \end{bmatrix} \\ (\text{ dim=0 }) \end{matrix}$$

$$C = \begin{matrix} \begin{bmatrix} A \\ B \end{bmatrix} \\ (\text{ dim=1 }) \end{matrix}$$

where $C \in \mathbb{R}^{mA \times (nA+nB)}$ or $C \in \mathbb{R}^{(mA+mB) \times nA}$, $A \in \mathbb{R}^{mA \times nA}$, $B \in \mathbb{R}^{mB \times nB}$.

Description of Inputs

Sparse A An instance of a `Sparse` matrix.

Sparse B An instance of a `Sparse` matrix.

Int dim Eighter 0 for rowwise concatenation or 1 for columnwise concatenation.

Description of Outputs

Sparse C A new `Sparse` C.

Calling Syntax

```
// A and B are instances of Sparse of appropriate dimensions
Sparse C;
C = Sparse.Concatenate(A, B, 0);
C = Sparse.Concatenate(A, B, 1);
```

10.2.3 Alter value

Description
Set a matrix element to a given value. where $A \in \mathbb{R}^{m \times n}$, i integer, j integer, val is a scalar.

Description of Inputs

Sparse A	An instance of a `Sparse` matrix.
int i	Integer equal to the row index.
int j	Integer equal to the column index.
double val	The new element in A.

Description of Outputs

A new	The modified sparse matrix.

Calling Syntax

```
// ok
A[1,2] = 3.14;

// bad
// A[1,200] = 1.337;
// Raises an exception with a message similar to
// "Index (1,200) not in Matrix (Sparse) of size (3 x 5)."
```

10.3 Properties

The following properties are available to obtain information about the sparse matrices.

10.3.1 Sparse.Cols

Description
Get the number of columns of the `Sparse`.

$$n \quad = \quad \text{number of columns in } A$$

where $A \in \mathbb{R}^{m \times n}$.

Description of Outputs

int n An integer equal to the number of columns in A.

Calling Syntax

```
int n =  A.Cols;
```

10.3.2 Sparse.Rows

Description
Get the number of rows of the `Sparse`.

$$m \quad = \quad \text{number of rows in } A$$

where $A \in \mathbb{R}^{m \times n}$.

Description of Outputs

m An integer equal to the number of rows in A.

Calling Syntax

```
int m =  A.Rows;
```

10.3.3 Sparse.Nnz

Description
Get the number of non zeros in a `Sparse`.

$$nnz = \quad \text{number of nonzeros in } A$$

where $A \in \mathbb{R}^{m \times n}$.

Description of Outputs

int nnz An integer equal to the number of non zeros in A.

Calling Syntax

```
int nnz =  A.Nnz;
```

10.3.4 get Value

Description
Get a matrix element value from a sparse 2-D matrix.

$$val \quad = \quad A[i,j]$$

where *val* is a scalar, $A \in \mathbb{R}^{m \times n}$, i integer, j integer.

Description of Inputs

Sparse A	Sparse matrix A.
int i	Integer equal to the row index.
int j	Integer equal to the column index.

Description of Outputs

double val	The element in A in the i:th row and j:th column.

Calling Syntax

```
double val =  A[1,2];
```

10.4 Operators

The following functions are available to perform calculations with sparse matrices.

10.4.1 Operator *

Description
multiplies two instances of `Sparse` matrices.

$$C = AB$$

where $C \in \mathbb{R}^{mA \times nB}$, $A \in \mathbb{R}^{mA \times nA}$ and $B \in \mathbb{R}^{mB \times nB}$.

Description of Inputs

Sparse A Sparse matrix A.

Sparse B Sparse matrix B.

Description of Outputs

C Sparse matrix C.

Calling Syntax

```
// A and B are instances of Sparse
// of appropriate size.
Sparse C = A * B;
```

A TOMNETProblem - the problem description

The class `TOMNETProblem`, is one of the most central aspects of working with TOMNET. It contains numerous members and methods.

A.1 Important Members and Methods

This section contains details about the base class `TOMNETProblem`. This class is inherited to the more specific classes:

- `LinearProblem` (see 7.1).
- `QuadraticProblem` (see 7.2).
- `NonlinearProblem` (see 7.3).
- `ConProblem` (see 7.4).
- `ConLinProblem` (see 7.5).
- `LinearLeastSquaresProblem` (see 7.6).
- `NonlinearLeastSquaresProblem` (see 7.7).

The table 43 contains a list of the most important members of `TOMNETProblem`. To get familiar with how to use these parameters consult the quickguide examples in *tomnet/quickguide*.

A list of the most important methods are found in table 44.

Table 43: List of the most important members of `TOMNETProblem`.

Member	Description
A	`LinearConstraints` object for linear constrains.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
c_L	Lower bounds on the nonlinear constraints.
c_U	Upper bounds on the nonlinear constraints.
ConsDiff	Numerical approximation of the constraint derivatives. Constructors to Problems where ConsDiff is applicable automatically sets ConsDiff when appropriate. See Section 8 for instructions on how to force numerical differentiation of constraints.
N	Problem dimension (number of variables).
mLin	Number of linear constraints.

Table 43: List of the most important members of `TOMNETProblem`, continued.

Member	Description
mNonlin	Number of nonlinear constraints.
Name	Problem name.
NumDiff	Numerical approximation of the derivatives of the objective function. Constructors to Problems where NumDiff is applicable automatically sets NumDiff when appropriate. See Section 8 for instructions on how to force numerical differentiation of the objective function.
x_0	Starting point.
x_L	Lower bounds on the decision variables x.
x_U	Upper bounds on the decision variables x.
x_opt	Stationary points x^*, one per row (if known). It is possible to define an extra column, in which a zero (0) indicates a minimum point, a one (1) a saddle point, and a two (2) a maximum. As default, minimum points are assumed. The corresponding function values for each row in *x_opt* should be given in *Prob.f_opt*.

Table 44: List of the most important methods of `TOMNETProblem`.

Method	Description
Constructor	All classes that inherit `TOMNETProblem` have appropriate Constructors that can be used to create instances of the specific class.
f	Method for evaluating the objective function.
g	Method for evaluating the gradient of the objective function. If numerical differentiation is used `g` is never called.
c	Method for evaluating the nonlinear constraints.
dc	Method for evaluating the Jacobian (gradients of the nonlinear constraints.) If numerical differentiation is used `dc` is never called.

A.2 Creating a Problem.

In order to implement a custom `TOMNETProblem` there are a few items to consider:

1. Find the `TOMNETProblem` class with the best problem fit.
2. Implement the objective function (using an `IFunction` or an `IDFunction`).
3. Implement the nonlinear constraints (using an `IConstraints` or an `IDConstraints`).
4. Create objects needed to call the constructor.
5. Solve using an appropriate `TOMNETSolver`.

In this section we will create a problem with a custom nonlinear objective function, and two nonlinear constraints. This example is included in the TOMNET distribution, in (usersguide/CreateProblem/CreateProblem.cs).

A.2.1 Find the `TOMNETProblem` class that matched the problem.

Suppose we want to model the below problem:

$$\begin{array}{lrcl} \min\limits_{x} & f(x) = x[0] + x[1] \cdot x[2] - x[3]^{2.37} & & \\ & & & \\ & 0.1 \leq & x[i] & \leq 100, for i = 0,1,2,3 \\ & -10 \leq & x[0] + x[1] & \\ & -10 \leq & x[1] + x[2] & \\ s/t & -10 \leq & x[2] + x[3] & \\ & -10 \leq & x[3] + x[0] & \\ & -100 \leq & x[0] \cdot x[1] \cdot x[2] \cdot x[3] - x[0]^4 & \leq 1000 \\ & 0.0 \leq & x[0]^2 + x[1]^2 + x[2]^2 + x[3]^2 & \leq 1000 \end{array} \tag{10}$$

where $x, x_L, x_U \in \mathbb{R}^n$, $f(x) \in \mathbb{R}$, $A \in \mathbb{R}^{m_1 \times n}$, $b_L, b_U \in \mathbb{R}^{m_1}$ and $c_L, c(x), c_U \in \mathbb{R}^{m_2}$.

It is clear that this problem has linear and nonlinear constraints and a nonlinear objective function. The `ConLinProblem` (see 7.5) is the most suitable class.

A.2.2 Implement the objective function

We must create the objective function and the gradient. In the following example an `IDFunction` (see 7.9) is implemented. (Note that the parameter `fx` is an array - this is for compliance with problems like `LinearConstraintsProblem`, For regular problems this is an array of length 1.)

```
class myFunc : IDFunction
{
  public void Evaluate(double[] fx, double[] x)
  {
    fx[0] = x[0] + x[1] * x[2] - Math.Pow(x[3], 2.37);
  }

  public  void Grad(double[] gx, double[] x)
  {
    gx[0] = 1;
    gx[1] = x[2];
```

```
    gx[2] = x[1];
    gx[3] = -2.37 * Math.Pow(x[3], 1.37);
  }
}
```

A.2.3 Implement the nonlinear constraints

It is always best to separate the linear and nonlinear constraints since this improves the performance of most solvers. The linear constraints will be dealt with in the constructor and two nonlinear constraints remains to be implemented using an `IDConstraints` (see 7.11) or an `IConstraints` (see 7.10).

```
class myCons : IDConstraints
{
  public  void Evaluate(double[] cx, double[] x)
  {
    cx[0] = x[0] * x[1] * x[2] * x[3] - Math.Pow(x[0], 4.0);
    cx[1] = Math.Pow(x[0], 2.0) +
            Math.Pow(x[1], 2.0) +
            Math.Pow(x[2], 2.0) +
            Math.Pow(x[3], 2.0);
  }

  public double[] Evaluate(double[] x)
  {
    double[] cx = new double[2];
    this.Evaluate(cx, x);
    return cx;
  }

  public void dc(Jacobian dcx, double[] x)
  {
    dcx[0, 0] = x[1] * x[2] * x[3] - 4 * Math.Pow(x[0], 3.0);
    dcx[0, 1] = x[0] * x[2] * x[3];
    dcx[0, 2] = x[0] * x[1] * x[3];
    dcx[0, 3] = x[0] * x[1] * x[2];

    dcx[1, 0] = 2 * x[0];
    dcx[1, 1] = 2 * x[1];
    dcx[1, 2] = 2 * x[2];
    dcx[1, 3] = 2 * x[3];
  }
}
```

A.2.4 Create objects needed to call the constructor.

In this small example it is suitable to create all arrays, the linear constraints and set a name to use in the constructor call.

```
static void Main(string[] args)
{
  //
  // Name of the problem
  //
  string Name = "my problem";

  //
  // Bounds on x and initial point.
  //
  double[] x_L = new double[4] {   0.1,   0.1,   0.1,   0.1 };
  double[] x_U = new double[4] { 100.0, 100.0, 100.0, 100.0 };
  double[] x_0 = new double[4] {  10.0,  10.0,  10.0,  10.0 };

  //
  // Linear constraints
  //
  int mLin = 4;
  int nnz = 8;
  int[] ColIdx = new int[8] { 1, 2, 2, 3, 3, 4, 4, 1 };
  int[] RowIdx = new int[8] { 1, 1, 2, 2, 3, 3, 4, 4 };
  double[] Values = new double[8] { 1, 1, 1, 1, 1, 1, 1, 1 };
  Sparse A = new Sparse(mLin, 4, nnz, RowIdx, ColIdx, Values, 0);

  //
  // Bounds on linear constraints
  //
  double Inf = double.PositiveInfinity;
  double[] b_L = new double[4] { -10.0, -10.0, -10.0, -10.0 };
  double[] b_U = new double[4] { Inf, Inf, Inf, Inf };

  //
  // Create instances of the
  // objective function and the nonlinear constraints
  //
  myFunc F = new myFunc();
  myCons C = new myCons();

  //
  // Bounds on nonlinear constraints
  //
  double[] c_L = new double[2] { -100, 0.0 };
  double[] c_U = new double[2] { 1000, 1000 };

  //
  // Call constructor
  //
```

```
  ConLinProblem Prob = new ConLinProblem(F, A, b_L, b_U,
    C, c_L, c_U, x_L, x_U, x_0, Name, double.NegativeInfinity,
    null, null);

  //
  // The problem is created
  //

}
```

A.2.5 Solve using a `TOMNETSolver`.

In order to solve a `TOMNETProblem` we always need three objects: the problem, a solver and a result. Above we created a `TOMNETProblem`, now we need to add a solver, a result and solve the problem. By adding the following three lines of code we achieve this:

```
NPSOL solver = new NPSOL();
Result result;
solver.Solve(Prob, out result);
```

To display the solution on the console - we also add the following:

```
Console.WriteLine(" * Solved '{0}'", Prob.Name);
Console.WriteLine(" * Objective: {0}", result.f_k);
Console.WriteLine(" * Solution");
Console.WriteLine("        x_L[i] <=     x_k[i] <=     x_U[i]");
for (int i = 0; i < Prob.N; i++)
{
  Console.WriteLine("    {0,10:g3} <= {1,10:g3} <= {2,10:g3} ",
    Prob.x_L[i], result.x_k[i], Prob.x_U[i]);
}
```

The output from this problem should be as follows.

```
Creating my problem.
 * Solved my problem
 * Objective: -3588.98175192509
 * Solution
        x_L[i] <=     x_k[i] <=     x_U[i]
           0.1 <=        0.1 <=        100
           0.1 <=        0.1 <=        100
           0.1 <=        0.1 <=        100
           0.1 <=       31.6 <=        100
```

B Result - results from the solver

The results of the optimization attempts are stored in a containter class named *Result*. The most important members of the class are shown in Table 45.

Table 45: Information stored in the optimization result structure *Result*.

Field	Description
Name	Problem name.
P	Problem number.
probType	TOMNET problem type, according to Table 1, page 8.
Solver	Solver used.
SolverAlgorithm	Solver algorithm used.
solvType	TOMNET solver type.
ExitFlag	0 if convergence to local min. Otherwise errors.
ExitText	Text string describing the result of the optimization.
Inform	Information parameter, type of convergence.
CPUtime	CPU time used in seconds.
REALtime	Real time elapsed in seconds.
Iter	Number of major iterations.
MinorIter	Number of minor iterations (for some solvers).
maxTri	Maximum rectangle size.
FuncEv	Number of function evaluations needed.
GradEv	Number of gradient evaluations needed.
HessEv	Number of Hessian evaluations needed.
ConstrEv	Number of constraint evaluations needed.
ConJacEv	Number of constraint Jacobian evaluations needed.
ConHessEv	Number of nonlinear constraint Hessian evaluations needed.
ResEv	Number of residual evaluations needed (least squares).
JacEv	Number of Jacobian evaluations needed (least squares).
x_k	Optimal point.
f_k	Function value at optimum.
g_k	Gradient value at optimum.
B_k	Quasi-Newton approximation of the Hessian at optimum.
H_k	Hessian value at optimum.
y_k	Dual parameters.

Table 45: Information stored in the optimization result structure *Result*, continued.

Field	Description
v_k	Lagrange multipliers for constraints on variables, linear and nonlinear constraints.
r_k	Residual vector at optimum.
J_k	Jacobian matrix at optimum.
Ax	Value of linear constraints at optimum.
c_k	Value of nonlinear constraints at optimum.
cJac	Constraint Jacobian at optimum.
x_0	Starting point.
f_0	Function value at start i.e. $f(x\_0)$.
c_0	Value of nonlinear constraints at start.
Ax0	Value of linear constraints at start.
xState	State of each variable, described in Table 46.
bState	State of each linear constraint, described in Table 47.
cState	State of each general constraint, described in Table 48.

The field *xState* describes the state of each of the variables. In Table 46 the different values are described. The different conditions for linear constraints are defined by the state variable in field *bState*. In Table 47 the different values are described.

Table 46: The state variable *xState* for the variable.

Value	Description
0	A free variable.
1	Variable on lower bound.
2	Variable on upper bound.
3	Variable is fixed, lower bound is equal to upper bound.

Table 47: The state variable *bState* for each linear constraint.

Value	Description
0	Inactive constraint.
1	Linear constraint on lower bound.
2	Linear constraint on upper bound.
3	Linear equality constraint.

Table 48: The state variable *cState* for each nonlinear constraint.

Value	Description
0	Inactive constraint.
1	Nonlinear constraint on lower bound.
2	Nonlinear constraint on upper bound.
3	Nonlinear equality constraint.

C Motivation and Background

Many scientists and engineers are using .NET as a platform for modeling, analysis and computation. But for the solving optimization problems, the support is weak. That was one motive for starting the development of TOMNET.

To solve optimization problems the user has been forced to write significant amounts of own code or use basic entry-level solvers.

In recent years several modeling languages have been developed, like AIMMS [2], AMPL [7], ASCEND [20], GAMS [3, 4] and LINGO [1]. The modeling system acts as a preprocessor. The user describes the details of his problem in a very verbal language; an opposite to the concise mathematical description of the problem. The problem description file is normally modified in a text editor, with help from example files solving the same type of problem. Much effort is directed to the development of more user friendly interfaces. The model system processes the input description file and calls any of the available solvers. For a solver to be accessible in the modeling system, special types of interfaces are developed.

The modeling language approach is suitable for many management and decision problems, but may not always be the best way for engineering problems, which often are nonlinear with a complicated problem description. Until recently, the support for nonlinear problems in the modeling languages has been crude. This is now rapidly changing [6].

For people with a mathematical background, modeling languages often seems to be a very tedious way to define an optimization problem. There has been several attempts to find languages more suitable than Fortran or C/C++ to describe mathematical problems, like the compact and powerful APL language [15, 21].

The concept of TOMNET is to integrate various modeling classes and interfaces, getting access to the best of all worlds. TOMNET should be seen as a complement to existing, often more specific, modeling platforms, for the user needing more power and flexibility than given by a just a programming language.

References

[1] *LINGO - The Modeling Language and Optimizer*. LINDO Systems Inc., Chicago, IL, 1995.

[2] J. Bisschop and R. Entriken. *AIMMS - The Modeling System*. Paragon Decision Technology, Haarlem, The Netherlands, 1993.

[3] J. Bisschop and A. Meeraus. On the development of a general algebraic modeling system in a strategic planning environment. *Mathematical Programming Study*, 20:1–29, 1982.

[4] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS - A User's Guide*. The Scientific Press, Redwood City, CA, 1988.

[5] C. D. Perttunen D. R. Jones and B. E. Stuckman. Lipschitzian optimization without the Lipschitz constant. October 1993.

[6] Arne Stolbjerg Drud. Interactions between nonlinear programing and modeling systems. *Mathematical Programming, Series B*, 79:99–123, 1997.

[7] R. Fourer, D. M. Gay, and B. W.Kernighan. *AMPL - A Modeling Language for Mathematical Programming*. The Scientific Press, Redwood City, CA, 1993.

[8] Philip E. Gill, Sven J. Hammarling, Walter Murray, Michael A. Saunders, and Margaret H. Wright. User's guide for LSSOL ((version 1.0): A Fortran package for constrained linear least-squares and convex quadratic programming. Technical Report SOL 86-1, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California 94305-4022, 1986.

[9] Philip E. Gill, Walter Murray, and Michael A. Saunders. User's guide for QPOPT 1.0: A Fortran package for Quadratic programming. Technical Report SOL 95-4, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California 94305-4022, 1995.

[10] Philip E. Gill, Walter Murray, and Michael A. Saunders. SNOPT: An SQP algorithm for Large-Scale constrained programming. Technical Report SOL 97-3, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California 94305-4022, 1997.

[11] Philip E. Gill, Walter Murray, and Michael A. Saunders. User's guide for SQOPT 5.3: A Fortran package for Large-Scale linear and quadratic programming. Technical Report Draft October 1997, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California 94305-4022, 1997.

[12] Philip E. Gill, Walter Murray, and Michael A. Saunders. User's guide for SNOPT 5.3: A Fortran package for Large-Scale nonlinear programming. Technical Report SOL 98-1, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California 94305-4022, 1998.

[13] Philip E. Gill, Walter Murray, Michael A. Saunders, and Margaret H. Wright. User's guide for NPSOL 5.0: A Fortran package for nonlinear programming. Technical Report SOL 86-2, Revised July 30, 1998, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California 94305-4022, 1998.

[14] K. Holmström. The TOMLAB Optimization Environment in Matlab. *Advanced Modeling and Optimization*, 1(1):47–69, 1999.

[15] Kenneth Iverson. *A Programming Language*. John Wiley and Sons, New York, 1962.

[16] Donald R. Jones. *Encyclopedia of Optimization*. To be published, 2001.

[17] David G. Luenberger. *Linear and Nonlinear Programming*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition, 1984.

[18] Bruce A. Murtagh and Michael A. Saunders. MINOS 5.5 USER'S GUIDE. Technical Report SOL 83-20R, Revised July 1998, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California 94305-4022, 1998.

[19] G. L. Nemhauser and L. A. Wolsey. Integer programming. In G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, editors, *Optimization*, volume 1 of *Handbooks in Operations Research and Management Science*. Elsevier/North Holland, Amsterdam, The Netherlands, 1989.

[20] P. C. Piela, T. G. Epperly, K. M. Westerberg, and A. W. Westerberg. ASCEND: An object-oriented computer environment for modeling and analysis: The modeling language. *Computers and Chemical Engineering*, 15:53–72, 1991.

[21] Raymond P. Polivka and Sandra Pakin. *APL: The Language and Its Usage*. Prentice Hall, Englewood Cliffs, N. J., 1975.

[22] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.