

TOMNET QUICK START GUIDE

Marcus M. Edvall¹ and Per Strandberg²

January 25, 2007



¹Tomlab Optimization Inc., 1260 SE Bishop Blvd Ste E, Pullman, WA, USA, medvall@tomopt.com.

²Tomlab Optimization AB, Västerås Technology Park, Trefasgatan 4, SE-721 30 Västerås, Sweden, per@tomopt.com.

Contents

Contents	2
1 QuickGuide Overview	3
2 LP Problem	4
3 QP Problem	7
4 NLP Problem	10
5 LPCON Problem	14
6 QPCON Problem	19
7 LLS Problem	24
8 NLLS Problem	27
9 GLB Problem	32
10 GLC Problem	36
11 Important Information	41
11.1 Using Patterns	41
11.2 Solver Timings	41

1 QuickGuide Overview

This guide illustrates the very basics associated with solving problems using the TOMNET Optimization Environment. After this session you will have solved several different problems with a variety of solvers. The solvers you can use will depend on what you are licensed for. The total time required for these exercises has been estimated to about 45 minutes.

Please make a copy of a suitable case for further implementations as needed.

The test cases were developed using Microsoft Visual C# 2005 and are best run from a command prompt (CMD.EXE or DOS). Observe that the DLL folder in the TOMNET installation needs to be included in the Environment variable PATH for the software to work properly

(Start -- > Control Panel -- > Performance and Maintenance -- > System -- > Advanced Tab -- > Environment Variable -- > In the PATH variable append the location of DLL files (e.g. d:\TOMNET\dll)).

It is also possible to set this temporarily from the command prompt.

Possible errors as a result of missing PATH variable:

```
Unhandled Exception: System.DllNotFoundException: Unable to load DLL
'TOMNETSnoptSub.dll': The specified module could not be found.
(Exception from HRESULT: 0x8 007007E)
  at TOMNET.SNOPT.tnInitProb(Char[] printFile, Char[] summFile, Char[] specsFile,
  Int32 nObj, Int32 priLev, Int32 minPriLev, Int32 numDiff, Int32 consDiff,
  Int32& fileID1, Int32& fileID2, Int32& fileID3, Int32& fileID4,
  Int32& fileID5, Int32& fileID6, Int32& fileID7)
  at TOMNET.SNOPT.Solve(TOMNETProblem Prob, Result& result)
  at TOMNET.lpQG.Main(String[] args)
```

2 LP Problem

The general formulation in TOMNET for a linear programming problem is:

$$\begin{aligned} \min_x \quad & f(x) = c^T x \\ \text{s/t} \quad & x_L \leq x \leq x_U \\ & b_L \leq Ax \leq b_U \end{aligned} \tag{1}$$

where $c, x, x_L, x_U \in \mathbb{R}^n$, $A \in \mathbb{R}^{m_1 \times n}$, and $b_L, b_U \in \mathbb{R}^{m_1}$. Equality constraints are defined by setting the lower bound to the upper bound.

Example problem:

$$\begin{aligned} \min_{x_1, x_2} \quad & f(x_1, x_2) = -7x_1 - 5x_2 \\ \text{s/t} \quad & x_1 + 2x_2 \leq 6 \\ & 4x_1 + x_2 \leq 12 \\ & x_1, x_2 \geq 0 \end{aligned} \tag{2}$$

The following code (maintained from the solution file: `quickguide/lpQG/lpQG.sln`) defines the problem in TOMNET.

Observe that there are three ways to define the linear constraints, firstly as a dense matrix row-wise, secondly by giving the indices for the entries, or by using the internal sparse matrix format.

```
using System;
using TOMNET;

namespace TOMNET
{
    /// <summary>
    /// Quick guide class for linear problem.
    /// </summary>
    static public class lpQG
    {
        /// <summary>
        /// Creates a simple linear problem.
        /// </summary>
        /// <returns>A simple linear problem.</returns>
        static public LinearProblem getLpQG()
        {
            const int N = 2;    // Number of decision variables
            const int mLin = 2; // Number of linear constraints
            const int nnz = 4;  // Number of nonzeros in linear
                               // constraints

            string Name = "lpQG"; // Problem name
            double Inf = double.PositiveInfinity;
```

```

// Linear objective function entries
double[] c = new double[N] { -7, -5 };

// Lower bound for linear constraints
double[] b_L = new double[mLin] { -Inf, -Inf };

// Upper bound for linear constraints
double[] b_U = new double[mLin] { 6, 12 };

// Lower and upper bounds for decision variables as well
// as the starting point for the solver
double[] x_L = new double[N] { 0, 0 };
double[] x_U = new double[N] { Inf, Inf };
double[] x_0 = new double[N] { 0, 0 };

// Method one (dense matrix with linear constraints)
// double[] A = new double[N*mLin] { 1, 2, 4, 1 };

// Method two (the standard sparse user format)
// double[] Av = new double[nnz] { 1, 4, 2, 1 };
// int[] Ai = new int[nnz] { 0, 1, 0, 1 };
// int[] Aj = new int[nnz] { 0, 0, 1, 1 };
// Sparse A = new Sparse(mLin, N, nnz, Ai, Aj, Av, 0);

// Method three (the internal representation)
double[] Apr = new double[nnz] { 1, 4, 2, 1 };
int[] Air = new int[nnz] { 0, 1, 0, 1 };
int[] Ajc = new int[N + 1] { 0, 2, 4 };
Sparse A = new Sparse(mLin, N, nnz, Air, Ajc, Apr);

// Create the linear problem
LinearProblem Prob = new LinearProblem(c, A, b_L, b_U, x_L,
    x_U, x_0, Name, double.NegativeInfinity, null, null);
return Prob;
}

static void Main(string[] args)
{
    LinearProblem Prob = lpQG.getLpQG();

    // Create a solver instance
    SNOPT solver = new SNOPT();
    // Create the result structure
    Result result;

    // Define a print file for SNOPT

```

```

string printfilename = "SnoptLP.txt";
solver.Options.PrintFile = printfilename;

// Solve the problem
solver.Solve(Prob, out result);

// Display the solution on the screen
Console.WriteLine(" * * * * *");
Console.WriteLine(" * Solved an LP problem");
Console.WriteLine(" * Printfile: " + printfilename);
Console.WriteLine(" * Objective: {0}", result.f_k);
Console.WriteLine(" * Solution (x_L[i] <= x_k[i] <= x_U[i]): ");
for (int i = 0; i < Prob.N; i++)
    Console.WriteLine("    {0} <= {1} <= {2} ", Prob.x_L[i], result.x_k[i], Prob.x_U[i]);

double[] Ax = Prob.A.Evaluate(result.x_k);
Console.WriteLine(" * Linear constraints (b_L[i] <= (A*x)[i] <= b_U[i]):");
for (int i = 0; i < Prob.mLin; i++)
    Console.WriteLine("    {0} <= {1} <= {2}", Prob.A.GetLowerBound(i),
        Ax[i], Prob.A.GetUpperBound(i));
}
}
}

```

After opening *lpQG.sln* build the solution (results in an exe file). The executable can be run from the command prompt, which should display the following output results:

```

C:\WINDOWS\system32\cmd.exe
D:\source\v5>cd quickguide
D:\source\v5\quickguide>cd lpQG
D:\source\v5\quickguide\lpQG>lpQG
* * * * *
* Solved an LP problem
* Printfile: SnoptLP.txt
* Objective: -26.5714285714286
* Solution (x_L[i] <= x_k[i] <= x_U[i]):
  0 <= 2.57142857142857 <= Infinity
  0 <= 1.71428571428571 <= Infinity
* Linear constraints (b_L[i] <= (A*x)[i] <= b_U[i]):
-Infinity <= 6 <= 6
-Infinity <= 12 <= 12
D:\source\v5\quickguide\lpQG>

```

3 QP Problem

The general formulation in TOMNET for a quadratic programming problem is:

$$\begin{aligned} \min_x \quad & f(x) = \frac{1}{2}x^T Fx + c^T x \\ \text{s/t} \quad & \begin{array}{l} x_L \leq x \leq x_U \\ b_L \leq Ax \leq b_U \end{array} \end{aligned} \tag{3}$$

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 $b_L, b_U \in \mathbb{R}^{m_1}$. Equality constraints are defined by setting the lower bound equal to the upper bound, i.e. for constraint i : $b_L(i) = b_U(i)$. Fixed variables are handled the same way.

Example problem:

$$\begin{aligned} \min_x \quad & f(x) = 4x_1^2 + 1x_1x_2 + 4x_2^2 + 3x_1 - 4x_2 \\ \text{s/t} \quad & \begin{array}{l} x_1 + x_2 \leq 5 \\ x_1 - x_2 = 0 \\ x_1 \geq 0 \\ x_2 \geq 0 \end{array} \end{aligned} \tag{4}$$

The following code (maintained from the solution file: `quickguide/qpQG/qpQG.sln`) defines the problem in TOMNET.

```
using System;
using TOMNET;

namespace TOMNET
{
    /// <summary>
    /// Quick guide class for quadratic problem.
    /// </summary>
    static public class qpQG
    {
        /// <summary>
        /// Creates a simple quadratic problem.
        /// </summary>
        /// <returns>A simple quadratic problem.</returns>
        static public QuadraticProblem getQpQG()
        {
            double Inf = double.PositiveInfinity;

            string Name = "QP Example";
            double[] F = new double[4] { 8, 1, 1, 8 }; // Matrix F in 1/2 * x' * F * x + c' * x
            double[] c = new double[2] { 3, -4 }; // Vector c in 1/2 * x' * F * x + c' * x
            double[] A = new double[4] { 1, 1, 1, -1 }; // Constraint matrix
            double[] b_L = new double[2] { -Inf, 0 }; // Lower bounds on the linear constraints
        }
    }
}
```

```

double[] b_U = new double[2] { 5, 0 }; // Upper bounds on the linear constraints
double[] x_L = new double[2] { 0, 0 }; // Lower bounds on the variables
double[] x_U = new double[2] { Inf, Inf }; // Upper bounds on the variables
double[] x_0 = new double[2] { 0, 1 }; // Starting point

// Create the quadratic problem
QuadraticProblem Prob = QuadraticProblem.qpAssign(F,c,A,b_L,b_U,x_L,x_U,x_0,Name);
return Prob;
}

static void Main(string[] args)
{
    QuadraticProblem Prob = qpQG.getQpQG();

    // Create a solver instance
    SNOPT solver = new SNOPT();

    // Create the result structure
    Result result;

    Prob.Name += " (solved with SNOPT)";

    // Define a print file for SNOPT
    string printfilename = "SnoptQP.txt";
    solver.Options.PrintFile = printfilename;

    // Solve the problem
    solver.Solve(Prob, out result);

    // Get the solution results
    double[] solution = result.x_k;
    double objective = result.f_k;

    // Display the solution on the screen
    Console.WriteLine(" * * * * *");
    Console.WriteLine(" * Solved a QP problem");
    Console.WriteLine(" * Printfile: " + printfilename);
    Console.WriteLine(" * Objective: {0}", objective);
    Console.WriteLine(" * Solution (x_L[i] <= x_k[i] <= x_U[i]): ");
    for (int i = 0; i < solution.Length; i++)
        Console.WriteLine(" {0} <= {1} <= {2} ", Prob.x_L[i], solution[i], Prob.x_U[i]);

    double[] Ax = Prob.A.Evaluate(result.x_k);
    Console.WriteLine(" * Linear constraints (b_L[i] <= (A*x)[i] <= b_U[i]):");
    for (int i = 0; i < Prob.mLin; i++)
        Console.WriteLine(" {0} <= {1} <= {2}", Prob.A.GetLowerBound(i),
            Ax[i], Prob.A.GetUpperBound(i));
}

```

```
}  
}  
}
```

The solution file *qpQG.sln* can be compiled to build an exe file. The executable can be run from the command prompt, which should display the following output results:

```
C:\WINDOWS\system32\cmd.exe  
D:\source\v5>cd quickguide\qpQG  
D:\source\v5\quickguide\qpQG>qpQG  
* * * * *  
* Solved a QP problem  
* Printfile: SnoptQP.txt  
* Objective: -0.02777777777777778  
* Solution (x_L[i] <= x_k[i] <= x_U[i]):  
  0 <= 0.0555555555555555 <= Infinity  
  0 <= 0.0555555555555555 <= Infinity  
* Linear constraints (b_L[i] <= (A*x)[i] <= b_U[i]):  
-Infinity <= 0.111111111111111 <= 5  
  0 <= 0 <= 0  
D:\source\v5\quickguide\qpQG>
```

4 NLP Problem

TOMNET requires that general nonlinear problems are defined in .NET classes's. The function to be optimized must always be supplied. It is recommended that the user supply as many analytical functions (i.e. the gradient and constraint Jacobian) as possible, since this greatly increases robustness and results in a faster execution time.

The constrained nonlinear programming problem is defined as:

$$\begin{aligned}
 & \min_x f(x) \\
 & s/t \quad \begin{array}{ccccc} x_L & \leq & x & \leq & x_U \\ b_L & \leq & Ax & \leq & b_U \\ c_L & \leq & c(x) & \leq & c_U \end{array}
 \end{aligned} \tag{5}$$

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 $c_L, c(x), c_U \in \mathbb{R}^{m_2}$.

Example problem:

$$\begin{aligned}
 & \min_x f(x) = \alpha (x_2 - x_1^2)^2 + (1 - x_1)^2 \\
 & s/t \quad \begin{array}{ccccc} -10 & \leq & x_1 & \leq & 2 \\ -10 & \leq & x_2 & \leq & 2 \\ -inf & \leq & x_1 * x_2 & \leq & 0.5 \end{array} \\
 & \alpha = 100
 \end{aligned} \tag{6}$$

The following code (maintained from the solution file: `quickguide/nlpQG/nlpQG.sln`) defines the problem in TOMNET.

```

using System;
using TOMNET;

namespace TOMNET
{
    /// <summary>
    /// Quick guide class for nonlinear problem (Rosenbrocks Banana).
    /// </summary>
    public class nlpQG : TOMNETProblem
    {
        /// <summary>
        /// Constant used in objective function and gradient.
        /// </summary>
        public double alpha;

        /// <summary>
        /// Constructor for nlpQG (Rosenbrocks Banana).
        /// </summary>
        /// <param name="alpha">Parameter Alpha</param>
        /// <remarks>

```

```

/// Calls base constructor with
/// 2 (for the number of variables),
/// 0 (for the number of linear constraints) and
/// 1 (for the number of nonlinear constraints).
/// </remarks>
public nlpQG(double alpha)
    : base(2, 0, 1)
{
    this.alpha = alpha;
    this.Name = "RBB";

    // Starting point
    this.x_0[0] = -1.2;
    this.x_0[1] = 1.0;

    // Lower and upper bounds
    this.x_L[0] = -10.0;
    this.x_L[1] = -10.0;
    this.x_U[0] = 2.0;
    this.x_U[1] = 2.0;

    // Lower and upper bounds for nonlinear constraints
    this.c_L[0] = -1000.0;
    this.c_U[0] = 0.0;
}

// Objective function
public override void f(double[] fx, double[] x)
{
    fx[0] = alpha * Math.Pow(x[1] - Math.Pow(x[0], 2), 2) + Math.Pow((1 - x[0]), 2);
}

// Gradient (first derivative) for the objective function
public override void g(double[] g, double[] x)
{
    g[0] = -4 * alpha * x[0] * (x[1] - Math.Pow(x[0], 2)) - 2 * (1 - x[0]);
    g[1] = 2 * alpha * (x[1] - Math.Pow(x[0], 2));
}

// Nonlinear constraint vector
public override void c(double[] c, double[] x)
{
    c[0] = -Math.Pow(x[0], 2) - x[1];
}

// Nonlinear constraint Jacobian (first derivative)
public override void dc(Jacobian Jac, double[] x)

```

```

{
    Jac[0, 0] = -2 * x[0];
    Jac[0, 1] = -1;
}

static void Main(string[] args)
{
    // Creating problem with given alpha value
    double alpha = 100.0;
    nlpQG Prob = new nlpQG(alpha);

    // Create a solver instance
    SNOPT solver = new SNOPT();

    // Create the result structure
    Result result;

    // Define a print file for SNOPT
    string printfilename = "SnoptNlpQG.txt";
    solver.Options.PrintFile = printfilename;

    // Solve the problem
    solver.Solve(Prob, out result);

    // Get the solution results
    double[] solution = result.x_k;
    double objective = result.f_k;

    // Display the solution on the screen
    Console.WriteLine(" * * * * *");
    Console.WriteLine(" * Solved " + Prob.Name);
    Console.WriteLine(" * Printfile: " + printfilename);
    Console.WriteLine(" * Objective: {0}", objective);
    Console.WriteLine(" * Solution (x_L[i] <= x_k[i] <= x_U[i]): ");
    for (int i = 0; i < solution.Length; i++)
        Console.WriteLine("    {0} <= {1} <= {2} ", Prob.x_L[i], solution[i], Prob.x_U[i]);

    Console.WriteLine(" * Nonlinear constraints (c_L[i] <= c(x)[i] <= c_U[i]):");
    double[] cx = new double[Prob.mNonLin];
    Prob.c(cx, result.x_k);
    for (int i = 0; i < Prob.mNonLin; i++)
        Console.WriteLine("    {0} <= {1} <= {2}", Prob.c_L[i], cx[i], Prob.c_U[i]);
}
}
}

```

Observe that the code defines four functions that are used in the callback from the solvers.

f: Function value
g: Gradient vector
c: Nonlinear constraint vector
dc: Nonlinear constraint gradient matrix

The *nlpQG.sln* can be opened by a suitable editor and compiled to an exe file. When running the executable from a command prompt the following output results are displayed:

```
C:\WINDOWS\system32\cmd.exe
D:\source\v5\quickguide\nlpQG>nlpQG
*****
* Solved RBB
* Printfile: SnoptNlpQG.txt
* Objective: 1.03756205935079E-18
* Solution (x_L[i] <= x_k[i] <= x_U[i]):
  -10 <= 1.00000000074403 <= 2
  -10 <= 1.0000000014185 <= 2
* Nonlinear constraints (c_L[i] <= c(x)[i] <= c_U[i]):
  -1000 <= -2.00000000290657 <= 0
D:\source\v5\quickguide\nlpQG>
```

5 LPCON Problem

When solving a problem with a linear objective and nonlinear constraints there is no need to explicitly code the gradient or Hessian since these are implicitly given (the gradient is a constant array and the Hessian is all zeros). TOMNET automatically supplies these when inheriting from the linear problem class.

The linear constrained nonlinear programming problem is defined as:

$$\begin{aligned} \min_x \quad & f(x) = d^T x \\ \text{s/t} \quad & x_L \leq x \leq x_U \\ & b_L \leq Ax \leq b_U \\ & c_L \leq c(x) \leq c_U \end{aligned} \tag{7}$$

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

The solution quickguide/lpconQG/lpconQG.sln defines an example problem in TOMNET.

```
using System;
using TOMNET;

namespace TOMNET
{
    /// <summary>
    /// Quick guide class for linear problem
    /// with both linear and nonlinear constraints.
    /// </summary>
    public class lpconQG : LinearProblem
    {
        /// <summary>
        /// Constructor for Quick Guide Example for a linear
        /// problem with both linear and nonlinear constraints.
        /// </summary>
        /// <param name="n">Number of variables</param>
        /// <param name="mLin">Number of linear constraints</param>
        /// <param name="mNonLin">Number of nonlinear constraints</param>
        /// <param name="c">Linear goal function</param>
        /// <param name="A">Linear constraints (row-based)</param>
        /// <param name="b_L">Lower limits of linear constraints</param>
        /// <param name="b_U">Upper limits of linear constraints</param>
        /// <param name="x_0">Initial point</param>
        /// <param name="x_L">Lower limits of variables</param>
        /// <param name="x_U">Upper limits of variables</param>
        /// <param name="c_L">Lower limits of nonlinear constraints</param>
        /// <param name="c_U">Upper limits of nonlinear constraints</param>
        /// <param name="Name">Name of problem</param>
        public lpconQG(int n, int mLin, int mNonLin, double[] c, double[] A,
            double[] b_L, double[] b_U, double[] x_0, double[] x_L, double[] x_U,
            double[] c_L, double[] c_U, String Name)
    }
}
```

```

: base(c, A, b_L, b_U, x_L, x_U, x_0, Name, double.NegativeInfinity, null, null)
{
// Nonlinear constraints
this.mNonLin = mNonLin;
this.c_L = new double[mNonLin];
this.c_U = new double[mNonLin];

// Setting the nonlinear constraint bounds
for (int i = 0; i < mNonLin; i++)
{
    this.c_L[i] = c_L[i];
    this.c_U[i] = c_U[i];
}

this.Name = Name;
this.NumDiff = 1;
}

/// <summary>
/// Nonlinear constraints
/// </summary>
/// <param name="c">Vector with nonlinear constraints in x.</param>
/// <param name="x">The decision variables.</param>
public override void c(double[] c, double[] x)
{
    if (x[0] == 0 && x[1] == 0)
        c[0] = 1.0;
    else
        c[0] = Math.Pow(x[0] * x[2] + x[1] * x[3], 2) / (Math.Pow(x[0], 2) +
            Math.Pow(x[1], 2)) - Math.Pow(x[2], 2) - Math.Pow(x[3], 2);
}

/// <summary>
/// Jacobian of nonlinear constraints
/// </summary>
/// <param name="Jac">Jacobian of nonlinear constraints</param>
/// <param name="x">The decision variables.</param>
public override void dc(Jacobian Jac, double[] x)
{
    double h1 = Math.Pow(x[0], 2) + Math.Pow(x[1], 2);
    double h2 = x[0] * x[2] + x[1] * x[3];

    if (h1 == 0)
    {
        Jac[0, 0] = double.PositiveInfinity;
        Jac[0, 1] = double.PositiveInfinity;
        Jac[0, 2] = double.PositiveInfinity;
    }
}

```

```

    Jac[0, 3] = double.PositiveInfinity;
}
else
{
    Jac[0, 0] = 2 * h2 * x[2] / h1 - 2 * x[0] * Math.Pow(h2 / h1, 2);
    Jac[0, 1] = 2 * h2 * x[3] / h1 - 2 * x[1] * Math.Pow(h2 / h1, 2);
    Jac[0, 2] = 2 * h2 * x[0] / h1 - 2 * x[2];
    Jac[0, 3] = 2 * h2 * x[1] / h1 - 2 * x[3];
}
}

static void Main()
{
    double Inf = double.PositiveInfinity;

    string Name = "lpconQG";
    const int mLin = 3;    // Number of linear constraints
    const int mNonLin = 1; // Number of nonlinear constraints
    const int n = 4;      // Number of decision variables

    // Linear constraints, lower and upper bounds
    double[] A = new double[n * mLin] { 1, 0, -1, 0, 0, 1, 0, -1, 0, 0, 1, -1 };
    double[] b_L = new double[mLin] { 1, 1, 0 };
    double[] b_U = new double[mLin] { Inf, Inf, Inf };

    // Nonlinear constraint bounds
    double[] c_L = new double[mNonLin] { -1 };
    double[] c_U = new double[mNonLin] { -1 };

    // Starting point, lower and upper bounds for decision variables
    double[] x_0 = new double[n] { 1, 0, 2, 0 };
    double[] x_L = new double[n] { -Inf, -Inf, -Inf, 1 };
    double[] x_U = new double[n] { 100, 100, 100, 100 };

    // Entries for linear objective function
    double[] c = new double[n] { 3, 2, 0, 0 };

    // Create a problem instance
    lpconQG Prob = new lpconQG(n, mLin, mNonLin, c, A, b_L, b_U,
        x_0, x_L, x_U, c_L, c_U, Name);

    // Create a solver and result instance
    SNOPT solver = new SNOPT();
    Result result;

    // Define a print file for SNOPT
    string printfilename = "SnoptlpconQP.txt";

```

```

solver.Options.PrintFile = printfilename;

// Solve the problem
solver.Solve(Prob, out result);

// Get the solution results
double[] solution = result.x_k;
double objective = result.f_k;

// Display the solution on the screen
Console.WriteLine(" * * * * *");
Console.WriteLine(" * Solved " + Prob.Name);
Console.WriteLine(" * Printfile: " + printfilename);
Console.WriteLine(" * Objective: {0}", objective);
Console.WriteLine(" * Solution (x_L[i] <= x_k[i] <= x_U[i]): ");
for (int i = 0; i < solution.Length; i++)
    Console.WriteLine("    {0} <= {1} <= {2} ", Prob.x_L[i], solution[i], Prob.x_U[i]);

Console.WriteLine(" * Linear constraints (b_L[i] <= (A*x)[i] <= b_U[i]):");
double[] Ax = Prob.A.Evaluate(result.x_k);
for (int i = 0; i < Prob.mLin; i++)
    Console.WriteLine("    {0} <= {1} <= {2}", Prob.A.GetLowerBound(i),
        Ax[i], Prob.A.GetUpperBound(i));

Console.WriteLine(" * Nonlinear constraints (c_L[i] <= c(x)[i] <= c_U[i]):");
double[] cx = new double[Prob.mNonLin];
Prob.c(cx, result.x_k);
for (int i = 0; i < Prob.mNonLin; i++)
    Console.WriteLine("    {0} <= {1} <= {2}", Prob.c_L[i], cx[i], Prob.c_U[i]);
}
}
}

```

Observe that the code defines two functions that are used in the callback from the solvers. The `d2c` function is currently not implemented.

```

c:   Nonlinear constraint vector
dc:  Nonlinear constraint gradient matrix
d2c: The second part of the Hessian to the Lagrangian
      function for the nonlinear constraints.

```

The `lpconQG.sln` may be opened by a suitable editor and compiled to an exe file. When running the executable from a command prompt the following output results are displayed:

```
C:\WINDOWS\system32\cmd.exe

D:\source\v5\quickguide\lpconQG>lpconQG
* * * * *
* Solved lpconQG
* Printfile: SnoptlpconQP.txt
* Objective: 16.9282019102968
* Solution <x_L[i] <= x_k[i] <= x_U[i]>:
  -Infinity <= 3.1546957170477 <= 100
  -Infinity <= 3.73205737957685 <= 100
  -Infinity <= 2.1546957170477 <= 100
  1 <= 1 <= 100
* Linear constraints <b_L[i] <= (A*x)[i] <= b_U[i]>:
  1 <= 1 <= Infinity
  1 <= 2.73205737957685 <= Infinity
  0 <= 1.1546957170477 <= Infinity
* Nonlinear constraints <c_L[i] <= c(x)[i] <= c_U[i]>:
  -1 <= -0.999999624258366 <= -1

D:\source\v5\quickguide\lpconQG>
```

6 QPCON Problem

When solving a problem with a quadratic objective and nonlinear constraints TOMNET automatically supplies objective derivatives (gradient and Hessian) if inheriting from the quadratic problem class, as shown for the LPCON 5 problem.

The quadratic constrained nonlinear programming problem is defined as:

$$\begin{aligned} \min_x \quad & f(x) = \frac{1}{2}x^T Fx + d^T x \\ \text{s/t} \quad & \begin{array}{l} x_L \leq x \leq x_U \\ b_L \leq Ax \leq b_U \\ c_L \leq c(x) \leq c_U \end{array} \end{aligned} \tag{8}$$

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

The following file (maintained from `quickguide/qpconQG/qpconQG.sln`) defines and solves an example problem in TOMNET:

```
using System;
using TOMNET;

namespace TOMNET
{
    /// <summary>
    /// Quick guide class for quadratic problem with both
    /// linear and nonlinear constraints.
    /// </summary>
    public class qpconQG : QuadraticProblem
    {
        /// <summary>
        /// Constructor for Quadratic problem with both
        /// linear and nonlinear constraints.
        /// </summary>
        /// <param name="n">Number of variables</param>
        /// <param name="mLin">Number of linear constraints</param>
        /// <param name="mNonLin">Number of nonlinear constraints</param>
        /// <param name="c">Linear part of goal function</param>
        /// <param name="F">Quadratic part of goal function</param>
        /// <param name="A">Linear constraints (row-based)</param>
        /// <param name="b_L">Lower limits of linear constraints</param>
        /// <param name="b_U">Upper limits of linear constraints</param>
        /// <param name="x_0">Initial point</param>
        /// <param name="x_L">Lower limits of variables</param>
        /// <param name="x_U">Upper limits of variables</param>
        /// <param name="c_L">Lower limits of nonlinear constraints</param>
        /// <param name="c_U">Upper limits of nonlinear constraints</param>
        /// <param name="Name">Name of problem</param>
        public qpconQG(int n, int mLin, int mNonLin, double[] c, double[] F,
```

```

double[] A, double[] b_L, double[] b_U, double[] x_0, double[] x_L,
double[] x_U, double[] c_L, double[] c_U, String Name)
: base(n, mLin, F, c, A, b_L, b_U, x_0, x_L, x_U)
{

    // Nonlinear constraints
    this.mNonLin = mNonLin;
    this.c_L = new double[mNonLin];
    this.c_U = new double[mNonLin];

    // Setting the nonlinear constraint bounds
    c_L.CopyTo(this.c_L, 0);
    c_U.CopyTo(this.c_U, 0);

    this.Name = Name;
    this.NumDiff = 1;
}

/// <summary>
/// Nonlinear constraints
/// </summary>
/// <param name="c">Values of nonlinear constraints in x.</param>
/// <param name="x">The decision variables.</param>
public override void c(double[] c, double[] x)
{
    c[0] = 0.0;
    for (int i = 0; i < this.N; i++)
        c[0] += Math.Pow(x[i], 2.0) / (1.0 + ((double)i) / 3.0);
}

/// <summary>
/// Jacobian for the nonlinear constraints
/// </summary>
/// <param name="Jac">Jacobian of nonlinear constraints.</param>
/// <param name="x">The decision variables.</param>
public override void dc(Jacobian Jac, double[] x)
{
    for (int i = 0; i < this.N; i++)
        Jac[0, i] = 2.0 * x[i] / (1.0 + ((double)i) / 3.0);
}

static void Main()
{
    double Inf = double.PositiveInfinity;

    string Name = "qpconQG";
    const int n = 10;        // Number of decision variables

```

```

const int mLin = 8;    // Number of linear constraints
const int mNonLin = 1; // Number of nonlinear constraints

// Lower and upper bounds for the linear constraints
double[] b_L = new double[mLin] { 1, 1, 1, 1, 1, 1, 1, 1 };
double[] b_U = new double[mLin] { 1, 1, 1, 1, 1, 1, 1, 1 };

// Nonlinear constraint bounds
double[] c_L = new double[mNonLin] { 4 };
double[] c_U = new double[mNonLin] { 4 };

// Starting point, lower and upper bounds for decision variables
double[] x_0 = new double[n] { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };
double[] x_L = new double[n] { -Inf, -Inf, -Inf, -Inf, -Inf,
    -Inf, -Inf, -Inf, -Inf, -Inf };
double[] x_U = new double[n] { Inf, Inf, Inf, Inf, Inf,
    Inf, Inf, Inf, Inf, Inf };

// Entries for linear objective function
double[] c = new double[n] { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };

// Linear constraints
double[] A = new double[n * mLin];

// Quadratic part of objective function
double[] F = new double[100];

// Setting the linear constraints
for (int i = 0; i < mLin; i++)
    for (int j = 0; j < n; j++)
        if (i == j)
            A[i * n + j] = 0.5;
        else
            A[i * n + j] = 1;

// Setting the quadratic objective
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 10; j++)
        if (j == i)
            F[i + 10 * j] = -2;
        else
            F[i + 10 * j] = 0.0;

// Create a problem instance
qpconQG Prob = new qpconQG(n, mLin, mNonLin, c, F, A, b_L, b_U,
    x_0, x_L, x_U, c_L, c_U, Name);

```

```

// Create a solver and result instance
SNOPT solver = new SNOPT();
Result result;

// Define a print file for SNOPT
string printfilename = "SnoptqpconQP.txt";
solver.Options.PrintFile = printfilename;

// Solve the problem
solver.Solve(Prob, out result);

// Get the solution results
double[] solution = result.x_k;
double objective = result.f_k;

// Display the solution on the screen
Console.WriteLine(" * * * * *");
Console.WriteLine(" * Solved " + Prob.Name);
Console.WriteLine(" * Printfile: " + printfilename);
Console.WriteLine(" * Objective: {0}", objective);
Console.WriteLine(" * Solution (x_L[i] <= x_k[i] <= x_U[i]): ");
for (int i = 0; i < solution.Length; i++)
    Console.WriteLine("    {0} <= {1} <= {2} ", Prob.x_L[i], solution[i], Prob.x_U[i]);

Console.WriteLine(" * Linear constraints (b_L[i] <= (A*x)[i] <= b_U[i]):");
double[] Ax = Prob.A.Evaluate(result.x_k);
for (int i = 0; i < Prob.mLin; i++)
    Console.WriteLine("    {0} <= {1} <= {2}", Prob.A.GetLowerBound(i),
        Ax[i], Prob.A.GetUpperBound(i));

Console.WriteLine(" * Nonlinear constraints (c_L[i] <= c(x)[i] <= c_U[i]):");
double[] cx = new double[Prob.mNonLin];
Prob.c(cx, result.x_k);
for (int i = 0; i < Prob.mNonLin; i++)
    Console.WriteLine("    {0} <= {1} <= {2}", Prob.c_L[i], cx[i], Prob.c_U[i]);
}
}
}

```

The constraints and an analytical Jacobian are defined in the example class.

```

c:   Nonlinear constraint vector
dc:  Nonlinear constraint gradient matrix
d2c: The second part of the Hessian to the Lagrangian
     function for the nonlinear constraints.

```

The executable is created by compiling *qpconQG.sln*. When running it from a command prompt the output results should be as in the following picture:

```
C:\WINDOWS\system32\cmd.exe
D:\source\v5\quickguide\qpconQG>qpconQG
* * * * *
* Solved qpconQG
* Printfile: SnoptqpconQP.txt
* Objective: -14.6758365568834
* Solution (x_L[i] <= x_k[i] <= x_U[i]):
  -Infinity <= -0.116378199197356 <= Infinity
  -Infinity <= -0.116378199197356 <= Infinity
  -Infinity <= -0.116378199197356 <= Infinity
  -Infinity <= -0.116378199197356 <= Infinity
  -Infinity <= -0.116378199197356 <= Infinity
  -Infinity <= -0.116378199197357 <= Infinity
  -Infinity <= -0.116378199197356 <= Infinity
  -Infinity <= -0.116378199197356 <= Infinity
  -Infinity <= -1.68613185822291 <= Infinity
  -Infinity <= 3.55896835220308 <= Infinity
* Linear constraints (b_L[i] <= (A*x)[i] <= b_U[i]):
  1 <= 1 <= 1
  1 <= 1 <= 1
  1 <= 1 <= 1
  1 <= 1 <= 1
  1 <= 1 <= 1
  1 <= 1 <= 1
  1 <= 1 <= 1
  1 <= 1 <= 1
  1 <= 1 <= 1
  1 <= 1 <= 1
* Nonlinear constraints (c_L[i] <= c(x)[i] <= c_U[i]):
  4 <= 4.00000000000658 <= 4
D:\source\v5\quickguide\qpconQG>
```

7 LLS Problem

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

$$\begin{aligned} \min_x \quad & f(x) = \frac{1}{2} \|Cx - d\| \\ \text{s/t} \quad & \begin{array}{l} x_L \leq x \leq x_U, \\ b_L \leq Ax \leq b_U \end{array} \end{aligned} \tag{9}$$

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

The following code defines and solves a problem in TOMNET.

```
using System;
using TOMNET;

namespace TOMNET
{
    /// <summary>
    /// Quick guide class for linear least squares problem.
    /// </summary>
    public class llsQG
    {
        static void Main(string[] args)
        {
            // For unbounded variables
            double Inf = double.PositiveInfinity;

            // Number of decision variables, linear constraints and observations
            const int n = 9;
            const int mLin = 3;
            const int k = 10;

            // Lower and upper bounds for the decision variables
            double[] x_L = new double[n] { -2, -2, -Inf, -2, -2, -2, -2, -2, -2 };
            double[] x_U = new double[n] { 2, 2, 2, 2, 2, 2, 2, 2, 2 };

            // Matrix defining linear constraints
            double[] A = new double[n * mLin]
            {1, 1, 1, 1, 1, 1, 1, 1, 4,
            1, 2, 3, 4, -2, 1, 1, 1, 1,
            1, -1, 1, -1, 1, 1, 1, 1, 1};

            // Lower and upper bounds on the linear constraints
            double[] b_L = new double[mLin] { 2, -Inf, -4 };
            double[] b_U = new double[mLin] { Inf, -2, -2 };

            // Vector k x 1 with observations in objective ||Cx - d||
        }
    }
}
```

```

double[] d = new double[k] { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 };

// Matrix m x n in objective ||Cx - d||
double[] C = new double[90]
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 2, 1, 1, 1, 1, 2, 0, 0,
 1, 1, 3, 1, 1, 1, -1, -1, -3,
 1, 1, 1, 4, 1, 1, 1, 1, 1,
 1, 1, 1, 3, 1, 1, 1, 1, 1,
 1, 1, 2, 1, 1, 0, 0, 0, -1,
 1, 1, 1, 1, 0, 1, 1, 1, 1,
 1, 1, 1, 0, 1, 1, 1, 1, 1,
 1, 1, 0, 1, 1, 1, 2, 2, 3,
 1, 0, 1, 1, 1, 1, 0, 2, 2};

// Starting point.
double[] x_0 = new double[n];
for (int i = 0; i < n; i++)
    x_0[i] = 1.0 / i;

// f_opt and x_opt estimated optimal values
double f_opt = 0.1390587318;
double[] x_opt = new double[n] {2.0, 1.57195927, -1.44540327, -0.03700275,
    0.54668583, 0.17512363, -1.65670447, -0.39474418, 0.31002899};

// Creating the problem, solver and result
LinearLeastSquaresProblem Prob = new LinearLeastSquaresProblem(n, mLin,
    k, C, d, A, b_L, b_U, x_0, x_L, x_U);
SNOPT solver = new SNOPT();
Result result;

// Setting a print file for SNOPT
string printfilename = "SnoptllsQG.txt";
solver.Options.PrintFile = printfilename;

// Calling Solve to get a Result object
solver.Solve(Prob, out result);

// Extract solution and objective from result
double[] solution = result.x_k;
double objective = result.f_k;

// Print information to the screen
Console.WriteLine(" * * * * *");
Console.WriteLine(" * Solved " + Prob.Name);
Console.WriteLine(" * Printfile: " + printfilename);
Console.WriteLine(" * Objective: {0} (estimated value {1})", objective, f_opt);

```

```

Console.WriteLine(" * Solution (x_L[i] <= x_k[i] <= x_U[i]): ");
for (int i = 0; i < solution.Length; i++)
    Console.WriteLine("    {0} <= {1} <= {2} ", Prob.x_L[i], solution[i], Prob.x_U[i]);

// Get and print |x_k - x_opt|
double dist = 0.0;
for (int i = 0; i < n; i++)
    dist += Math.Pow(solution[i] - x_opt[i], 2);
dist = Math.Sqrt(dist);
Console.WriteLine(" * |x_k - x_opt| = {0}", dist);

// Print linear constraints
Console.WriteLine(" * Linear constraints (b_L[i] <= (A*x)[i] <= b_U[i]):");
double[] Ax = Prob.A.Evaluate(result.x_k);
for (int i = 0; i < Prob.mLin; i++)
    Console.WriteLine("    {0} <= {1} <= {2}", Prob.A.GetLowerBound(i),
        Ax[i], Prob.A.GetUpperBound(i));
}
}
}

```

The following picture shows the correct output results:

```

C:\WINDOWS\system32\cmd.exe
D:\source\v5\quickguide\llsQG>llsQG
* * * * *
* Solved
* Printfile: SnoptllsQG.txt
* Objective: 0.139058731821439 (estimated value 0.1390587318)
* Solution (x_L[i] <= x_k[i] <= x_U[i]):
-2 <= 2 <= 2
-2 <= 1.59192152363756 <= 2
-Infinity <= -1.45538444693436 <= 2
-2 <= -0.0370027380775076 <= 2
-2 <= 0.546685815462397 <= 2
-2 <= 0.175123769212785 <= 2
-2 <= -1.67167611913228 <= 2
-2 <= -0.389783763034111 <= 2
-2 <= 0.310028989716378 <= 2
* |x_k - x_opt| = 0.0273289494404865
* Linear constraints (b_L[i] <= (A*x)[i] <= b_U[i]):
2 <= 2 <= Infinity
-Infinity <= -2 <= -2
-4 <= -2.03992454026924 <= -2
D:\source\v5\quickguide\llsQG>

```

8 NLLS Problem

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

$$\begin{aligned} \min_x \quad & f(x) = \frac{1}{2}r(x)^T r(x) \\ \text{s/t} \quad & \begin{aligned} x_L &\leq x \leq x_U, \\ b_L &\leq Ax \leq b_U \\ c_L &\leq c(x) \leq c_U \end{aligned} \end{aligned} \tag{10}$$

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 $c_L, c(x), c_U \in \mathbb{R}^{m_2}$. The following file defines and solves a problem in TOMNET.

The following code defines a problem in TOMNET.

```
using System;
using TOMNET;

namespace TOMNET
{
    /// <summary>
    /// Quick guide class for nonlinear least squares problem.
    /// </summary>
    static public class nllsQG
    {
        /// <summary>
        /// Class for the residual function for nllsQG.
        /// </summary>
        public class nllsQG_r : IDFunction
        {
            private double[] y;
            private double[] t;
            private double uP;

            bool hasYT = false;

            /// <summary>
            /// Constructor
            /// </summary>
            /// <param name="uP">Parameter passed to function from problem.</param>
            public nllsQG_r(double uP)
            {
                this.uP = uP;
            }

            /// <summary>
            /// Sets information for problem.
            /// </summary>

```

```

/// <param name="Prob"></param>
public void initFunc(NonlinearLeastSquaresProblem Prob)
{
    this.y = Prob.y;
    this.t = Prob.t;
    hasYT = true;
}

/// <summary>
/// Evaluates the residual.
/// </summary>
/// <param name="r"></param>
/// <param name="x"></param>
public void Evaluate(double[] r, double[] x)
{
    if (!hasYT)
        throw new Exception("Function not initiated yet");

    for (int i = 0; i < y.Length; i++)
        r[i] = this.uP * x[0] * (Math.Exp(-x[1] * t[i]) -
            Math.Exp(-x[0] * t[i])) / (x[2] * (x[0] - x[1]));
}

/// <summary>
/// Evaluates the gradient of the residual Jacobain.
/// </summary>
/// <param name="gx"></param>
/// <param name="x"></param>
public void Grad(double[] gx, double[] x)
{
    if (gx.Length != y.Length * x.Length)
        throw new TOMNETErrors.MismatchException("gx has illegal length " +
            gx.Length.ToString() + " (should have " +
            (y.Length * x.Length).ToString() + ")");

    double a = this.uP * x[0] / (x[2] * (x[0] - x[1]));
    double b = x[0] - x[1];

    double e1, e2;

    for (int i = 0; i < y.Length; i++)
    {
        e1 = Math.Exp(-x[0] * t[i]);
        e2 = Math.Exp(-x[1] * t[i]);
        gx[i*3+0] = a*(t[i]*e1+(e2-e1)*(1-1/b));
        gx[i*3+1] = a*(-t[i]*e2+(e2-e1)/b);
        gx[i*3+2] = -a * (e2 - e1) / x[2];
    }
}

```

```

    }
  }
}

/// <summary>
/// Jacobian matrix class for nllsQG.
/// </summary>
public class nllsQG_J : IDConstraints
{
    public void Evaluate(double[] cx, double[] f)
    {
        cx = this.Evaluate(f);
    }

    public double[] Evaluate(double[] f)
    {
        double[] blank = new double[28];
        blank.Initialize();
        return blank;
    }

    public void dc(Jacobian Jac, double[] x)
    { }
}

static void Main()
{
    // The observation times
    double[] t = new double[28] { 0.25, 0.5, 0.75, 1, 1.5, 2, 3, 4, 6, 8,
        12, 24, 32, 48, 54, 72, 80, 96, 121, 144,
        168, 192, 216, 246, 276, 324, 348, 386 };
    // The observations
    double[] y = new double[28] {30.5, 44, 43, 41.5, 38.6, 38.6, 39, 41,
        37, 37, 24, 32, 29, 23, 21, 19, 17, 14,
        9.5, 8.5, 7, 6, 6, 4.5, 3.6, 3, 2.2, 1.6};

    // Starting point, lower and upper bounds for decision variables
    double[] x_0 = new double[3] {6.8729, 0.0108, 0.1248};
    double[] x_L = ArrayUtils.getx_L(3);
    double[] x_U = ArrayUtils.getx_U(3);

    // Nonlinear constraint bounds
    double[] c_L = new double[0];
    double[] c_U = new double[0];

    // Linear constraints, lower and upper bounds
    double[] A = new double[3] { 1, 0, 0 };

```

```

double[] b_L = new double[1] { double.NegativeInfinity };
double[] b_U = new double[1] { double.PositiveInfinity };

// The residuals and Jacobian
nllsQG_r R = new nllsQG_r(5);
nllsQG_J C = new nllsQG_J();

// Create an instance of the problem
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);
R.initFunc(Prob);

// Create a solver, print file and result instance
SNOPT solver = new SNOPT();
string printfilename = "SnoptNllsQP.txt";
solver.Options.PrintFile = printfilename;
Result result;

// Solve the problem
solver.Solve(Prob, out result);

// Get the solution results
double[] solution = result.x_k;
double objective = result.f_k;

// Display the solution on the screen
Console.WriteLine(" * * * * *");
Console.WriteLine(" * Solved " + Prob.Name);
Console.WriteLine(" * Printfile: " + printfilename);
Console.WriteLine(" * Objective: {0}", objective);
Console.WriteLine(" * Solution (x_L[i] <= x_k[i] <= x_U[i]): ");
for (int i = 0; i < solution.Length; i++)
    Console.WriteLine("    {0} <= {1} <= {2} ", Prob.x_L[i], solution[i], Prob.x_U[i]);
Console.WriteLine(" * * * * *");
}
}
}

```

There are two callback function defined in the code:

```

r: Residual function
J: Residual gradient matrix (Jacobian)

```

It is also possible to define nonlinear constraints as needed.

The results should match the following command prompt outputs:

```
C:\WINDOWS\system32\cmd.exe

D:\source\v5\quickguide\nllsQG>nllsQG
* * * * *
* Solved NllsProb
* Printfile: SnoptNllsQP.txt
* Objective: 103.648825264008
* Solution (x_L[i] <= x_k[i] <= x_U[i]):
  -Infinity <= 6.87283349768409 <= Infinity
  -Infinity <= 0.0108118644226535 <= Infinity
  -Infinity <= 0.124842838132209 <= Infinity
* * * * *

D:\source\v5\quickguide\nllsQG>
```

9 GLB Problem

The **unconstrained global optimization (glb)** problem is defined as

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s/t} \quad & -\infty < x_L \leq x \leq x_U < \infty \end{aligned} \tag{11}$$

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

The following file (maintained from the solution file: `quickguide/glbQG/glbQG.sln`) illustrates how to solve an unconstrained global optimization problem in TOMNET.

```
using System;
using TOMNET;

namespace TOMNET
{
    /// <summary>
    /// Quick guide class for unconstrained global optimization (glb).
    /// </summary>
    public static class glbQG
    {
        /// <summary>
        /// Computes function value for Shekel 5.
        /// </summary>
        public class glbQG_f : IFunction
        {
            // Constants
            const int ACols = 4;
            const int ARows = 5;
            double[] c = new double[ARows] { 0.1, 0.2, 0.2, 0.4, 0.4 };
            double[] A = new double[ACols * ARows] { 4.0, 4.0, 4.0, 4.0, 1.0, 1.0,
                1.0, 1.0, 8.0, 8.0, 8.0, 8.0, 6.0, 6.0, 6.0, 6.0, 3.0, 7.0, 3.0, 7.0 };

            // The objective function
            public void Evaluate(double[] f, double[] x)
            {
                f[0] = 0.0;
                double xai = 0.0;
                for (int i = 0; i < ARows; i++)
                {
                    xai = 0.0;
                    for (int j = 0; j < ACols; j++)
                        xai += Math.Pow(x[j] - A[i * ACols + j], 2);

                    f[0] -= 1.0 / (xai + c[i]);
                }
            }
        }
    }
}
```

```

    }
}

/// <summary>
/// Creates and returns a glbProblem.
/// </summary>
/// <returns>glbProblem.</returns>
public static NonlinearProblem getGlbQG()
{
    // Number of decision variables
    const int N = 4;

    string Name = "glbQG";

    // Starting point, lower and upper bounds for decision variables
    double[] x_L = new double[N] { 0, 0, 0, 0};
    double[] x_U = new double[N] { 10, 10, 10, 10};
    double[] x_0 = new double[N] { 1, 2, 4, 8};

    // Optimum information
    double[] f_opt = new double[1] {-10.1531996790582};
    double f_Low = -20.0;

    // Objective function instance
    glbQG_f F = new glbQG_f();

    // Create an instance of the unconstrained problem
    return new NonlinearProblem(F, null, null, null, x_L, x_U,
        x_0, Name, f_Low, f_opt, null);
}

static void Main(string[] args)
{
    // Create an instance of the problem
    NonlinearProblem Prob = getGlbQG();

    // Create a solver and result instance
    SNOPT solver = new SNOPT();
    Result result;

    // Define a print file for SNOPT
    string printfilename = "SnoptGlbQG.txt";
    solver.Options.PrintFile = printfilename;

    // Solve the problem
    solver.Solve(Prob, out result);
}

```

```

// Get the solution results
double[] solution = result.x_k;
double objective = result.f_k;

// Display the solution on the screen for SNOPT
Console.WriteLine(" * * * * *");
Console.WriteLine(" * Solved a glb problem with SNOPT");
Console.WriteLine(" * Printfile: " + printfilename);
Console.WriteLine(" * Objective: {0}", objective);
Console.WriteLine(" * Solution (x_L[i] <= x_k[i] <= x_U[i]): ");
for (int i = 0; i < solution.Length; i++)
    Console.WriteLine("    {0} <= {1} <= {2} ", Prob.x_L[i], solution[i], Prob.x_U[i]);
Console.WriteLine();

// Solve the problem with glbDirect as well
GlbDirect solver2 = new GlbDirect();
printfilename = "GlbDirectGlbQG.txt";
solver2.Options.LogFile = printfilename;
solver2.Options.PrintLevel = 4;
solver2.Options.Iterprint = 10;
Result result2;

solver2.Solve(Prob, out result2);

solution = result2.x_k;
objective = result2.f_k;

// Display the solution on the screen for glbDirect
Console.WriteLine(" * * * * *");
Console.WriteLine(" * Solved a glb problem with glbDirect");
Console.WriteLine(" * Printfile: " + printfilename);
Console.WriteLine(" * Objective: {0}", objective);
Console.WriteLine(" * Solution (x_L[i] <= x_k[i] <= x_U[i]): ");
for (int i = 0; i < solution.Length; i++)
    Console.WriteLine("    {0} <= {1} <= {2} ", Prob.x_L[i], solution[i], Prob.x_U[i]);
}
}
}

```

Only the objective function can be defined for unconstrained (box-bounded) problems:

f: Function

The following results should be displayed after running the executable:

```
C:\WINDOWS\system32\cmd.exe

D:\source\v5\quickguide\glbQG>glbQG
* * * * *
* Solved a glb problem with SNOPT
* Printfile: SnoptGlbQG.txt
* Objective: -10.1531986791008
* Solution (x_L[i] <= x_k[i] <= x_U[i]):
  0 <= 3.99998716404544 <= 10
  0 <= 4.00008325152628 <= 10
  0 <= 3.99998714561694 <= 10
  0 <= 4.00008328849506 <= 10

* * * * *
* Solved a glb problem with glbDirect
* Printfile: GlbDirectGlbQG.txt
* Objective: -10.1531969488372
* Solution (x_L[i] <= x_k[i] <= x_U[i]):
  0 <= 4.00015241579028 <= 10
  0 <= 4.00015241579028 <= 10
  0 <= 4.00015241579028 <= 10
  0 <= 4.00015241579028 <= 10

D:\source\v5\quickguide\glbQG>
```

10 GLC Problem

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

$$\begin{aligned}
 & \min_x f(x) \\
 & s/t \quad -\infty < x_L \leq x \leq x_U < \infty \\
 & \quad \quad b_L \leq Ax \leq b_U \\
 & \quad \quad c_L \leq c(x) \leq c_U, \quad x_j \in \mathbb{N} \quad \forall j \in I,
 \end{aligned} \tag{12}$$

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 $1, \dots, n$, are restricted to be integers.

The following file defines a test problem in TOMNET.

```

using System;
using TOMNET;

namespace TOMNET
{
    /// <summary>
    /// Quick guide class for global mixed-integer nonlinear programming (glc).
    /// </summary>
    public static class glcQG
    {
        /// <summary>
        /// Computes function value for Hock-Schittkowski 59.
        /// </summary>
        public class glcQG_f : IFunction
        {
            // Constants
            public double[] u = new double[19]
            {75.196, 3.8112, 0.0020567, 1.0345E-5,
            6.8306, 0.030234, 1.28134E-3, 2.266E-7,
            0.25645, 0.0034604, 1.3514E-5, 28.106,
            5.2375E-6, 6.3E-8, 7E-10, 3.405E-4,
            1.6638E-6, 2.8673, 3.5256E-5};

            // Objective function
            public void Evaluate(double[] f, double[] x)
            {
                f[0] = -u[0] + u[1] * x[0] +
                u[2] * Math.Pow(x[0], 3) - u[3] * Math.Pow(x[0], 4) +
                u[4] * x[1] - u[5] * x[0] * x[1] +
                u[6] * x[1] * Math.Pow(x[0], 2) +
                u[7] * Math.Pow(x[0], 4) * x[1] -
                u[8] * Math.Pow(x[1], 2) + u[9] * Math.Pow(x[1], 3) -
                u[10] * Math.Pow(x[1], 4) + u[11] / (x[1] + 1) +
            }
        }
    }
}

```

```

        u[12] * Math.Pow(x[0], 2) * Math.Pow(x[1], 2) +
        u[13] * Math.Pow(x[0], 3) * Math.Pow(x[1], 2) -
        u[14] * Math.Pow(x[0], 3) * Math.Pow(x[1], 3) +
        u[17] * Math.Exp(0.0005 * x[0] * x[1]) -
        u[18] * Math.Pow(x[0], 3) * x[1] -
        u[15] * x[0] * Math.Pow(x[1], 2) +
        u[16] * x[0] * Math.Pow(x[1], 3) - 0.12694 * Math.Pow(x[0], 2);
    }
}

/// <summary>
/// Computes constraints for Hock-Schittkowski 59.
/// </summary>
public class glcQG_c : IDConstraints
{
    // Nonlinear constraint evaluation
    public void Evaluate(double[] cx, double[] x)
    {
        cx[0] = x[0] * x[1] - 700;
        cx[1] = x[1] - Math.Pow(x[0], 2) / 125;
        cx[2] = Math.Pow(x[1] - 50, 2) - 5 * (x[0] - 55);
    }

    public double[] Evaluate(double[] x)
    {
        double[] cx = new double[3];
        this.Evaluate(cx, x);
        return cx;
    }

    // Constraint Jacobian evaluation
    public void dc(Jacobian Jac, double[] x)
    {
        Jac[0,0] = x[1];
        Jac[0,1] = x[0];

        Jac[1,0] = 1;
        Jac[1,1] = x[0] / 62.5;

        Jac[2,0] = -5;
        Jac[2,1] = 2 * x[1] - 100;
    }
}

/// <summary>
/// Creates and returns a glcProblem.
/// </summary>

```

```

/// <returns>glcProblem</returns>
public static ConProblem getglcQG()
{
    const int N = 2;          // Number of decision variables
    const int mNonLin = 3; // Number of nonlinear constraints

    double inf = double.PositiveInfinity;

    string Name = "glcQG";

    // Starting point, lower and upper bounds for decision variables
    double[] x_0 = new double[N] { 10, 50 };
    double[] x_L = new double[2] { 0, 0 };
    double[] x_U = new double[2] { 75, 65};

    // Nonlinear constraint bounds
    double[] c_L = new double[mNonLin] {0, 0, 0};
    double[] c_U = new double[mNonLin] {inf, inf, inf};

    // Optimum information
    double[] x_opt = new double[N] { 13.55010424, 51.66018129 };
    double[] f_opt = new double[1] { -7.804226324 };
    double f_Low = -20.0;

    // Instances of objective function and nonlinear constraints
    glcQG_f F = new glcQG_f();
    glcQG_c C = new glcQG_c();

    // Creating an instance of the problem
    return new ConProblem(F,C,c_L,c_U,x_L,x_U,x_0,Name,f_Low,f_opt,x_opt);
}

static void Main()
{
    // Create an instance of the problem
    ConProblem Prob = getglcQG();

    // Create a solver and result instance
    SNOPT solver = new SNOPT();
    Result result;

    // Define a print file for SNOPT
    string printfilename = "SnoptGlcQG.txt";
    solver.Options.PrintFile = printfilename;

    // Solve the problem
    solver.Solve(Prob, out result);
}

```

```

// Get the solution results
double[] solution = result.x_k;
double objective = result.f_k;

// Display the solution on the screen
Console.WriteLine(" * * * * *");
Console.WriteLine(" * Solved a glcProblem");
Console.WriteLine(" * Printfile: " + printfilename);
Console.WriteLine(" * Objective: {0}", objective);
Console.WriteLine(" * Solution (x_L[i] <= x_k[i] <= x_U[i]): ");
for (int i = 0; i < solution.Length; i++)
    Console.WriteLine("    {0} <= {1} <= {2} ", Prob.x_L[i], solution[i], Prob.x_U[i]);

// Get the solution results for nonlinear constraints
double[] cx = new double[Prob.mNonLin];
Prob.c(cx, result.x_k);

// Display the nonlinear constraints on the screen
Console.WriteLine(" * Constraints (c_L[i] <= cx[i] <= c_U[i]): ");
for (int i = 0; i < Prob.mNonLin; i++)
    Console.WriteLine("    {0} <= {1} <= {2} ", Prob.c_L[i], cx[i], Prob.c_U[i]);
}
}
}

```

Only the objective function and nonlinear constraints are given for this problem type.

```

f:  Function
c:  Constraints

```

The following picture illustrates the output displayed:

```
C:\WINDOWS\system32\cmd.exe

D:\source\v5\quickguide\glcQG>glcQG
* * * * *
* Solved a glcProblem
* Printfile: SnoptGlcQG.txt
* Objective: -7.80278947345994
* Solution (x_L[i] <= x_k[i] <= x_U[i]):
  0 <= 13.550148553115 <= 75
  0 <= 51.6599502274477 <= 65
* Constraints (c_L[i] <= cx[i] <= c_U[i]):
  0 <= -1.71554802363971E-07 <= Infinity
  0 <= 50.1910980209558 <= Infinity
  0 <= 210.004691992029 <= Infinity

D:\source\v5\quickguide\glcQG>
```

11 Important Information

Setting patterns is especially important for large-scale problems as memory needs to be managed more properly (a dense problem is normally assumed otherwise).

11.1 Using Patterns

For most problems it is critical to set the proper problem patterns (e.g. *ConsPattern*) for memory allocation purposes and to speed up numerical differentiation. If analytical derivatives are given only the memory benefit will be seen.

11.2 Solver Timings

Comparing different solvers with solution times under 5-10 seconds may yield incorrect results. When running a problem and a solver for the first time general overhead and loading of dll's consume the majority of the time. It is recommended to run several times to get a good average runtime.