USER'S GUIDE FOR TOMLAB 7[1]

Kenneth Holmström[2], Anders O. Göran[3] and Marcus M. Edvall[4]

May 5, 2010

-TOMLAB NOW INCLUDES THE MODELING ENGINE, TomSym [See section 4.3]!

[1]More information available at the TOMLAB home page: http://tomopt.com/ and at the Applied Optimization and Modeling **TOM** home page http://www.ima.mdh.se/tom. E-mail: **tomlab@tomopt.com**.

[2]Professor in Optimization, Mälardalen University, Department of Mathematics and Physics, P.O. Box 883, SE-721 23 Västerås, Sweden, kenneth.holmstrom@mdh.se.

[3]Tomlab Optimization AB, Västerås Technology Park, Trefasgatan 4, SE-721 30 Västerås, Sweden, anders@tomopt.com.

[4]Tomlab Optimization Inc., 1260 SE Bishop Blvd Ste E, Pullman, WA 99163, USA, medvall@tomopt.com.

Contents

1 Introduction

1.1 What is TOMLAB?

TOMLAB is a general purpose development, modeling and optimal control environment in Matlab for research, teaching and practical solution of optimization problems.

TOMLAB 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 make some new interface routine to make everything work. Therefore the first obvious and basic design principle in TOMLAB is: *Define your problem once, run all available solvers.* The system takes care of all interface problems, whether between languages or due to different demands on the problem specification.

In the process of optimization one sometimes wants to graphically view the problem and the solution process, especially for ill-conditioned nonlinear problems. Sometimes it is not clear what solver is best for the particular type of problem and tests on different solvers can be of use. In teaching one wants to view the details of the algorithms and look more carefully at the different algorithmic steps. When developing new algorithms tests on thousands of problems are necessary to fully access the pros and cons of the new algorithm. One might want to solve a practical problem very many times, with slightly different conditions for the run. Or solve a control problem looping in real-time and solving the optimization problem each time slot.

All these issues and many more are addressed with the TOMLAB optimization environment. TOMLAB gives easy access to a large set of standard test problems, optimization solvers and utilities.

1.2 The Organization of This Guide

Section 2 presents the general design of TOMLAB.

Section 3 contains strict mathematical definitions of the optimization problem types. All solver routines available in TOMLAB are described.

Section 4 describes the input format and modeling environment. The functionality of the modeling engine TomSym is discussed in 4.3 and also in appendix C.

Sections 5, 6, 7 and 8 contain examples on the process of defining problems and solving them. All test examples are available as part of the TOMLAB distribution.

Section 9 shows how to setup and define multi layer optimization problems in TOMLAB.

Section 11 contains detailed descriptions of many of the functions in TOMLAB. The TOM solvers, originally developed by the Applied Optimization and Modeling (TOM) group, are described together with TOMLAB driver routine and utility functions. Other solvers, like the Stanford Optimization Laboratory (SOL) solvers are not described, but documentation is available for each solver.

Section 12 describes the utility functions that can be used, for example *tomRun* and *SolverList*.

Section 13 introduces the different options for derivatives, automatic differentiation.

Section 14 discusses a number of special system features such as partially separable functions and user supplied parameter information for the function computations.

Appendix A contains tables describing all elements defined in the problem structure. Some subfields are either empty, or filled with information if the particular type of optimization problem is defined. To be able to set different parameter options for the optimization solution, and change problem dependent information, the user should consult the tables in this Appendix.

Appendix B contains tables describing all elements defined in the output result structure returned from all solvers and driver routines.

Appendix D is concerned with the global variables used in TOMLAB and routines for handling important global variables enabling recursive calls of any depth.

Appendix E describes the available set of interfaces to other optimization software, such as CUTE, AMPL, and The Mathworks' Optimization Toolbox.

Appendix F gives some motivation for the development of TOMLAB.

1.3 Further Reading

TOMLAB has been discussed in several papers and at several conferences. The main paper on TOMLAB v1.0 is [42]. The use of TOMLAB for nonlinear programming and parameter estimation is presented in [45], and the use of linear and discrete optimization is discussed in [46]. Global optimization routines are also implemented, one is described in [8].

In all these papers TOMLAB was divided into two toolboxes, the NLPLIB TB and the OPERA TB. TOMLAB v2.0 was discussed in [43], [40]. and [41]. TOMLAB v4.0 and how to solve practical optimization problems with TOMLAB is discussed in [44].

The use of TOMLAB for costly global optimization with industrial applications is discussed in [9]; costly global optimization with financial applications in [37, 38, 39]. Applications of global optimization for robust control is presented in [25, 26]. The use of TOMLAB for exponential fitting and nonlinear parameter estimation are discussed in e.g. [49, 4, 22, 23, 47, 48].

The manuals for the add-on solver packages are also recommended reading material.

2 Overall Design

The scope of TOMLAB is large and broad, and therefore there is a need of a well-designed system. It is also natural to use the power of the Matlab language, to make the system flexible and easy to use and maintain. The concept of structure arrays is used and the ability in Matlab to execute Matlab code defined as string expressions and to execute functions specified by a string.

2.1 Structure Input and Output

Normally, when solving an optimization problem, a direct call to a solver is made with a long list of parameters in the call. The parameter list is solver-dependent and makes it difficult to make additions and changes to the system.

TOMLAB solves the problem in two steps. First the problem is defined and stored in a Matlab structure. Then the solver is called with a single argument, the problem structure. Solvers that were not originally developed for the TOMLAB environment needs the usual long list of parameters. This is handled by the driver routine *tomRun.m* which can call any available solver, hiding the details of the call from the user. The solver output is collected in a standardized result structure and returned to the user.

2.2 Introduction to Solver and Problem Types

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

The global variable *probType* contains the current type of optimization problem to be solved. An optimization solver is defined to be of type *solvType*, where *solvType* is any of the *probType* entries in Table 1. It is clear that a solver of a certain *solvType* is able to solve a problem defined to be of another type. For example, a constrained nonlinear programming solver should be able to solve unconstrained problems, linear and quadratic programs and constrained nonlinear least squares problems. In the graphical user interface and menu system an additional variable *optType* is defined to keep track of what type of problem is defined as the main subject. As an example, the user may select the type of optimization to be quadratic programming ($optType == 2$), then select a particular problem that is a linear programming problem ($probType == 8$) and then as the solver choose a constrained NLP solver like MINOS ($solvType == 3$).

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

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.

Table 1: The different types of optimization problems defined in TOMLAB, continued

probType		Type of optimization problem
minlp	12	Constrained mixed-integer nonlinear optimization.
lmi	13	Semi-definite programming with Linear Matrix Inequalities.
bmi	14	Semi-definite programming with Bilinear Matrix Inequalities.
exp	15	Exponential fitting problems.
nts	16	Nonlinear Time Series.
lcp	22	Linear Mixed-Complimentary Problems.
mcp	23	Nonlinear Mixed-Complimentary Problems.

Please note that since the actual numbers used for *probType* may change in future releases, it is recommended to use the text abbreviations. See help for *checkType* for further information.

Define *probSet* to be a set of defined optimization problems belonging to a certain class of problems of type *probType*. Each *probSet* is physically stored in one file, an *Init File*. In Table 2 the currently defined problem sets are listed, and new *probSet* sets are easily added.

Table 2: Defined test problem sets in TOMLAB. **probSets** marked with * are not part of the standard distribution

probSet	**probType**	**Description of test problem set**
uc	1	Unconstrained test problems.
qp	2	Quadratic programming test problems.
con	3	Constrained test problems.
ls	4	Nonlinear least squares test problems.
lls	5	Linear least squares problems.
cls	6	Linear constrained nonlinear least squares problems.
mip	7	Mixed-integer programming problems.
lp	8	Linear programming problems.
glb	9	Box-bounded global optimization test problems.
glc	10	Global MINLP test problems.
miqp	11	Constrained mixed-integer quadratic problems.
minlp	12	Constrained mixed-integer nonlinear problems.
lmi	13	Semi-definite programming with Linear Matrix Inequalities.
bmi	14	Semi-definite programming with Bilinear Matrix Inequalities.
exp	15	Exponential fitting problems.
nts	16	Nonlinear time series problems.
lcp	22	Linear mixed-complimentary problems.
mcp	23	Nonlinear mixed-complimentary problems.
mgh	4	More, Garbow, Hillstrom nonlinear least squares problems.
chs	3	Hock-Schittkowski constrained test problems.
uhs	1	Hock-Schittkowski unconstrained test problems.

The names of the predefined Init Files that do the problem setup, and the corresponding low level routines are constructed as two parts. The first part being the abbreviation of the relevant *probSet*, see Table 2, and the

second part denotes the computed task, shown in Table 3. The user normally does not have to define the more complicated functions ⋄_*d2c* and ⋄_*d2r*. It is recommended to supply this information when using solvers which can utilize second order information, such as TOMLAB /KNITRO and TOMLAB /CONOPT.

Table 3: Names for predefined Init Files and low level m-files in TOMLAB.

Generic name	**Purpose** (⋄ is any *probSet*, e.g. ⋄=**con**)
⋄_prob	Init File that either defines a string matrix with problem names or a structure *prob* for the selected problem.
⋄_f	Compute objective function $f(x)$.
⋄_g	Compute the gradient vector $g(x)$.
⋄_H	Compute the Hessian matrix $H(x)$.
⋄_c	Compute the vector of constraint functions $c(x)$.
⋄_dc	Compute the matrix of constraint normals, $\partial c(x)/dx$.
⋄_d2c	Compute the 2nd part of 2nd derivative matrix of the Lagrangian function, $\sum_i \lambda_i \partial^2 c_i(x)/dx^2$.
⋄_r	Compute the residual vector $r(x)$.
⋄_J	Compute the Jacobian matrix $J(x)$.
⋄_d2r	Compute the 2nd part of the Hessian matrix, $\sum_i r_i(x) \partial^2 r_i(x)/dx^2$

The Init File has two modes of operation; either return a string matrix with the names of the problems in the *probSet* or a structure with all information about the selected problem. All fields in the structure, named *Prob*, are presented in tables in Section A. Using a structure makes it easy to add new items without too many changes in the rest of the system. For further discussion about the definition of optimization problems in TOMLAB, see Section 4.

There are default values for everything that is possible to set defaults for, and all routines are written in a way that makes it possible for the user to just set an input argument empty and get the default.

2.3 The Process of Solving Optimization Problems

A flow-chart of the process of optimization in TOMLAB is shown in Figure 1. It is inefficient to use an interactive system. This is symbolized with the *Standard User* box in the figure, which directly runs the *Optimization Driver*, *tomRun*. The direct solver call is possible for all TOMLAB solvers, if the user has executed *ProbCheck* prior to the call. See Section 3 for a list of the TOMLAB solvers.

Depending on the type of problem, the user needs to supply the *low-level* routines that calculate the objective function, constraint functions for constrained problems, and also if possible, derivatives. To simplify this coding process so that the work has to be performed only once, TOMLAB provides *gateway* routines that ensure that any solver can obtain the values in the correct format.

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 $f(x) = \frac{1}{2} r^T(x) r(x)$. Such issues are automatically handled by the gateway functions.

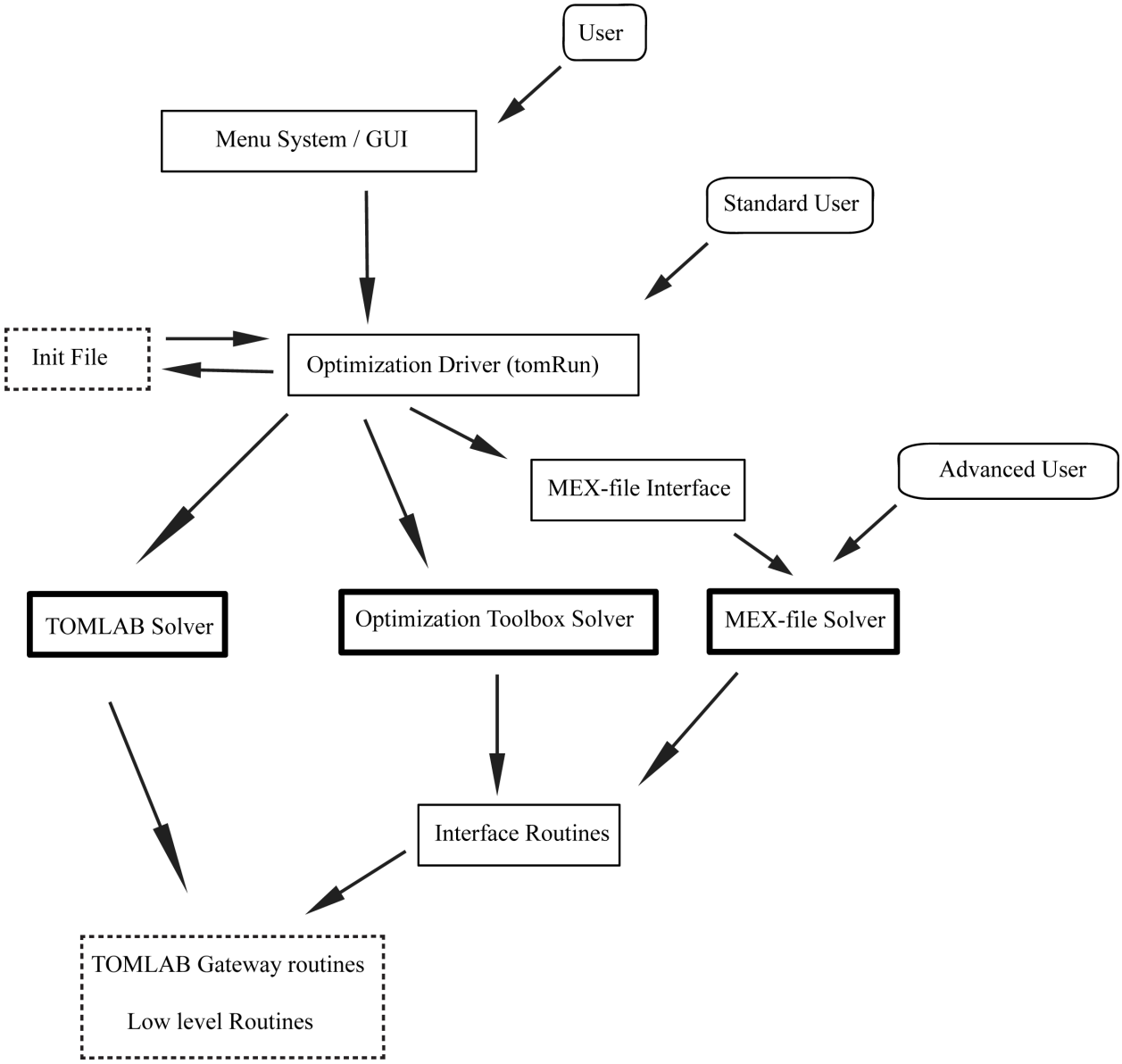


Figure 1: The process of optimization in TOMLAB.

2.4 Low Level Routines and Gateway Routines

Low level routines are the routines that compute:

- The objective function value
- The gradient vector
- The Hessian matrix (second derivative matrix)
- The residual vector (for nonlinear least squares problems)
- The Jacobian matrix (for nonlinear least squares problems)
- The vector of constraint functions
- The matrix of constraint normals (the constraint Jacobian)
- The second part of the second derivative of the Lagrangian function.

The last three routines are only needed for constrained problems.

The names of these routines are defined in the structure fields *Prob.FUNCS.f*, *Prob.FUNCS.g*, *Prob.FUNCS.H* etc. It is the task for the *Assign* routine to set the names of the low level m-files. This is done by a call to the routine *conAssign* with the names as arguments for example. There are *Assign* routines for all problem types handled by TOMLAB. As an example, see 'help conAssign' in MATLAB.

```
Prob = conAssign('f', 'g', 'H', HessPattern, x_L, x_U, Name,x_0,...
       pSepFunc, fLowBnd, A, b_L, b_U, 'c', 'dc', 'd2c', ConsPattern,...
       c_L, c_U, x_min, x_max, f_opt, x_opt);
```

Only the low level routines relevant for a certain type of optimization problem need to be coded. There are dummy routines for the others. Numerical differentiation is automatically used for gradient, Jacobian and constraint gradient if the corresponding user routine is non present or left out when calling *conAssign*. However, the solver always needs more time to estimate the derivatives compared to if the user supplies a code for them. Also the numerical accuracy is lower for estimated derivatives, so it is recommended that the user always tries to code the derivatives, if it is possible. Another option is automatic differentiation with TOMLAB /MAD.

TOMLAB uses gateway routines (*nlp_f*, *nlp_g*, *nlp_H*, *nlp_c*, *nlp_dc*, *nlp_d2c*, *nlp_r*, *nlp_J*, *nlp_d2r*). These routines extract the search directions and line search steps, count iterations, handle separable functions, keep track of the kind of differentiation wanted etc. They also handle the separable NLLS case and NLLS weighting. By the use of global variables, unnecessary evaluations of the user supplied routines are avoided.

To get a picture of how the low-level routines are used in the system, consider Figure 2 and 3. Figure 2 illustrates the chain of calls when computing the objective function value in *ucSolve* for a nonlinear least squares problem defined in *mgh_prob*, *mgh_r* and *mgh_J*. Figure 3 illustrates the chain of calls when computing the numerical approximation of the gradient (by use of the routine *fdng*) in *ucSolve* for an unconstrained problem defined in *uc_prob* and *uc_f*.

Information about a problem is stored in the structure variable *Prob*, described in detail in the tables in Appendix A. This variable is an argument to all low level routines. In the field element *Prob.user*, problem specific information

```
ucSolve <==> nlp_f <==> ls_f  <==> nlp_r  <==> mgh_r
```

Figure 2: The chain of calls when computing the objective function value in *ucSolve* for a nonlinear least squares problem defined in *mgh_prob*, *mgh_r* and *mgh_J*.

```
ucSolve <==> nlp_g <==> fdng  <==> nlp_r  <==> uc_f
```

Figure 3: The chain of calls when computing the numerical approximation of the gradient in *ucSolve* for an unconstrained problem defined in *uc_prob* and *uc_f*.

needed to evaluate the low level routines are stored. This field is most often used if problem related questions are asked when generating the problem. It is often the case that the user wants to supply the low-level routines with additional information besides the variables x that are optimized. Any unused fields could be defined in the structure *Prob* for that purpose. To avoid potential conflicts with future TOMLAB releases, it is recommended to use subfields of *Prob.user*. It the user wants to send some variables a, B and C, then, after creating the *Prob* structure, these extra variables are added to the structure:

```
Prob.user.a=a;
Prob.user.B=B;
Prob.user.C=C;
```

Then, because the *Prob* structure is sent to all low-level routines, in any of these routines the variables are picked out from the structure:

```
a = Prob.user.a;
B = Prob.user.B;
C = Prob.user.C;
```

A more detailed description of how to define new problems is given in sections 5, 6 and 8. The usage of *Prob.user* is described in Section 14.2.

Different solvers all have different demand on how information should be supplied, i.e. the function to optimize, the gradient vector, the Hessian matrix etc. To be able to code the problem only once, and then use this formulation to run all types of solvers, interface routines that returns information in the format needed for the relevant solver were developed.

Table 4 describes the low level test functions and the corresponding Init File routine needed for the predefined constrained optimization (**con**) problems. For the predefined unconstrained optimization (**uc**) problems, the global optimization (**glb, glc**) problems and the quadratic programming problems (**qp**) similar routines have been defined.

To conclude, the system design is flexible and easy to expand in many different ways.

Table 4: Generally constrained nonlinear (**con**) test problems.

Function	**Description**
con_prob	Init File. Does the initialization of the **con** test problems.
con_f	Compute the objective function $f(x)$ for **con** test problems.
con_g	Compute the gradient $g(x)$ for **con** test problems. x
con_H	Compute the Hessian matrix $H(x)$ of $f(x)$ for **con** test problems.
con_c	Compute the constraint residuals $c(x)$ for **con** test problems.
con_dc	Compute the derivative of the constraint residuals for **con** test problems.
con_d2c	Compute the ^{2}nd part of ^{2}nd derivative matrix of the Lagrangian function, $\sum_i \lambda_i \partial^2 c_i(x)/dx^2$ for **con** test problems.
con_fm	Compute merit function $\theta(x_k)$.
con_gm	Compute gradient of merit function $\theta(x_k)$.

3 Problem Types and Solver Routines

Section 3.1 defines all problem types in TOMLAB. Each problem definition is accompanied by brief suggestions on suitable solvers. This is followed in Section 3.2 by a complete list of the available solver routines in TOMLAB and the various available extensions, such as /SOL and /CGO.

3.1 Problem Types Defined in TOMLAB

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

$$\begin{aligned} \min_x \quad & f(x) \\ s/t \quad & x_L \leq x \leq x_U, \end{aligned} \tag{1}$$

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: **TOMLAB /KNITRO and TOMLAB /SNOPT**.

The TOMLAB Base Module routine *ucSolve* includes several of the most popular search step methods for unconstrained optimization. Bound constraints are treated as described in Gill et. al. [28]. The search step methods for unconstrained optimization included in *ucSolve* are: the Newton method, the quasi-Newton BFGS and DFP method, the Fletcher-Reeves and Polak-Ribiere conjugate-gradient method, and the Fletcher conjugate descent method. The quasi-Newton methods may either update the inverse Hessian (standard) or the Hessian itself. The Newton method and the quasi-Newton methods updating the Hessian use a subspace minimization technique to handle rank problems, see Lindström [53]. The quasi-Newton algorithms also use safe guarding techniques to avoid rank problem in the updated matrix.

Another TOMLAB Base Module solver suitable for unconstrained problems is the structural trust region algorithm *sTrustr*, combined with an initial trust region radius algorithm. The code is based on the algorithms in [13] and [67], and treats partially separable functions. Safeguarded BFGS or DFP are used for the quasi-Newton update, if the analytical Hessian is not used. The set of constrained nonlinear solvers could also be used for unconstrained problems.

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

$$\begin{aligned} \min_x \quad & f(x) = \tfrac{1}{2} x^T F x + c^T x \\ s/t \quad & \begin{aligned} x_L &\leq x \leq x_U, \\ b_L &\leq Ax \leq b_U \end{aligned} \end{aligned} \tag{2}$$

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: **TOMLAB /KNITRO, TOMLAB /SNOPT and TOMLAB /MINLP**.

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 [54], implemented in the TOM routine *qpSolve*. *qpSolve* explicitly treats both inequality and equality constraints, as well as lower and upper bounds on the variables (simple bounds). It converges to a local minimum for indefinite quadratic programs.

In TOMLAB *MINOS* in the general or the QP-version (*QP-MINOS*), or the dense QP solver *QPOPT* may be used. In the TOMLAB /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 TOMLAB /KNITRO or TOMLAB /BARNLP.

TOMLAB /CPLEX, TOMLAB /Xpress and TOMLAB /XA should always be considered for large-scale QP problems.

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

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

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: **TOMLAB /SNOPT, TOMLAB /NPSOL and TOMLAB /KNITRO**.

For general constrained nonlinear optimization a sequential quadratic programming (SQP) method by Schittkowski [69] is implemented in the TOMLAB Base Module solver *conSolve*. *conSolve* also includes an implementation of the Han-Powell SQP method. There are also a TOMLAB Base Module routine *nlpSolve* implementing the Filter SQP by Fletcher and Leyffer presented in [21].

Another constrained solver in TOMLAB is the structural trust region algorithm *sTrustr*, described above. Currently, *sTrustr* only solves problems where the feasible region, defined by the constraints, is convex. TOMLAB /MINOS solves constrained nonlinear programs. The TOMLAB /SOL extension gives an additional set of general solvers for dense or sparse problems.

sTrustr, *pdco* and *pdsco* in TOMLAB Base Module handle nonlinear problems with *linear* constraints only.

There are many other options for large-scale nonlinear optimization to consider in TOMLAB. TOMLAB /CONOPT, TOMLAB /BARNLP, TOMLAB /MINLP, TOMLAB /NLPQL and TOMLAB /SPRNLP.

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

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

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 solvers: **TOMLAB /LGO and TOMLAB /CGO with TOMLAB /SOL**.

The TOM solver *glbSolve* implements the DIRECT algorithm [14], which is a modification of the standard Lipschitzian approach that eliminates the need to specify a Lipschitz constant. In *glbSolve* no derivative information is used. For global optimization problems with expensive function evaluations the TOMLAB /CGO routine *ego* implements the Efficient Global Optimization (EGO) algorithm [16]. The idea of the EGO algorithm is to first fit a response surface to data collected by evaluating the objective function at a few points. Then, EGO balances between finding the minimum of the surface and improving the approximation by sampling where the prediction error may be high.

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

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

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: **TOMLAB /OQNLP**.

The TOMLAB Base Module solver *glcSolve* implements an extended version of the DIRECT algorithm [52], that handles problems with both nonlinear and integer constraints.

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

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

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: **TOMLAB /CPLEX, TOMLAB /Xpress and TOMLAB /XA**.

The TOMLAB Base Module solver *lpSimplex* implements a simplex algorithm for **lp** problems.

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

The recommended solvers normally outperforms all other solvers.

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

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

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 solvers: **TOMLAB /CPLEX, TOMLAB /Xpress and TOMLAB /XA**.

Mixed-integer programs can be solved using the TOMLAB Base Module routine *mipSolve* that implements a standard branch-and-bound algorithm, see Nemhauser and Wolsey [58, chap. 8]. When dual feasible solutions are available, mipSolve is using the TOMLAB dual simplex solver DualSolve to solve subproblems, which is significantly faster than using an ordinary linear programming solver, like the TOMLAB lpSimplex. *mipSolve* also implements user defined priorities for variable selection, and different tree search strategies. For 0/1 - knapsack problems a round-down primal heuristic is included. Another TOMLAB Base Module solver is the cutting plane

routine *cutplane*, using Gomory cuts. It is recommended to use *mipSolve* with the LP version of *MINOS* with warm starts for the subproblems, giving great speed improvement. The TOMLAB /Xpress extension gives access to the state-of-the-art LP, QP, MILP and MIQP solver Xpress-MP. For many MIP problems, it is necessary to use such a powerful solver, if the solution should be obtained in any reasonable time frame. TOMLAB /CPLEX is equally powerful as TOMLAB /Xpress and also includes a network solver.

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

$$\begin{aligned} \min_x \quad & f(x) = \tfrac{1}{2}||Cx - d|| \\ s/t \quad & \begin{array}{rcccl} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U \end{array} \end{aligned} \tag{8}$$

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: **TOMLAB /LSSOL**.

Tlsqr solves unconstrained sparse **lls** problems. *lsei* solves the general dense problems. *Tnnls* is a fast and robust replacement for the Matlab nnls. The general least squares solver *clsSolve* may also be used. In the TOMLAB /NPSOL or TOMLAB /SOL extension the *LSSOL* solver is suitable for dense linear least squares problems.

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

$$\begin{aligned} \min_x \quad & f(x) = \tfrac{1}{2} r(x)^T r(x) \\ s/t \quad & \begin{array}{rcccl} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U \\ c_L & \leq & c(x) & \leq & c_U \end{array} \end{aligned} \tag{9}$$

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: **TOMLAB /NLSSOL**.

The TOMLAB Base Module nonlinear least squares solver *clsSolve* treats problems with bound constraints in a similar way as the routine *ucSolve*. This strategy is combined with an active-set strategy to handle linear equality and inequality constraints [7].

clsSolve includes seven optimization methods for nonlinear least squares problems, among them: the Gauss-Newton method, the Al-Baali-Fletcher [2] and the Fletcher-Xu [19] hybrid method, and the Huschens TSSM method [50]. If rank problems occur, the solver uses subspace minimization. The line search algorithm used is the same as for unconstrained problems.

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

One important utility routine is the TOMLAB Base Module line search algorithm *LineSearch*, used by the solvers *conSolve*, *clsSolve* and *ucSolve*. It implements a modified version of an algorithm by Fletcher [20, chap. 2]. The line search algorithm uses quadratic and cubic interpolation, see Section 12.10.

Another TOMLAB Base Module routine, *preSolve*, is running a presolve analysis on a system of linear equalities, linear inequalities and simple bounds. An algorithm by Gondzio [36], somewhat modified, is implemented in *preSolve*. See [7] for a more detailed presentation.

The **linear semi-definite programming problem with linear matrix inequalities** (**sdp**) is defined as

$$
\begin{aligned}
\min_x \quad & f(x) = c^T x \\
s/t \quad & \begin{array}{rcccl} x_L & \leq & x & \leq & x_U \\ b_L & \leq & Ax & \leq & b_U \end{array} \\
& Q_0^i + \sum_{k=1}^{n} Q_k^i x_k \preccurlyeq 0, \qquad i = 1, \ldots, m.
\end{aligned} \tag{10}
$$

where $c, x, x_L, x_U \in \mathbb{R}^n$, $A \in \mathbb{R}^{m_l \times n}$, $b_L, b_U \in \mathbb{R}^{m_l}$ and Q_k^i are symmetric matrices of similar dimensions in each constraint i. If there are several LMI constraints, each may have it's own dimension.

Recommended solvers: **TOMLAB /PENSDP and TOMLAB /PENBMI**.

The **linear semi-definite programming problem with bilinear matrix inequalities** (**bmi**) is defined similarly to but with the matrix inequality

$$
Q_0^i + \sum_{k=1}^{n} Q_k^i x_k + \sum_{k=1}^{n} \sum_{l=1}^{n} x_k x_l K_{kl}^i \preccurlyeq 0 \tag{11}
$$

The MEX solvers *pensdp* and *penbmi* treat **sdp** and **bmi** problems, respectively. These are available in the TOMLAB /PENSDP and TOMLAB /PENBMI toolboxes.

The MEX-file solver *pensdp* available in the TOMLAB /PENSDP toolbox implements a penalty method aimed at large-scale dense and sparse **sdp** problems. The interface to the solver allows for data input in the sparse SDPA input format as well as a TOMLAB specific format corresponding to.

The MEX-file solver *penbmi* available in the TOMLAB /PENBMI toolbox is similar to *pensdp*, with added support for the bilinear matrix inequalities.

3.2 Solver Routines in TOMLAB

3.2.1 TOMLAB Base Module

TOMLAB includes a large set of optimization solvers. Most of them were originally developed by the Applied Optimization and Modeling group (TOM) [42]. Since then they have been improved e.g. to handle Matlab sparse arrays and been further developed. Table 5 lists the main set of TOM optimization solvers in all versions of TOMLAB.

Table 5: The optimization solvers in TOMLAB Base Module.

Function	Description	Section	Page
clsSolve	Constrained nonlinear least squares. Handles simple bounds and linear equality and inequality constraints using an active-set strategy. Implements Gauss-Newton, and hybrid quasi-Newton and Gauss-Newton methods.	11.1.1	86
conSolve	Constrained nonlinear minimization solver using two different sequential quadratic programming methods.	11.1.2	90
cutplane	Mixed-integer programming using a cutting plane algorithm.	11.1.3	94
DualSolve	Solves a linear program with a dual feasible starting point.	11.1.4	97
expSolve	Solves exponential fitting problems.	11.1.5	100
glbSolve	Box-bounded global optimization, using only function values.	11.1.7	106
glcCluster	Hybrid algorithm for constrained mixed-integer global optimization. Uses a combination of *glcFast* (DIRECT) and a clustering algorithm.	11.1.8	109
glcSolve	Global mixed-integer nonlinear programming, using no derivatives.	11.1.10	118
infSolve	Constrained minimax optimization. Reformulates problem and calls any suitable nonlinear solver.	11.1.12	125
lpSimplex	Linear programming using a simplex algorithm.	11.1.14	129
MILPSOLVE	Mixed-integer programming using branch and bound.	11.1.16	133
L1Solve	Constrained L1 optimization. Reformulates problem and calls any suitable nonlinear solver.	11.1.15	131
mipSolve	Mixed-integer programming using a branch-and-bound algorithm.	11.1.18	150
nlpSolve	Constrained nonlinear minimization solver using a filter SQP algorithm.	11.1.21	160
pdco	Linearly constrained nonlinear minimization solver using a primal-dual barrier algorithm.	11.1.22	163
pdsco	Linearly constrained nonlinear minimization solver using a primal-dual barrier algorithm, for separable functions.	11.1.23	166
qpSolve	Non-convex quadratic programming.	11.1.24	169
slsSolve	Sparse constrained nonlinear least squares. Reformulates problem and calls any suitable sparse nonlinear solver.	11.1.25	171
sTrustr	Constrained convex optimization of partially separable functions, using a structural trust region algorithm.	11.1.26	174

Table 5: The optimization solvers in TOMLAB Base Module, continued.

Field	Description		
ucSolve	Unconstrained optimization with simple bounds on the parameters. Implements Newton, quasi-Newton and conjugate-gradient methods.	11.1.29	180

Additional Fortran solvers in TOMLAB are listed in Table 6. They are called using a set of MEX-file interfaces developed in TOMLAB.

Table 6: Additional solvers in TOMLAB.

Function	Description	Reference	Page
goalSolve	Solves sparse multi-objective goal attainment problems		
lsei	Dense constrained linear least squares		
qld	Convex quadratic programming		
Tfzero	Finding a zero to f(x) in an interval, x is one-dimensional.	[70, 15]	
Tlsqr	Sparse linear least squares.	[60, 59, 68]	
Tnnls	Nonnegative constrained linear least squares		

3.2.2 TOMLAB /BARNLP

The add-on toolbox TOMLAB /BARNLP solves large-scale nonlinear programming problems using a sparse primal-dual interior point algorithm, in conjunction with a filter for globalization. The solver package was developed in co-operation with Boeing Phantom Works. The TOMLAB /BARNLP package is described in a separate manual available at http://tomopt.com.

3.2.3 TOMLAB /CGO

The add-on toolbox TOMLAB /CGO solves costly global optimization problems. The solvers are listed in Table 7. They are written in a combination of Matlab and Fortran code, where the Fortran code is called using a set of MEX-file interfaces developed in TOMLAB.

Table 7: Additional solvers in TOMLAB /CGO.

Function	Description	Reference
rbfSolve	Costly constrained box-bounded optimization using a RBF algorithm.	[9]
ego	Costly constrained box-bounded optimization using the Efficient Global Optimization (EGO) algorithm.	[16]

3.2.4 TOMLAB /CONOPT

The add-on toolbox TOMLAB /CONOPT solves large-scale nonlinear programming problems with a feasible path GRG method . The solver package was developed in co-operation with Arki Consulting and Development A/S. The TOMLAB /CONOPT is described in a separate manual available at http://tomopt.com. There is also m-file help available.

3.2.5 TOMLAB /CPLEX

The add-on toolbox TOMLAB /CPLEX solves large-scale mixed-integer linear and quadratic programming problems. The solver package was developed in co-operation with ILOG S.A. The TOMLAB /CPLEX solver package and interface are described in a manual available at http://tomopt.com.

3.2.6 TOMLAB /KNITRO

The add-on toolbox TOMLAB /KNITRO solves large-scale nonlinear programming problems with interior (barrier) point trust-region methods. The solver package was developed in co-operation with Ziena Optimization Inc. The TOMLAB /KNITRO manual is available at http://tomopt.com.

3.2.7 TOMLAB /LGO

The add-on toolbox TOMLAB /LGO solves global nonlinear programming problems. The LGO solver package is developed by Pintér Consulting Services, Inc. [63] The TOMLAB /LGO manual is available at http://tomopt.com.

3.2.8 TOMLAB /MINLP

The add-on toolbox TOMLAB /MINLP provides solvers for continuous and mixed-integer quadratic programming (**qp**,**miqp**) and continuous and mixed-integer nonlinear constrained optimization (**con**, **minlp**).

All four solvers, written by Roger Fletcher and Sven Leyffer, University of Dundee, Scotland, are available in sparse and dense versions. The solvers are listed in Table 8.

The TOMLAB /MINLP manual is available at http://tomopt.com.

3.2.9 TOMLAB /MINOS

Another set of Fortran solvers were developed by the Stanford Optimization Laboratory (SOL). Table 9 lists the solvers included in TOMLAB /MINOS. The solvers are called using a set of MEX-file interfaces developed as part of TOMLAB. All functionality of the SOL solvers are available and changeable in the TOMLAB framework in Matlab.

3.2.10 TOMLAB /OQNLP

The add-on toolbox TOMLAB /OQNLP uses a multistart heuristic algorithm to find global optima of smooth constrained nonlinear programs (NLPs) and mixed-integer nonlinear programs (MINLPs). The solver package

Table 8: Solver routines in TOMLAB /MINLP.

Function	Description	Reference
bqpd	Quadratic programming using a null-space method.	*bqpdTL.m*
miqpBB	Mixed-integer quadratic programming using *bqpd* as subsolver.	*miqpBBTL.m*
filterSQP	Constrained nonlinear minimization using a Filtered Sequential QP method.	*filterSQPTL.m*
minlpBB	Constrained, mixed-integer nonlinear minimization using a branch-and-bound search scheme. *filterSQP* is used as NLP solver.	*minlpBBTL.m*

Table 9: The SOL optimization solvers in TOMLAB /MINOS.

Function	Description	Reference	Page
MINOS 5.51	Sparse linear and nonlinear programming with linear and nonlinear constraints.	[57]	
LP-MINOS	A special version of the *MINOS 5.5* MEX-file interface for sparse linear programming.	[57]	
QP-MINOS	A special version of the *MINOS 5.5* MEX-file interface for sparse quadratic programming.	[57]	
LPOPT 1.0-10	Dense linear programming.	[30]	
QPOPT 1.0-10	Non-convex quadratic programming with dense constraint matrix and sparse or dense quadratic matrix.	[30]	

was developed in co-operation with Optimal Methods Inc. The TOMLAB /OQNLP manual is available at http://tomopt.com.

3.2.11 TOMLAB /NLPQL

The add-on toolbox TOMLAB /NLPQL solves dense nonlinear programming problems, multi criteria optimization problems and nonlinear fitting problems. The solver package was developed in co-operation with Klaus Schittkowski. The TOMLAB /NLPQL manual is available at http://tomopt.com.

3.2.12 TOMLAB /NPSOL

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

3.2.13 TOMLAB /PENBMI

The add-on toolbox TOMLAB /PENBMI solves linear semi-definite programming problems with linear and bilinear matrix inequalities. The solvers are listed in Table 10. They are written in a combination of Matlab and C code. The TOMLAB /PENBMI manual is available at http://tomopt.com.

Table 10: Additional solvers in TOMLAB /PENSDP.

Function	Description	Reference	Page
penbmi	Sparse and dense linear semi-definite programming using a penalty algorithm.		
penfeas_bmi	Feasibility check of systems of linear matrix inequalities, using *penbmi*.		

3.2.14 TOMLAB /PENSDP

The add-on toolbox TOMLAB /PENSDP solves linear semi-definite programming problems with linear matrix inequalities. The solvers are listed in Table 11. They are written in a combination of Matlab and C code. The TOMLAB /PENSDP manual is available at http://tomopt.com.

Table 11: Additional solvers in TOMLAB /PENSDP.

Function	Description	Reference	Page
pensdp	Sparse and dense linear semi-definite programming using a penalty algorithm.		
penfeas_sdp	Feasibility check of systems of linear matrix inequalities, using *pensdp*.		

3.2.15 TOMLAB /SNOPT

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

3.2.16 TOMLAB /SOL

The extension toolbox TOMLAB /SOL gives access to the complete set of Fortran solvers developed by the Stanford Systems Optimization Laboratory (SOL). These solvers are listed in Table 9 and 12.

3.2.17 TOMLAB /SPRNLP

The add-on toolbox TOMLAB /SPRNLP solves large-scale nonlinear programming problems. SPRNLP is a state-of-the-art sequential quadratic programming (SQP) method, using an augmented Lagrangian merit function and safeguarded line search. The solver package was developed in co-operation with Boeing Phantom Works. The TOMLAB /SPRNLP package is described in a separate manual available at http://tomopt.com.

Table 12: The optimization solvers in the TOMLAB /SOL toolbox.

Function	Description	Reference	Page
NPSOL 5.02	Dense linear and nonlinear programming with linear and nonlinear constraints.	[34]	
SNOPT 6.2-2	Large, sparse linear and nonlinear programming with linear and nonlinear constraints.	[33, 31]	
SQOPT 6.2-2	Sparse convex quadratic programming.	[32]	
NLSSOL 5.0-2	Constrained nonlinear least squares. NLSSOL is based on NPSOL. No reference except for general NPSOL reference.	[34]	
LSSOL 1.05-4	Dense linear and quadratic programs (convex), and constrained linear least squares problems.	[29]	

3.2.18 TOMLAB /XA

The add-on toolbox TOMLAB /XA solves large-scale linear, binary, integer and semi-continuous linear programming problems, as well as quadratic programming problems. The solver package was developed in co-operation with Sunset Software Technology. The TOMLAB /XA package is described in a separate manual available at http://tomopt.com.

3.2.19 TOMLAB /Xpress

The add-on toolbox TOMLAB /Xpress solves large-scale mixed-integer linear and quadratic programming problems. The solver package was developed in co-operation with Dash Optimization Ltd. The TOMLAB /Xpress solver package and interface are described in the html manual that comes with the installation package. There is also a TOMLAB /Xpress manual available at http://tomopt.com.

3.2.20 Finding Available Solvers

To get a list of all available solvers, including Fortran, C and Matlab Optimization Toolbox solvers, for a certain *solvType*, call the routine *SolverList* with *solvType* as argument. *solvType* should either be a string ('uc', 'con' etc.) or the corresponding *solvType* number as given in Table 1, page 11. For example, if wanting a list of all available solvers of *solvType* **con**, then

```
SolverList('con')
```

gives the output

```
>> SolverList('con');

Tomlab recommended choice for Constrained Nonlinear Programming (NLP)

npsol

Other solvers for NLP

   Licensed:

   nlpSolve
   conSolve
   sTrustr
   constr
   minos
   snopt
   fmincon
   filterSQP
   PDCO
   PDSCO


   Non-licensed:

   NONE

Solvers also handling NLP

   Licensed:

   glcSolve
   glcFast
   glcCluster
   rbfSolve
   minlpBB

   Non-licensed:
```

```
NONE
```

SolverList also returns two output arguments: all available solvers as a string matrix and a vector with the corresponding *solvType* for each solver.

Note that solvers for a more general problem type may be used to solve the problem. In Table 13 an attempt has been made to classify these relations.

Table 13: 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).

	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 TOMLAB

TOMLAB is based on the principle of creating a problem structure that defines the problem and includes all relevant information needed for the solution of the user problem. One unified format is defined, the TOMLAB format. The TOMLAB format gives the user a fast way to setup a problem structure and solve the problem from the Matlab command line using any suitable TOMLAB solver.

TOMLAB also includes a modeling engine (or advanced Matlab class), TomSym, see Section 4.3. The package uses Matlab objects and operator overloading to capture Matlab procedures, and generates source code for derivatives of any order.

In this section follows a more detailed description of the TOMLAB format.

4.1 The TOMLAB Format

The TOMLAB format is a quick way to setup a problem and easily solve it using any of the TOMLAB solvers. The principle is to put all information in a Matlab structure, which then is passed to the solver, which extracts the relevant information. The structure is passed to the user function routines for nonlinear problems, making it a convenient way to pass other types of information.

The solution process for the TOMLAB format has four steps:

1. Define the problem structure, often called Prob.
2. Call the solver or the universal driver routine *tomRun*.
3. Postprocessing, e.g. print the result of the optimization.

Step 1 could be done in several ways in TOMLAB. Recommended is to call one of the following routines dependent on the type of optimization problem, see Table 14.

Step 2, the solver call, is either a direct call, e.g. conSolve:

```
Prob = ProbCheck(Prob, 'conSolve');
Result = conSolve(Prob);
```

or a call to the multi-solver driver routine *tomRun*, e.g. for constrained optimization:

```
Result = tomRun('conSolve', Prob);
```

Note that *tomRun* handles several input formats.

Step 3 could be a call to PrintResult.m:

```
PrintResult(Result);
```

The 3rd step could be included in Step 2 by increasing the print level to 1, 2 or 3 in the call to the driver routine

```
Result = tomRun('conSolve',Prob, 3);
```

See the different demo files that gives examples of how to apply the TOMLAB format: *conDemo.m*, *ucDemo.m*, *qpDemo.m*, *lsDemo.m*, *lpDemo.m*, *mipDemo.m*, *glbDemo.m* and *glcDemo.m*.

Table 14: Routines to create a problem structure in the TOMLAB format.

Matlab call	probTypes	Type of optimization problem
Prob = bmiAssign(...)	14	Semi-definite programming with bilinear matrix inequalities.
Prob = clsAssign(...)	4,5,6	Unconstrained and constrained nonlinear least squares.
Prob = conAssign(...)	1,3	Unconstrained and constrained nonlinear optimization.
Prob = expAssign(...)	17	Exponential fitting problems.
Prob = glcAssign(...)	9,10,15	Box-bounded or mixed-integer constrained global programming.
Prob = lcpAssign(...)	22	Linear mixed-complimentary problems.
Prob = llsAssign(...)	5	Linear least-square problems.
Prob = lpAssign(...)	8	Linear programming.
Prob = lpconAssign(...)	3	Linear programming with nonlinear constraints.
Prob = mcpAssign(...)	23	Nonlinear mixed-complimentary problems.
Prob = minlpAssign(...)	12	Mixed-Integer nonlinear programming.
Prob = mipAssign(...)	7	Mixed-Integer programming.
Prob = miqpAssign(...)	11	Mixed-Integer quadratic programming.
Prob = miqqAssign(...)	18	Mixed-Integer quadratic programming with quadratic constraints.
Prob = qcpAssign(...)	23	Quadratic mixed-complimentary problems.
Prob = qpblockAssign(...)	2	Quadratic programming (factorized).
Prob = qpAssign(...)	2	Quadratic programming.
Prob = qpconAssign(...)	3	Quadratic programming with nonlinear constraints.
Prob = sdpAssign(...)	13	Semi-definite programming with linear matrix inequalities.
Prob = amplAssign(...)	1-3,7,8,11,12	For AMPL problems defined as *nl*-files.
Prob = simAssign(...)	1,3-6,9-10	General routine, functions and constraints calculated at the same time .

4.2 Modifying existing problems

It is possible to modify an existing *Prob* structure by editing elements directly, however this is not recommended since there are dependencies for memory allocation and problem sizes that the user may not be aware of.

There are a set of routines developed specifically for modifying linear constraints (do not modify directly, *Prob.mLin* need to be set to a proper value if so). All the static information can be set with the following routines.

4.2.1 add_A

Purpose
Adds linear constraints to an existing problem.

Calling Syntax
Prob = add_A(Prob, A, b_L, b_U)

Description of Inputs

Prob Existing TOMLAB problem.
A The additional linear constraints.
b_L The lower bounds for the new linear constraints.
b_U The upper bounds for the new linear constraints.

Description of Outputs

Prob Modified TOMLAB problem.

4.2.2 keep_A

Purpose
Keeps the linear constraints specified by idx.

Calling Syntax
Prob = keep_A(Prob, idx)

Description of Inputs

Prob Existing TOMLAB problem.
idx The row indices to keep in the linear constraints.

Description of Outputs

Prob Modified TOMLAB problem.

4.2.3 remove_A

Purpose
Removes the linear constraints specified by idx.

Calling Syntax
Prob = remove_A(Prob, idx)

Description of Inputs

Prob Existing TOMLAB problem.
idx The row indices to remove in the linear constraints.

Description of Outputs

Prob	Modified TOMLAB problem.

4.2.4 replace_A

Purpose
Replaces the linear constraints.

Calling Syntax
Prob = replace_A(Prob, A, b_L, b_U)

Description of Inputs

Prob	Existing TOMLAB problem.
A	New linear constraints.
b_L	Lower bounds for linear constraints.
b_U	Upper bounds for linear constraints.

Description of Outputs

Prob	Modified TOMLAB problem.

4.2.5 modify_b_L

Purpose
Modify lower bounds for linear constraints. If idx is not given b_L will be replaced.

Calling Syntax
Prob = modify_b_L(Prob, b_L, idx)

Description of Inputs

Prob	Existing TOMLAB problem.
b_L	New lower bounds for the linear constraints.
idx	Indices for the modified constraint bounds (optional).

Description of Outputs

Prob Modified TOMLAB problem.

4.2.6 modify_b_U

Purpose
Modify upper bounds for linear constraints. If idx is not given b_U will be replaced.

Calling Syntax
Prob = modify_b_U(Prob, b_U, idx)

Description of Inputs

Prob Existing TOMLAB problem.
b_U New upper bounds for the linear constraints.
idx Indices for the modified constraint bounds (optional).

Description of Outputs

Prob Modified TOMLAB problem.

4.2.7 modify_c

Purpose
Modify linear objective (LP/QP only).

Calling Syntax
Prob = modify_c(Prob, c, idx)

Description of Inputs

Prob Existing TOMLAB problem.
c New linear coefficients.
idx Indices for the modified linear coefficients (optional).

Description of Outputs

Prob Modified TOMLAB problem.

4.2.8 modify_c_L

Purpose
Modify lower bounds for nonlinear constraints. If idx is not given c_L will be replaced.

Calling Syntax
Prob = modify_c_L(Prob, c_L, idx)

Description of Inputs

Prob	Existing TOMLAB problem.
c_L	New lower bounds for the nonlinear constraints.
idx	Indices for the modified constraint bounds (optional).

Description of Outputs

Prob	Modified TOMLAB problem.

4.2.9 modify_c_U

Purpose
Modify upper bounds for nonlinear constraints. If idx is not given c_U will be replaced.

Calling Syntax
Prob = modify_c_U(Prob, c_U, idx)

Description of Inputs

Prob	Existing TOMLAB problem.
c_U	New upper bounds for the nonlinear constraints.
idx	Indices for the modified constraint bounds (optional).

Description of Outputs

Prob	Modified TOMLAB problem.

4.2.10 modify_x_0

Purpose
Modify starting point. If x_0 is outside the bounds an error will be returned. If idx is not given x_0 will be replaced.

Calling Syntax
Prob = modify_x_0(Prob, x_0, idx)

Description of Inputs

Prob	Existing TOMLAB problem.
x_0	New starting points.
idx	Indices for the modified starting points (optional).

Description of Outputs

Prob	Modified TOMLAB problem.

4.2.11 modify_x_L

Purpose
Modify lower bounds for decision variables. If idx is not given x_L will be replaced. x_0 will be shifted if needed.

Calling Syntax
Prob = modify_x_L(Prob, x_L, idx)

Description of Inputs

Prob	Existing TOMLAB problem.
x_L	New lower bounds for the decision variables.
idx	Indices for the modified lower bounds (optional).

Description of Outputs

Prob	Modified TOMLAB problem.

4.2.12 modify_x_U

Purpose
Modify upper bounds for decision variables. If idx is not given x_U will be replaced. x_0 will be shifted if needed.

Calling Syntax
Prob = modify_x_U(Prob, x_U, idx)

Description of Inputs

Prob	Existing TOMLAB problem.
x_U	New upper bounds for the decision variables.
idx	Indices for the modified upper bounds (optional).

Description of Outputs

Prob	Modified TOMLAB problem.

4.3 TomSym

For further information about TomSym, please visit http://tomsym.com/ - the pages contain detailed modeling examples and real life applications. All illustrated examples are available in the folder `/tomlab/tomsym/examples/` in the TOMLAB installation. The modeling engine supports all problem types in TOMLAB with some minor exceptions.

A detailed function listing is available in Appendix C.

TomSym combines the best features of symbolic differentiation, i.e. produces source code with simplifications and optimizations, with the strong point of automatic differentiation where the result is a procedure, rather than an expression. Hence it does not grow exponentially in size for complex expressions.

Both forward and reverse modes are supported, however, reverse is default when computing the derivative with respect to more than one variable. The command *derivative* results in forward mode, and *derivatives* in reverse mode.

TomSym produces very efficient and fully vectorized code and is compatible with TOMLAB /MAD for situations where automatic differentiation may be the only option for parts of the model.

It should also be noted that TomSym automatically provides first and second order derivatives as well as problem sparsity patterns. With the use of TomSym the user no longer needs to code cumbersome derivative expressions and Jacobian/Hessian sparsity patterns for most optimization and optimal control problems.

The main features in TomSym can be summarized with the following list:

- A full modeling environment in Matlab with support for most built-in mathematical operators.
- Automatically generates derivatives as Matlab code.
- A complete integration with PROPT (optimal control platform).
- Interfaced and compatible with MAD, i.e. MAD can be used when symbolic modeling is not suitable.
- Support for if, then, else statements.
- Automated code simplification for generated models.
- Ability to analyze most p-coded files (if code is vectorized).

4.3.1 Modeling

One of the main strength of TomSym is the ability to automatically and quickly compute symbolic derivatives of matrix expressions. The derivatives can then be converted into efficient Matlab code.

The matrix derivative of a matrix function is a fourth rank tensor - that is, a matrix each of whose entries is a matrix. Rather than using four-dimensional matrices to represent this, TomSym continues to work in two dimensions. This makes it possible to take advantage of the very efficient handling of sparse matrices in Matlab (not available for higher-dimensional matrices).

In order for the derivative to be two-dimensional, TomSym's derivative reduces its arguments to one-dimensional vectors before the derivative is computed. In the returned J, each row corresponds to an element of F, and each column corresponds to an element of X. As usual in Matlab, the elements of a matrix are taken in column-first order.

For vectors F and X, the resulting J is the well-known Jacobian matrix.

Observe that the TomSym notation is slightly different from commonly used mathematical notation. The notation used in tomSym was chosen to minimize the amount of element reordering needed to compute gradients for common expressions in optimization problems. It needs to be pointed out that this is different from the commonly used mathematical notation, where the tensor $(\frac{dF}{dX})$ is flattened into a two-dimensional matrix as it is written (There are actually two variations of this in common use - the indexing of the elements of X may or may not be transposed).

For example, in common mathematical notation, the so-called self derivative matrix $(\frac{dX}{dX})$ is a mn-by-mn square (or mm-by-nn rectangular in the non-transposed variation) matrix containing mn ones spread out in a random-looking manner. In tomSym notation, the self-derivative matrix is the mn-by-mn identity matrix.

The difference in notation only involves the ordering of the elements, and reordering the elements to a different notational convention should be trivial if tomSym is used to generate derivatives for applications other than for TOMLAB and PROPT.

Example:

```
>> toms     y
>> toms 3x1 x
>> toms 2x3 A
>> f = (A*x).^(2*y)

f = tomSym(2x1):

   (A*x).^(2*y)

>> derivative(f,A)

ans = tomSym(2x6):

   (2*y)*setdiag((A*x).^(2*y-1))*kron(x',eye(2))
```

In the above example, the 2x1 symbol f is differentiated with respect to the 2x3 symbol A. The result is a 2x6 matrix, representing $\frac{d(vec(f))}{d(vec(A))}$.

The displayed text is not necessarily identical to the m-code that will be generated from an expression. For example, the identity matrix is generated using speye in m-code, but displayed as eye (Derivatives tend to involve many sparse matrices, which Matlab handles efficiently). The mcodestr command converts a tomSym object to a Matlab code string.

```
>> mcodestr(ans)
ans =
setdiag((2*y)*(A*x).^(2*y-1))*kron(x',[1 0;0 1])
```

Observe that the command mcode and not mcodestr should be used when generating efficient production code.

4.3.2 Ezsolve

TomSym provides the function ezsolve, which needs minimal input to solve an optimization problem: only the objective function and constraints. For example, the miqpQG example from the tomlab quickguide can be reduced to the following:

```
toms integer x
toms         y

objective = -6*x + 2*x^2 + 2*y^2 - 2*x*y;
constraints = {x+y<=1.9,x>=0, y>=0};

solution = ezsolve(objective,constraints)
```

Ezsolve calls tomDiagnose to determine the problem type, getSolver to find an appropriate solver, then sym2prob, tomRun and getSoluton in sequence to obtain the solution.

Advanced users might not use ezsolve, and instead call sym2prob and tomRun directly. This provides for the possibility to modify the Prob struct and set flags for the solver.

4.3.3 Usage

TomSym, unlike most other symbolic algebra packages, focuses on **vectorized** notation. This should be familiar to Matlab users, but is very different from many other programming languages. When computing the derivative of a vector-valued function with respect to a vector valued variable, tomSym attempts to give a derivative as vectorized Matlab code. However, this only works if the original expressions use vectorized notation. For example:

```
toms 3x1 x
f = sum(exp(x));
g = derivative(f,x);
```

results in the efficient $g = exp(x)'$. In contrast, the mathematically equivalent but slower code:

```
toms 3x1 x
f = 0;
for i=1:length(x)
   f = f+exp(x(i));
end
g = derivative(f,x);
```

results in $g = (exp(x(1)) * [100] + exp(x(2)) * [010]) + exp(x(3)) * [001]$ as each term is differentiated individually. Since tomSym has no way of knowing that all the terms have the same format, it has to compute the symbolic derivative for each one. In this example, with only three iterations, that is not really a problem, but large for-loops can easily result in symbolic calculations that require more time than the actual numeric solution of the underlying optimization problem.

It is thus recommended to avoid for-loops as far as possible when working with tomSym.

Because tomSym computes derivatives with respect to whole symbols, and not their individual elements, it is also a good idea not to group several variables into one vector, when they are mostly used individually. For example:

```
toms 2x1 x
f = x(1)*sin(x(2));
g = derivative(f,x);
```

results in $g = sin(x(2)) * [10] + x(1) * (cos(x(2)) * [01])$. Since x is never used as a 2x1 vector, it is better to use two independent 1x1 variables:

```
toms a b
f = a*sin(b);
g = derivative(f,[a; b]);
```

which results in $g = [sin(b)\ a * cos(b)]$. The main benefit here is the increased readability of the auto-generated code, but there is also a slight performance increase (Should the vector x later be needed, it can of course easily be created using the code $x = [a; b]$).

4.3.4 Scaling variables

Because tomSym provides analytic derivatives (including second order derivatives) for the solvers, badly scaled problems will likely be less problematic from the start. To further improve the model, tomSym also makes it very easy to manually scale variables before they are presented to the solver. For example, assuming that an optimization problem involves the variable x which is of the order of magnitude $1e6$, and the variable y, which is of the order of $1e-6$, the code:

```
toms xScaled yScaled
x = 1e+6*xScaled;
y = 1e-6*yScaled;
```

will make it possible to work with the tomSym expressions x and y when coding the optimization problem, while the solver will solve for the symbols xScaled and yScaled, which will both be in the order of 1. It is even possible to provide starting guesses on x and y (in equation form), because tomSym will solve the linear equation to obtain starting guesses for the underlying xScaled and yScaled.

The solution structure returned by ezsolve will of course only contain xScaled and yScaled, but numeric values for x and y are easily obtained via, e.g. subs(x,solution).

4.3.5 SDP/LMI/BMI interface

An interface for bilinear semidefinite problems is included with tomSym. It is also possible to solve nonlinear problems involving semidefinite constraints, using any nonlinear solver (The square root of the semidefinte matrix is then introduced as an extra set of unknowns).

See the examples *optimalFeedbackBMI* and *example_sdp*.

4.3.6 Interface to MAD and finite differences

If a user function is incompatible with tomSym, it can still be used in symbolic computations, by giving it a "wrapper". For example, if the cos function was not already overloaded by tomSym, it would still be possible to do the equivalent of cos(3*x) by writing feval('cos',3*x).

MAD then computes the derivatives when the Jacobian matrix of a wrapped function is needed. If MAD is unavailable, or unable to do the job, numeric differentiation is used.

Second order derivatives cannot be obtained in the current implementation.

It is also possible to force the use of automatic or numerical differentiation for any function used in the code. The follow examples shows a few of the options available:

```
toms x1 x2
alpha = 100;

% 1. USE MAD FOR ONE FUNCTION.
% Create a wrapper function. In this case we use sin, but it could be any
% MAD supported function.
y = wrap(struct('fun','sin','n',1,'sz1',1,'sz2',1,'JFuns','MAD'),x1/x2);
f = alpha*(x2-x1^2)^2 + (1-x1)^2 + y;

% Setup and solve the problem
c = -x1^2 - x2;
con = {-1000 <= c <= 0
    -10 <= x1 <= 2
    -10 <= x2 <= 2};

x0 = {x1 == -1.2
    x2 == 1};

solution1 = ezsolve(f,con,x0);

% 2. USE MAD FOR ALL FUNCTIONS.
options = struct;
options.derivatives = 'automatic';
f = alpha*(x2-x1^2)^2 + (1-x1)^2 + sin(x1/x2);
solution2 = ezsolve(f,con,x0,options);

% 3. USE FD (Finite Differences) FOR ONE FUNCTIONS.
% Create a new wrapper function. In this case we use sin, but it could be
% any function since we use numerical derivatives.
y = wrap(struct('fun','sin','n',1,'sz1',1,'sz2',1,'JFuns','FDJac'),x1/x2);
f = alpha*(x2-x1^2)^2 + (1-x1)^2 + y;
solution3 = ezsolve(f,con,x0);

% 4. USE FD and MAD FOR ONE FUNCTION EACH.
y1 = 0.5*wrap(struct('fun','sin','n',1,'sz1',1,'sz2',1,'JFuns','MAD'),x1/x2);
y2 = 0.5*wrap(struct('fun','sin','n',1,'sz1',1,'sz2',1,'JFuns','FDJac'),x1/x2);
f = alpha*(x2-x1^2)^2 + (1-x1)^2 + y1 + y2;
solution4 = ezsolve(f,con,x0);

% 5. USE FD FOR ALL FUNCTIONS.
options = struct;
options.derivatives = 'numerical';
f = alpha*(x2-x1^2)^2 + (1-x1)^2 + sin(x1/x2);
solution5 = ezsolve(f,con,x0,options);
```

```
% 6. USE MAD FOR OBJECTIVE, FD FOR CONSTRAINTS
options = struct;
options.derivatives = 'numerical';
options.use_H   = 0;
options.use_d2c = 0;
options.type    = 'con';
Prob = sym2prob(f,con,x0,options);
madinitglobals;
Prob.ADObj = 1;
Prob.ConsDiff = 1;
Result = tomRun('snopt', Prob, 1);
solution6 = getSolution(Result);
```

4.3.7 Simplifications

The code generation function detects sub-expressions that occur more than once, and optimizes by creating temporary variables for those since it is very common for a function to share expressions with its derivative, or for the derivative to contain repeated expressions.

Note that it is not necessary to complete code generation in order to evaluate a tomSym object numerically. The subs function can be used to replace symbols by their numeric values, resulting in an evaluation.

TomSym also automatically implements algebraic simplifications of expressions. Among them are:

- Multiplication by 1 is eliminated: $1 * A = A$
- Addition/subtraction of 0 is eliminated: $0 + A = A$
- All-same matrices are reduced to scalars: $[3; 3; 3] + x = 3 + x$
- Scalars are moved to the left in multiplications: $A * y = y * A$
- Scalars are moved to the left in addition/subtraction: $A - y = -y + A$
- Expressions involving element-wise operations are moved inside setdiag: $setdiag(A)+setdiag(A) = setdiag(A+A)$
- Inverse operations cancel: $sqrt(x)^2 = x$
- Multiplication by inverse cancels: $A * inv(A) = eye(size(A))$
- Subtraction of self cancels: $A - A = zeros(size(A))$
- Among others...

Except in the case of scalar-matrix operations, tomSym does not reorder multiplications or additions, which means that some expressions, like (A+B)-A will not be simplified (This might be implemented in a later version, but must be done carefully so that truncation errors are not introduced).

Simplifications are also applied when using subs. This makes it quick and easy to handle parameterized problems. For example, if an intricate optimization problem is to be solved for several values of a parameter a, then one

might first create the symbolic functions and gradients using a symbolic a, and then substitute the different values, and generate m-code for each substituted function. If some case, like $a = 0$ results in entire sub-expressions being eliminated, then the m-code will be shorter in that case.

It is also possible to generate complete problems with constants as decision variables and then change the bounds for these variables to make them "real constants". The backside of this is that the problem will be slightly larger, but the problem only has to be generated once.

The following problem defines the variable alpha as a toms, then the bounds are adjusted for alpha to solve the problem for all alphas from 1 to 100.

```
toms x1 x2

% Define alpha as a toms although it is a constant
toms alpha

% Setup and solve the problem
f = alpha*(x2-x1^2)^2 + (1-x1)^2;
c = -x1^2 - x2;
con = {-1000 <= c <= 0
    -10 <= x1 <= 2
    -10 <= x2 <= 2};
x0 = {x1 == -1.2; x2 == 1};

Prob = sym2prob(f,con,x0);

% Now solve for alpha = 1:100, while reusing x_0
obj = zeros(100,1);
for i=1:100
    Prob.x_L(Prob.tomSym.idx.alpha) = i;
    Prob.x_U(Prob.tomSym.idx.alpha) = i;
    Prob.x_0(Prob.tomSym.idx.alpha) = i;
    Result = tomRun('snopt', Prob, 1);
    Prob.x_0 = Result.x_k;
    obj(i) = Result.f_k;
end
```

4.3.8 Special functions

TomSym adds some functions that duplicates the functionality of Matlab, but that are more suitable for symbolic treatment. For example:

- **setDiag** and **getDiag** - Replaces some uses of Matlab's diag function, but clarifies whether diag(x) means "create a matrix where the diagonal elements are the elements of x" or "extract the main diagonal from the matrix x".
- **subsVec** applies an expression to a list of values. The same effect can be achieved with a for-loop, but subsVec gives more efficient derivatives.

- ifThenElse - A replacement for the if ... then ... else constructs (See below).

If ... then ... else:

A common reason that it is difficult to implement a function in tomSym is that it contains code like the following:

```
if x<2
    y = 0;
else
    y = x-2;
end
```

Because x is a symbolic object, the expression $x < 2$ does not evaluate to true or false, but to another symbolic object.

In tomSym, one should instead write:

```
y = ifThenElse(x<2,0,x-2)
```

This will result in a symbolic object that contains information about both the "true" and the "false" scenario. However, taking the derivative of this function will result in a warning, because the derivative is not well-defined at $x = 2$.

The "smoothed" form:

```
y = ifThenElse(x<2,0,x-2,0.1)
```

yields a function that is essentially the same for $abs(x-2) > 3*0.1$, but which follows a smooth curve near $x = 2$, ensuring that derivatives of all orders exist. However, this introduces a local minimum which did not exist in the original function, and invalidates the convexity.

It is recommended that the smooth form ifThenElse be used for nonlinear problems whenever it replaces a discontinuous function. However, for convex functions (like the one above) it is usually better to use code that helps tomSym know that convexity is preserved. For example, instead of the above $ifThenElse(x < 2, 0, x-2, 0.1)$, the equivalent $max(0, x-2)$ is preferred.

4.3.9 Procedure vs parse-tree

TomSym works with procedures. This makes it different from many symbolic algebra packages, that mainly work with parse-trees.

In optimization, it is not uncommon for objectives and constraints to be defined using procedures that involve loops. TomSym is built to handle these efficiently. If a function is defined using many intermediate steps, then tomSym will keep track of those steps in an optimized procedure description. For example, consider the code:

```
toms x
y = x*x;
z = sin(y)+cos(y);
```

In the tomSym object z, there is now a procedure, which looks something like:

```
temp = x*x;
result = cos(temp)+sin(temp);
```

Note: It is not necessary to use the intermediate symbol y. TomSym, automatically detects duplicated expressions, so the code sin(x*x)+cos(x*x) would result in the same optimized procedure for z.

On the other hand, the same corresponding code using the symbolic toolbox:

```
syms x
y = x*x;
z = sin(y)+cos(y);
```

results in an object z that contains $cos(x^2) + sin(x^2)$, resulting in a double evaluation of x^2.

This may seem like a small difference in this simplified example, but in real-life applications, the difference can be significant.

Numeric stability:

For example, consider the following code, which computes the Legendre polynomials up to the 100th order in tomSym (The calculation takes about two seconds on a modern computer).

```
toms x
p{1}=1;
p{2}=x;
for i=1:99
   p{i+2} = ((2*i+1)*x.*p{i+1}-i*p{i})./(i+1);
end
```

Replacing "toms" by "syms" on the first line should cause the same polynomials to be computed using Mathwork's Symbolic Toolbox. But after a few minutes, when only about 30 polynomials have been computed, the program crashes as it fails to allocate more memory. This is because the expression grows exponentially in size. To circumvent the problem, the expression must be algebraically simplified after each step. The following code succeeds in computing the 100 polynomials using the symbolic toolbox.

```
syms x
p{1}=1;
p{2}=x;
for i=1:99
   p{i+2} = simplify(((2*i+1)*x.*p{i+1}-i*p{i})./(i+1));
end
```

However, the simplification changes the way in which the polynomial is computed. This is clearly illustrated if we insert $x = 1$ into the 100th order polynomial. This is accomplished by using the command subs(p101,x,1) for both the tomSym and the Symbolic Toolbox expressions. TomSym returns the result 1.0000, which is correct. The symbolic toolbox, on the other hand, returns $2.6759e + 020$, which is off by 20 orders of magnitude. The reason is that the "simplified" symbolic expressions involves subtracting very large numbers. Note: It is of course possible to get correct results from the Symbolic Toolbox using exact arithmetic instead of machine-precision floating-point, but at the cost of much slower evaluation.

In tomSym, there are also simplifications, for example identifying identical sub-trees, or multiplication by zero, but the simplifications are not as extensive, and care is taken to avoid simplifications that can lead to truncation errors. Thus, an expression computed using tomSym should be exactly as stable (or unstable) as the algorithm used to generate it.

Another example:

The following code, iteratively defines q as a function of the tomSym symbol x, and computes its derivative:

```
toms x
q=x;
for i=1:4
   q = x*cos(q+2)*cos(q);
end
derivative(q,x)
```

This yields the following tomSym procedure:

```
tempC3   = x+2;
tempC4   = cos(tempC3);
tempC5   = x*tempC4;
tempC10  = cos(x);
tempC12  = tempC10*(tempC4-x*sin(tempC3))-tempC5*sin(x);
tempC13  = tempC5*tempC10;
tempC16  = tempC13+2;
tempC17  = cos(tempC16);
tempC18  = x*tempC17;
tempC24  = cos(tempC13);
tempC26  = tempC24*(tempC17-x*(sin(tempC16)*tempC12))-tempC18*(sin(tempC13)*tempC12);
tempC27  = tempC18*tempC24;
tempC30  = tempC27+2;
tempC31  = cos(tempC30);
tempC32  = x*tempC31;
tempC38  = cos(tempC27);
tempC40  = tempC38*(tempC31-x*(sin(tempC30)*tempC26))-tempC32*(sin(tempC27)*tempC26);
tempC41  = tempC32*tempC38;
tempC44  = tempC41+2;
tempC45  = cos(tempC44);
out      = cos(tempC41)*(tempC45-x*(sin(tempC44)*tempC40))-(x*tempC45)*(sin(tempC41)*tempC40);
```

Now, complete the same task using the symbolic toolbox:

```
syms x
q=x;
for i=1:4
   q = x*cos(q+2)*cos(q);
end
diff(q,x)
```

This yields the following symbolic expression:

```
cos(x*cos(x*cos(cos(x+2)*x*cos(x)+2)*cos(cos(x+2)*x*cos(x))+2)*cos(x*cos(cos(x+2)*x*cos(x)+2)*...
cos(cos(x+2)*x*cos(x)))+2)*cos(x*cos(x*cos(cos(x+2)*x*cos(x)+2)*cos(cos(x+2)*x*cos(x))+2)*cos(x*...
cos(cos(x+2)*x*cos(x)+2)*cos(cos(x+2)*x*cos(x))))-x*sin(x*cos(x*cos(cos(x+2)*x*cos(x)+2)*cos(...
...
and 23 more lines of code.
```

And this example only had four iterations of the loop. Increasing the number of iterations, the Symbolic toolbox expressions quickly becomes unmanageable, while the tomSym procedure only grows linearly.

4.3.10 Problems and error messages

- **Warning: Directory c:\Temp\tp563415 could not be removed (or similar).** When tomSym is used to automatically create m-code it places the code in a temporary directory given by Matlab's tempname function. Sometimes Matlab chooses a name that already exists, which results in this error message (The temporary directory is cleared of old files regularly by most modern operating systems. Otherwise the temporary Matlab files can easily be removed manually).
- **Attempting to call SCRIPT as a function (or similar).** Due to a bug in the Matlab syntax, the parser cannot know if $f(x)$ is a function call or the x:th element of the vector f. Hence, it has to guess. The Matlab parser does not understand that toms creates variables, so it will get confused if one of the names is previously used by a function or script (For example, "cs" is a script in the systems identification toolbox). Declaring toms cs and then indexing cs(1) will work at the Matlab prompt, but not in a script. The bug can be circumvented by assigning something to each variable before calling toms.

4.3.11 Example

A TomSym model is to a great extent independent upon the problem type, i.e. a linear, nonlinear or mixed-integer nonlinear model would be modeled with about the same commands. The following example illustrates how to construct and solve a MINLP problem using TomSym.

```
Name='minlp1Demo - Kocis/Grossman.';

toms 2x1 x
toms 3x1 integer y

objective = [2 3 1.5 2 -0.5]*[x;y];

constraints = { ...
   x(1) >= 0, ...
   x(2) >= 1e-8, ...
   x <= 1e8, ...
   0 <= y <=1, ...
   [1 0 1 0 0]*[x;y] <= 1.6, ...
   1.333*x(2) + y(2) <= 3, ...
   [-1 -1 1]*y <= 0, ...
```

```
   x(1)^2+y(1) == 1.25, ...
   sqrt(x(2)^3)+1.5*y(2) == 3, ...
};

guess = struct('x',ones(size(x)),'y',ones(size(y)));
options = struct;
options.name = Name;
Prob = sym2prob('minlp',objective,constraints,guess,options);
Prob.DUNDEE.optPar(20) = 1;
Result = tomRun('minlpBB',Prob,2);
```

The TomSym engine automatically completes the separation of simple bounds, linear and nonlinear constraints.

5 Solving Linear, Quadratic and Integer Programming Problems

This section describes how to define and solve linear and quadratic programming problems, and mixed-integer linear programs using TOMLAB. Several examples are given on how to proceed, depending on if a quick solution is wanted, or more advanced tests are needed. TOMLAB is also compatible with MathWorks Optimization TB. See Appendix E for more information and test examples.

The test examples and output files are part of the standard distribution of TOMLAB, available in directory *usersguide*, and all tests can be run by the user. There is a file *RunAllTests* that goes through and runs all tests for this section.

Also see the files *lpDemo.m*, *qpDemo.m*, and *mipDemo.m*, in the directory *examples*, where in each file a set of simple examples are defined. The examples may be ran by giving the corresponding file name, which displays a menu, or by running the general TOMLAB help routine *tomHelp.m*.

5.1 Linear Programming Problems

The general formulation in TOMLAB for a linear programming problem is

$$\begin{aligned} \min_x \quad & f(x) = c^T x \\ s/t \quad & \begin{array}{ccccc} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U \end{array} \end{aligned} \tag{12}$$

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 equal to the upper bound, i.e. for constraint i: $b_L(i) = b_U(i)$.

To illustrate the solution of LPs consider the simple linear programming test problem

$$\begin{array}{lccc} \min\limits_{x_1,x_2} & f(x_1,x_2) & = & -7x_1 - 5x_2 \\ s/t & x_1 + 2x_2 & \leq & 6 \\ & 4x_1 + x_2 & \leq & 12 \\ & x_1, x_2 & \geq & 0 \end{array} \tag{13}$$

named *LP Example*.

The following statements define this problem in Matlab

File: tomlab/usersguide/lpExample.m

```
Name  = 'lptest';
c     = [-7 -5]';   % Coefficients in linear objective function
A     = [ 1  2
          4  1 ];   % Matrix defining linear constraints
b_U   = [ 6 12 ]';  % Upper bounds on the linear inequalities
x_L   = [ 0  0 ]';  % Lower bounds on x

% x_min and x_max are only needed if doing plots
x_min = [ 0  0 ]';
x_max = [10 10 ]';
```

```
% b_L, x_U and x_0 have default values and need not be defined.
% It is possible to call lpAssign with empty [] arguments instead
b_L   = [-inf -inf]';
x_U   = [];
x_0   = [];
```

5.1.1 A Quick Linear Programming Solution

The quickest way to solve this problem is to define the following Matlab statements using the TOMLAB format:

File: tomlab/usersguide/lpTest1.m

```
lpExample;

Prob   = lpAssign(c, A, b_L, b_U, x_L, x_U, x_0, 'lpExample');
Result = tomRun('lpSimplex', Prob, 1);
```

lpAssign is used to define the standard Prob structure, which TOMLAB always uses to store all information about a problem. The three last parameters could be left out. The upper bounds will default be Inf, and the problem name is only used in the printout in *PrintResult* to make the output nicer to read. If x_0, the initial value, is left out, an initial point is found by *lpSimplex* solving a feasible point (Phase I) linear programming problem. In this test the given x_0 is empty, so a Phase I problem must be solved. The solution of this problem gives the following output to the screen

File: tomlab/usersguide/lpTest1.out

```
===== * * * ========================================================================= * * *
TOMLAB /SOL + /CGO + /Xpress MEX + /CPLEX Parallel 2-CPU + 21 more - Tomlab Optimizat
=============================================================================================
Problem: ---  1: lpExample                       f_k     -26.571428571428569000

Solver: lpSimplex.  EXIT=0.
Simplex method. Minimum reduced cost.
Optimal solution found

FuncEv    3 Iter    3
CPU time: 0.046800 sec. Elapsed time: 0.019000 sec.
```

Having defined the *Prob* structure is it easy to call any solver that can handle linear programming problems,

```
Result = tomRun('qpSolve', Prob, 1);
```

Even a nonlinear solver may be used.

```
Result = tomRun('nlpSolve',Prob, 3);
```

All TOMLAB solvers may either be called directly, or by using the driver routine *tomRun*, as in this case.

5.2 Quadratic Programming Problems

The general formulation in TOMLAB for a quadratic programming problem is

$$\begin{array}{lll} \min\limits_{x} & f(x) = \frac{1}{2}x^T F x + c^T x & \\ s/t & \begin{array}{ccccc} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U \end{array} & \end{array} \tag{14}$$

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.

To illustrate the solution of QPs consider the simple quadratic programming test problem

$$\begin{array}{ll} \min\limits_{x} & f(x) = 4x_1^2 + 1x_1x_2 + 4x_2^2 + 3x_1 - 4x_2 \\ s/t & x_1 + x_2 \leq 5 \\ & x_1 - x_2 = 0 \\ & x_1 \geq 0 \\ & x_2 \geq 0, \end{array} \tag{15}$$

named *QP Example*. The following statements define this problem in Matlab.

File: tomlab/usersguide/qpExample.m

```
Name  = 'QP Example';  % File qpExample.m
F     = [ 8   1        % Matrix F in 1/2 * x' * F * x + c' * x
          1   8 ];
c     = [ 3  -4 ]';    % Vector c in 1/2 * x' * F * x + c' * x
A     = [ 1   1        % Constraint matrix
          1  -1 ];
b_L   = [-inf  0  ]';  % Lower bounds on the linear constraints
b_U   = [  5   0  ]';  % Upper bounds on the linear constraints
x_L   = [  0   0  ]';  % Lower bounds on the variables
x_U   = [ inf inf ]';  % Upper bounds on the variables
x_0   = [  0   1  ]';  % Starting point
x_min = [-1 -1 ];      % Plot region lower bound parameters
x_max = [ 6  6 ];      % Plot region upper bound parameters
```

5.2.1 A Quick Quadratic Programming solution

The quickest way to solve this problem is to define the following Matlab statements using the TOMLAB format:

File: tomlab/usersguide/qpTest1.m

```
qpExample;

Prob = qpAssign(F, c, A, b_L, b_U, x_L, x_U, x_0, 'qpExample');
```

```
Result = tomRun('qpSolve', Prob, 1);
```

The solution of this problem gives the following output to the screen.

File: tomlab/usersguide/qpTest1.out

```
===== * * * ========================================================================= * * *
TOMLAB /SOL + /CGO + /Xpress MEX + /CPLEX Parallel 2-CPU + 21 more - Tomlab Optimizat
================================================================================
Problem: ---  1: qpExample                       f_k      -0.027777777777777790

Solver: qpSolve.  EXIT=0.  INFORM=1.
Active set strategy
Optimal point found
First order multipliers >= 0

Iter    4
CPU time: 0.046800 sec. Elapsed time: 0.037000 sec.
```

qpAssign is used to define the standard Prob structure, which TOMLAB always uses to store all information about a problem. The three last parameters could be left out. The upper bounds will default be Inf, and the problem name is only used in the printout in *PrintResult* to make the output nicer to read. If x_0, the initial value, is left out, a initial point is found by *qpSolve* solving a feasible point (Phase I) linear programming problem calling the TOMLAB *lpSimplex* solver. In fact, the output shows that the given $x_0 = (0, -1)^T$ was rejected because it was infeasible, and instead a Phase I solution lead to the initial point $x_0 = (0, 0)^T$.

5.3 Mixed-Integer Programming Problems

This section describes how to solve mixed-integer programming problems efficiently using TOMLAB. To illustrate the solution of MIPs consider the simple knapsack 0/1 test problem *Weingartner 1*, which has 28 binary variables and two knapsacks. The problem is defined

$$\begin{array}{lll} \min\limits_{x} & & c^T x \\ s/t & 0 \leq x \leq 1, \\ & Ax = b, \end{array} \tag{16}$$

where $b = (600, 600)^T$,

$$c = -(\begin{array}{cccccccccccccc} 1898 & 440 & 22507 & 270 & 14148 & 3100 & 4650 & 30800 & 615 & 4975 & 1160 & 4225 & 510 & 11880 \\ 479 & 440 & 490 & 330 & 110 & 560 & 24355 & 2885 & 11748 & 4550 & 750 & 3720 & 1950 & 10500 \end{array})^T$$

and the A matrix is

$$\begin{pmatrix} 45 & 0 & 85 & 150 & 65 & 95 & 30 & 0 & 170 & 0 & 40 & 25 & 20 & 0 & 0 & 25 & 0 & 0 & 25 & 0 \\ 165 & 0 & 85 & 0 & 0 & 0 & 0 & 100 & & & & & & & & & & & & \\ 30 & 20 & 125 & 5 & 80 & 25 & 35 & 73 & 12 & 15 & 15 & 40 & 5 & 10 & 10 & 12 & 10 & 9 & 0 & 20 \\ 60 & 40 & 50 & 36 & 49 & 40 & 19 & 150 & & & & & & & & & & & & \end{pmatrix}$$

The following statements define this problem in Matlab using the TOMLAB format:

File: tomlab/usersguide/mipExample.m

```
Name='Weingartner 1 - 2/28 0-1 knapsack';
% Problem formulated as a minimum problem
A = [ 45      0     85     150     65     95     30      0    170  0 ...
      40     25     20       0      0     25      0      0     25  0 ...
      165     0     85       0      0      0      0    100  ; ...
      30     20    125       5     80     25     35     73     12  15 ...
      15     40      5      10     10     12     10      9      0  20 ...
      60     40     50      36     49     40     19    150];
b_U = [600;600];  % 2 knapsack capacities
c   = [1898  440  22507   270  14148  3100  4650  30800   615  4975 ...
       1160 4225    510 11880    479   440   490    330   110   560 ...
      24355 2885  11748  4550    750  3720  1950  10500]'; % 28 weights

% Make problem on standard form for mipSolve
[m,n]   = size(A);
A       = [A eye(m,m)];
c       = [-c;zeros(m,1)]; % Change sign to make a minimum problem
[mm nn] = size(A);
x_L     = zeros(nn,1);
x_U     = [ones(n,1);b_U];
x_0     = [zeros(n,1);b_U];

fprintf('Knapsack problem. Variables %d. Knapsacks %d\n',n,m);
```

```
   fprintf('Making standard form with   %d variables\n',nn);

   % All original variables should be integer, but also slacks in this case
   IntVars = nn;  % Could also be set as: IntVars=1:nn; or IntVars=ones(nn,1);
   x_min   = x_L; x_max  = x_U; f_Low  = -1E7; % f_Low <= f_optimal must hold
   n       = length(c);
   b_L     = b_U;
   f_opt   = -141278;
```

The quickest way to solve this problem is to define the following Matlab statements:

File: tomlab/usersguide/mipTest1.m

```
mipExample;

nProblem  = 7;  % Use the same problem number as in mip_prob.m
fIP       = []; % Do not use any prior knowledge
xIP       = []; % Do not use any prior knowledge
setupFile = []; % Just define the Prob structure, not any permanent setup file
x_opt     = []; % The optimal integer solution is not known
VarWeight = []; % No variable priorities, largest fractional part will be used
KNAPSACK  = 0;  % First run without the knapsack heuristic

Prob = mipAssign(c, A, b_L, b_U, x_L, x_U, x_0, Name, setupFile, nProblem,...
                 IntVars, VarWeight, KNAPSACK, fIP, xIP, ...
                 f_Low, x_min, x_max, f_opt, x_opt);

Prob.Solver.Alg       = 2;    % Depth First, then Breadth (Default Depth First)
Prob.optParam.MaxIter = 5000; % Must increase iterations from default 500
Prob.optParam.IterPrint = 0;
Prob.PriLev = 1;
Result                = tomRun('mipSolve', Prob, 0);

% ------------------------------------------------
% Add priorities on the variables
% ------------------------------------------------
VarWeight = c;
% NOTE. Prob is already defined,  could skip mipAssign, directly set:
% Prob.MIP.VarWeight=c;

Prob = mipAssign(c, A, b_L, b_U, x_L, x_U, x_0, Name, setupFile, nProblem,...
                 IntVars, VarWeight, KNAPSACK, fIP, xIP, ...
                 f_Low, x_min, x_max, f_opt, x_opt);


Prob.Solver.Alg       = 2;             % Depth First, then Breadth search
Prob.optParam.MaxIter = 5000;          % Must increase number of iterations
```

```
Prob.optParam.IterPrint = 0;
Prob.PriLev = 1;
Result                = tomRun('mipSolve', Prob, 0);

% ------------------------------------------
% Use the knapsack heuristic, but not priorities
% ------------------------------------------
KNAPSACK  = 1; VarWeight = [];
% NOTE. Prob is already defined,  could skip mipAssign, directly set:
% Prob.MIP.KNAPSACK=1;
% Prob.MIP.VarWeight=[];

Prob      = mipAssign(c, A, b_L, b_U, x_L, x_U, x_0, Name, setupFile, ...
                      nProblem, IntVars, VarWeight, KNAPSACK, fIP, xIP, ...
                      f_Low, x_min, x_max, f_opt, x_opt);

Prob.Solver.Alg = 2;              % Depth First, then Breadth search
Prob.optParam.IterPrint = 0;
Prob.PriLev = 1;
Result                = tomRun('mipSolve', Prob, 0);

% ------------------------------------------------
% Now use both the knapsack heuristic and priorities
% ------------------------------------------------
VarWeight = c; KNAPSACK  = 1;
% NOTE. Prob is already defined,  could skip mipAssign, directly set:
% Prob.MIP.KNAPSACK=1;
% Prob.MIP.VarWeight=c;

Prob = mipAssign(c, A, b_L, b_U, x_L, x_U, x_0, Name, setupFile, nProblem,...
                 IntVars, VarWeight, KNAPSACK, fIP, xIP, ...
                 f_Low, x_min, x_max, f_opt, x_opt);

Prob.Solver.Alg = 2;              % Depth First, then Breadth search
Prob.optParam.IterPrint = 0;
Prob.PriLev = 1;
Result                = tomRun('mipSolve', Prob, 0);
```

To make it easier to see all variable settings, the first lines define the needed variables. Several of them are just empty arrays, and could be directly set as empty in the call to *mipAssign*. *mipAssign* is used to define the standard *Prob* structure, which TOMLAB always uses to store all information about a problem. After mipAssign has defined the structure *Prob* it is then easy to set or change fields in the structure. The solver *mipSolve* is using three different strategies to search the branch-and-bound tree. The default is the *Depth first* strategy, which is also the result if setting the field *Solver.Alg* as zero. Setting the field as one gives the *Breadth first* strategy and setting it as two gives the *Depth first, then breadth search* strategy. In the example our choice is the last strategy. The number of iterations might be many, thus the maximal number of iterations must be increased, the field *optParam.MaxIter*.

Tests show two ways to improve the convergence of MIP problems. One is to define a priority order in which the different non-integer variables are selected as variables to branch on. The field *MIP.VarWeight* is used to set priority numbers for each variable. Note that the lower the number, the higher the priority. In our test case the coefficients of the objective function is used as priority weights. The other way to improve convergence is to use a heuristic. For binary variables a simple knapsack heuristic is implemented in *mipSolve*. Setting the field *MIP.KNAPSACK* as true instructs *mipSolve* to use the heuristic.

Running the four different tests on the knapsack problem gives the following output to the screen

File: tomlab/usersguide/mipTest1.out

```
mipTest1
Knapsack problem. Variables 28. Knapsacks 2
Branch and bound. Depth First, then Breadth.

--- Branch & Bound converged! Iterations (nodes visited) =   714 Total LP Iterations =    713

    Optimal Objective function =    -141278.0000000000000000
x: 0  0  1 -0  1  1  1  1  0  1  0  1  1  1
   0  0  0  0  1  0  1  0  1  1  0  1  0  0
B: -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
Branch & bound. Depth First, then Breadth. Priority weights.

--- Branch & Bound converged! Iterations (nodes visited) =   470 Total LP Iterations =    469

    Optimal Objective function =    -141278.0000000000000000
x: 0  0  1 -0  1  1  1  1  0  1  0  1  1  1
   0  0  0  0  1  0  1  0  1  1  0  1  0  0
B: -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
Branch and bound. Depth First, then Breadth. Knapsack Heuristic.
    Found new BEST Knapsack. Nodes left      0. Nodes deleted      0.
    Best IP function value          -139508.0000000000000000
    Found new BEST Knapsack. Nodes left      1. Nodes deleted      0.
    Best IP function value          -140768.0000000000000000
    Found new BEST Knapsack. Nodes left      4. Nodes deleted      0.
    Best IP function value          -141278.0000000000000000

--- Branch & Bound converged! Iterations (nodes visited) =    96 Total LP Iterations =     95

    Optimal Objective function =    -141278.0000000000000000
x: 0  0  1 -0  1  1  1  1  0  1  0  1  1  1
   0  0  0  0  1  0  1  0  1  1  0  1  0  0
B: 1 1 -1 1 -1 -1 -1 -1 1 -1 0 -1 -1 -1 1 1 1 1 -1 1 -1 0 -1 -1 1 -1 -1 1
Branch & bound. Depth First, then Breadth. Knapsack heuristic. Priority weights.
    Found new BEST Knapsack. Nodes left      0. Nodes deleted      0.
    Best IP function value          -139508.0000000000000000
    Found new BEST Knapsack. Nodes left      1. Nodes deleted      0.
    Best IP function value          -140768.0000000000000000
    Found new BEST Knapsack. Nodes left      4. Nodes deleted      0.
```

```
    Best IP function value          -141278.0000000000000000

--- Branch & Bound converged! Iterations (nodes visited) =    94 Total LP Iterations =     93

    Optimal Objective function =    -141278.0000000000000000
x: 0  0  1 -0  1  1  1  1  0  1  0  1  1  1
   0  0  0  0  1  0  1  0  1  1  0  1  0  0
B: 1 1 -1 1 -1 -1 -1 -1 1 -1 0 -1 -1 -1 1 1 1 1 -1 1 -1 0 -1 -1 1 -1 -1 1
diary off
```

Note that there is a large difference in the number of iterations if the additional heuristic and priorities are used. Similar results are obtained if running the other two tree search strategies.

6 Solving Unconstrained and Constrained Optimization Problems

This section describes how to define and solve unconstrained and constrained optimization problems. Several examples are given on how to proceed, depending on if a quick solution is wanted, or more advanced runs are needed. See Appendix E for more information on your external solvers.

All demonstration examples that are using the TOMLAB format are collected in the directory *examples*. It is also possible to run each example separately. The examples relevant to this section are *ucDemo* and *conDemo*. The full path to these files are always given in the text. Throughout this section the test problem *Rosenbrock's banana*,

$$\begin{array}{lrcccl} \min\limits_{x} & f(x) = \alpha \left(x_2 - x_1^2\right)^2 + (1 - x_1)^2 & & & & \\ s/t & -10 & \leq & x_1 & \leq & 2 \\ & -10 & \leq & x_2 & \leq & 2 \end{array} \tag{17}$$

is used to illustrate the solution of unconstrained problems. The standard value is $\alpha = 100$. In this formulation simple bounds are added on the variables, and also constraints in illustrative purpose. This problem is called *RB BANANA* in the following descriptions to avoid mixing it up with problems already defined in the problem definition files.

6.1 Defining the Problem in Matlab m-files

TOMLAB demands that the general nonlinear problem is defined in Matlab m-files, and not given as an input text string. A file defining the function to be optimized must always be supplied. For linear constraints the constraint coefficient matrix and the right hand side vector are given directly. Nonlinear constraints are defined in a separate file. First order derivatives and second order derivatives, if available, are stored in separate files, both function derivatives and constraint derivatives.

TOMLAB is compatible with MathWorks Optimization TB, which in various ways demands both functions, derivatives and constraints to be returned by the same function. TOMLAB handle all this by use of interface routines, hidden for the user.

It is generally recommended to use the TOMLAB format instead, because having defined the files in this format, all MathWorks Optimization TB solvers are accessible through the TOMLAB multi-solver driver routines.

The rest of this section shows how to make the m-files for the cases of unconstrained and constrained optimization. In Section 6.2 and onwards similar m-files are used to solve unconstrained optimization using the TOMLAB format.

The most simple way to write the m-file to compute the function value is shown for the example in (17) assuming $\alpha = 100$:

File: tomlab/usersguide/rbbs_f.m

```
% rbbs_f - function value for Constrained Rosenbrocks Banana
%
% function f = rbbs_f(x)

function f = rbbs_f(x)

f = 100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
```

Running TOMLAB it is recommended to use a more general format for the m-files, adding one extra parameter,

the *Prob* problem definition structure described in detail in Appendix A. TOMLAB will handle the simpler format also, but the more advanced features of TOMLAB are not possible to use.

If using this extra parameter, then any information needed in the low-level function computation routine may be sent as fields in this structure. For single parameter values, like the above α parameter in the example, the field *Prob.user* is recommended.

Using the above convention, then the new m-file for the example in (17) is defined as

File: tomlab/usersguide/rbb_f.m

```
% rbb_f - function value for Rosenbrocks Banana, Problem RB BANANA
%
% function f = rbb_f(x, Prob)

function f = rbb_f(x, Prob)

alpha = Prob.user.alpha;

f = alpha*(x(2)-x(1)^2)^2 + (1-x(1))^2;
```

The value in the field *Prob.user* is the α value. It is defined before calling the solver by explicit setting the *Prob* structure. In a similar way the gradient routine is defined as

File: tomlab/usersguide/rbb_g.m

```
% rbb_g - gradient vector for Rosenbrocks Banana, Problem RB BANANA
%
% function g = rbb_g(x, Prob)

function g = rbb_g(x, Prob)

alpha = Prob.user.alpha;

g = [-4*alpha*x(1)*(x(2)-x(1)^2)-2*(1-x(1)); 2*alpha*(x(2)-x(1)^2)];
```

If the gradient routine is not supplied, TOMLAB will use finite differences (or automatic differentiation) if the gradient vector is needed for the particular solver. In this case it is also easy to compute the Hessian matrix, which gives the following code

File: tomlab/usersguide/rbb_H.m

```
% rbb_H - Hessian matrix for Rosenbrocks Banana, Problem RB BANANA
%
% function H = crbb_H(x, Prob)

function H = rbb_H(x, Prob)

alpha = Prob.user.alpha;
```

```
H = [ 12*alpha*x(1)^2-4*alpha*x(2)+2 , -4*alpha*x(1);
                -4*alpha*x(1)        ,    2*alpha  ];
```

If the Hessian routine is not supplied, TOMLAB will use finite differences (or automatic differentiation) if the Hessian matrix is needed for the particular solver. Often a positive-definite approximation of the Hessian matrix is estimated during the optimization , and the second derivative routine is then not used.

If using the constraints defined for the example in (17) then a constraint routine needs to defined for the single nonlinear constraint, in this case

File: tomlab/usersguide/rbb_c.m

```
% rbb_c - nonlinear constraint vector for Rosenbrocks Banana, Problem RB BANANA
%
% function c = rbb_c(x, Prob)

function cx = rbb_c(x, Prob)

cx = -x(1)^2 - x(2);
```

The constraint Jacobian matrix is also of interest and is defined as

File: tomlab/usersguide/rbb_dc.m

```
% rbb_dc - nonlinear constraint gradient matrix
%          for Rosenbrocks Banana, Problem RB BANANA
%
% function dc = crbb_dc(x, Prob)

function dc = rbb_dc(x, Prob)

% One row for each constraint, one column for each variable.

dc = [-2*x(1),-1];
```

If the constraint Jacobian routine is not supplied, TOMLAB will use finite differences (or automatic differentiation) to estimate the constraint Jacobian matrix if it is needed for the particular solver.

The solver *nlpSolve* is also using the second derivatives of the constraint vector. The result is returned as a weighted sum of the second derivative matrices with respect to each constraint vector element, the weights being the Lagrange multipliers supplied as input to the routine. For the example problem the routine is defined as

File: tomlab/usersguide/rbb_d2c.m

```
% rbb_d2c - The second part of the Hessian to the Lagrangian function for the
%     nonlinear constraints for Rosenbrocks Banana, Problem RB BANANA, i.e.
%
%           lam' * d2c(x)
%
```

```
% in
%
%   L(x,lam) =   f(x) - lam' * c(x)
% d2L(x,lam) = d2f(x) - lam' * d2c(x) = H(x) - lam' * d2c(x)
%
% function d2c=crbb_d2c(x, lam, Prob)

function d2c=rbb_d2c(x, lam, Prob)

% The only nonzero element in the second derivative matrix for the single
% constraint is the (1,1) element, which is a constant -2.

d2c = lam(1)*[-2 0; 0 0];
```

6.1.1 Communication between user routines

It is often the case that mathematical expressions that occur in the function computation also is part of the gradient and Hessian computation. If these operations are costly it is natural to avoid recomputing these and reuse them when computing the gradient and Hessian.

The function routine is always called before the gradient routine in TOMLAB, and the gradient routine is always called before the Hessian routine. The constraint routine is similarly called before the computation of the constraint gradient matrix. However, the TOM solvers call the function before the constraint routine, but the SOL solvers do the reverse.

Thus it is safe to use global variables to communicate information from the function routine to the gradient and Hessian, and similarly from the constraint routine to the constraint gradient routine. Any non-conflicting name could be used as global variable, see Table 154 in Appendix D to find out which names are in use. However, the recommendation is to always use a predefined global variable named *US_A* for this communication. TOMLAB is designed to handle recursive calls, and any use of new global variables may cause conflicts. The variable *US_A* (and also *US_B*) is automatically saved in a stack, and any level of recursions may safely be used. The user is free to use *US_A* both as variable, and as a structure. If much information is to be communicated, defining *US_A* as a structure makes it possible to send any amount of information between the user routines.

In the *examples* directory the constrained optimization example in *conDemo* is using the defined functions *con1_f*, *con1_g* and *con1_H*. They include an example of communicating one exponential expression between the routines.

The *lsDemo* example file in the *examples* directory communicates two exponential expressions between *ls1_r* and *ls1_J* with the use of *US_A* and *US_B*. In *ls1_r* the main part is

```
...
global US_A

t  = Prob.LS.t(:);

% Exponential computations takes time, and may be done once, and
% reused when computing the Jacobian
US_A = exp(-x(1)*t);
US_B = exp(-x(2)*t);
```

```
r = K*x(1)*(US_B - US_A) / (x(3)*(x(1)-x(2))) - Prob.LS.y;
```

In *ls1_J* then *US_A* is used

```
...
global US_A
% Pick up the globally saved exponential computations
e1 = US_A;
e2 = US_B;

% Compute the three columns in the Jacobian, one for each of variable
J = a * [ t.*e1+(e2-e1)*(1-1/b), -t.*e2+(e2-e1)/b, (e1-e2)/x(3)];
```

For more discussions on global variables and the use of recursive calls in TOMLAB, see Appendix D.

In the following sections it is described how to setup problems in TOMLAB and use the defined m-files. First comes the simplest way, to use the TOMLAB format.

6.2 Unconstrained Optimization Problems

The use of the TOMLAB format is best illustrated by examples

The following is the first example in the *ucDemo* demonstration file. It shows an example of making a call to *conAssign* to create a structure in the TOMLAB format, and solve the problem with a call to *ucSolve*.

```
% ----------------------------------------------------------------------
function uc1Demo
% ----------------------------------------------------------------------

format compact
fprintf('=======================================================\n');
fprintf('Rosenbrocks banana with explicit f(x), g(x) and H(x)\n');
fprintf('=======================================================\n');

Name    = 'RB Banana';
x_0     = [-1.2 1]';   % Starting values for the optimization.
x_L     = [-10;-10];   % Lower bounds for x.
x_U     = [2;2];       % Upper bounds for x.
fLowBnd = 0;           % Lower bound on function.

% Generate the problem structure using the TOMLAB format (short call)
Prob = conAssign('uc1_f', 'uc1_g', 'uc1_H', [], x_L, x_U, Name, ...
x_0, [], fLowBnd);
```

```
Result  = tomRun('ucSolve', Prob, 1);
```

In its more general form *conAssign* is used to define constrained problems. It also takes as input the nonzero pattern of the Hessian matrix, stored in the matrix *HessPattern*. In this case all elements of the Hessian matrix are nonzero, and either *HessPattern* is set as empty or as a matrix with all ones. Also the parameter *pSepFunc* should be set. It defines if the objective function is partially separable, see Section 14.5. Setting this parameter empty (the default), then this property is not used. In the above example the call would be

```
...
HessPattern = ones(2,2);   % The pattern of nonzeros
pSepFunc    = [];          % No partial separability used

% conAssign is used to generate the TOMLAB problem structure
Prob    = conAssign('uc1_f', 'uc1_g', 'uc1_H', HessPattern, ...
                    x_L, x_U, Name, x_0, pSepFunc, fLowBnd);
...
```

Also see the other examples in *ucDemo* on how to solve the problem, when gradients routines are not present, and numerical differentiation must be used. An example on how to solve a sequence of problems is also presented.

If the gradient is not possible to define one just sets the corresponding gradient function name empty.

The example *uc3Demo* in file *ucDemo* show how to solve a sequence of problems in TOMLAB, in this case changing the steepness parameter α in (17). It is important to point out that it is only necessary to define the *Prob* structure once and then just change the varying parameters, in this case the α value. The α value is sent to the user routines using the field *user* in the *Prob* structure. Any field in the *Prob* structure could be used that is not conflicting with the predefined fields. In this example the a vector of *Result* structures are saved for later preprocessing.

```
% ----------------------------------------------------------------------
function uc3Demo - Sequence of Rosenbrocks banana functions
% ----------------------------------------------------------------------

% conAssign is used to generate the TQ problem structure
% Prob = conAssign(f,g,H, HessPattern, x_L, x_U, Name, x_0, pSepFunc, fLowBnd);

Prob = conAssign('uc3_f',[],[],[],[-10;-10], [2;2], [-1.2;1], 'RB Banana',[],0)

% The different steepness parameters to be tried
Steep = [100 500 1000 10000];

for i = 1:4
    Prob.user.alpha      = Steep(i);
    Result(i)            = tomRun('ucSolve', Prob, 1);
end
```

6.3 Direct Call to an Optimization Routine

When wanting to solve a problem by a direct call to an optimization routine there are two possible ways of doing it. The difference is in the way the problem dependent parameters are defined. The most natural way is to use a Init File, like the predefined TOMLAB Init Files $\diamond$*_prob* (e.g. *uc_prob* if the problem is of the type unconstrained) to define those parameters. The other way is to call the routine *conAssign*. In this subsection, examples of two different approaches are given.

First, solve the problem *RB BANANA* in (17) as an unconstrained problem. In this case, define the problem in the files *ucnew_prob*, *ucnew_f*, *ucnew_g* and *ucnew_H*. Using the problem definition files in the working directory solve the problem and print the result by the following calls.

File: tomlab/usersguide/ucnewSolve1.m

```
probFile = 'ucnew_prob';       % Problem definition file.
P = 18;                        % Problem number for the added RB Banana.
Prob = probInit(probFile, P);  % Setup Prob structure.

Result = tomRun('ucSolve', Prob, 1);
```

Now, solve the same problem as in the example above but define the parameters *x_0*, *x_L* and *x_L* by calling the routine *conAssign*. Note that in this case the file *ucnew_prob* is not used, only the files *ucnew_f* and *ucnew_g*. The file *ucnew_H* is not needed because a quasi-Newton BFGS algorithm is used.

File: tomlab/usersguide/ucnewSolve2.m

```
x_0     = [-1.2;1];    % Starting values for the optimization.
x_L     = [-10;-10];   % Lower bounds for x.
x_U     = [2;2];       % Upper bounds for x.
Prob = conAssign('ucnew_f','ucnew_g', [], [], x_L, x_U,...
'ucNew', x_0);

Prob.P  = 18;                          % Problem number.
Prob.Solver.Alg=1;                     % Use quasi-Newton BFGS
Prob.user.uP = 100;                    % Set alpha parameter
Result  = tomRun('ucSolve',Prob,1);
```

6.4 Constrained Optimization Problems

Study the following constrained exponential problem, *Exponential problem III*,

$$\begin{array}{lrcccl} \min\limits_{x} & f(x) = exp(x_1)(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1) & & & & \\ & -10 & \leq & x_1 & \leq & 10 \\ & -10 & \leq & x_2 & \leq & 10 \\ s/t & 0 & \leq & x_1 + x_2 & \leq & 0 \\ & 1.5 & \leq & -x_1x_2 + x_1 + x_2 & & \\ & -10 & \leq & x_1x_2 & & \end{array} \quad . \tag{18}$$

The first two constraints are simple bounds, the third is a linear equality constraint, because lower and upper bounds are the same. The last two constraints are nonlinear inequality constraints. To solve the problem, define the following statements, available as *con1Demo* in file *conDemo*.

```
Name    = 'Exponential problem III';
A       = [1 1];        % One linear constraint
b_L     = 0;            % Lower bound on linear constraint
b_U     = 0;            % b_L == b_U implies equality
c_L     = [1.5;-10]     % Two nonlinear inequality constraints
c_U     = [];           % Empty means Inf (default) for the two constraints
x_0     = [-5;5];       % Initial value for x
x_L     = [-10;-10];    % Lower bounds on x
x_U     = [10;10];      % Upper bounds on x
fLowBnd = 0;            % A lower bound on the optimal function value
x_min   = [-2;-2];      % Used for plotting, lower bounds
x_max   = [4;4];        % Used for plotting, upper bounds

x_opt=[-3.162278, 3.162278; -1.224745, 1.224745]; % Two stationary points
f_opt=[1.156627; 1.8951];

HessPattern = [];  % All elements in Hessian are nonzero.
ConsPattern = [];  % All elements in the constraint Jacobian are nonzero.
pSepFunc    = [];  % The function f is not defined as separable

% Generate the problem structure using the TOMLAB format
Prob = conAssign('con1_f', 'con1_g', 'con1_H', HessPattern, x_L, x_U, ...
          Name, x_0, pSepFunc, fLowBnd, A, b_L, b_U,  'con1_c', 'con1_dc',...
          [], ConsPattern, c_L, c_U, x_min, x_max, f_opt, x_opt);

Result  = tomRun('conSolve',Prob);
PrintResult(Result);
```

The following example, *con2Demo* in file *conDemo*, illustrates numerical estimates of the gradient and constrained Jacobian matrix. Only the statements different from the previous example is given. Note that the gradient routine is not given at all, but the constraint Jacobian routine is given. Setting *Prob.ConsDiff* greater than zero will overrule the use of the constraint Jacobian routine. The solver *conSolve* is in this case called directly.

```
% Generate the problem structure using the TOMLAB format
Prob = conAssign('con1_f', [], [], HessPattern,  x_L, x_U, Name, x_0, ...
          pSepFunc, fLowBnd, A, b_L, b_U, 'con1_c', 'con1_dc', [], ...
          ConsPattern, c_L, c_U, x_min, x_max, f_opt, x_opt);

Prob.NumDiff  = 1;   % Use standard numerical differences
Prob.ConsDiff = 5;   % Use the complex variable method to estimate derivatives

Prob.Solver.Alg = 0; % Use default algorithm in conSolve

Result  = tomRun('conSolve', Prob, 1);
```

The third example, *con3Demo* in file *conDemo*, shows how to solve the same problem for a number of different initial values on x. The initial values are stored in the matrix $X0$, and in each loop step *Prob.x_0* is set to one of the columns in $X0$. In a similar way any of the values in the Prob structure may be changed in a loop step, if e.g. the

loop is part of a control loop. The Prob structure only needs to be defined once. The different initial points reveal that this problem is nasty, and that several points fulfill the convergence criteria. Only the statements different from the previous example is given. A different solver is called dependent on which TOMLAB version is used.

```
X0              = [ -1 -5  1 0 -5 ;
                    1  7 -1 0  5];

% Generate the problem structure using the TOMLAB format
Prob = conAssign('con1_f', 'con1_g', 'con1_H', HessPattern, x_L, x_U, Name, ...
          X0(:,1), pSepFunc, fLowBnd, A, b_L, b_U, 'con1_c', 'con1_dc',...
          [], ConsPattern, c_L, c_U, x_min, x_max, f_opt, x_opt);

Prob.Solver.Alg = 0;
TomV            = tomlabVersion;

for i = 1:size(X0,2)
    Prob.x_0 = X0(:,i);

    if TomV(1:1) ~= 'M'
       % Users of v3.0 may instead call MINOS (or SNOPT, NPSOL in v3.0 /SOL)
       Result   = tomRun('minos',Prob, 2);
    else
       Result   = tomRun('conSolve',Prob, 2);
    end
end
```

The constrained optimization solvers all have proven global convergence to a local minimum. If the problem is not convex, then it is always difficult to assure that a global minimum has been reached. One way to make it more likely that the global minimum is found is to optimize very many times with different initial values. The fifth example, *con5Demo* in file *conDemo* illustrates this approach by solving the exponential problem 50 times with randomly generated initial points.

If the number of variables are not that many, say fifteen, another approach is to use a global optimization solver like *glcSolve* to crunch the problem and search for the global minimum. If letting it run long enough, it is very likely to find the global minimum, but maybe not with high precision. To run *glcSolve* the problem must be box-bounded, and the advise is to try to squeeze the size of the box down as much as possible. The sixth example, *con6Demo* in file *conDemo*, illustrates a call to *glcSolve*. It is very simple to do this call if the problem has been defined in the TOMLAB format. The statements needed are the following

```
Prob.optParam.MaxFunc = 5000; % Define maximal number of function evaluations
Result                = tomRun('glcSolve',Prob,2);
```

A more clever approach, using warm starts and successive checks on the best function value obtained, is discussed in Section 7. It is also better to use *glcAssign* and not *conAssign* if the intension is to use global optimization.

6.5 Efficient use of the TOMLAB solvers

To follow the iterations in the TOMLAB Base Module solvers, it is useful to set the *IterPrint* parameter as true. This gives one line of information for each iteration. This parameter is part of the *optParam* subfield:

```
Prob.optParam.IterPrint = 1;
```

Note that *ucSolve* implements a whole set of methods for unconstrained optimization. If the user explicitly wants Newtons method to be used, utilizing second order information, then set

```
Prob.Solver.Alg=1;            % Use Newtons Method
```

But *ucSolve* will switch to the default BFGS method if no routine has been given to return the Hessian matrix. If the user still wants to run Newtons method, then the Hessian routine must be defined and return an empty Hessian. That triggers a numerical estimation of the Hessian. Do *help ucSolve* to see the different algorithmic options and other comments on how to run the solver.

Both *ucSolve* and *conSolve* use line search based methods. The parameter σ influences the accuracy of the line search each step. The default value is

```
Prob.LineParam.sigma = 0.9;   % Inaccurate line search
```

However, using the conjugate gradient methods in *ucSolve*, they benefit from a more accurate line search

```
Prob.LineParam.sigma = 0.01;   % Default accurate line search for C-G methods
```

as do quasi-Newton DFP methods (default $\sigma = 0.2$). The test for the last two cases are made for $\sigma = 0.9$. If the user really wishes these methods to use $\sigma = 0.9$, the value must be set slightly different to fool the test:

```
Prob.LineParam.sigma = 0.9001; % Avoid the default value for C-G methods
```

The choice of line search interpolation method is also important, a cubic or quadratic interpolation. The default is to use cubic interpolation.

```
Prob.LineParam.LineAlg = 1; % 0 = quadratic, 1 = cubic
```

7 Solving Global Optimization Problems

Global Optimization deals with optimization problems that might have more than one local minimum. To find the global minimum out of a set of local minimum demands other types of methods than for the problem of finding local minimum. The TOMLAB routines for global optimization are based on using only function or constraint values, and no derivative information. Two different types are defined, Box-bounded global optimization **glb** and global mixed-integer nonlinear programming **glc**. For the second case, still the problem should be box-bounded.

All demonstration examples that are using the TOMLAB format are collected in the directory *examples*. Running the menu program *tomMenu*, it is possible to run all demonstration examples. It is also possible to run each example separately. The examples relevant to this section are *glbDemo* and *glcDemo*.

7.1 Box-Bounded Global Optimization Problems

Box-bounded global optimization problems are simple to define, only one function routine is needed, because the global optimization routines in TOMLAB does not utilize information about derivatives. To define the *Shekel 5* test problem in a routine *glb1_f*, the following statements are needed

```
function f = glb1_f(x, Prob)

A = [ 4 4 4 4; 1 1 1 1; 8 8 8 8; 6 6 6 6; 3 7 3 7]';
f=0;   c = [.1 .2 .2 .4 .4]';
for i = 1:5
    z = x-A(:,i);
    f = f - 1/(z'*z + c(i) ); % Shekel 5
end
```

To solve the *Shekel 5* test problem define the following statements, available as *glb1Demo* in *glbDemo*.

```
function glb1Demo

Name   = 'Shekel 5';
x_L    = [ 0  0  0  0]'; % Lower bounds in the box
x_U    = [10 10 10 10]'; % Upper bounds in the box

% Generate the problem structure using the TOMLAB format (short call)
Prob   = glcAssign('glb1_f', x_L, x_U, Name);
Result = tomRun('glbSolve', Prob, 1); % Solve using the default of 200 iterations
```

If the user knows the optimal function value or some good approximation, it could be set as a target for the optimization, and the solver will stop if the target value is achieved within a relative tolerance. For the *Shekel 5* problem, the optimal function value is known and could be set as target value with the following statements.

```
Prob.optParam.fGoal = -10.1532; % The optimal value set as target
Prob.optParam.eps_f =   0.01;   % Convergence tolerance one percent
```

Convergence will occur if the function value sampled is within one percent of the optimal function value.

Without additional knowledge about the problem, like the function value at the optimum, there is no convergence criteria to be used. The global optimization routines continues to sample points until the maximal number of function evaluations or the maximum number of iteration cycles are reached. In practice, it is therefore important to be able to do warm starts, starting once again without having to recompute the past history of iterations and function evaluations. Before doing a new warm start, the user can evaluate the results and determine if to continue or not. If the best function value has not changed for long it is a good chance that there are no better function value to be found.

In TOMLAB warm starts are automatically handled, the only thing the user needs to do is to set one flag, *Prob.WarmStart*, as true. The solver *glbSolve* defines a binary *mat*-file called *glbSave.mat* to store the information needed for a warm start. It is important to avoid running other problems with this solver when doing warm starts. The warm start information would then be overwritten. The example *glb3Demo* in *glbDemo* shows how to do warm starts. The number of iterations per call is set very low to be able to follow the process.

```
Name   = 'Shekel 5';
x_L    = [ 0  0  0  0]';
x_U    = [10 10 10 10]';

% Generate the problem structure using the TOMLAB format (short call)
Prob                  = glcAssign('glb1_f', x_L, x_U, Name);
Prob.optParam.MaxIter = 5; % Do only five iterations per call
Result                = tomRun('glbSolve',Prob,2); pause(1)
Prob.WarmStart        = 1; % Set the flag for warm start
for i = 1:6                % Do 6 warm starts
    Result = tomRun('glbSolve',Prob,2); pause(1)
end
```

The example *glb4Demo* in *glbDemo* illustrates how to send parameter values down to the function routine from the calling routine. Change the *Shekel 5* test problem definition so that A and c are given as input to the function routine

```
function f = glb4_f(x, Prob)

% A and c info are sent using Prob structure
f = 0; A = Prob.user.A; c = Prob.user.c;
for i = 1:5
    z = x-A(:,i);
    f = f - 1/(z'*z + c(i) );  % Shekel 5
end
```

Then the following statements solve the *Shekel 5* test problem.

```
Name   = 'Shekel 5';
x_L    = [ 0  0  0  0]';
x_U    = [10 10 10 10]';

% Generate the problem structure using the TOMLAB format (short call)
Prob   = glcAssign('glb4_f', x_L, x_U, Name);
```

```
% Add information to be sent to glb4_f. Used in f(x) computation
Prob.user.A = [4 4 4 4;1 1 1 1;8 8 8 8;6 6 6 6;3 7 3 7]';
Prob.user.c = [.1 .2 .2 .4 .4]';

Result = tomRun('glbSolve',Prob,2);
```

7.2 Global Mixed-Integer Nonlinear Problems

To solve global mixed-integer nonlinear programming problems with the TOMLAB format, only two routines need to be defined, one routine that defines the function and one that defines the constraint vector. No derivative information is utilized by the TOMLAB solvers. To define the *Floudas-Pardalos 3.3* test problem, one routine *glc1_f*

```
function f = fp3_3f(x, Prob)
f = -25*(x(1)-2)^2-(x(2)-2)^2-(x(3)-1)^2-(x(4)-4)^2-(x(5)-1)^2-(x(6)-4)^2;
```

and one routine *glc1_c*

```
function c = fp3_3c(x, Prob)
c = [(x(3)-3)^2+x(4); (x(5)-3)^2+x(6)]; % Two nonlinear constraints (QP)
```

is needed. Below is the example *glc1Demo* in *glcDemo* that shows how to solve this problem doing ten warm starts. The warm starts are automatically handled, the only thing the user needs to do is to set one flag as true, *Prob.WarmStart*. The solver *glcSolve* defines a binary *mat*-file called *glcSave.mat* to store the information needed for the warm start. It is important to avoid running other problems with *glcSolve* when doing warm starts. Otherwise the warm start information will be overwritten with the new problem. The original *Floudas-Pardalos 3.3* test problem, has no upper bounds on x_1 and x_2, but such bounds are implicit from the third linear constraint, $x_1 + x_2 \leq 6$. This constraint, together with the simple bounds $x_1 \geq 0$ and $x_2 \geq 0$ immediately leads to $x_1 \leq 6$ and $x_2 \leq 6$. Using these inequalities a finite box-bounded problem can be defined.

```
Name   = 'Floudas-Pardalos 3.3';  % This example is number 16 in glc_prob.m

x_L   = [ 0  0 1 0 1 0]'; % Lower bounds on x
A =     [ 1 -3 0 0 0 0
         -1  1 0 0 0 0
          1  1 0 0 0 0];  % Linear equations
b_L   = [-inf -inf  2 ]'; % Upper bounds for linear equations
b_U   = [  2    2   6 ]'; % Lower bounds for linear equations
x_U   = [6 6 5 6 5 10]';  % Upper bounds after x(1),x(2) values inserted
c_L   = [4 4]';           % Lower bounds on two nonlinear constraints
c_U   = [];               % Upper bounds are infinity for nonlinear constraints
x_opt = [5 1 5 0 5 10]';  % Optimal x value
f_opt = -310;             % Optimal f(x) value
x_min = x_L; x_max = x_U; % Plotting bounds

% Set the rest of the arguments as empty
IntVars = []; VarWeight = [];
```

```
fIP     = []; xIP = []; fLowBnd = []; x_0       = [];

%IntVars = [1:5];  % Indices of the variables that should be integer valued

Prob = glcAssign('glc1_f', x_L, x_U, Name, A, b_L, b_U, 'glc1_c', ...
          c_L, c_U, x_0, IntVars, VarWeight, ...
          fIP, xIP, fLowBnd, x_min, x_max, f_opt, x_opt);

% Increase the default max number of function evaluations in glcSolve
Prob.optParam.MaxFunc = 500;

Result = tomRun('glcSolve', Prob, 3);

Prob.WarmStart = 1;
% Do 10 restarts, call driver tomRun, PriLev = 2 gives call to PrintResult
for i=1:10
    Result = tomRun('glcSolve',Prob,2);
end
```

8 Solving Least Squares and Parameter Estimation Problems

This section describes how to define and solve different types of linear and nonlinear least squares and parameter estimation problems. Several examples are given on how to proceed, depending on if a quick solution is wanted, or more advanced tests are needed. TOMLAB is also compatible with MathWorks Optimization TB. See Appendix E for more information and test examples.

All demonstration examples that are using the TOMLAB format are collected in the directory *examples*. The examples relevant to this section are *lsDemo* and *llsDemo*. The full path to these files are always given in the text.

Section 8.5 (page 81) contains information on solving extreme large-scale **ls** problems with *Tlsqr*.

8.1 Linear Least Squares Problems

This section shows examples how to define and solve linear least squares problems using the TOMLAB format. As a first illustration, the example *lls1Demo* in file *llsDemo* shows how to fit a linear least squares model with linear constraints to given data. This test problem is taken from the Users Guide of *LSSOL* [29].

```
Name='LSSOL test example';

% In TOMLAB it is best to use Inf and -Inf, not big numbers.
n = 9;  % Number of unknown parameters
x_L = [-2 -2 -Inf, -2*ones(1,6)]';
x_U = 2*ones(n,1);

A   = [ ones(1,8) 4; 1:4,-2,1 1 1 1; 1 -1 1 -1, ones(1,5)];
b_L = [2    -Inf -4]';
b_U = [Inf    -2 -2]';

y = ones(10,1);
C = [ ones(1,n); 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];

x_0 = 1./[1:n]';

t          = [];   % No time set for y(t) (used for plotting)
weightY    = [];   % No weighting
weightType = [];   % No weighting type set
x_min      = [];   % No lower bound for plotting
x_max      = [];   % No upper bound for plotting

Prob = llsAssign(C, y, x_L, x_U, Name, x_0, t, weightType, weightY, ...
                 A, b_L, b_U,  x_min, x_max);

Result  = tomRun('lsei',Prob,2);
```

It is trivial to change the solver in the call to *tomRun* to a nonlinear least squares solver, e.g. *clsSolve*, or a general nonlinear programming solver.

8.2 Linear Least Squares Problems using the SOL Solver LSSOL

The example *lls2Demo* in file *llsDemo* shows how to fit a linear least squares model with linear constraints to given data using a direct call to the SOL solver *LSSOL*. The test problem is taken from the Users Guide of *LSSOL* [29].

```
% Note that when calling the LSSOL MEX interface directly, avoid using
% Inf and -Inf. Instead use big numbers that indicate Inf.
% The standard for the MEX interfaces is 1E20 and -1E20, respectively.

n = 9; % There are nine unknown parameters, and 10 equations
x_L = [-2 -2 -1E20, -2*ones(1,6)]';
x_U = 2*ones(n,1);

A   = [ ones(1,8) 4; 1:4,-2,1 1 1 1; 1 -1 1 -1, ones(1,5)];
b_L = [2    -1E20 -4]';
b_U = [1E20    -2 -2]';
% Must put lower and upper bounds on variables and constraints together
bl = [x_L;b_L];
bu = [x_U;b_U];

H   = [ ones(1,n); 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];
y   = ones(10,1);

x_0 = 1./[1:n]';

% Set empty indicating default values for most variables
c          = [];          % No linear coefficients, they are for LP/QP
Warm       = [];          % No warm start
iState     = [];          % No warm start
Upper      = [];          % C is not factorized
kx         = [];          % No warm start
SpecsFile  = [];          % No parameter settings in a SPECS file
PriLev     = [];          % PriLev is not really used in LSSOL
ProbName   = [];          % ProbName is not really used in LSSOL
optPar(1)  = 50;          % Set print level at maximum
PrintFile  = 'lssol.txt'; % Print result on the file with name lssol.txt

z0 = (y-H*x_0);
f0 = 0.5*z0'*z0;
fprintf('Initial function value %f\n',f0);
```

```
[x, Inform, iState, cLamda, Iter, fObj, r, kx] = ...
    lssol( A, bl, bu, c, x_0, optPar, H, y, Warm, ...
           iState, Upper, kx, SpecsFile, PrintFile, PriLev, ProbName );

% We could equally well call with the following shorter call:
% [x, Inform, iState, cLamda, Iter, fObj, r, kx] = ...
%     lssol( A, bl, bu, c, x, optPar, H, y);

z = (y-H*x);
f = 0.5*z'*z;
fprintf('Optimal function value %f\n',f);
```

8.3 Nonlinear Least Squares Problems

This section shows examples how to define and solve nonlinear least squares problems using the TOMLAB format. As a first illustration, the example *ls1Demo* in file *lsDemo* shows how to fit a nonlinear model of exponential type with three unknown parameters to experimental data. This problem, *Gisela*, is also defined as problem three in *ls_prob*. A weighting parameter K is sent to the residual and Jacobian routine using the *Prob* structure. The solver *clsSolve* is called directly. Note that the user only defines the routine to compute the residual vector and the Jacobian matrix of derivatives. TOMLAB has special routines *ls_f*, *ls_g* and *ls_H* that computes the nonlinear least squares objective function value, given the residuals, as well as the gradient and the approximative Hessian, see Table 39. The residual routine for this problem is defined in file *ls1_r* in the directory *example* with the statements

```
function r = ls_r(x, Prob)

% Compute residuals to nonlinear least squares problem Gisela

% US_A is the standard TOMLAB global parameter to be used in the
% communication between the residual and the Jacobian routine

global US_A

% The extra weight parameter K is sent as part of the structure
K  = Prob.user.K;
t  = Prob.LS.t(:);     % Pick up the time points

% Exponential expressions to be later used when computing the Jacobian
US_A.e1 = exp(-x(1)*t); US_A.e2 = exp(-x(2)*t);

r = K*x(1)*(US_A.e2 - US_A.e1) / (x(3)*(x(1)-x(2))) - Prob.LS.y;
```

Note that this example also shows how to communicate information between the residual and the Jacobian routine. It is best to use any of the predefined global variables *US_A* and *US_B*, because then there will be no conflicts with respect to global variables if recursive calls are used. In this example the global variable *US_A* is used as structure array storing two vectors with exponential expressions. The Jacobian routine for this problem is defined in the file *ls1_J* in the directory *example*. The global variable *US_A* is accessed to obtain the exponential expressions, see the statements below.

```
function J = ls1_J(x, Prob)

% Computes the Jacobian to least squares problem Gisela. J(i,j) is dr_i/d_x_j

% Parameter K is input in the structure Prob
a  = Prob.user.K * x(1)/(x(3)*(x(1)-x(2)));
b  = x(1)-x(2);
t = Prob.LS.t;

% Pick up the globally saved exponential computations
global US_A
e1 = US_A.e1; e2 = US_A.e2;

% Compute the three columns in the Jacobian, one for each of variable
J = a * [ t.*e1+(e2-e1)*(1-1/b), -t.*e2+(e2-e1)/b, (e1-e2)/x(3)];
```

The following statements solve the *Gisela* problem.

```
% ----------------------------------------------------------------------
function ls1Demo - Nonlinear parameter estimation with 3 unknowns
% ----------------------------------------------------------------------

Name='Gisela';

% Time values
t  = [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];

% Observations
y = [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];

x_0 = [6.8729,0.0108,0.1248]'; % Initial values for unknown x

% Generate the problem structure using the TOMLAB format (short call)
% Prob = clsAssign(r, J, JacPattern, x_L, x_U, Name, x_0, ...
%                  y, t, weightType, weightY, SepAlg, fLowBnd, ...
%                  A, b_L, b_U, c, dc, ConsPattern, c_L, c_U, ...
%                  x_min, x_max, f_opt, x_opt);

Prob = clsAssign('ls1_r', 'ls1_J', [], [], [], Name, x_0, y, t);

% Weighting parameter K in model is sent to r and J computation using Prob
Prob.user.K = 5;

Result  = tomRun('clsSolve', Prob, 2);
```

The second example *ls2Demo* in file *lsDemo* solves the same problem as *ls1Demo*, but using numerical differences to compute the Jacobian matrix in each iteration. To make TOMLAB avoid using the Jacobian routine, the variable *Prob.NumDiff* has to be set nonzero. Also in this example the flag *Prob.optParam.IterPrint* is set to enable one line of printing for each iteration. The changed statements are

```
...
Prob.NumDiff            = 1; % Use standard numerical differences
Prob.optParam.IterPrint = 1; % Print one line each iteration

Result  = tomRun('clsSolve',Prob,2);
```

The third example *ls3Demo* in file *lsDemo* solves the same problem as *ls1Demo*, but six times for different values of the parameter K in the range $[3.8, 5.0]$. It illustrates that it is not necessary to remake the problem structure *Prob* for each optimization, but instead just change the parameters needed. The *Result* structure is saved as an vector of structure arrays, to enable post analysis of the results. The changed statements are

```
for i=1:6
    Prob.user.K = 3.8 + 0.2*i;

    Result(i)  = tomRun('clsSolve',Prob,2);

    fprintf('\nWEIGHT PARAMETER K is %9.3f\n\n\n',Prob.user.K);
end
```

Table 39 describes the low level routines and the initialization routines needed for the predefined constrained nonlinear least squares (**cls**) test problems. Similar routines are needed for the nonlinear least squares (**ls**) test problems (here no constraint routines are needed).

Table 39: Constrained nonlinear least squares (**cls**) test problems.

Function	**Description**
cls_prob	Initialization of **cls** test problems.
cls_r	Compute the residual vector $r_i(x), i = 1, ..., m$. $x \in \mathbb{R}^n$ for **cls** test problems.
cls_J	Compute the Jacobian matrix $J_{ij}(x) = \partial r_i / dx_j, i = 1, ..., m, j = 1, ..., n$ for **cls** test problems.
cls_c	Compute the vector of constraint functions $c(x)$ for **cls** test problems.
cls_dc	Compute the matrix of constraint normals $\partial c(x)/dx$ for **cls** test problems.
cls_d2c	Compute the second part of the second derivative of the Lagrangian function for **cls** test problems.
ls_f	General routine to compute the objective function value $f(x) = \frac{1}{2} r(x)^T r(x)$ for nonlinear least squares type of problems.
ls_g	General routine to compute the gradient $g(x) = J(x)^T r(x)$ for nonlinear least squares type of problems.
ls_H	General routine to compute the Hessian approximation $H(x) = J(x)^T * J(x)$ for nonlinear least squares type of problems.

8.4 Fitting Sums of Exponentials to Empirical Data

In TOMLAB the problem of fitting sums of positively weighted exponential functions to empirical data may be formulated either as a nonlinear least squares problem or a separable nonlinear least squares problem [66]. Several empirical data series are predefined and artificial data series may also be generated. There are five different types of exponential models with special treatment in TOMLAB, shown in Table 40. In research in cooperation with Todd Walton, Vicksburg, USA, TOMLAB has been used to estimate parameters using maximum likelihood in simulated Weibull distributions, and Gumbel and Gamma distributions with real data. TOMLAB has also been useful for parameter estimation in stochastic hydrology using real-life data.

Table 40: Exponential models treated in TOMLAB.

$f(t) = \sum_i^p \alpha_i e^{-\beta_i t},$	$\alpha_i \geq 0,$	$0 \leq \beta_1 < \beta_2 < ... < \beta_p.$
$f(t) = \sum_i^p \alpha_i (1 - e^{-\beta_i t}),$	$\alpha_i \geq 0,$	$0 \leq \beta_1 < \beta_2 < ... < \beta_p.$
$f(t) = \sum_i^p t\alpha_i e^{-\beta_i t},$	$\alpha_i \geq 0,$	$0 \leq \beta_1 < \beta_2 < ... < \beta_p.$
$f(t) = \sum_i^p (t\alpha_i - \gamma_i) e^{-\beta_i t},$	$\alpha_i, \gamma_i \geq 0,$	$0 \leq \beta_1 < \beta_2 < ... < \beta_p.$
$f(t) = \sum_i^p t\alpha_i e^{-\beta_i (t-\gamma_i)},$	$\alpha_i \geq 0,$	$0 \leq \beta_1 < \beta_2 < ... < \beta_p.$

Algorithms to find starting values for different number of exponential terms are implemented. Test results show that these initial value algorithms are very close to the true solution for equidistant problems and fairly good for non-equidistant problems, see the thesis by Petersson [61]. Good initial values are extremely important when solving real life exponential fitting problems, because they are so ill-conditioned. Table 41 shows the relevant routines.

Table 41: Exponential fitting test problems.

Function	**Description**
expAssign	Assign exponential fitting problem.
exp_ArtP	Generate artificial exponential sum problems.
expInit	Find starting values for the exponential parameters λ.
expSolve	Solve exponential fitting problems.
exp_prob	Defines a exponential fitting type of problem, with data series (t, y). The file includes data from several different empirical test series.
Helax_prob	Defines 335 medical research problems supplied by Helax AB, Uppsala, Sweden, where an exponential model is fitted to data. The actual data series (t, y) are stored on one file each, i.e. 335 data files, 8MB large, and are not distributed. A sample of five similar files are part of *exp_prob*.
exp_r	Compute the residual vector $r_i(x), i = 1, ..., m.\ x \in \mathbb{R}^n$
exp_J	Compute the Jacobian matrix $\partial r_i / dx_j, i = 1, ..., m, j = 1, ..., n.$
exp_d2r	Compute the 2nd part of the second derivative for the nonlinear least squares exponential fitting problem.
exp_c	Compute the constraints $\lambda_1 < \lambda_2 < ...$ on the exponential parameters $\lambda_i, i = 1, ..., p.$
exp_dc	Compute matrix of constraint normals for constrained exponential fitting problem.

Table 41: Exponential fitting test problems, continued

Function	Description
exp_d2c	Compute second part of second derivative matrix of the Lagrangian function for constrained exponential fitting problem. This is a zero matrix, because the constraints are linear.
exp_q	Find starting values for exponential parameters $\lambda_i, i = 1, ..., p$.
exp_p	Find optimal number of exponential terms, p.

The algorithmic development implemented in TOMLAB is further discussed in [49]. An overview of the field is also given in this reference.

8.5 Large Scale LS problems with Tlsqr

The *Tlsqr* MEX solver provides special parameters for advanced memory handling, enabling the user to solve extremely large linear least squares problems.

We'll take the problem of solving $Ax = b$ in the least squares sense as a prototype problem for this section. Here, $A \in \mathbb{R}^{m \times n}$ is a dense or sparse matrix and $b \in \mathbb{R}^m$.

Controlling memory allocation in *Tlsqr*

The normal mode of operation of *Tlsqr* is that memory for the A matrix is allocated and deallocated each time the solver is called. In a real-life situation with a very large A and where the solver is called repeatedly, this may become inefficient and even cause problems getting memory because of memory fragmenting.

The *Tlsqr* solver provides a parameter *Alloc*, given as the second element of the first input parameter to control the memory handling. The possible values of *Alloc* and their meanings are given in Table 42.

Table 42: *Alloc* values for *Tlsqr*

Alloc (m(2))	**Meaning**
0	Normal operation: allocate – solve – deallocate
1	Only allocate, no results returned
2	Allocate and solve, no deallocate
3	Only solve, no allocate/deallocate
4	Solve and deallocate
5	Deallocate only, no results returned

An example of the calling sequence is given below.

```
>> m = 60000; n = 1000; d = 0.01; % Size and density of A
>> A = sprand(m,n,d);             % Sparse random matrix
>> b = ones(m,1);                 % Right hand side
>> whos A

  Name      Size         Bytes  Class

  A     60000x500      3584784  sparse array

Grand total is 298565 elements using 3584784 bytes

% ========================================================================
% Simple standard call to Tlsqr, Alloc is set to default 0 if m is scalar

>> x=Tlsqr(m,n,A,[],[],b);

% ========================================================================
% To solve repeatedly with e.g. the same A but different b,
% the user may do:
```

```
% Indicate to Tlsqr to allocate and solve the problem

>> m(2) = 2
m =
       60000       2

>> x = Tlsqr(m,n,A,[],[],b);  % First solution

% Indicate to Tlsqr that memory is already allocated,
% and that no deallocation should occur on exit

>> m(2) = 3
m =
       60000       3

% Loop 100 times, calling Tlsqr each time - without re-allocation of memory

>> for k=1:100
>>     b = (...);                % E.g. alter the right hand side each time
>>     x = Tlsqr(m,n,A,[],[],b); % Call Tlsqr, now with m(2)=3
>> end

% Final call, with m(2) = 4: Solve and deallocate

>> m(2) = 4
m =
       60000       4

>> x=Tlsqr(m,n,A,[],[],b);

% Alternatively, to just deallocate, the user could do

>> m(2) = 5;
>> Tlsqr(m,n,A,[],[],b); % Nothing is returned
```

Further Memory Control: The maxneA Parameter

If the number of non-zero elements in a sparse A matrix increases in the middle of a *Tlsqr*-calling loop, the initially allocated space will not be sufficient. One solution is that the user checks this prior to calling *Tlsqr* and reallocating if necessary. The other solution is to set $m(3)$ to an upper limit ($maxneA$) of the number of nonzero elements in A in the first allocation call. For example:

```
>> m = [ 60000  1 1E6 ]

m =
    60000    1    1000000
```

will initiate a *Tlsqr* session, allocating sufficient memory to allow A matrices with up to 1.000.000 nonzeros. If the allocated memory is still insufficient, *Tlsqr* will try to reallocate enough space for the operation to continue.

Using Global Variables with Tlsqr and Tlsqrglob.m

For cases where it is not possible to send the A matrix to *Tlsqr* because it is simply too large, the user may choose to use the *tomlab/mex/Tlsqrglob.m* routine.

This function, which more often than not needs to be customized to the application in mind, should provide the following functionality:

```
function y = Tlsqrglob( mode, m, n, x, Aname, rw )

global A

if mode==1
 y = A*x;
else
 y = A'*x;
end
```

The purpose is to provide the possibility to define a global variable A and perform the multiplication without transferring this potentially very large matrix to the MEX function *Tlsqr*.

If several matrices are involved, for example if $A = [A_1; A_2]$, this approach can be used to eliminate the need to explicitly repeatedly form the composite matrix A during a run. *Tlsqrglob.m* should then be (copied and) modified as:

```
function y = Tlsqrglob( mode, m, n, x, Aname, rw )

global A1 A2

if mode==1
 y = A1*x;
 y = [y ; A2*x];
else
 M = size(A1,1);
 y =  A1' * x(1:M) + ...
      A2' * x(M+1:end);
end
```

To use the global approach, *Tlsqr* must be called with the name of the global multiplication routine, for example:

```
 [ x, ... ] = Tlsqr(m,n,'Tlsqrglob',...);
```

9 Multi Layer Optimization

TOMLAB supports optimization with any level of recursion assuming that a MEX interface is not permanently allocated in memory. For example, SNOPT cannot use SNOPT as a sub-solver however, it is possible with a solver using a QP MEX solver internally as the MEX solver finishes on each run. The sub optimization problems can be defined as constraints or objectives.

In order to use a sub-solver, a special driver routine *tomSolve* is needed. The universal driver routine *tomRun* goes through several steps before initializing the solution process, so only a pre-check on the sub problem should be done.

The following steps should be followed when setting up a multi layer optimization problem.

- Create a TOMLAB problem using the appropriate assign routine. For example, *Prob = conAssign(...)*
- Create the sub problem in the same manner, *Prob2 = conAssign(...)*. Then check that the problem is correctly setup by executing, Prob = ProbCheck(Prob, Solver, solvType, probType);
- The subproblem should be included in the main problem in the 'user' field, *Prob.user.Prob2 = Prob2*.
- Call the universal driver routine, *tomRun*. *Result = tomRun('snopt', Prob, 1)*.
- In the routine that executes the sub-optimization extract the subproblem, *Prob2 = Prob.user.Prob2* and supply it to *tomSolve*
- Before calling *tomSolve*, parameters depending on the decision variables, x, from the outer problem should be set in *Prob2*. For example lower and upper bounds as well as user parameters could be modified.

When doing multi layer optimization the user can still define the derivatives for known parts of the problem. *Prob.CheckNaN* should be set to obtain derivatives for subproblems.

10 tomHelp - The Help Program

tomHelp is a graphical interface for quick help on all problem types defined in TOMLAB The interface is started by entering *tomHelp* in the MATLAB command prompt. The menu system will be displayed as in Figure 4.

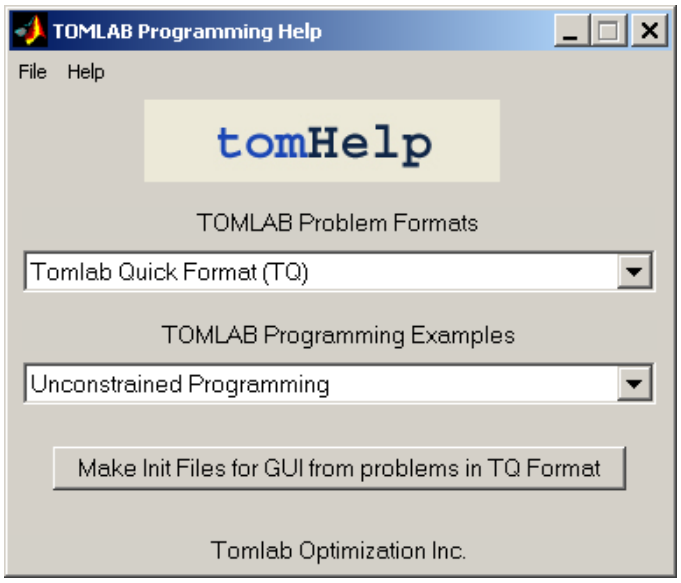


Figure 4: tomHelp start menu.

If a specific problem category is selected then a new menu is displayed. Text help in the MATLAB command window is displayed by choosing help from the interface. It is recommended that the individual demo files are viewed to get an understanding about the specific problem. The files can be used as a starting point for defining problem in the TOMLAB format.

The llsDemo menu is illustrated below in Figure 5

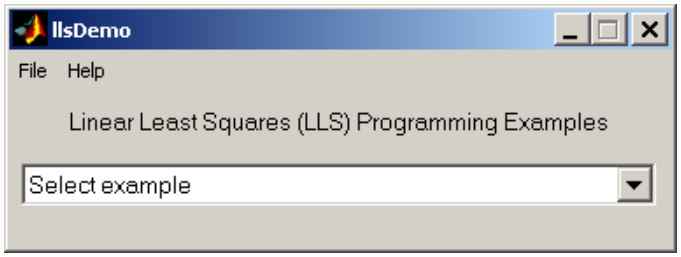


Figure 5: llsDemo menu.

11 TOMLAB Solver Reference

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

11.1 TOMLAB Base Module

For a description of solvers called using the MEX-file interface, see the M-file help, e.g. for the MINOS solver *minosTL.m*. For more details, see the User's Guide for the particular solver.

11.1.1 clsSolve

Purpose
Solves dense and sparse nonlinear least squares optimization problems with linear inequality and equality constraints and simple bounds on the variables.

clsSolve solves problems of the form

$$\begin{array}{lccccc} \min\limits_{x} & f(x) & = & \frac{1}{2} r(x)^T r(x) & & \\ s/t & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \end{array}$$

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

Calling Syntax
Result = clsSolve(Prob, varargin)
Result = tomRun('clsSolve', Prob);

Description of Inputs

Prob Problem description structure. The following fields are used:

Solver.Alg Solver algorithm to be run:
0: Gives default, the Fletcher - Xu hybrid method;
1: Fletcher - Xu hybrid method; Gauss-Newton/BFGS.
2: Al-Baali - Fletcher hybrid method; Gauss-Newton/BFGS.
3: Huschens method. SIAM J. Optimization. Vol 4, No 1, pp 108-129 jan 1994.
4: The Gauss-Newton method.
5: Wang, Li, Qi Structured MBFGS method.
6: Li-Fukushima MBFGS method.
7: Broydens method.

Prob Problem description structure. The following fields are used:, continued

Recommendations: Alg=5 is theoretically best, and seems best in practice as well. Alg=1 and Alg=2 behave very similar, and are robust methods. Alg=4 may be good for ill-conditioned problems. Alg=3 and Alg=6 may sometimes fail. Alg=7 tries to minimize Jacobian evaluations, but might need more residual evaluations. Also fails more often that other algorithms. Suitable when analytic Jacobian is missing and evaluations of the Jacobian is costly. The problem should not be too ill-conditioned.

Solver.Method Method to solve linear system:
0: QR with pivoting (both sparse and dense).
1: SVD (dense).
2: The inversion routine (inv) in Matlab (Uses QR).
3: Explicit computation of pseudoinverse, using pinv(J_k).

Search method technique (if Prob.LargeScale = 1, then Method = 0 always):
Prob.Solver.Method = 0 Sparse iterative QR using Tlsqr.

LargeScale If = 1, then sparse iterative QR using Tlsqr is used to find search directions

x_0 Starting point.
x_L Lower bounds on the variables.
x_U Upper bounds on the variables.

b_L Lower bounds on the linear constraints.
b_U Upper bounds on the linear constraints.
A Constraint matrix for linear constraints.

c_L Lower bounds on the nonlinear constraints.
c_U Upper bounds on the nonlinear constraints.

f_Low A lower bound on the optimal function value, see LineParam.fLowBnd below.

SolverQP Name of the solver used for QP subproblems. If empty, the default solver is used. See GetSolver.m and tomSolve.m.

PriLevOpt Print Level.
optParam Structure with special fields for optimization parameters, see Table 141. Fields used are: *bTol*, *eps_absf*, *eps_g*, *eps_Rank*, *eps_x*, *IterPrint*, *MaxIter*, *PreSolve*, *size_f*, *size_x*, *xTol*, *wait*, and *QN_InitMatrix* (Initial Quasi-Newton matrix, if not empty, otherwise use identity matrix).
LineParam Structure with line search parameters. Special fields used:
LineAlg If *Alg* = 7
0 = Fletcher quadratic interpolation line search
3 = Fletcher cubic interpolation line search
otherwise Armijo-Goldstein line search (LineAlg == 2)

Prob Problem description structure. The following fields are used:, continued

If $Alg! = 7$
0 = Fletcher quadratic interpolation line search
1 = Fletcher cubic interpolation line search
2 = Armijo-Goldstein line search
otherwise Fletcher quadratic interpolation line search (LineAlg == 0)

If Fletcher, see help LineSearch for the LineParam parameters used. Most important is the accuracy in the line search: sigma - Line search accuracy tolerance, default 0.9.

If LineAlg == 2, then the following parameters are used

agFac	Armijo Goldsten reduction factor, default 0.1
sigma	Line search accuracy tolerance, default 0.9

fLowBnd	A lower bound on the global optimum of f(x). NLLS problems always have f(x) values >= 0 The user might also give lower bound estimate in Prob.f_Low clsSolve computes LineParam.fLowBnd as: LineParam.fLowBnd = max(0,Prob.f_Low,Prob.LineParam.fLowBnd) fLow = LineParam.fLowBnd is used in convergence tests.

varargin Other parameters directly sent to low level routines.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Optimal point.
v_k	Lagrange multipliers (not used).
f_k	Function value at optimum.
g_k	Gradient value at optimum.
x_0	Starting point.
f_0	Function value at start.
r_k	Residual at optimum.
J_k	Jacobian matrix at optimum.
xState	State of each variable, described in Table 150.
bState	State of each linear constraint, described in Table 151.
Iter	Number of iterations.
ExitFlag	Flag giving exit status. 0 if convergence, otherwise error. See *Inform*.
Inform	Binary code telling type of convergence: 1: Iteration points are close.

Result	Structure with result from optimization. The following fields are changed:, continued
	2: Projected gradient small.
	3: Iteration points are close and projected gradient small.
	4: Function value close to 0.
	5: Iteration points are close and function value close to 0.
	6: Projected gradient small and function value close to 0.
	7: Iteration points are close, projected gradient small and function value close to 0.
	8: Relative function value reduction low for $LowIts = 10$ iterations.
	11: Relative f(x) reduction low for LowIts iter. Close Iters.
	16: Small Relative f(x) reduction.
	17: Close iteration points, Small relative f(x) reduction.
	18: Small gradient, Small relative f(x) reduction.
	32: Local minimum with all variables on bounds.
	99: The residual is independent of x. The Jacobian is 0.
	101: Maximum number of iterations reached.
	102: Function value below given estimate.
	104: *x_k* not feasible, constraint violated.
	105: The residual is empty, no NLLS problem.
Solver	Solver used.
SolverAlgorithm	Solver algorithm used.
Prob	Problem structure used.

Description

The solver *clsSolve* includes seven optimization methods for nonlinear least squares problems: the Gauss-Newton method, the Al-Baali-Fletcher [3] and the Fletcher-Xu [19] hybrid method, the Huschens TSSM method [50] and three more. If rank problem occur, the solver is using subspace minimization. The line search is performed using the routine *LineSearch* which is a modified version of an algorithm by Fletcher [20]. Bound constraints are partly treated as described in Gill, Murray and Wright [28]. *clsSolve* treats linear equality and inequality constraints using an active set strategy and a null space method.

M-files Used

ResultDef.m, *preSolve.m*, *qpSolve.m*, *tomSolve.m*, *LineSearch.m*, *ProbCheck.m*, *secUpdat.m*, *iniSolve.m*, *endSolve.m*

See Also

conSolve, *nlpSolve*, *sTrustr*

Limitations

When using the *LargeScale* option, the number of residuals may not be less than 10 since the sqr2 algorithm may run into problems if used on problems that are not really large-scale.

Warnings

Since no second order derivative information is used, *clsSolve* may not be able to determine the type of stationary point converged to.

11.1.2 conSolve

Purpose
Solve general constrained nonlinear optimization problems.

conSolve solves problems of the form

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

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 $b_L, b_U \in \mathbb{R}^{m_2}$.

Calling Syntax
Result = conSolve(Prob, varargin)
Result = tomRun('conSolve', Prob);

Description of Inputs

Prob Problem description structure. The following fields are used:

Solver.Alg	Choice of algorithm. Also affects how derivatives are obtained. See following fields and the table on page 92. 0,1,2: Schittkowski SQP. 3,4: Han-Powell SQP.
x_0	Starting point.
x_L	Lower bounds on the variables.
x_U	Upper bounds on the variables.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
A	Constraint matrix for linear constraints.
c_L	Lower bounds on the general constraints.
c_U	Upper bounds on the general constraints.
NumDiff	How to obtain derivatives (gradient, Hessian).
ConsDiff	How to obtain the constraint derivative matrix.
SolverQP	Name of the solver used for QP subproblems. If empty, the default solver is used. See GetSolver.m and tomSolve.m.
f_Low	A lower bound on the optimal function value, see LineParam.fLowBnd below. Used in convergence tests, f_k(x_k) <= f_Low. Only a feasible point x_k is accepted.

Prob Problem description structure. The following fields are used:, continued

FUNCS.f	Name of m-file computing the objective function $f(x)$.
FUNCS.g	Name of m-file computing the gradient vector $g(x)$.
FUNCS.H	Name of m-file computing the Hessian matrix $H(x)$.
FUNCS.c	Name of m-file computing the vector of constraint functions $c(x)$.
FUNCS.dc	Name of m-file computing the matrix of constraint normals $\partial c(x)/dx$.
PriLevOpt	Print level.
optParam	Structure with optimization parameters, see Table 141. Fields that are used: *bTol*, *cTol*, *eps_absf*, *eps_g*, *eps_x*, *eps_Rank*, *IterPrint*, *MaxIter*, *QN_InitMatrix*, *size_f*, *size_x*, *xTol* and *wait*.
LineParam	Structure with line search parameters. See Table 140.
fLowBnd	A lower bound on the global optimum of f(x). The user might also give lower bound estimate in Prob.f_Low conSolve computes LineParam.fLowBnd as: LineParam.fLowBnd = max(Prob.f_Low,Prob.LineParam.fLowBnd).

varargin Other parameters directly sent to low level routines.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Optimal point.
v_k	Lagrange multipliers.
f_k	Function value at optimum.
g_k	Gradient value at optimum.
H_k	Hessian value at optimum.
x_0	Starting point.
f_0	Function value at start.
c_k	Value of constraints at optimum.
cJac	Constraint Jacobian at optimum.
xState	State of each variable, described in Table 150 .
bState	State of each linear constraint, described in Table 151.
cState	State of each nonlinear constraint.
Iter	Number of iterations.
ExitFlag	Flag giving exit status.
ExitText	Text string giving *ExitFlag* and *Inform* information.
Inform	Code telling type of convergence: 1: Iteration points are close.

Result Structure with result from optimization. The following fields are changed:, continued

	2: Small search direction. 3: Iteration points are close and Small search direction. 4: Gradient of merit function small. 5: Iteration points are close and gradient of merit function small. 6: Small search direction and gradient of merit function small. 7: Iteration points are close, small search direction and gradient of merit function small. 8: Small search direction p and constraints satisfied. 101: Maximum number of iterations reached. 102: Function value below given estimate. 103: Iteration points are close, but constraints not fulfilled. Too large penalty weights to be able to continue. Problem is maybe infeasible. 104: Search direction is zero and infeasible constraints. The problem is very likely infeasible. 105: Merit function is infinity. 106: Penalty weights too high.
Solver	Solver used.
SolverAlgorithm	Solver algorithm used.
Prob	Problem structure used.

Description

The routine *conSolve* implements two SQP algorithms for general constrained minimization problems. One implementation, $Solver.Alg = 0, 1, 2$, is based on the SQP algorithm by Schittkowski with Augmented Lagrangian merit function described in [69]. The other, $Solver.Alg = 3, 4$, is an implementation of the Han-Powell SQP method.

The Hessian in the QP subproblems are determined in one of several ways, dependent on the input parameters. The following table shows how the algorithm and Hessian method is selected.

Solver.Alg	NumDiff	AutoDiff	isempty(FUNCS.H)	Hessian computation	Algorithm
0	0	0	0	Analytic Hessian	Schittkowski SQP
0	any	any	any	BFGS	Schittkowski SQP
1	0	0	0	Analytic Hessian	Schittkowski SQP
1	0	0	1	Numerical differences H	Schittkowski SQP
1	> 0	0	any	Numerical differences g,H	Schittkowski SQP
1	< 0	0	any	Numerical differences H	Schittkowski SQP
1	any	1	any	Automatic differentiation	Schittkowski SQP
2	0	0	any	BFGS	Schittkowski SQP
2	$= 0$	0	any	BFGS, numerical gradient g	Schittkowski SQP
2	any	1	any	BFGS, automatic diff gradient	Schittkowski SQP
3	0	0	0	Analytic Hessian	Han-Powell SQP
3	0	0	1	Numerical differences H	Han-Powell SQP
3	> 0	0	any	Numerical differences g,H	Han-Powell SQP
3	< 0	0	any	Numerical differences H	Han-Powell SQP
3	any	1	any	Automatic differentiation	Han-Powell SQP
4	0	0	any	BFGS	Han-Powell SQP
4	$= 0$	0	any	BFGS, numerical gradient g	Han-Powell SQP
4	any	1	any	BFGS, automatic diff gradient	Han-Powell SQP

M-files Used

ResultDef.m, *tomSolve.m*, *LineSearch.m*, *iniSolve.m*, *endSolve.m*, *ProbCheck.m*.

See Also

nlpSolve, *sTrustr*

11.1.3 cutPlane

Purpose
Solve mixed integer linear programming problems (MIP).

cutplane solves problems of the form

$$\begin{array}{lrcccl} \min\limits_{x} & f(x) & = & c^T x & & \\ \text{subject to} & 0 & \leq & x & \leq & x_U \\ & & & Ax & = & b, \quad x_j \in \mathbb{N} \forall j \in \mathrm{I} \end{array}$$

where $c, x, x_U \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. The variables $x \in I$, the index subset of $1, ..., n$ are restricted to be integers.

Calling Syntax
Result = cutplane(Prob); or
Result = tomRun('cutplane', Prob);

Description of Inputs

Prob Problem description structure. The following fields are used:

c	Constant vector.
A	Constraint matrix for linear constraints.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
x_L	Lower bounds on the variables (assumed to be 0).
x_U	Upper bounds on the variables.
x_0	Starting point.
QP.B	Active set *B_0* at start: $B(i) = 1$: Include variable $x(i)$ in basic set. $B(i) = 0$: Variable $x(i)$ is set on it's lower bound. $B(i) = -1$: Variable $x(i)$ is set on it's upper bound. B empty: *lpSimplex* solves Phase I LP to find a feasible point.
Solver.Method	Variable selection rule to be used: 0: Minimum reduced cost. (default) 1: Bland's anti-cycling rule. 2: Minimum reduced cost, Dantzig's rule.
MIP.IntVars	Which of the n variables are integers.
SolverLP	Name of the solver used for initial LP subproblem.

Prob Problem description structure. The following fields are used:, continued

SolverDLP	Name of the solver used for dual LP subproblems.
optParam	Structure with special fields for optimization parameters, see Table 141. Fields used are: *MaxIter*, *PriLev*, *wait*, *eps_f*, *eps_Rank*, *xTol* and *bTol*.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Optimal point.
f_k	Function value at optimum.
g_k	Gradient value at optimum, *c*.
v_k	Lagrange multipliers.
QP.B	Optimal active set. See input variable *QP.B*.
xState	State of each variable, described in Table 150 .
x_0	Starting point.
f_0	Function value at start.
Iter	Number of iterations.
FuncEv	Number of function evaluations. Equal to *Iter*.
ConstrEv	Number of constraint evaluations. Equal to *Iter*.
ExitFlag	0: OK. 1: Maximal number of iterations reached. 4: No feasible point *x_0* found.
Inform	If $ExitFlag > 0$, $Inform = ExitFlag$.
Solver	Solver used.
SolverAlgorithm	Solver algorithm used.
Prob	Problem structure used.

Description

The routine *cutplane* is an implementation of a cutting plane algorithm with Gomorov cuts. *cutplane* normally uses the linear programming routines *lpSimplex* and *DualSolve* to solve relaxed subproblems. By changing the setting of the structure fields *Prob.Solver.SolverLP* and *Prob.Solver.SolverDLP*, different sub-solvers are possible to use.

cutplane can interpret *Prob.MIP.IntVars* in two different ways:

- Vector of length less than dimension of problem: the elements designate indices of integer variables, e.g. $IntVars = [1\ 3\ 5]$ restricts x_1, x_3 and x_5 to take integer values only.
- Vector of same length as *c*: non-zero values indicate integer variables, e.g. with five variables ($x \in \mathbb{R}^5$), $IntVars = [1\ 1\ 0\ 1\ 1]$ demands all but x_3 to take integer values.

M-files Used
lpSimplex.m, *DualSolve.m*

See Also
mipSolve, *balas*, *lpsimp1*, *lpsimp2*, *lpdual*, *tomSolve*.

11.1.4 DualSolve

Purpose
Solve linear programming problems when a dual feasible solution is available.

DualSolve solves problems of the form

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

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

Finite upper bounds on x are added as extra inequality constraints. Finite nonzero lower bounds on x are added as extra inequality constraints. Fixed variables are treated explicitly. Adding slack variables and making necessary sign changes gives the problem in the standard form

$$\begin{array}{ccccc} \min\limits_{x} & f_P(x) & = & c^T x \\ s/t & \hat{A}x & = & b \\ & x & \geq & 0 \end{array}$$

and the following dual problem is solved,

$$\begin{array}{ccccc} \max\limits_{y} & f_D(y) & = & b^T y \\ s/t & \hat{A}^T y & \leq & c \\ & y & urs & \end{array}$$

with $x, c \in \mathbb{R}^n$, $A \in \mathbb{R}^{\hat{m} \times n}$ and $b, y \in \mathbb{R}^m$.

Calling Syntax
[Result] = DualSolve(Prob)

Description of Inputs

Prob Problem description structure. The following fields are used:

QP.c	Constant vector.
A	Constraint matrix for linear constraints.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
x_L	Lower bounds on the variables.
x_U	Upper bounds on the variables.
x_0	Starting point, must be dual feasible.
y_0	Dual parameters (Lagrangian multipliers) at *x_0*.
QP.B	Active set *B_0* at start:

Prob Problem description structure. The following fields are used:, continued

$B(i) = 1$: Include variable $x(i)$ is in basic set.
$B(i) = 0$: Variable $x(i)$ is set on its lower bound.
$B(i) = -1$: Variable $x(i)$ is set on its upper bound.

Solver.Alg Variable selection rule to be used:
0: Minimum reduced cost (default).
1: Bland's anti-cycling rule.
2: Minimum reduced cost. Dantzig's rule.

PriLevOpt Print Level.

optParam Structure with special fields for optimization parameters, see Table 141.
Fields used are: *MaxIter*, *wait*, *eps_f*, *eps_Rank* and *xTol*.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Optimal primal solution x.
f_k	Function value at optimum.
v_k	Optimal dual parameters. Lagrange multipliers for linear constraints.
x_0	Starting point.
Iter	Number of iterations.
QP.B	Optimal active set.
ExitFlag	Exit flag: 0: Optimal solution found. 1: Maximal number of iterations reached. No primal feasible solution found. 2: Infeasible Dual problem. 4: Illegal step length due to numerical difficulties. Should not occur. 6: No dual feasible starting point found. 7: Illegal step length due to numerical difficulties. 8: Convergence because $f_k >= QP.DualLimit$. 9: $x_L(i) > x_U(i) + xTol$ for some i. No solution exists.

Solver Solver used.
SolverAlgorithm Solver algorithm used.

c	Constant vector in standard form formulation.
A	Constraint matrix for linear constraints in standard form.
b	Right hand side in standard form.

Description
When a dual feasible solution is available, the dual simplex method is possible to use. *DualSolve* implements this method using the algorithm in [35, pages 105-106]. There are three rules available for variable selection. Bland's cycling prevention rule is the choice if fear of cycling exist. The other two are variants of minimum reduced cost variable selection, the original Dantzig's rule and one which sorts the variables in increasing order in each step (the default choice).

M-files Used
cpTransf.m

See Also
lpSimplex

11.1.5 expSolve

Purpose
Solve exponential fitting problems for given number of terms p.

Calling Syntax
Prob = expAssign(...);
Result = expSolve(Prob, PriLev); or
Result = tomRun('expSolve', PriLev);

Description of Inputs

Prob	Problem created with expAssign.
PriLev	Print level in tomRun call.
Prob.SolverL2	Name of solver to use. If empty, TOMLAB selects dependent on license.

Description of Outputs

Result	TOMLAB Result structure as returned by solver selected by input argument *Solver*.
LS	Statistical information about the solution. See Table 153, page 239.

Global Parameters Used

Description
expSolve solves a **cls** (constrained least squares) problem for exponential fitting formulates by expAssign. The problem is solved with a suitable or given **cls** solver.

The aim is to provide a quicker interface to exponential fitting, automating the process of setting up the problem structure and getting statistical data.

M-files Used
GetSolver, *expInit*, *StatLS* and *expAssign*

Examples
Assume that the Matlab vectors t, y contain the following data:

t_i	0	1.00	2.00	4.00	6.00	8.00	10.00	15.00	20.00
y_i	905.10	620.36	270.17	154.68	106.74	80.92	69.98	62.50	56.29

To set up and solve the problem of fitting the data to a two-term exponential model

$$f(t) = \alpha_1 e^{-\beta_1 t} + \alpha_2 e^{-\beta_2 t},$$

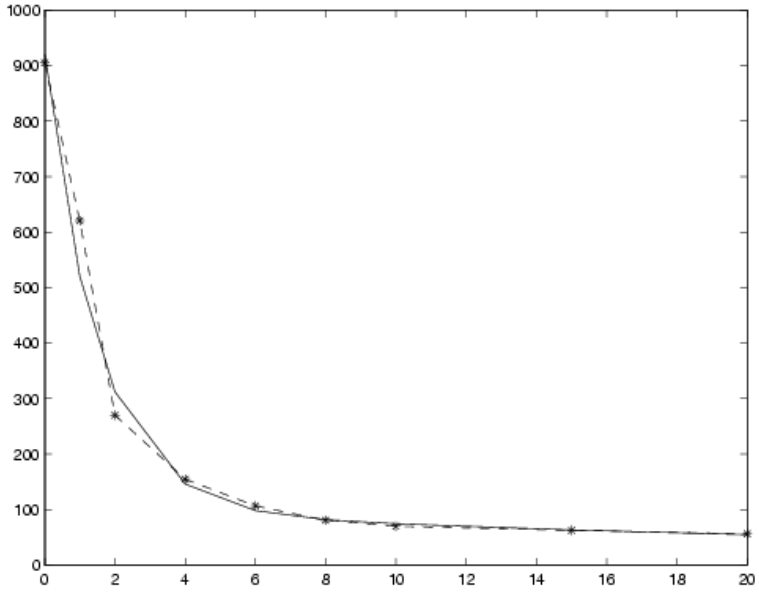
give the following commands:

```
>> p      = 2;                          % Two terms
>> Name   = 'Simple two-term exp fit'; % Problem name, can be anything
>> wType  = 0;                          % No weighting
>> SepAlg = 0;                          % Separable problem
>> Prob = expAssign(p,Name,t,y,wType,[],SepAlg);

>> Result = tomRun('expSolve',Prob,1);
>> x = Result.x_k'

x =
          0.01          0.58         72.38        851.68
```

The x vector contains the parameters as $x = [\beta_1, \beta_2, \alpha_1, \alpha_2]$ so the solution may be visualized with

```
>> plot(t,y,'-*', t,x(3)*exp(-t*x(1)) + x(4)*exp(-t*x(2)) );
```

Figure 6: Results of fitting experimental data to two-term exponential model. Solid line: final model, dash-dot: data.



11.1.6 glbDirect

Purpose
Solve box-bounded global optimization problems.

glbDirect solves problems of the form

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

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

glbDirect is a Fortran MEX implementation of *glbSolve*.

Calling Syntax
Result = glbDirectTL(Prob,varargin)
Result = tomRun('glbDirect', Prob);

Description of Inputs

Prob Problem description structure. The following fields 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.
FUNCS.f	Name of m-file computing the objective function $f(x)$.

Prob Problem description structure. The following fields are used:, continued

PriLevOpt	Print Level. 0 = Silent. 1 = Errors. 2 = Termination message and warm start info. 3 = Option summary.
WarmStart	If true, > 0, *glbDirect* reads the output from the last run from *Prob.glbDirect.WarmStartInto* if it exists. If it doesn't exist, *glbDirect* attempts to open and read warm start data from mat-file glbDirectSave.mat. *glbDirect* uses this warm start information to continue from the last run.
optParam	Structure in Prob, Prob.optParam. Defines optimization parameters. Fields used:
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.
MaxIter	Maximal number of iterations, default 200.
MaxFunc	Maximal number of function evaluations, default 10000 (roughly).
EpsGlob	Global/local weight parameter, default 1E-4.
fGoal	Goal for function value, if empty not used.
eps_f	Relative accuracy for function value, $fTol == eps_f$. Stop if $abs(f - fGoal) <= abs(fGoal) * fTol$, if $fGoal = 0$. Stop if $abs(f - fGoal) <= fTol$, if $fGoal == 0$.
eps_x	Convergence tolerance in x. All possible rectangles are less than this tolerance (scaled to (0,1)). See the output field maxTri.
glbDirect	Structure in Prob, Prob.glbDirect. Solver specific.
options	Structure with options. These options have precedence over all other options in the Prob struct. Available options are: PRILEV: Equivalent to Prob.PrilevOpt. Default: 0 MAXFUNC: Eq. to Prob.optParam.MaxFunc. Default: 10000 MAXITER: Eq. to Prob.optParam.MaxIter. 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 WARMSTART: Eq. to Prob.WarmStart. Default: 0 ITERPRINT: Eq. to Prob.optParam.IterPrint. Default: 0 FUNTOL: Eq. to Prob.optParam.eps_f. Default: 1e-2 VARTOL: Eq. to Prob.optParam.eps_x. Default: 1e-13 GLWEIGHT: Eq. to Prob.optParam.EpsGlob. Default: 1e-4
WarmStartInfo	Structure with WarmStartInfo. Use WarmDefDIRECT.m to define it.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Matrix with optimal points as columns.
f_k	Function value at optimum.
Iter	Number of iterations.
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.
glbDirect	Substructure for glbDirect specific result data.
nextIterFunc	If optimization algorithm was stopped because of maximum number of function evaluations reached, this is the number of function evaluations required to complete the next iteration.
maxTri	Maximum size of any triangles.
WarmStartInfo	Structure containing warm start data. Could be used to continue optimization where glbDirect stopped.

To make a warm start possible, glbDirect saves the following information in the structure Result.glbDirect.WarmStartInfo and file glbDirectSave.mat (for internal solver use only):

points	Matrix with all rectangle centerpoints, in [0,1]-space.
dRect	Vector with distances from centerpoint to the vertices.
fPoints	Vector with function values.
nIter	Number of iterations.
lRect	Matrix with all rectangle side lengths in each dimension.
Name	Name of the problem. Used for security if doing warm start.
dMin	Row vector of minimum function value for each distance.
ds	Row vector of all different distances, sorted.
glbfMin	Best function value found at a feasible point.

Result Structure with result from optimization. The following fields are changed:, continued

iMin	The index in D which has lowest function value, i.e. the rectangle which minimizes $(F-glbfMin+E)./D$ where $E = max(EpsGlob*abs(glbfMin), 1E-8)$.
ign	Rectangles to be ignored in the rect. selection procedure.

Description

The global optimization routine *glbDirect* is an implementation of the DIRECT algorithm presented in [14]. The algorithm in *glbDirect* is a Fortran MEX implementation of the 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* a flag *Prob.WarmStart* should be set to one and *WarmDefDIRECT* run. Then *glbDirect* is using output previously obtained to make the restart. The m-file *glbSolve* also includes the subfunction *conhull* (in MEX) which is an implementation of the algorithm GRAHAMHULL in [65, 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.

M-files Used

iniSolve.m, *endSolve.m glbSolve.m*.

11.1.7 glbSolve

Purpose
Solve box-bounded global optimization problems.

glbSolve solves problems of the form

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

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

Calling Syntax
Result = glbSolve(Prob,varargin)
Result = tomRun('glbSolve', Prob);

Description of Inputs

Prob Problem description structure. The following fields 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.
FUNCS.f	Name of m-file computing the objective function $f(x)$.
PriLevOpt	Print Level. 0 = silent. 1 = some printing. 2 = print each iteration.
WarmStart	If true, > 0, glbSolve reads the output from the last run from the mat-file glbSave.mat, and continues from the last run.
MaxCPU	Maximal CPU Time (in seconds) to be used.
optParam	Structure in Prob, Prob.optParam. Defines optimization parameters. Fields used:

IterPrint	Print iteration #, # of evaluated points and best f(x) each iteration.
MaxIter	Maximal number of iterations, default $max(5000, n * 1000)$.
MaxFunc	Maximal number of function evaluations, default $max(10000, n * 2000)$.
EpsGlob	Global/local weight parameter, default 1E-4.
fGoal	Goal for function value, if empty not used.
eps_f	Relative accuracy for function value, $fTol == eps_f$. Stop if $abs(f - fGoal) <= abs(fGoal) * fTol$, if $fGoal = 0$. Stop if $abs(f - fGoal) <= fTol$, if $fGoal == 0$.

If warm start is chosen, the following fields saved to *glbSave.mat* are also used
and contains information from the previous run:

Prob Problem description structure. The following fields are used:, continued

C	Matrix with all rectangle centerpoints, in [0,1]-space.
D	Vector with distances from centerpoint to the vertices.
DMin	Row vector of minimum function value for each distance.
DSort	Row vector of all different distances, sorted.
E	Computed tolerance in rectangle selection.
F	Vector with function values.
L	Matrix with all rectangle side lengths in each dimension.
Name	Name of the problem. Used for security if doing warm start.
glbfMin	Best function value found at a feasible point.
iMin	The index in D which has lowest function value, i.e. the rectangle which minimizes $(F-glbfMin+E)./D$ where $E = max(EpsGlob*abs(glbfMin), 1E-8)$.

varargin Other parameters directly sent to low level routines.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Matrix with all points giving the function value *f_k*.
f_k	Function value at optimum.
Iter	Number of iterations.
FuncEv	Number function evaluations.
maxTri	Maximum size of any triangle.
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. 0 = Normal Exit. 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. $abs(f - fGoal)/abs(fGoal) <= fTol$. 9 = Max CPU Time reached.
Solver	Solver used, 'glbSolve'.

Description

The global optimization routine *glbSolve* is an implementation of the DIRECT algorithm presented in [14]. 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 *glbSolve* 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** *glbSolve* with the final status of all parameters from the previous run, a so called *warm start* Assume that a run has been made with *glbSolve* 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 *glbSolve* a flag *Prob.WarmStart* should be set to one. Then *glbSolve* is using output previously written to the file *glbSave.mat* to make the restart. The m-file *glbSolve* also includes the subfunction *conhull* (in MEX) which is an implementation of the algorithm GRAHAMHULL in [65, 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.

M-files Used
iniSolve.m, *endSolve.m*

11.1.8 glcCluster

Purpose
Solve general constrained mixed-integer global optimization problems using a hybrid algorithm.

glcCluster solves problems of the form

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

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.

Calling Syntax
Result = glcCluster(Prob, maxFunc1, maxFunc2, maxFunc3, ProbL)
Result = tomRun('glcCluster', Prob, PriLev) (driver call)

Description of Inputs

Prob Problem description structure. The following fields are used:

A	Constraint matrix for linear constraints.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
c_L	Lower bounds on the general constraints.
c_U	Upper bounds on the general constraints.
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.
FUNCS.f	Name of m-file computing the objective function $f(x)$.
FUNCS.c	Name of m-file computing the vector of constraint functions $c(x)$.
PriLevOpt	Print level. 0=Silent. 1=Some output from each glcCluster phase. 2=More output from each phase. 3=Further minor output from each phase. 6=Use PrintResult(,1) to print summary from each global and local run. 7 = Use PrintResult(,2) to print summary from each global and local run. 8 = Use PrintResult(,3) to print summary from each global and local run.
WarmStart	If true, > 0, glcCluster warm starts the DIRECT solver. The DIRECT solver will utilize all points sampled in last run, from one or two calls, dependent on the success in last run. Note: The DIRECT solver may not be changed if doing WarmStart mat-file glcFastSave.mat, and continues from the last run.

Prob Problem description structure. The following fields are used:, continued

Name	Name of the problem. *glcCluster* uses the warmstart capability in *glcFast* and needs the name for security reasons.
GO	Structure in *Prob*, *Prob.GO*. Fields used:
maxFunc1	Number of function evaluations in 1st call to glcFast. Should be odd number (automatically corrected). Default $100 * dim(x) + 1$.
maxFunc2	Number of function evaluations in 2nd call to glcFast.
maxFunc3	If glcFast is not feasible after maxFunc1 function evaluations, it will be repeatedly called (warm start) doing maxFunc1 function evaluations until maxFunc3 function evaluations reached.
ProbL	Structure to be used in the local search. By default the same problem structure as in the global search is used, Prob (see below). Using a second structure is important if optimal continuous variables may take values on bounds. glcFast used for the global search only converges to the simple bounds in the limit, and therefore the simple bounds may be relaxed a bit in the global search. Also, if the global search has difficulty fulfilling equality constraints exactly, the lower and upper bounds may be slightly relaxed. But being exact in the local search. Note that the local search is using derivatives, and can utilize given analytic derivatives. Otherwise the local solver is using numerical derivatives or automatic differentiation. If routines to provide derivatives are given in ProbL, they are used. If only one structure Prob is given, glcCluster uses the derivative routines given in the this structure.
localSolver	Optionally change local solver used ('snopt' or 'npsol' etc.).
DIRECT	DIRECT subsolver, either glcSolve or glcFast (default).
localTry	Maximal number of local searches from cluster points. If <= 0, glcCluster stops after clustering. Default 100.
maxDistmin	The minimal number used for clustering, default 0.05.
optParam	Structure with special fields for optimization parameters, see Table 141. Fields used are: *PriLev*, *cTol*, *IterPrint*, *MaxIter*, *MaxFunc*, *EpsGlob*, *fGoal*, *eps_f*, *eps_x*.
MIP.IntVars	Structure in Prob, Prob.MIP. If empty, all variables are assumed non-integer (LP problem). If $length(IntVars) > 1 ==> length(IntVars) == length(c)$ should hold. Then $IntVars(i) == 1 ==> x(i)$ integer. $IntVars(i) == 0 ==> x(i)$ real. If $length(IntVars) < n$, 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 done more efficiently.

varargin Other parameters directly sent to low level routines.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Matrix with all points giving the function value *f_k*.
f_k	Function value at optimum.
c_k	Nonlinear constraints values at *x_k*.
Iter	Number of iterations.
FuncEv	Number function evaluations.
maxTri	Maximum size of any triangle.
ExitText	Text string giving ExitFlag and Inform information.
Cluster	Subfield with clustering information
x_k	Matrix with best cluster points.
f_k	Row vector with f(x) values for each column in Cluster.x_k.
maxDist	maxDist used for clustering.
minDist	vector of all minimal distances between points.

Description

The routine *glcCluster* implements an extended version of DIRECT, see [52], that handles problems with both nonlinear and integer constraints.

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 *glcCluster* 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** *glcCluster* with the final status of all parameters from the previous run, a so called *warm start* Assume that a run has been made with *glcCluster* 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 *glcCluster* a flag *Prob.WarmStart* should be set to one. Then *glcCluster* is using output previously written to the file *glcSave.mat* to make the restart.

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.

M-files Used

iniSolve.m, *endSolve.m*, *glcFast.m*

11.1.9 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 a Fortran MEX implementation of *glcSolve*.

Calling Syntax
Result = glcDirectTL(Prob,varargin)
Result = tomRun('glcDirect', Prob);

Description of Inputs

Prob Problem description structure. The following fields are used:

Name	Problem name. Used for safety when doing warm starts.
FUNCS.f	Name of m-file computing the objective function $f(x)$.
FUNCS.c	Name of m-file computing the vector of constraint functions $c(x)$.
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 general constraints.
c_U	Upper bounds on the general 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.
PriLevOpt	Print Level. This controls both regular printing from glcDirect and the amount of iteration log information to print. 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.

Prob Problem description structure. The following fields are used:, continued

See optParam.IterPrint for more information on iteration log printing.

WarmStart If true, > 0, glcDirect reads the output from the last run from Prob.glcDirect.WarmStartInfo if it exists. If it doesn't exist, glcDirect attempts to open and read warm start data from mat-file glcDirectSave.mat. glcDirect uses this warm start information to continue from the last run.

MaxCPU Maximum CPU Time (in seconds) to be used.

MIP Structure in *Prob*, *Prob.MIP*.

Intvars If empty, all variables are assumed non-integer (LP problem) If length(IntVars) > 1 ==> length(IntVars) == length(c) should hold Then IntVars(i) == 1 ==> x(i) integer. IntVars(i) == 0 ==> x(i) real If length(IntVars) < n, 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 done more efficiently.

fIP An upper bound on the optimal f(x) value. If empty, set as Inf.

xIP The x-values giving the fIP value. If fIP empty and xIP given, fIP will be computed if xIP nonempty, its feasibility is checked

glcDirect Structure with DIRECT algorithm specific parameters. Fields used:

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.

Prob Problem description structure. The following fields are used:, continued

=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, similar to *mipSolve*, see Prob.MIP.VarWeight.

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.
=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.

Prob Problem description structure. The following fields are used:, continued

optParam Structure with special fields for optimization parameters, see Table 141 on page 229.
Fields used by *glcDirect* are: *IterPrint*, *bTol*, *cTol*, *MaxIter*, *MaxFunc*, *EpsGlob*, *fGoal*, *eps_f*, *eps_x*.

varargin Other parameters directly sent to low level routines.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Matrix with all points giving the function value *f_k*.
f_k	Function value at optimum.
c_k	Nonlinear constraints values at *x_k*.
Iter	Number of iterations.
FuncEv	Number function evaluations.
maxTri	Maximum size of any triangle.
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.

Result Structure with result from optimization. The following fields are changed:, continued

glcDirect	Substructure for glcDirect specific result data.
convFlag	Converge status flag from solver.
WarmStartInfo	Structure with warm start information. Use WarmDefDIRECT to reuse this information in another run.
glcDirectSave.mat	To make a warm start possible, glcDirect saves the following information in the structure Result.glcDirect.WarmStartInfo and file glcDirectSave.mat (for internal solver use only):

C	Matrix with all rectangle centerpoints, in [0,1]-space.
D	Vector with distances from centerpoint to the vertices.
F	Vector with function values.
G	Matrix with constraint values for each point.
Iter	Number of iterations.
Name	Name of the problem. Used for security if doing warm start.
Split	$Split(i, j)$ is the number of splits along dimension i of rectangle j.
Tr	$Tr(i)$ is the number of times rectangle i has been trisected.
fMinIdx	Indices of the currently best points.
fMinEQ	sum(abs(infeasibilities)) for minimum points, 0 if no equalities.
glcfMin	Best function value found at a feasible point.
feasible	Flag indicating if a feasible point has been found.
ignoreidx	Rectangles to be ignored in the rectangle selection procedure.
roc	Rate of change s, for each constraint.
s0	Sum of observed rate of change s0 in the objective.
t	$t(i)$ is the total number of splits along dimension i.

Description

The routine *glcDirect* implements an extended version of DIRECT, see [52], that handles problems with both nonlinear and integer constraints. The algorithm in *glcDirect* is a Fortran MEX implementation of the 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* a flag *Prob.WarmStart* should be set to one. Then *glcDirect* will use output previously written to the file *glcDirectSave.mat* (or the warm start structure) to make the restart.

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.

M-files Used
iniSolve.m, *endSolve.m* and *glcSolve.m*.

Warnings
A significant portion of *glcDirect* is coded in Fortran MEX 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. To reduce the risk of trouble, do "`clear mex`" if a run has been aborted.

11.1.10 glcSolve

Purpose
Solve general constrained mixed-integer global optimization problems.

glcSolve 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).

Calling Syntax
Result = glcSolve(Prob,varargin)
Result = tomRun('glcSolve', Prob);

Description of Inputs

Prob Problem description structure. The following fields are used:

A	Constraint matrix for linear constraints.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
c_L	Lower bounds on the general constraints.
c_U	Upper bounds on the general constraints.
MIP	Structure in *Prob*, *Prob.MIP*.
Intvars	If empty, all variables are assumed non-integer (LP problem) If length(IntVars) > 1 ==> length(IntVars) == length(c) should hold Then IntVars(i) == 1 ==> x(i) integer. IntVars(i) == 0 ==> x(i) real If length(IntVars) < n, 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 done more efficiently.
fIP	An upper bound on the optimal f(x) value. If empty, set as Inf.
xIP	The x-values giving the fIP value. If fIP empty and xIP given, fIP will be computed if xIP nonempty, its feasibility is checked

Prob Problem description structure. The following fields are used:, continued

x_L	Lower bounds for x, must be given to restrict the search space. Any lower bounds that are inf are changed to -10000.
x_U	Upper bounds for x, must be given to restrict the search space. Any upper bounds that are inf are changed to 10000.
FUNCS.f	Name of m-file computing the objective function $f(x)$.
FUNCS.c	Name of m-file computing the vector of constraint functions $c(x)$.
Name	Name of the problem. Used for security if doing warm start.
PriLevOpt	Print level. 0 = silent. 1 = some printing. 2 = print each iteration.
WarmStart	If true (> 0), *glcSolve* reads the output from the last run from the mat-file *glcSave.mat*, and continues from the last run. NOTE: All rectangles that are fathomed in the previous run are deleted. This saves space and computational time and enables solving larger problems and more function evaluations to be done.
MaxCPU	Maximal CPU Time (in seconds) to be used.
glcDirect	Structure with DIRECT algorithm specific parameters. Fields used:
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.

Prob Problem description structure. The following fields are used:, continued

=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, similar to *mipSolve*, see Prob.MIP.VarWeight.

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.
=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 glcSolve 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.

Prob Problem description structure. The following fields are used:, continued

If WarmStart is chosen, the following fields in *glcSave.mat* are also used and contains information from the previous run:

C	Matrix with all rectangle centerpoints.
D	Vector with distances from centerpoint to the vertices.
F	Vector with function values.
G	Matrix with constraint values for each point.
Name	Name of the problem. Used for security if doing warm start.
Split	$Split(i, j)$ is the number of splits along dimension i of rectangle j.
T	$T(i)$ is the number of times rectangle i has been trisected.
fMinEQ	sum(abs(infeasibilities)) for minimum points, 0 if no equalities.
fMinIdx	Indices of the currently best points.
feasible	Flag indicating if a feasible point has been found.
glcfmin	Best function value found at a feasible point.
iL	$iL(i, j)$ is the lower bound for rectangle j in integer dimension $I(i)$.
iU	$iU(i, j)$ is the upper bound for rectangle j in integer dimension $I(i)$.
ignoreidx	Rectangles to be ignored in the rectangle selection procedure.
s	$s(j)$ is the sum of observed rates of change for constraint j.
s_0	s_0 is used as $s(0)$.
t	$t(i)$ is the total number of splits along dimension i.
SubRes	Additional output from nlp_f, if suboptimization done.

optParam Structure with special fields for optimization parameters, see Table 141 on page 229.
Fields used by *glcSolve* are: *IterPrint*, *bTol*, *cTol*, *MaxIter* (default $max(5000, n*1000)$), *MaxFunc* (default $max(10000, n*2000)$), *EpsGlob*, *fGoal*, *eps_f*, *eps_x*.

varargin Other parameters directly sent to low level routines.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Matrix with all points giving the function value *f_k*.
f_k	Function value at optimum.
c_k	Nonlinear constraints values at *x_k*.

glcSave.mat Special file containing:

C	Matrix with all rectangle centerpoints.
D	Vector with distances from centerpoint to the vertices.
F	Vector with function values.
G	Matrix with constraint values for each point.

Result Structure with result from optimization. The following fields are changed:, continued

Name	Name of the problem. Used for security if doing warm start.
Split	$Split(i, j)$ is the number of splits along dimension i of rectangle j.
T	$T(i)$ is the number of times rectangle i has been trisected.
fMinEQ	sum(abs(infeasibilities)) for minimum points, 0 if no equalities.
fMinIdx	Indices of the currently best points.
feasible	Flag indicating if a feasible point has been found.
glcf_min	Best function value found at a feasible point.
iL	$iL(i, j)$ is the lower bound for rectangle j in integer dimension $I(i)$.
iU	$iU(i, j)$ is the upper bound for rectangle j in integer dimension $I(i)$.
ignoreidx	Rectangles to be ignored in the rectangle selection procedure.
s	$s(j)$ is the sum of observed rates of change for constraint j.
s_0	s_0 is used as $s(0)$.
t	$t(i)$ is the total number of splits along dimension i.
Iter	Number of iterations.
FuncEv	Number function evaluations.
maxTri	Maximum size of any triangle.
ExitText	Text string giving ExitFlag and Inform information.
ExitFlag	0 - Reached maxFunc or maxIter. 2 - Some upper bounds below lower bounds. 7 - Reached maxFunc or maxIter, NOT feasible. 8 - Empty domain for integer variables.
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. 9 = Max CPU Time reached. 91= Infeasible. 99= Input error, see ExitFlag.

Description

The routine *glcSolve* implements an extended version of DIRECT, see [52], that handles problems with both nonlinear and integer constraints.

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 *glcSolve* 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** *glcSolve* with the final status of all parameters from the previous run, a so called *warm start* Assume that a run has been made with *glcSolve* 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 *glcSolve* a flag *Prob.WarmStart* should be set to one. Then *glcSolve* is using output previously written to the file *glcSave.mat* to make the restart.

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.

M-files Used

iniSolve.m, *endSolve.m*

11.1.11 infLinSolve

Purpose

Finds a linearly constrained minimax solution of a function of several variables with the use of any suitable TOMLAB solver. The decision variables may be binary or integer.

infLinSolve solves problems of the type:

$$\begin{array}{lllll} \min\limits_{x} & \max Dx & & & \\ \text{subject to} & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \end{array}$$

where $x, x_L, x_U \in \mathbb{R}^n$, $b_L, b_U \in \mathbb{R}^{m_1}$, $A \in \mathbb{R}^{m_1 \times n}$ and $D \in \mathbb{R}^{m_2 \times n}$. The variables $x \in I$, the index subset of $1, ..., n$ are restricted to be integers. The different objectives are stored in D row-wise.

Calling Syntax
Result=infLinSolve(Prob,PriLev)

Description of Inputs

Prob Structure Prob. Prob must be defined. Best is to use Prob = lp/mipAssign(.....), if using the TQ format. Prob.QP.D matrix should then be set to the rows (Prob.QP.c ignored).

PriLev Print level in *infLinSolve*.
- $= 0$ Silent except for error messages.
- > 0 Print summary information about problem transformation. Calls *PrintResult* with specified *PriLev*.
- $= 2$ Standard output from *PrintResult* (default).

Extra fields used in Prob:

SolverInf Name of the TOMLAB solver. Valid names are: cplex, minos, snopt, xa and more. See SolverList('lp'); or SolverList('mip');

QP.D The rows with the different objectives.

f_Low Lower bound on the objective (optional).

f_Upp Upper bound on the objective (optional).

Description of Outputs

Result Structure with results from optimization. Output depends on the solver used.

The fields *x_k*, *f_k*, *x_0*, *xState*, *bState*, *v_k* are transformed back to match the original problem.

The output in Result.Prob is the result after infLinSolve transformed the problem, i.e. the altered Prob structure

Description
The linear minimax problem is solved in infLinSolve by rewriting the problem as a linear optimization problem. One additional variable $z \in \mathbb{R}$, stored as x_{n+1} is added and the problem is rewritten as:

$$\min_x z$$

$$\begin{array}{lccccc} \text{subject to} & x_L & \leq & (x_1, x_2, \ldots, x_n)^T & \leq & x_U \\ & -\infty & \leq & z & \leq & \infty \\ & b_L & \leq & Ax & \leq & b_U \\ & -\infty & \leq & Dx - ze & \leq & 0 \end{array}$$

where $e \in \mathbb{R}^N,\ e(i) = 1\ \forall i$.

To handle cases where a row in D*x is taken the absolute value of: $minmax|D * x|$, expand the problem with extra residuals with the opposite sign: $[D * x; -D * x]$.

See Also
lpAssign.

11.1.12 infSolve

Purpose
Find a constrained minimax solution with the use of any suitable TOMLAB solver.

infSolve solves problems of the type:

$$\begin{array}{lllll} \min\limits_{x} & \max r(x) & & & \\ \text{subject to} & x_L & \leq \quad x & \leq & x_U \\ & b_L & \leq \quad Ax & \leq & b_U \\ & c_L & \leq \quad c(x) & \leq & c_U \end{array}$$

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

Calling Syntax
Result=infSolve(Prob,PriLev)

Description of Inputs

Prob Problem description structure. Should be created in the **cls** format. infSolve uses two special fields in *Prob*:

SolverInf	Name of solver used to solve the transformed problem. Valid choices are *conSolve*, *nlpSolve*, *sTrustr* and *clsSolve*. If TOMLAB /SOL is installed: *minos*, *snopt*, *npopt*.
InfType	1 - constrained formulation (default). 2 - LS penalty approach (experimental).

The remaining fields of *Prob* should be defined as required by the selected subsolver.

PriLev Print level in *infSolve*.

$= 0$	Silent except for error messages.
> 0	Print summary information about problem transformation. Calls *PrintResult* with specified *PriLev*.
$= 2$	Standard output from *PrintResult* (default).

Description of Outputs

Result Structure with results from optimization. Output depends on the solver used.

The fields *x_k*, *r_k*, *J_k*, *c_k*, *cJac*, *x_0*, *xState*, *cState*, *v_k* are transformed back to match the original problem.

g_k is calculated as $J\_k^T \cdot r\_k$.

The output in Result.Prob is the result after infSolve transformed the problem, i.e. the altered Prob structure

Description
The minimax problem is solved in infSolve by rewriting the problem as a general constrained optimization problem. One additional variable $z \in \mathbb{R}$, stored as x_{n+1} is added and the problem is rewritten as:

$$\min_x z$$

$$\begin{array}{lccccc} \text{subject to} & x_L & \leq & (x_1, x_2, \ldots, x_n)^T & \leq & x_U \\ & -\infty & \leq & z & \leq & \infty \\ & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \\ & -\infty & \leq & r(x) - ze & \leq & 0 \end{array}$$

where $e \in \mathbb{R}^N,\ e(i) = 1\ \forall i$.

To handle cases where an element $r_i(x)$ in $r(x)$ appears in absolute value: $\min \max |r_i(x)|$, expand the problem with extra residuals with the opposite sign: $[r_i(x); -r_i(x)]$

Examples
minimaxDemo.m.

See Also
clsAssign.

11.1.13 linRatSolve

Purpose

Finds a linearly constrained solution of a function of the ratio of two linear functions with the use of any suitable TOMLAB solver. Binary and integer variables are not supported.

linRatSolve solves problems of the type:

$$\begin{array}{lllll} \min\limits_{x} & \frac{(c1x)}{(c2x)} & & & \\ \text{subject to} & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \end{array}$$

where $c1, c2, x, x_L, x_U \in \mathbb{R}^n$, $b_L, b_U \in \mathbb{R}^{m_1}$ and $A \in \mathbb{R}^{m_1 \times n}$.

Calling Syntax
Result=linRatSolve(Prob,PriLev)

Description of Inputs

Prob Structure Prob. Prob must be defined. Best is to use Prob = lpAssign(.....), if using the TQ format. Prob.QP.c1/c2 matrices should then be set (Prob.QP.c ignored).

PriLev Print level in *linRatSolve*.
- = 0 Silent except for error messages.
- > 0 Print summary information about problem transformation. Calls *PrintResult* with specified *PriLev*.
- = 2 Standard output from *PrintResult* (default).

Extra fields used in Prob:

SolverRat Name of the TOMLAB solver. Valid names are: cplex, minos, snopt, xa and more. See SolverList('lp');

QP.c1 The numerator in the objective.

QP.c2 The denominator in the objective.

z1_L Lower bound for z1 (default 1e-5). See description below

Description of Outputs

Result Structure with results from optimization. Output depends on the solver used.

The fields *x_k*, *f_k*, *x_0*, *xState*, *bState*, *v_k* are transformed back to match the original problem.

The output in Result.Prob is the result after linRatSolve transformed the problem, i.e. the altered Prob structure

Description

The linear ratio problem is solved by linRatSolve by rewriting the problem as a linear constrained optimization problem. n+1 variables z1 and z2(2:n+1) are needed, stored as x(1:n+1). The n original variables are removed so one more variable exists in the final problem.

$$\begin{array}{rcc} z1 & = & 1/(c2x) \\ z2 & = & xz1 \\ z1(c1x) & = & (c1z1x) = c1z2 \end{array}$$

The problem then becomes:

$$\min_x c1z2$$

$$\begin{array}{lrcccc} \text{subject to} & z1_L & \leq & z1 & \leq & \infty \\ & 1 & \leq & c2z2 & \leq & 1 \\ & 0 & \leq & Az2 - z1beq & \leq & 0 \\ & -\infty & \leq & Az2 - z1b_U & \leq & 0 \\ & -\infty & \leq & -Az2 + z1b_L & \leq & 0 \\ \\ & 0 & \leq & A1z2 - z1xeq & \leq & 0 \\ & -\infty & \leq & A1z2 - z1x_U & \leq & 0 \\ & -\infty & \leq & -A1z2 + z1x_L & \leq & 0 \end{array}$$

where $A1 \in \mathbb{R}^N,\ A1 = speye(N)$.

OBSERVE the denominator $c2x$ must always be positive. It is normally a good a idea to run the problem with both signs (multiply each side by -1).

See Also
lpAssign.

11.1.14 lpSimplex

Purpose
Solve general linear programming problems.

lpSimplex solves problems of the form

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

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

Calling Syntax
Result = lpSimplex(Prob) or
Result = tomRun('lpSimplex', Prob, 1);

Description of Inputs

Prob Problem description structure. The following fields are used:

QP.c	Constant vector.
A	Constraint matrix for linear constraints.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
x_L	Lower bounds on the variables.
x_U	Upper bounds on the variables.
x_0	Starting point.
Solver.Alg	Variable selection rule to be used: 0: Minimum reduced cost. 1: Bland's rule (default). 2: Minimum reduced cost. Dantzig's rule.
QP.B	Active set *B_0* at start: $B(i) = 1$: Include variable $x(i)$ is in basic set. $B(i) = 0$: Variable $x(i)$ is set on its lower bound. $B(i) = -1$: Variable $x(i)$ is set on its upper bound.
optParam	Structure with special fields for optimization parameters, see Table 141. Fields used are: *MaxIter*, *PriLev*, *wait*, *eps_f*, *eps_Rank*, *xTol* and *bTol*.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Optimal point.

Result Structure with result from optimization. The following fields are changed:, continued

f_k	Function value at optimum.
g_k	Gradient value at optimum, c.
v_k	Lagrange multipliers.
x_0	Starting point.
f_0	Function value at start.
xState	State of each variable, described in Table 150.
ExitFlag	0: Optimal solution found. 1: Maximal number of iterations reached. 2: Unbounded feasible region. 5: Too many active variables in given initial point. 6: No feasible point found with Phase 1. 10: Errors in input parameters. 11: Illegal initial x as input.
Inform	If $ExitFlag > 0$, $Inform = ExitFlag$.
QP.B	Optimal active set. See input variable *QP.B*.
Solver	Solver used.
SolverAlgorithm	Solver algorithm used.
Iter	Number of iterations.
FuncEv	Number of function evaluations. Equal to *Iter*.
ConstrEv	Number of constraint evaluations. Equal to *Iter*.
Prob	Problem structure used.

Description
The routine *lpSimplex* implements an active set strategy (Simplex method) for Linear Programming using an additional set of slack variables for the linear constraints. If the given starting point is not feasible then a Phase I objective is used until a feasible point is found.

M-files Used
ResultDef.m

See Also
qpSolve

11.1.15 L1Solve

Purpose
Find a constrained L1 solution of a function of several variables with the use of any suitable nonlinear TOMLAB solver.

L1Solve solves problems of the type:

$$\begin{array}{lccccc} \min\limits_{x} & \sum_i |r_i(x)| & & & & \\ \text{subject to} & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \end{array}$$

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

Calling Syntax
Result = L1Solve(Prob,PriLev)

Description of Inputs

Prob Problem description structure. *Prob* should be created in the **cls** constrained nonlinear format.

L1Solve uses one special field in *Prob*:

SolverL1	Name of the TOMLAB solver used to solve the augmented general nonlinear problem generated by *L1Solve*.

Any other fields are passed along to the solver specified by *Prob.SolverL1*. In particular:

A	Linear constraint matrix.
b_L	Lower bounds on variables.
b_U	Upper bounds on variables.
c_L	Lower bounds for nonlinear constraints.
c_U	Upper bounds for nonlinear constraints..
x_L	Lower bounds on variables.
x_U	Upper bounds on variables.
x_0	Starting point.
ConsPattern	Nonzero patterns of constraint and residual Jacobians.
JacPattern	*Prob.LS.y* must have the correct residual length if *JacPattern* is empty but *ConsPattern* is not. *L1Solve* will create the new patterns for the sub-solver using the information supplied in these two fields.
PriLev	Print level in *L1Solve*.

Prob Problem description structure. *Prob* should be created in the **cls** constrained nonlinear format, continued

$= 0$	silent except for error messages.
> 0	print summary information about problem transformation. Calls *PrintResult* with specified *PriLev*.
$= 2$	standard output from *PrintResult*.

Description of Outputs

Result Structure with results from optimization. Fields changed depends on which solver was used for the extended problem.

The fields *x_k*, *r_k*, *J_k*, *c_k*, *cJac*, *x_0*, *xState*, *cState*, *v_k*, are transformed back to the format of the original L1 problem. *g_k* is calculated as $J\_k^T \cdot r\_k$. The returned problem structure *Result.Prob* is the result after *L1Solve* transformed the problem, i.e. the altered *Prob* structure.

Description

L1Solve solves the L1 problem by reformulating it as the general constrained optimization problem

$$\begin{array}{lccccc} \min\limits_{x} & \sum_i (y_i + z_i) & & & & \\ \text{subject to} & x_L & \leq & x & \leq & x_U \\ & 0 & \leq & y & \leq & \infty \\ & 0 & \leq & z & \leq & \infty \\ & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \\ & 0 & \leq & r(x) + y - z & \leq & 0 \end{array}$$

A problem with N residuals is extended with $2N$ nonnegative variables $y, z \in \mathbb{R}^N$ along with N equality constraints $r_i(x) + y_i - z_i = 0$.

See Also
infSolve

11.1.16 MILPSOLVE

Purpose
Solve mixed integer linear programming problems (MILP).

MILPSOLVE solves problems of the form

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

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

Calling Syntax
Result = tomRun('MILPSOLVE',Prob, 1); or
Prob = ProbCheck(Prob, 'MILPSOLVE');
Result = milpsolveTL(Prob);
PrintResult(Result,1);

Description of Inputs

Prob Problem description structure. The following fields are used:

x_L, x_U	Lower and upper bounds on variables. (Must be dense).
b_L, b_U	Lower and upper bounds on linear constraints. (Must be dense).
A	Linear constraint matrix. (Sparse or dense).
QP.c	Linear objective function coefficients, size $nx1$.
BIG	Definition of infinity. Default is 1e30.
LargeScale	Defines if milpsolveTL will convert the A matrix to a sparse matrix or not. Largescale ! = 0 - sparse LargeScale = 0 - dense Default is to use the A matrix just as it is defined.
PriLevOpt	Specifies the printlevel that will be used by MILPSOLVE. 0 (NONE) No outputs 1 (NEUTRAL) Only some specific debug messages in debug print routines are reported. 2 (CRITICAL) Only critical messages are reported. Hard errors like instability, out of memory. 3 (SEVERE) Only severe messages are reported. Errors. 4 (IMPORTANT) Only important messages are reported. Warnings and Errors.

Prob Problem description structure. The following fields are used:, continued

5 (NORMAL) Normal messages are reported.
6 (DETAILED) Detailed messages are reported. Like model size, continuing B&B improvements.
7 (FULL) All messages are reported. Useful for debugging purposes and small models.

Default print level is 0, no outputs. PriLevOpt < 0 is interpreted as 0, and larger than 7 is interpreted as 7.

MaxCPU Maximal CPU Time (in seconds) to be used by MILPSOLVE, stops with best point found.

Fields used in **Prob.MILPSOLVE** (Structure with MILPSOLVE specific parameters)

ANTI_DEGEN Binary vector. If empty, no anti-degeneracy handling is applied. If the length (i) of the vector is less than 8 elements,only the i first modes are considered. Also if i is longer than 8 elements, the elements after element 8 are ignored.

ANTI_DEGEN specifies if special handling must be done to reduce degeneracy/cycling while solving. Setting this flag can avoid cycling, but can also increase numerical instability.

ANTIDEGEN_FIXEDVARS ! = 0 Check if there are equality slacks in the basis and try to drive them out in order to reduce chance of degeneracy in Phase 1.
ANTIDEGEN_COLUMNCHECK != 0
ANTIDEGEN_STALLING != 0
ANTIDEGEN_NUMFAILURE != 0
ANTIDEGEN_LOSTFEAS != 0
ANTIDEGEN_INFEASIBLE != 0
ANTIDEGEN_DYNAMIC != 0
ANTIDEGEN_DURINGBB != 0

basis If empty or erroneous, default basis is used. Default start base is the all slack basis (the default simplex starting basis).

Prob.MILPSOLVE.basis stores the basic variables. If an element is less then zero then it means on lower bound, else on upper bound. Element 0 of the array is unused. The default initial basis is bascolumn[x] = -x. By MILPSOLVE convention, a basic variable is always on its lower bound, meaning that basic variables is always represented with a minus sign.

When a restart is done, the basis vector must be assigned a correct starting basis.

Prob Problem description structure. The following fields are used:, continued

BASIS_CRASH The set_basiscrash function specifies which basis crash mode MILPSOLVE will used.

When no base crash is done (the default), the initial basis from which MILPSOLVE starts to solve the model is the basis containing all slack or artificial variables that is automatically associates with each constraint.

When base crash is enabled, a heuristic "crash procedure" is executed before the first simplex iteration to quickly choose a basis matrix that has fewer artificial variables. This procedure tends to reduce the number of iterations to optimality since a number of iterations are skipped. MILPSOLVE starts iterating from this basis until optimality.

BASIS_CRASH ! = 2 - No basis crash
BASIS_CRASH = 2 - Most feasible basis

Default is no basis crash.

BB_DEPTH_LIMIT Sets the maximum branch-and-bound depth. This value makes sense only if there are integer, semi-continuous or SOS variables in the model so that the branch-and-bound algorithm is used to solve the model. The branch-and-bound algorithm will not go deeper than this level. When BB_DEPTH_LIMIT i set to 0 then there is no limit to the depth. The default value is -50. A positive value means that the depth is absolute. A negative value means a relative B&B depth. The "order" of a MIP problem is defined to be 2 times the number of binary variables plus the number of SC and SOS variables. A relative value of -x results in a maximum depth of x times the order of the MIP problem.

BB_FLOOR_FIRST Specifies which branch to take first in branch-and-bound algorithm. Default value is 1.
BB_FLOOR_FIRST = 0 (BRANCH_CEILING) Take ceiling branch first
BB_FLOOR_FIRST = 1 (BRANCH_FLOOR) Take floor branch first
BB_FLOOR_FIRST = 2 (BRANCH_AUTOMATIC) MILPSOLVE decides which branch being taken first

BB_RULE Specifies the branch-and-bound rule. Default value is 0.
BB_RULE = 0 (NODE_FIRSTSELECT) Select lowest indexed non-integer column
BB_RULE = 1 (NODE_GAPSELECT) Selection based on distance from the current bounds
BB_RULE = 2 (NODE_RANGESELECT) Selection based on the largest current bound

Prob Problem description structure. The following fields are used:, continued

BB_RULE = 3 (NODE_FRACTIONSELECT) Selection based on largest fractional value
BB_RULE = 4 (NODE_PSEUDOCOSTSELECT4) Simple, unweighted pseudo-cost of a variable
BB_RULE = 5 (NODE_PSEUDONONINTSELECT) This is an extended pseudo-costing strategy based on minimizing the number of integer infeasibilities.
BB_RULE = 6 (NODE_PSEUDORATIOSELECT) This is an extended pseudo-costing strategy based on maximizing the normal pseudo-cost divided by the number of infeasibilities. Effectively, it is similar to (the reciprocal of) a cost/benefit ratio.
BB_RULE = 7 (NODE_USERSELECT)

BB_RULE_ADD Additional values for the BB_RULE. BB_RULE is a vector. If the length i of the vector is less than 10 elements, only the i first modes are considered. Also if i is longer than 10 elements, the elements after element 10 is ignored.

BB_RULE_ADD(1) ! = 0 (NODE_WEIGHTREVERSEMODE)
BB_RULE_ADD(2) ! = 0 (NODE_BRANCHREVERSEMODE) In case when get_bb_floorfirst is BRANCH_AUTOMATIC, select the opposite direction (lower/upper branch) that BRANCH_AUTOMATIC had chosen.
BB_RULE_ADD(3) ! = 0 (NODE_GREEDYMODE)
BB_RULE_ADD(4) ! = 0 (NODE_PSEUDOCOSTMODE)
BB_RULE_ADD(5) ! = 0 (NODE_DEPTHFIRSTMODE) Select the node that has already been selected before the number of times
BB_RULE_ADD(6) ! = 0 (NODE_RANDOMIZEMODE)
BB_RULE_ADD(7) ! = 0 (NODE_DYNAMICMODE) When NODE_DEPTHFIRSTMODE is selected, switch off this mode when a first solution is found.
BB_RULE_ADD(8) ! = 0 (NODE_RESTARTMODE)
BB_RULE_ADD(9) ! = 0 (NODE_BREADTHFIRSTMODE) Select the node that has been selected before the fewest number of times or not at all BB_RULE_ADD(10) ! = 0 (NODE_AUTOORDER)

BFP Defines which Basis Factorization Package that will be used by MILPSOLVE.
BFP = 0 : LUSOL
BFP = 1 : built in etaPHI from MILPSOLVE v3.2
BFP = 2 : Additional etaPHI
BFP = 3 : GLPK

Default BFP is LUSOL.

Prob Problem description structure. The following fields are used:, continued

BREAK_AT_FIRST Specifies if the branch-and-bound algorithm stops at the first found solution (BREAK_AT_FIRST != 0) or not (BREAK_AT_FIRST = 0). Default is not to stop at the first found solution.

BREAK_AT_VALUE Specifies if the branch-and-bound algorithm stops when the object value is better than a given value. The default value is (-) infinity.

EPAGAP Specifies the absolute MIP gap tolerance for the branch and bound algorithm. This tolerance is the difference between the best-found solution yet and the current solution. If the difference is smaller than this tolerance then the solution (and all the sub-solutions) is rejected. The default value is 1e-9.

EPGAP Specifies the relative MIP gap tolerance for the branch and bound algorithm. The default value is 1e-9.

EPSB Specifies the value that is used as a tolerance for the Right Hand Side (RHS) to determine whether a value should be considered as 0. The default epsb value is 1.0e-10

EPSD Specifies the value that is used as a tolerance for reduced costs to determine whether a value should be considered as 0. The default epsd value is 1e-9. If EPSD is empty, EPSD is read from *Prob.optParam.eps_f*.

EPSEL Specifies the value that is used as a tolerance for rounding values to zero. The default epsel value is 1e-12.

EPSINT Specifies the tolerance that is used to determine whether a floating-point number is in fact an integer. The default value for epsint is 1e-7. Changing this tolerance value can result in faster solving times, but the solution is less integer.

EPSPERTURB Specifies the value that is used as perturbation scalar for degenerative problems. The default epsperturb value is 1e-5.

EPSPIVOT Specifies the value that is used as a tolerance pivot element to determine whether a value should be considered as 0. The default epspivot value is 2e-7

IMPROVEMENT_LEVEL Specifies the iterative improvement level.

IMPROVEMENT_LEVEL = 0 (IMPROVE_NONE) improve none
IMPROVEMENT_LEVEL = 1 (IMPROVE_FTRAN) improve FTRAN
IMPROVEMENT_LEVEL = 2 (IMPROVE_BTRAN) improve BTRAN

Prob Problem description structure. The following fields are used:, continued

IMPROVEMENT_LEVEL = 3 (IMPROVE_SOLVE) improve FTRAN + BTRAN.
IMPROVEMENT_LEVEL = 4 (IMPROVE_INVERSE) triggers automatic
inverse accuracy control in the dual simplex, and when an error gap is exceeded the basis is reinverted

Choice 1,2,3 should not be used with MILPSOLVE 5.1.1.3, because of problems with the solver. Default is 0.

LoadFile File that contains the model. If LoadFile is a nonempty string which corresponds to actual file, then the model is read from this file rather than from the Prob struct.

LoadMode 1 - LP - MILPSOLVE LP format
2 - MPS - MPS format
3 - FMPS - Free MPS format

A default value for this field does not exist. Both LoadFile and LoadMode must be set if a problem will be loaded.

If there is something wrong with LoadMode or LoadFile, an error message will be printed and MILPSOLVE will be terminated. Leave LoadMode and LoadFile empty if the problem not will be loaded from file.

LogFile Name of file to print MILPSOLVE log on.

MAXIMIZE If MAXIMIZE ! = 0, MILPSOLVE is set to maximize the objective function, default is to minimize.

MAX_PIVOT Sets the maximum number of pivots between a re-inversion of the matrix. Default is 42.

NEG_RANGE Specifies the negative value below which variables are split into a negative and a positive part. This value must always be zero or negative. If a positive value is specified, then 0 is taken. The default value is -1e6.

PRESOLVE Vector containing possible presolve options. If the length i of the vector is less than 7 elements, only the i first modes are considered. Also if i is longer than 7 elements, the elements after element 7 is ignored.

PRESOLVE(1) ! = 0 (PRESOLVE_ROWS) Presolve rows
PRESOLVE(2) ! = 0 (PRESOLVE_COLS) Presolve columns
PRESOLVE(3) ! = 0 (PRESOLVE_LINDEP) Eliminate linearly dependent rows

Prob Problem description structure. The following fields are used:, continued

PRESOLVE(4) ! = 0 (PRESOLVE_SOS) Convert constraints to SOSes (only SOS1 handled)
PRESOLVE(5) ! = 0 (PRESOLVE_REDUCEMIP) If the phase 1 solution process finds that a constraint is redundant then this constraint is deleted.
PRESOLVE(6) ! = 0 (PRESOLVE_DUALS) Calculate duals
PRESOLVE(7) ! = 0 (PRESOLVE_SENSDUALS) Calculate sensitivity if there are integer variables

Default is not to do any presolve.

PRICING_RULE The pricing rule can be one of the following rules.
PRICING_RULE = 0 Select first (PRICER_FIRSTINDEX)
PRICING_RULE = 1 Select according to Dantzig (PRICER_DANTZIG)
PRICING_RULE = 2 Devex pricing from Paula Harris (PRICER_DEVEX)
PRICING_RULE = 3 Steepest Edge (PRICER_STEEPESTEDGE)

PRICING_MODE Additional pricing settings, any combination of the modes below. This is a binary vector. If the length i of the vector is less than 7 elements, only the i first modes are considered. Also if i is longer than 7 elements, the elements after element 7 is ignored.

PRICE_PRIMALFALLBACK ! = 0 In case of Steepest Edge, fall back to DEVEX in primal.
PRICE_MULTIPLE ! = 0 Preliminary implementation of the multiple pricing scheme. This means that attractive candidate entering columns from one iteration may be used in the subsequent iteration, avoiding full updating of reduced costs. In the current implementation, MILPSOLVE only reuses the 2nd best entering column alternative.
PRICE_PARTIAL ! = 0 Enable partial pricing
PRICE_ADAPTIVE ! = 0 Temporarily use First Index if cycling is detected
PRICE_RANDOMIZE ! = 0 Adds a small randomization effect to the selected pricer
PRICE_LOOPLEFT ! = 0 Scan entering/leaving columns left rather than right
PRICE_LOOPALTERNATE ! = 0 Scan entering/leaving columns alternatingly left/right

Default basis is PRICER_DEVEX combined with PRICE_ADAPTIVE.

sa Struct containing information of the sensitivity analysis (SA) MILPSOLVE will perform.
sa.obj =! 0 Perform sensitivity analysis on the objective function

Prob Problem description structure. The following fields are used:, continued

sa.obj = 0 Do not perform sensitivity analysis on the objective function
sa.rhs =! 0 Perform sensitivity analysis on the right hand sides.
sa.rhs = 0 Do not perform sensitivity analysis on the right hand sides.

SaveFileAfter Name of a file to save the MILPSOLVE object after presolve. The name must be of type string (char), Example: Prob.MILPSOLVE.SaveFileAfter = 'save2' If the type is not char SaveFileBefore is set to save2.[file_extension].

SaveFileBefore Name of a file to save the MILPSOLVE object before presolve. The name must be of type string (char), Example: Prob.MILPSOLVE.SaveFileBefore = 'save1'. If the type is not char SaveFileBefore is set to save1.[file_extension].

SaveMode 1 - LP - MILPSOLVE LP format
2 - MPS - MPS format
3 - FMPS - Free MPS format
If empty, the default format LP is used.

SCALE_LIMIT Sets the relative scaling convergence criterion to the absolute value of SCALE_LIMIT for the active scaling mode. The integer part of SCALE_LIMIT specifies the maximum number of iterations. Default is 5.

SCALING_ALG Specifies which scaling algorithm will be used by MILPSOLVE.
0 No scaling algorithm
1 (SCALE_EXTREME) Scale to convergence using largest absolute value
2 (SCALE_RANGE) Scale based on the simple numerical range
3 (SCALE_MEAN) Numerical range-based scaling
4 (SCALE_GEOMETRIC) Geometric scaling
7 (SCALE_CURTISREID) Curtis-reid scaling

Default is 0, no scaling algorithm.

SCALING_ADD Vector containing possible additional scaling parameters. If the length (i) of the vector is less than 7 elements, only the i first modes are considered. Also if i is longer than 7 elements, the elements after element 7 is ignored.
SCALING_ADD ! = 0 (SCALE_QUADRATIC)
SCALING_ADD ! = 0 (SCALE_LOGARITHMIC) Scale to convergence using logarithmic mean of all values
SCALING_ADD ! = 0 (SCALE_USERWEIGHT) User can specify scalars
SCALING_ADD ! = 0 (SCALE_POWER2) also do Power scaling
SCALING_ADD ! = 0 (SCALE_EQUILIBRATE) Make sure that no scaled number is above 1

Prob Problem description structure. The following fields are used:, continued

SCALING_ADD ! = 0 (SCALE_INTEGERS) Also scaling integer variables
SCALING_ADD ! = 0 (SCALE_DYNUPDATE) Dynamic update

Default is 0, no additional mode.

Settings SCALE_DYNUPDATE is a way to make sure that scaling factors are recomputed. In that case, the scaling factors are recomputed also when a restart is done.

SIMPLEX_TYPE Sets the desired combination of primal and dual simplex algorithms.
5 (SIMPLEX_PRIMAL_PRIMAL) Phase1 Primal, Phase2 Primal
6 (SIMPLEX_DUAL_PRIMAL) Phase1 Dual, Phase2 Primal
9 (SIMPLEX_PRIMAL_DUAL) Phase1 Primal, Phase2 Dual
10 (SIMPLEX_DUAL_DUAL) Phase1 Dual, Phase2 Dual

Default is SIMPLEX_DUAL_PRIMAL (6).

SOLUTION_LIMIT Sets the solution number that will be returned. This value is only considered if there are integer, semi-continuous or SOS variables in the model so that the branch-and-bound algorithm is used. If there are more solutions with the same objective value, then this number specifies which solution must be returned. Default is 1.

sos List of structs containing data about Special Ordered Sets (SOS). See below for further description.

Fields used in **Prob.MIP** (Structure with MIP specific parameters)

IntVars Defines which variables are integers, of general type I or binary type B Variable indices should be in the range [1,...,n].

IntVars is a logical vector ==> $x(find(IntVars > 0))$ are integers

IntVars is a vector of indices ==> $x(IntVars)$ are integers (if [], then no integers of type I or B are defined) variables with x_L=0 and x_U=1, is are set to binary. It is possible to combine integer and semi-continuous type to obtain the semi-integer type.

fIP This parameter specifies the initial "at least better than" guess for objective function. This is only used in the branch-and-bound algorithm when integer variables exist in the model. All solutions with a worse objective value than this value are immediately rejected. The default is infinity.

Prob Problem description structure. The following fields are used:, continued

SC	A vector with indices for variables of type semi-continuous (SC), a logical vector or a scalar (see MIP.IntVars). A semi-continuous variable i takes either the value 0 or some value in the range [x_L(i), x_U(i)]. It is possible to combine integer and semi-continuous type to obtain the semi-integer type.
sos1	List of structures defining the Special Ordered Sets of Order One (SOS1). For SOS1 set k, sos1(k).var is a vector of indices for variables of type SOS1 in set k, sos1(k).row is the priority of SOS k in the set of SOS1 and sos1(k).weight is a vector of the same length as sos1(k).var and it describes the order MILPSOLVE will weight the variables in SOS1 set k. a low number of a row and a weight means high priority.
sos2	List of n structures defining the Special Ordered Sets (SOS) of Order Two (SOS2). (see MIP.sos1)

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Optimal solution (or some other solution if optimum could not been found)
f_k	Optimal objective value.
v_k	[rc; duals]. If Reduced cost and dual variables are not available, then v_k is empty.
ExitFlag	TOMLAB information parameter. 0 = Optimal solution found. 1 = Suboptimal solution or user abort. 2 = Unbounded solution. 3 = Numerical failure. 4 = Infeasible model. 10 = Out of memory. 11 = Branch and bound stopped.
ExitText	Status text from MILPSOLVE.
Inform	MILPSOLVE information parameter. -2 = Out of memory. 0 = Optimal solution found. 1 = Suboptimal solution.

Result Structure with result from optimization. The following fields are changed:, continued

	2 = Infeasible model. 3 = Unbounded solution. 4 = Degenerate solution. 5 = Numerical failure. 6 = User abort. 7 = Timeout. 10 = Branch and bound failed. 11 = Branch and bound stopped. 12 = Feasible branch and bound solution. 13 = No feasible branch and bound solution. Other = Unknown status.
Iter	The total number of nodes processed in the branch-and-bound algorithm. Is only applicable if the model contains integer variables. In the case of an LP model Result.Iter contains the number of iterations. This is however not documented.
MinorIter	The total number of Branch-and-bound iterations. When the problem is LP, MinorIter equals Result.Iter
MILPSOLVE.basis	Optimal basis, on the format described above under Prob.MILPSOLVE.basis.
MILPSOLVE.MaxLevel	The deepest Branch-and-bound level of the last solution. Is only applicable if the model contains integer variables.
MILPSOLVE.sa.objStatus	1 successful 0 SA not requested -1 Error: error from MILPSOLVE -3 no SA available
MILPSOLVE.sa.ObjLower	An array that will contain the values of the lower limits on the objective function.
MILPSOLVE.sa.ObjUpper	An array that will contain the values of the upper limits on the objective function.
MILPSOLVE.sa.RhsStatus	see MILPSOLVE.sa.objStatus.
MILPSOLVE.sa.RhsLower	An array that will contain the values of the lower limits on the RHS.
MILPSOLVE.sa.RhsUpper	An array that will contain the values of the upper limits on the RHS.

Result Structure with result from optimization. The following fields are changed:, continued

xState	State of each variable 0 - free variable, 1 - variable on lower bound, 2 - variable on upper bound, 3 - variable is fixed, lower bound = upper bound.
bState	State of each linear constraint 0 - Inactive constraint, 1 - Linear constraint on lower bound, 2 - Linear constraint on upper bound, 3 - Linear equality constraint.

11.1.17 minlpSolve

Purpose
Branch & Bound algorithm for Mixed-Integer Nonlinear Programming (MINLP) with convex or nonconvex sub problems using NLP relaxation (Formulated as minlp-IP).

The parameter Convex (see below) determines if to assume the NLP subproblems are convex or not.

minlpSolve depends on a suitable NLP solver.

minlpSolve solves problems of the form

$$\begin{array}{lccccc} \min\limits_{x} & f(x) & & & & \\ s/t & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \\ & & & x_j \in \mathbb{N} & \forall j \in \mathrm{I} & \end{array}$$

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.

Calling Syntax
Result = tomRun('minlpSolve',Prob,...)

Description of Inputs

Prob Problem description structure. The following fields are used:

x_L	Lower bounds on x.
x_U	Upper bounds on x.
A	The linear constraint matrix.
b_L	Lower bounds on linear constraints.
b_U	Upper bounds on linear constraints.
c_L	Lower bounds on nonlinear constraints.
c_U	Upper bounds on nonlinear constraints.

x_0	Starting point.
Convex	If Convex==1, assume NLP problems are convex, and only one local NLP solver call is used at each node. If Convex==0 (Default), multiMin is used to do many calls to a local solver to determine the global minima at each node. The global minimum with most components integer valued is chosen.
MaxCPU	Maximal CPU Time (in seconds) to be used by minlpSolve, stops with best point found
PriLev	Print level in minlpSolve (default 1). Also see optParam.IterPrint

Prob Problem description structure. The following fields are used:, continued

PriLevOpt Print level in sub solvers (SNOPT and other NLP solvers):
=0 No output; >0 Convergence results
>1 Output every iteration, >2 Output each step in the NLP alg
For other NLP solvers, see the documentation for the solver

WarmStart If true, >0, minlpSolve reads the output from the last run from Prob.minlpSolve, if it exists. If it doesn't exist, minlpSolve attempts to open and read warm start data from mat-file minlpSolveSave.mat. minlpSolve uses the warm start information to continue from the last run. The mat-file minlpSolveSave.mat is saved every Prob.MIP.SaveFreq iteration.

SolverNLP Name of the solver used for NLP subproblems. If empty, the default solver is found calling GetSolver('con',1); If TOMLAB /SOL installed, SNOPT is the default solver. If SolverNLP is a SOL solver (SNOPT, MINOS or NPSOL), the SOL.optPar and SOL.PrintFile is used: See help minosTL.m, npsolTL.m or snoptTL.m for how to set these parameters

RandState If Convex == 0, RandState is sent to multiMin to initialize the random generator. RandState is used as follows:
If > 0, rand('state',RandState) is set to initialize the pseudo-random generator
if < 0, rand('state',sum(100*clock)) is set to give a new set of random values each run
if RandState == 0, rand('state',) is not called. Default RandState = -1

MIP Structure in Prob, Prob.MIP. Defines integer optimization parameters. Fields used:

IntVars If empty, all variables are assumed non-integer. If islogical(IntVars) (=all elements are 0/1), then 1 = integer variable, 0 = continuous variable. If any element >1, IntVars is the indices for integer variables.

VarWeight Weight for each variable in the variable selection phase. A lower value gives higher priority. Setting Prob.MIP.VarWeight might improve convergence.

DualGap *minlpSolve* stops if the duality gap is less than DualGap. DualGap = 1, stop at first integer solution e.g. DualGap = 0.01, stop if solution < 1% from optimal solution.

fIP An upper bound on the IP value wanted. Makes it possible to cut branches and avoid node computations. Used even if xIP not given.

xIP The x-values giving the fIP value, if a solution (xIP,fIP) is known.

NodeSel Node selection method in branch and bound
= 0 Depth First. Priority on nodes with more integer components
= 1 Breadth First. Priority on nodes with more integer components
= 2 Depth First. When integer solution found, use NodeSel = 1 (default)
= 3 Pure LIFO (Last in, first out) Depth First

Prob Problem description structure. The following fields are used:, continued

= 4 Pure FIFO (First in, first out) Breadth First
= 5 Pure LIFO Depth First. When integer solution found, use NodeSel 4

VarSel Variable selection method in branch and bound:
= 1 Use variable with most fractional value
= 2 Use gradient and distance to nearest integer value

KNAPSACK If = 1, use a knapsack heuristic. Default 0.

ROUNDH If = 1, use a rounding heuristic. Default 0.

SaveFreq Warm start info saved on minlpSolveSave.mat every SaveFreq iteration (default -1, i.e. no warm start info is saved)

optParam Structure in Prob. Fields used in Prob.optParam, also in sub solvers:
MaxIter Maximal number of iterations, default 10000
IterPrint Print short information each iteration (PriLev > 0 ==> IterPrint = 1). Iteration number: Depth in tree (symbol L[] - empty list, symbol Gap - Dual Gap convergence), fNLP (Optimal f(x) current node), fIPMin (Best integer feasible f(x) found), LowBnd (Lower bound on optimal integer feasible f(x)), Dual Gap in absolut value and percent, The length of the node list L, —L—, The Inform and ExitFlag the solver returned at the current node, FuEv (Number of function evaluations used by solver at current node), date/time stamp.
bTol Linear constraint violation convergence tolerance.
cTol Constraint violation convergence tolerance.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

Iter Number of iterations.
ExitFlag 0: Global optimal solution found, or integer solution with duality gap less than user tolerance.
1: Maximal number of iterations reached.
2: Empty feasible set, no integer solution found.
4: No feasible point found running NLP relaxation.
5: Illegal *x_0* found in NLP relaxation.
99: Maximal CPU Time used (cputime > Prob.MaxCPU).

Inform Code telling type of convergence, returned from subsolver.
ExitText Text string giving ExitFlag and Inform information.
DualGap Relative duality gap, max(0,fIPMin-fLB)/—fIPMin—, if fIPMin =0; max(0,fIPMin-fLB) if fIPMin == 0. If fIPMin =0: Scale with 100, 100*DualGap, to get the percentage duality gap. For absolute value duality gap: scale with fIPMin, fIPMin * DualGap

Result Structure with result from optimization. The following fields are changed:, continued

x_k	Solution.
v_k	Lagrange multipliers. Bounds, Linear and Nonlinear Constraints, n + mLin + mNonLin.
f_k	Function value at optimum.
g_k	Gradient vector at optimum.
x_0	Starting point x_0.
f_0	Function value at start.
c_k	Constraint values at optimum.
cJac	Constraint derivative values at optimum.
xState	State of each variable, described in Table 150.
bState	State of each constraint, described in Table 151.
cState	State of each general constraint, described in Table 152.
Solver	Solver used ('mipSolve').
SolverAlgorithm	Text description of solver algorithm used.
Prob	Problem structure used.
minlpSolve	A structure with warm start information. Use with WarmDefGLOBAL, see example below.

Description
To make a restart (warm start), just set the warm start flag, and call minlpSolve once again:

```
Prob.WarmStart = 1;
Result         = tomRun('minlpSolve', Prob, 2);
```

minlpSolve will read warm start information from the minlpSolveSave.mat file.

Another warm start (with same MaxFunc) is made by just calling tomRun again:

```
Result = tomRun('minlpSolve', Prob, 2);
```

To make a restart from the warm start information in the Result structure, make a call to WarmDefGLOBAL before calling minlpSolve. WarmDefGLOBAL moves information from the Result structure to the Prob structure and sets the warm start flag, Prob.WarmStart = 1;

```
Prob = WarmDefGLOBAL('minlpSolve', Prob, Result);
```

where Result is the result structure returned by the previous run. A warm start (with same MaxIter) is done by just calling tomRun again:

```
Result = tomRun('minlpSolve', Prob, 2);
```

To make another warm start with new MaxIter 100, say, redefine MaxIter as:

```
Prob.optParam.MaxIter = 100;
```

Then repeat the two lines:

```
Prob = WarmDefGLOBAL('minlpSolve', Prob, Result);
Result = tomRun('minlpSolve', Prob, 2);
```

11.1.18 mipSolve

Purpose
Solve mixed integer linear programming problems (MIP).

mipSolve solves problems of the form

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

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

Starting with TOMLAB version 4.0, *mipSolve* accepts upper and lower bounds on the linear constraints like most other TOMLAB solvers. Thus it is no longer necessary to use slack variables to handle inequality constraints.

Calling Syntax
Result = tomRun('mipSolve',Prob,...)

Description of Inputs

Prob Problem description structure. The following fields are used:

c	The vector c in $c^T x$.
A	Constraint matrix for linear constraints.
b_L	Lower bounds on the linear constraints. If empty, *Prob.b_U* is used.
b_U	Upper bounds on the linear constraints.
x_L	Lower bounds on the variables.
x_U	Upper bounds on the variables.
x_0	Starting point.
MaxCPU	Maximal CPU Time (in seconds) to be used by mipSolve, stops with best point found.
QP.B	Active set *B_0* at start: $B(i) = 1$: Include variable $x(i)$ is in basic set. $B(i) = 0$: Variable $x(i)$ is set on its lower bound. $B(i) = -1$: Variable $x(i)$ is set on its upper bound.
SolverLP	Name of solver used for initial LP subproblem. Default solver is used if empty, see *GetSolver.m* and *tomSolve.m*.
SolverDLP	Name of solver used for the dual LP subproblems. Default solver is used if empty, see *GetSolver.m* and *tomSolve.m*.

Prob Problem description structure. The following fields are used:, continued

PriLevOpt	Print level in *lpSimplex* and *DualSolve*: 0: No output; > 0: Convergence result; > 1: Output every iteration; > 2: Output each step in simplex algorithm.
PriLev	Print level in *mipSolve*.
SOL.optPar	Parameters for the SOL solvers, if they are used as subsolvers.
SOL.PrintFile	Name of print file for SOL solvers, if they are used as subsolvers.
MIP	Structure with fields for integer optimization The following fields are used:
IntVars	The set of integer variables. If empty, all variables are assumed non-integer (LP problem)
VarWeight	Weight for each variable in the variable selection phase. A lower value gives higher priority. Setting *Prob.MIP.VarWeight* = *Prob.c* improves convergence for knapsack problems.
DualGap	*mipSolve* stops if the duality gap is less than *DualGap*. To stop at the first found integer solution, set *DualGap* =1. For example, *DualGap* = 0.01 makes the algorithm stop if the solution is $< 1\%$ from the optimal solution.
fIP	An upper bound on the IP value wanted. Makes it possible to cut branches and avoid node computations.
xIP	The x-value giving the *fIP* value.
KNAPSACK	If solving a knapsack problem, set to true (1) to use a knapsack heuristic.
optParam	Structure with special fields for optimization parameters, see Table 141. Fields used are: *IterPrint*, *MaxIter*, *PriLev*, *wait*, *eps_f* and *eps_Rank*.
Solver	Structure with fields for algorithm choices:
Alg	Node selection method: 0: Depth first 1: Breadth first 2: Depth first. When integer solution found, switch to Breadth.
method	Rule to select new variables in DualSolve/lpSimplex: 0: Minimum reduced cost, sort variables increasing. (Default) 1: Bland's rule (default). 2: Minimum reduced cost. Dantzig's rule.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Optimal point.
f_k	Function value at optimum.
g_k	Gradient value at optimum, c.

Result Structure with result from optimization. The following fields are changed:, continued

v_k	Lagrange multipliers, [Constraints + lower + upper bounds].
x_0	Starting point.
f_0	Function value at start.
xState	State of each variable, described in Table 150, page 239.
Inform	If $ExitFlag > 0$, $Inform = ExitFlag$.
QP.B	Optimal active set. See input variable *QP.B*.
QP.y	Dual parameters y (also part of *Result.v_k*.
p_dx	Search steps in x.
alphaV	Step lengths for each search step.
ExitFlag	0: OK. 1: Maximal number of iterations reached. 2: Empty feasible set, no integer solution found. 3: Rank problems. Can not find any solution point. 4: No feasible point found running LP relaxation. 5: Illegal *x_0* found in LP relaxation. 99: Maximal CPU Time used (cputime > Prob.MaxCPU).
Iter	Number of iterations.
Solver	Solver used ('mipSolve').
SolverAlgorithm	Text description of solver algorithm used.
Prob	Problem structure used.

Description
The routine *mipSolve* is an implementation of a branch and bound algorithm from Nemhauser and Wolsey [58, chap. 8.2]. *mipSolve* normally uses the linear programming routines *lpSimplex* and *DualSolve* to solve relaxed subproblems. *mipSolve* calls the general interface routines *SolveLP* and *SolveDLP*. By changing the setting of the structure fields *Prob.Solver.SolverLP* and *Prob.Solver.SolverDLP*, different sub-solvers are possible to use, see the help for the interface routines.

Algorithm
See [58, chap. 8.2] and the code in *mipSolve.m*.

Examples
See *exip39*, *exknap*, *expkorv*.

M-files Used
lpSimplex.m, *DualSolve.m*, *GetSolver.m*, *tomSolve.m*

See Also
cutplane, *balas*, *SolveLP*, *SolveDLP*

11.1.19 multiMin

Purpose
multiMin solves general constrained mixed-integer global optimization problems. It tries to find all local minima by a multi-start method using a suitable nonlinear programming subsolver.

multiMin solves problems of the form

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

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.

The integer components of every x_0 is rounded to the nearest integer value inside simple bounds, and these components are fixed during the nonlinear local search.

If generating random points and there are linear constraints, multiMin checks feasibility with respect to the linear constraints, and for each initial point tries 100 times to get linear feasibility before accepting the initial point.

Calling Syntax
Result = multiMin(Prob, xInit)
Result = tomRun('multiMin', Prob, PriLev) (driver call)

Description of Inputs

Prob Problem description structure. The following fields are used:

xInit Either, 1x1 number - The number of random initial points, default 10*Prob.N
dxm matrix - Every column is an initial point (of length d=Prob.N).
May also be set as Prob.xInit.

fCut If initial f(x_0) > fCut, no local optimization is done.

WarmStart If true, > 0, multiMin assumes the field multiMin defined with the output from a previous run on the same problem. See the Output fields of Result.multiMin. Use WarmDefGLOBAL to set the correct fields in the Prob structure. Necessary fields are fOpt and xOpt. If any of the other fields are missing, the corresponding variables are initialized to 0. These other fields are: localTry, Iter, FuncEv, GradEv, HessEv, ConstrEv Inform (is set to zeros(length(fOpt,1) if not defined).

In WarmDefGLOBAL the Result structure for the optimal run will be fed back to multiMin as Prob.multiMin.ResOpt In case this field is not defined, and no better point is found during the runs, a special call to the localSolver is used to generate a proper Result structure.

Prob Problem description structure. The following fields are used:, continued

RandState If $WarmStart$ and isscalar(xInit), RandState is used as follows: If > 0, rand('state',RandState) is set to initialize the pseudo-random generator if < 0, rand('state',sum(100*clock)) is set to give a new set of random values each run if RandState == 0, rand('state',) is not called Default RandState = -1.

xEqTol Tolerance to test if new point x_k already defined as optimum: $norm(x_k - xOpt(:,i)) <= xEqTol * max(1, norm(x_k))$ If test fulfilled x_k is assumed to be too close to xOpt(:,i) Default xEqTol = 1E-5

x_L Lower bounds for each element in x. If generating random points, -inf elements of x_L are set to -10000.

x_U Upper bounds for each element in x. If generating random points, inf elements of x_U are set to 10000.

A Constraint matrix for linear constraints.

b_L Lower bounds on the linear constraints.

b_U Upper bounds on the linear constraints.

c_L Lower bounds on the general constraints.

c_U Upper bounds on the general constraints.

PriLevOpt 0 = silent.

1 = Display one row for each unique local minima found. The minima are sorted, lowest value first (possibly the global minima)

The following 4 information values are displayed:

1. Order #
2. Function value f(x) at local minima
3. Point x at local minima. Only up to 10 values are displayed
4. Inform value returned from local Solver (normally 0)

2 = One row of output from each multiMin local optimization trial

The following 6 information values are displayed:

1. Step #
2. Text Old (previously found local minima), FAIL (solver failed to verify local minima) or blank (solver success, new local minima found)
3. Inform value from local solver
4. f(x_0) - function value f(x_0) for initial x_0
5. f(x) - best f(x) value found in this local search
6. x - point for which best f(x) value was found in this local search. Only up to 10 values are displayed.

3 = tomRun (PrintResult) output from every optimization, print level 1.

Prob Problem description structure. The following fields are used:, continued

4 = tomRun (PrintResult) output from every optimization, print level 2. For constrained problems output of sum(—constr—) and information if optimal point was accepted w.r.t. feasibility.
5 = tomRun (PrintResult) output from every optimization, print level 3.

GO Structure in *Prob*, *Prob.GO*. Fields used:

localSolver The local solver used to run all local optimizations. Default is the license dependent output of GetSolver('con',1,0).

optParam Defines optimization parameters. Fields used:

fGoal Goal for function value f(x), if empty not used. If fGoal is reached, no further local optimizations are done.

eps_f Relative accuracy for function value, fTol == eps_f. Stop if abs(f-fGoal) <= abs(fGoal) * fTol , if fGoal = 0. Stop if abs(f-fGoal) <= fTol , if fGoal ==0.

bTol The local solver used to run all local optimizations. Default is the license dependent output of GetSolver('con',1,0).

MIP.IntVars Structure in Prob, Prob.MIP. If empty, all variables are assumed non-integer (LP problem). If $length(IntVars) > 1$ ==> $length(IntVars) == length(c)$ should hold. Then $IntVars(i) == 1$ ==> $x(i)$ integer. $IntVars(i) == 0$ ==> $x(i)$ real. If $length(IntVars) < n$, 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 done more efficiently.

varargin Other parameters directly sent to low level routines.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

The following fields in Result are changed by multiMin before return:
ExitFlag = 0 normal output, of if fGoal set and found.
= 1 A solution reaching the user defined fGoal was not found
The Solver, SolverAlgorithm and ExitText fields are also reset.

A special field in Result is also returned, Result.multiMin:

xOpt Prob.N x k matrix with k distinct local optima, the test being $norm(x_k - xOpt(:,i)) <= xEqTol * max(1, norm(x_k))$ that if fulfilled assumes x_k to be to close to xOpt(:,i)
fOpt The k function values in the local optima xOpt(:,i),i=1,...,k.

Result Structure with result from optimization. The following fields are changed:, continued

Inform	The Inform value returned by the local solver when finding each of the local optima xOpt(:,i); i=1,...,k. The Inform value can be used to judge the validity of the local minimum reported.
localTry	Total number of local searches.
Iter	Total number of iterations.
FuncEv	Total number of function evaluations.
GradEv	Total number of gradient evaluations.
HessEv	Total number of Hessian evaluations.
ConstrEv	Total number of constraint function evaluations.
ExitText	Text string giving ExitFlag and Inform information.

11.1.20 multiMINLP

Purpose
multiMINLP solves general constrained mixed-integer global nonlinear optimization problems.

It is aimed for problems where the number of integer combinations nMax is huge and relaxation of the integer constraint is possible.

If no integer variables, multiMINLP calls multiMin. If nMax <= min(Prob.optParam.MaxFunc,5000), glcDirect is used. Otherwise, multiMINLP first finds a set M of local minima calling multiMin with no integer restriction on any variable. The best local minimum is selected. glcDirect is called to find the best integer feasible solution fIP in a small area (< +- 2 units) around the best local minimum found.

The other local minima are pruned, if fOpt(i) > fIP, no integer feasible solution could be found close to this local minimum i.

The area close to the remaining candidate local minima are searched one by one by calling glcDirect to find any fIPi < fIP.

multiMINLP solves problems of the form

$$\begin{array}{lccccc} \min\limits_{x} & f(x) & & & & \\ s/t & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \\ & & & x_j \in \mathbb{N} & \forall j \in \mathrm{I} & \end{array}$$

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.

Calling Syntax
Result = tomRun('multiMINLP',Prob,...)

Description of Inputs

Prob Problem description structure. The following fields are used:

Prob is a structure, defined as to solve a standard MINLP problem. The Prob structure is fed to the localSolver. See e.g. minlpAssign.
See multiMin and glcDirect for input to the subsolvers e.g. Prob.xInit is used in multiMin (and fCut, RandState, xEQTol).

x_L Lower bounds for each element in x. If generating random points, -inf elements of x_L are set to min(-L,xMin,x_U-L) xMin is the minimum of the finite x_L values.

x_U Upper bounds for each element in x. If generating random points, inf elements of x_U are set to max(L,xMax,x_L+L) xMax is the maximum of the finite x_U values.

L is 100 for nonlinear least squares, otherwise 1000.

Prob Problem description structure. The following fields are used:, continued

b_L	Lower bounds on linear constraints.
b_U	Upper bounds on linear constraints.
A	The linear constraint matrix.
c_L	Lower bounds on nonlinear constraints.
c_U	Upper bounds on nonlinear constraints.
PriLev	Print Level: 0 = Silent 1 = Display 2 summary rows 2 = Display some extra summary rows 5 = Print level 1 in tomRun call 6 = Print level 2 in tomRun call 7 = Print level 3 in tomRun call
xInit	Used in multiMin. See help for multiMin.
GO.localSolver	The local solver used to run all local optimizations. Default is the license dependent output of GetSolver('con',1,0).
optParam	Structure in Prob, Prob.optParam. Defines optimization parameters. Fields used:
MaxFunc	Max number of function evaluations in each subproblem
fGoal	Goal for function value f(x), if empty not used. If fGoal is reached, no further local optimizations are done
eps_f	Relative accuracy for function value, fTol == eps_f. Stop if abs(f-fGoal) <= abs(fGoal) * fTol , if fGoal = 0. Stop if abs(f-fGoal) <= fTol , if fGoal ==0. Default 1e-8.
bTol	Linear constraint feasibility tolerance. Default 1e-6
cTol	Nonlinear constraint feasibility tolerance. Default 1e-6
MIP	Structure in Prob, Prob.MIP. Defines integer optimization parameters. Fields used:
IntVars	If empty, all variables are assumed non-integer. If islogical(IntVars) (=all elements are 0/1), then 1 = integer variable, 0 = continuous variable. If any element >1, IntVars is the indices for integer variables.
nMax	Number of integer combinations possible, if empty multiMINLP computes nMax.
Rfac	Reduction factor for real variables in MINLP subproblem close to local multi-MINLP minimum. Bounds set to x_L = max(x_L,x-Rfac*(x_U-x_L)) and x_U = min(x_U,x+Rfac*(x_U-x_L)). Default 0.25.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

Result structure from the last good optimization step giving the best f(x) value, the possible global MINLP minimum.

The following fields in Result are changed by multiMINLP before return:

ExitFlag = 0 normal output, of if fGoal set and found.
= 1 A solution reaching the user defined fGoal was not found.
= 2 Unbounded problem.
= 4 Infeasible problem.

The Solver, SolverAlgorithm and ExitText fields are also reset.

A special field in Result is also returned, Result.multiMINLP:

xOpt Prob.N x k matrix with k distinct local optima, the test being norm(x_k-xOpt(:,i)) <= xEqTol*max(1,norm(x_k)) that if fulfilled assumes x_k to be to close to xOpt(:,i).

fOpt The k function values in the local optima xOpt(:,i),i=1,...,k.

Inform The Inform value returned by the local solver when finding each of the local optima xOpt(:,i); i=1,...,k. The Inform value can be used to judge the validity of the local minimum reported.

localTry Total number of local searches.

Iter Total number of iterations.

FuncEv Total number of function evaluations.

GradEv Total number of gradient evaluations.

HessEv Total number of Hessian evaluations.

ConstrEv Total number of constraint function evaluations.

ExitText Text string giving Inform information.

11.1.21 nlpSolve

Purpose
Solve general constrained nonlinear optimization problems.

nlpSolve solves problems of the form

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

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 $b_L, b_U \in \mathbb{R}^{m_2}$.

Calling Syntax
Result = nlpSolve(Prob, varargin)
Result = tomRun('nlpSolve', Prob);

Description of Inputs

Prob Problem description structure. The following fields are used:

A	Constraint matrix for linear constraints.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
c_L	Lower bounds on the general constraints.
c_U	Upper bounds on the general constraints.
x_L	Lower bounds on the variables.
x_U	Upper bounds on the variables.
x_0	Starting point.
PriLevOpt	Print level: 0 Silent, 1 Final result, 2 Each iteration, short, 3 Each iteration, more info, 4 Matrix update information.
FUNCS.f	Name of m-file computing the objective function $f(x)$.
FUNCS.g	Name of m-file computing the gradient vector $g(x)$.
FUNCS.H	Name of m-file computing the Hessian matrix $H(x)$.
FUNCS.c	Name of m-file computing the vector of constraint functions $c(x)$.
FUNCS.dc	Name of m-file computing the matrix of constraint normals $\partial c(x)/dx$.
FUNCS.d2c	Name of m-file computing the second derivatives of the constraints, weighted by an input Lagrange vector
NumDiff	How to obtain derivatives (gradient, Hessian).
ConsDiff	How to obtain the constraint derivative matrix.

Prob Problem description structure. The following fields are used:, continued

SolverQP	Name of the solver used for QP subproblems. If empty, the default solver is used. See GetSolver.m and tomSolve.m.
SolverFP	Name of the solver used for FP subproblems. If empty, the default solver is used. See GetSolver.m and tomSolve.m.
optParam	Structure with special fields for optimization parameters, see Table 141. Fields used are: *eps_g*, *eps_x*, *MaxIter*, *wait*, *size_x*, *method*, *IterPrint*, *xTol*, *bTol*, *cTol*, and *QN_InitMatrix*.

varargin Other parameters directly sent to low level routines.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Optimal point.
f_k	Function value at optimum.
g_k	Gradient value at optimum.
c_k	Value of constraints at optimum.
H_k	Hessian value at optimum.
v_k	Lagrange multipliers.
x_0	Starting point.
f_0	Function value at start.
cJac	Constraint Jacobian at optimum.
xState	State of each variable, described in Table 150.
bState	State of each linear constraint, described in Table 151.
cState	State of each general constraint.
Inform	Type of convergence.
ExitFlag	Flag giving exit status.
ExitText	0: Convergence. Small step. Constraints fulfilled. 1: Infeasible problem? 2: Maximal number of iterations reached. 3: No progress in either function value or constraint reduction.
Inform	1: Iteration points are close. 2: Small search direction 3: Function value below given estimate. Restart with lower fLow if minimum not reached. 4: Projected gradient small.

Result Structure with result from optimization. The following fields are changed:, continued

10: Karush-Kuhn-Tucker conditions fulfilled.

Iter	Number of iterations.
Solver	Solver used.
SolverAlgorithm	Solver algorithm used.
Prob	Problem structure used.

Description
The routine *nlpSolve* implements the Filter SQP by Roger Fletcher and Sven Leyffer presented in the paper [21].

M-files Used
tomSolve.m, *ProbCheck.m*, *iniSolve.m*, *endSolve.m*

See Also
conSolve, *sTrustr*

11.1.22 pdcoTL

Purpose
pdcoTL solves linearly constrained convex nonlinear optimization problems of the kind

$$\begin{array}{lrcccl} \min\limits_{x} & f(x) & & & & \\ \text{s/t} & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \end{array} \tag{19}$$

where $f(x)$ is a convex nonlinear function, $x, x_L, x_U \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b_L, b_U \in \mathbb{R}^m$.

Calling Syntax
Result=tomRun('pdco',Prob,...);

Description of Inputs

Prob Problem description structure. The following fields are used:

x_0	Initial x vector, used if non-empty.
A	The linear constraint matrix.
b_L,b_U	Lower and upper bounds for the linear constraints.
PriLevOpt	Print level in *pdsco* solver. If > 0: prints summary information.
SOL	Structure with SOL special parameters:
pdco	Options structure with fields as defined by *pdcoSet*.
d1	Primal regularization vector. Must be a positive vector (length n) or scalar, in which case $D1 = diag(d1)$ is used. Default: 10^{-4}.
d2	Dual regularization vector. Must be a positive vector (length m) or a scalar value, in which case $D2 = diag(d2)$ is used. Default: 10^{-4}.
y0	Initial dual parameters for linear constraints (default 0)
z0	Initial dual parameters for simple bounds (default $1/N$) *xsize,zsize* are used to scale (x, y, z). Good estimates should improve the performance of the barrier method.
xsize	Estimate of the biggest x at the solution. (default $1/N$)
zsize	Estimate of the biggest z at the solution. (default $1/N$)
optParam	Structure with optimization parameters. The following fields are used:
MaxIter	Maximum number of iterations. (*Prob.SOL.pdco.MaxIter*).
MinorIter	Maximum number of iterations in *LSQR*. (*Prob.SOL.pdco.LSQRMaxIter*).
eps_x	Accuracy for satisfying $x_1 .* z_1 = 0, x_2 .z_1 = 0$, where $z = z_1 - z_2$ and $z_1, z_2 > 0$. (*Prob.SOL.pdco.OptTol*)
bTol	Accuracy for satisfying $Ax + D_2 r = b$, $A^T y + z = \nabla f(x)$ and $x - x_1 = b_L, x + x2 = b_U$, where $x_1, x_2 > 0$ (*Prob.SOL.pdco.FeaTol*)

Prob Problem description structure. The following fields are used:, continued

wait	0 - solve the problem with default internal parameters; 1 - pause: allows interactive resetting of parameters. (*Prob.SOL.pdco.wait*)

Description of Outputs

Result Structure with result from optimization. The following fields are set by *pdcoTL*

x_k	Solution vector
f_k	Function value at optimum
g_k	Gradient of the function at the solution
H_k	Hessian of the function at the solution, diagonal only
x_0	Initial solution vector
f_0	Function value at start, x = x_0
xState	State of variables. Free == 0; On lower == 1; On upper == 2; Fixed == 3;
bState	State of linear constraints. Free == 0; Lower == 1; Upper == 2; Equality == 3;
v_k	Lagrangian multipliers (orignal bounds + constraints)
y_k	Lagrangian multipliers (for bounds + dual solution vector) The full $[z; y]$ vector as returned from *pdco*, including slacks and extra linear constraints after rewriting constraints: $-inf < b\_L < A * x < b\_U < inf$; non-inf lower AND upper bounds
ExitFlag	Tomlab Exit status from *pdco* MEX
Inform	*pdco* information parameter: 0 = Solution found;
0	Solution found
1	Too many iterations
2	Linesearch failed too often
Iter	Number of iterations
FuncEv	Number of function evaluations
GradEv	Number of gradient evaluations
HessEv	Number of Hessian evaluations
Solver	Name of the solver ('pdco')
SolverAlgorithm	Description of the solver

Description

pdco implements an primal-dual barrier method developed at Stanford Systems Optimization Laboratory (SOL).

The problem (19) is first reformulated into SOL PDCO form:

$$\begin{array}{lccccc} \min\limits_{x} & f(x) & & & & \\ \text{s/t} & x_L & \leq & x & \leq & x_U \\ & & & Ax & = & b \\ & & r \text{ unconstrained} & & & \end{array}$$

The problem actually solved by *pdco* is

$$\min_{x,r} \quad \phi(x) + \tfrac{1}{2}\|D_1 x\|^2 + \tfrac{1}{2}\|r\|^2$$

$$\begin{array}{lccccc} \text{s/t} & x_L & \leq & x & \leq & x_U \\ & Ax & + & D_2 r & = & b \end{array}$$

where D_1 and D_2 are positive-definite diagonal matrices defined from d_1, d_2 given in *Prob.SOL.d1* and *Prob.SOL.d2*.

In particular, d_2 indicates the accuracy required for satisfying each row of $Ax = b$. See *pdco.m* for a detailed discussion of D_1 and D_2. Note that in *pdco.m*, the objective $f(x)$ is denoted $\phi(x)$, $bl == x\_L$ and $bu == x\_U$.

Examples
Problem 14 and 15 in *tomlab/testprob/con_prob.m.m* are good examples of the use of *pdcoTL*.

M-files Used
pdcoSet.m, *pdco.m*, *Tlsqrmat.m*

See Also
pdscoTL.m

11.1.23 pdscoTL

Purpose
pdscoTL solves linearly constrained convex nonlinear optimization problems of the kind

$$\begin{array}{lccccc} \min\limits_{x} & f(x) & & & & \\ \text{s/t} & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \end{array} \tag{20}$$

where $f(x)$ is a convex separable nonlinear function, $x, x_L, x_U \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b_L, b_U \in \mathbb{R}^m$.

Calling Syntax
Result=tomRun('pdsco',Prob,...);

Description of Inputs

Prob Problem description structure. The following fields are used:

x_0	Initial x vector, used if non-empty.
A	The linear constraints coefficient matrix.
b_L,b_U	Lower and upper bounds for the linear constraints.
HessPattern	Non-zero pattern for the objective function. Only the diagonal is needed. Default if empty is the unit matrix.
PriLevOpt	Print level in *pdsco* solver. If > 0: prints summary information.
SOL	Structure with SOL special parameters:
pdco	Options structure with fields as defined by *pdscoSet*.
gamma	Primal regularization parameter.
delta	Dual regularization parameter.
y0	Initial dual parameters for linear constraints (default 0)
z0	Initial dual parameters for simple bounds (default $1/N$)
	xsize,zsize are used to scale (x, y, z). Good estimates should improve the performance of the barrier method.
xsize	Estimate of the biggest x at the solution. (default $1/N$)
zsize	Estimate of the biggest z at the solution. (default $1/N$)
optParam	Structure with optimization parameters. The following fields are used:
MaxIter	Maximum number of iterations. (*Prob.SOL.pdco.MaxIter*).
MinorIter	Maximum number of iterations in *LSQR* (*Prob.SOL.pdco.LSQRMaxIter*).

Prob Problem description structure. The following fields are used:, continued

eps_x	Accuracy for satisfying $x. * z = 0$
bTol	Accuracy for satisfying $Ax+r = b$, $A^T y+z = \nabla f(x)$ and $x-x_1 = b_L, x+x2 = b_U$, where $x_1, x_2 > 0$. (*Prob.SOL.pdco.FeaTol*)
wait	0 - solve the problem with default internal parameters; 1 - pause: allows interactive resetting of parameters. (*Prob.SOL.pdco.wait*)

Description of Outputs

Result Structure with result from optimization. The following fields are set by *pdscoTL*:

x_k	Solution vector
f_k	Function value at optimum
g_k	Gradient of the function at the solution
H_k	Hessian of the function at the solution, diagonal only
x_0	Initial solution vector
f_0	Function value at start, x = x_0
xState	State of variables. Free == 0; On lower == 1; On upper == 2; Fixed == 3;
bState	State of linear constraints. Free == 0; Lower == 1; Upper == 2; Equality == 3;
v_k	Lagrangian multipliers (orignal bounds + constraints)
y_k	Lagrangian multipliers (for bounds + dual solution vector) The full $[z; y]$ vector as returned from *pdsco*, including slacks and extra linear constraints after rewriting constraints: $-inf < b\_L < A * x < b\_U < inf$; non-inf lower AND upper bounds
ExitFlag	Tomlab Exit status from *pdsco* MEX
Inform	*pdsco* information parameter: 0 = Solution found;
0	Solution found
1	Too many iterations
2	Linesearch failed too often
Iter	Number of iterations
FuncEv	Number of function evaluations
GradEv	Number of gradient evaluations
HessEv	Number of Hessian evaluations
Solver	Name of the solver ('pdsco')
SolverAlgorithm	Description of the solver

Description
pdsco implements an primal-dual barrier method developed at Stanford Systems Optimization Laboratory (SOL). The problem (20) is first reformulated into SOL PDSCO form:

$$\begin{array}{lllll} \min\limits_{x} & f(x) & & \\ \text{s/t} & x & \geq & x_U \\ & Ax & = & b. \end{array}$$

The problem actually solved by *pdsco* is

$$\min_{x,r} \quad f(x) + \tfrac{1}{2}\|\gamma x\|^2 + \tfrac{1}{2}\|r/\delta\|^2$$

$$\begin{array}{llllll} \text{s/t} & & & x & \geq & 0 \\ & Ax & + & r & = & b \\ & r \text{ unconstrained} & & & & \end{array}$$

where γ is the primal regularization parameter, typically small but 0 is allowed. Furthermore, δ is the dual regularization parameter, typically small or 1; must be strictly greater than zero.

With positive γ, δ the primal-dual solution (x, y, z) is bounded and unique.

See *pdsco.m* for a detailed discussion of γ and δ. Note that in *pdsco.m*, the objective $f(x)$ is denoted $\phi(x)$, $bl == x\_L$ and $bu == x\_U$.

Examples
Problem 14 and 15 in `tomlab/testprob/con_prob.m` are good examples of the use of *pdscoTL*.

M-files Used
pdscoSet.m, *pdsco.m*, *Tlsqrmat.m*

See Also
pdcoTL.m

11.1.24 qpSolve

Purpose
Solve general quadratic programming problems.

qpSolve solves problems of the form

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

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

Calling Syntax
Result = qpSolve(Prob) or
Result = tomRun('qpSolve', Prob, 1);

Description of Inputs

Prob Problem description structure. The following fields are used:

QP.F	Constant matrix, the Hessian.
QP.c	Constant vector.
A	Constraint matrix for linear constraints.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
x_L	Lower bounds on the variables.
x_U	Upper bounds on the variables.
x_0	Starting point.
optParam	Structure with special fields for optimization parameters, see Table 141. Fields used are: *eps_f*, *eps_Rank*, *MaxIter*, *wait*, *bTol* and *PriLev*.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Optimal point.
f_k	Function value at optimum.
g_k	Gradient value at optimum.
H_k	Hessian value at optimum.
v_k	Lagrange multipliers.

Result Structure with result from optimization. The following fields are changed:, continued

x_0	Starting point.
f_0	Function value at start.
xState	State of each variable, described in Table 150 .
Iter	Number of iterations.
ExitFlag	0: OK, see *Inform* for type of convergence. 2: Can not find feasible starting point *x_0*. 3: Rank problems. Can not find any solution point. 4: Unbounded solution. 10: Errors in input parameters.
Inform	If $ExitFlag > 0$, $Inform = ExitFlag$, otherwise $Inform$ show type of convergence: 0: Unconstrained solution. 1: $\lambda \geq 0$. 2: $\lambda \geq 0$. No second order Lagrange mult. estimate available. 3: λ and 2nd order Lagr. mult. positive, problem is not negative definite. 4: Negative definite problem. 2nd order Lagr. mult. positive, but releasing variables leads to the same working set.
Solver	Solver used.
SolverAlgorithm	Solver algorithm used.
Prob	Problem structure used.

Description
Implements an active set strategy for Quadratic Programming. For negative definite problems it computes eigenvalues and is using directions of negative curvature to proceed. To find an initial feasible point the Phase 1 LP problem is solved calling *lpSimplex*. The routine is the standard QP solver used by *nlpSolve*, *sTrustr* and *conSolve*.

M-files Used
ResultDef.m, *lpSimplex.m*, *tomSolve.m*, *iniSolve.m*, *endSolve.m*

See Also
qpBiggs, *qpe*, *qplm*, *nlpSolve*, *sTrustr* and *conSolve*

11.1.25 slsSolve

Purpose
Find a Sparse Least Squares (sls) solution to a constrained least squares problem with the use of any suitable TOMLAB NLP solver.

slsSolve solves problems of the type:

$$\begin{array}{lllll} \min\limits_{x} & \frac{1}{2} r(x)^T r(x) & & & \\ \text{subject to} & x_L & \leq \quad x & \leq & x_U \\ & b_L & \leq \quad Ax & \leq & b_U \\ & c_L & \leq \quad c(x) & \leq & c_U \end{array}$$

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

The use of *slsSolve* is mainly for large, sparse problems, where the structure in the Jacobians of the residuals and the nonlinear constraints are utilized by a sparse NLP solver, e.g. *SNOPT*.

Calling Syntax
Result=slsSolve(Prob,PriLev)

Description of Inputs

Prob Problem description structure. Should be created in the **cls** format, preferably by calling *Prob=clsAssign(...)* if using the **TQ** format.

slsSolve uses two special fields in *Prob*:

SolverL2	Text string with name of the NLP solver used for solving the reformulated problem. Valid choices are *conSolve*, *nlpSolve*, *sTrustr*, *clsSolve*. Suitable SOL solvers, if available: *minos*, *snopt*, *npopt*.
L2Type	Set to 1 for standard constrained formulation. Currently this is the only allowed choice.

All other fields should be set as expected by the nonlinear solver selected. In particular:

A	Linear constraint matrix.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
c_L	Upper bounds on the nonlinear constraints.
c_U	Lower bounds on the nonlinear constraints.
x_L	Lower bounds on the variables.
x_U	Upper bounds on the variables.
x_0	Starting point.

Prob Problem description structure. Should be created in the **cls** format, preferably by calling *Prob=clsAssign(...)* if using the **TQ** format, continued

ConsPattern	The nonzero pattern of the constraint Jacobian.
JacPattern	The nonzero pattern of the residual Jacobian. Note that *Prob.LS.y* must be of correct length if *JacPattern* is empty (but ConsPattern is not). *slsSolve* will create the new *Prob.ConsPattern* to be used by the nonlinear solver using the information in the supplied *ConsPattern* and *JacPattern*.
PriLev	Print level in *slsSolve*. Default value is 2.
0	Silent except for error messages.
> 1	Print summary information about problem transformation. *slsSolve* calls *PrintResult(Result,PriLev)*.
2	Standard output in PrintResult.

Description of Outputs

Result Structure with results from optimization. The contents of *Result* depend on which nonlinear solver was used to solved

slsSolve transforms the following fields of *Result* back to the format of the original problem:

x_k	Optimal point.
r_k	Residual at optimum.
J_k	Jacobian of residuals at optimum.
c_k	Nonlinear constraint vector at optimum.
v_k	Lagrange multipliers.
g_k	The gradient vector is calculated as $J\_k^T \cdot r\_k$.
cJac	Jacobian of nonlinear constraints at optimum.
x_0	Starting point.
xState	State of variables at optimum.
cState	State of constraints at optimum.
Result.Prob	The problem structure defining the reformulated problem.

Description

The constrained least squares problem is solved in slsSolve by rewriting the problem as a general constrained optimization problem. A set of m (the number of residuals) extra variables $z = (z_1, z_2, \ldots, z_m)$ are added at the end of the vector of unknowns. The reformulated problem

$$\begin{array}{lrcccl} \min\limits_{x} & \frac{1}{2} z^T z & & & & \\ \text{subject to} & x_L & \leq & (x_1, x_2, \ldots, x_n) & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \\ & 0 & \leq & r(x) - z & \leq & 0 \end{array}$$

is then solved by the solver given by *Prob.SolverL2*.

Examples
slsDemo.m

M-files Used
iniSolve.m, *GetSolver.m*

11.1.26 sTrustr

Purpose
Solve optimization problems constrained by a convex feasible region.

sTrustr solves problems of the form

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

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 $b_L, b_U \in \mathbb{R}^{m_2}$.

Calling Syntax
Result = sTrustr(Prob, varargin)

Description of Inputs

Prob Problem description structure. The following fields are used:

A	Constraint matrix for linear constraints.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
c_L	Lower bounds on the general constraints.
c_U	Upper bounds on the general constraints.
x_L	Lower bounds on the variables.
x_U	Upper bounds on the variables.
x_0	Starting point.
FUNCS.f	Name of m-file computing the objective function $f(x)$.
FUNCS.g	Name of m-file computing the gradient vector $g(x)$.
FUNCS.H	Name of m-file computing the Hessian matrix $H(x)$.
FUNCS.c	Name of m-file computing the vector of constraint functions $c(x)$.
FUNCS.dc	Name of m-file computing the matrix of constraint normals $\partial c(x)/dx$.
optParam	Structure with special fields for optimization parameters, see Table 141. Fields used are: *eps_f*, *eps_g*, *eps_c*, *eps_x*, *eps_Rank*, *MaxIter*, *wait*, *size_x*, *size_f*, *xTol*, *LowIts*, *PriLev*, *method* and *QN_InitMatrix*.
PartSep	Structure with special fields for partially separable functions, see Table 142.

varargin Other parameters directly sent to low level routines.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Optimal point.
f_k	Function value at optimum.
g_k	Gradient value at optimum.
c_k	Value of constraints at optimum.
H_k	Hessian value at optimum.
v_k	Lagrange multipliers.
x_0	Starting point.
f_0	Function value at start.
cJac	Constraint Jacobian at optimum.
xState	State of each variable, described in Table 150.
Iter	Number of iterations.
ExitFlag	Flag giving exit status.
Inform	Binary code telling type of convergence: 1: Iteration points are close. 2: Projected gradient small. 3: Iteration points are close and projected gradient small. 4: Relative function value reduction low for *LowIts* iterations. 5: Iteration points are close and relative function value reduction low for LowIts iterations. 6: Projected gradient small and relative function value reduction low for LowIts iterations. 7: Iteration points are close, projected gradient small and relative function value reduction low for LowIts iterations. 8: Too small trust region. 9: Trust region small. Iteration points close. 10: Trust region and projected gradient small. 11: Trust region and projected gradient small, iterations close. 12: Trust region small, Relative f(x) reduction low. 13: Trust region small, Relative f(x) reduction low. Iteration points are close. 14: Trust region small, Relative f(x) reduction low. Projected gradient small. 15: Trust region small, Relative f(x) reduction low. Iteration points close, Projected gradient small. 101: Maximum number of iterations reached. 102: Function value below given estimate. 103: Convergence to saddle point (eigenvalues computed).
Solver	Solver used.
SolverAlgorithm	Solver algorithm used.
Prob	Problem structure used.

Description
The routine *sTrustr* is a solver for general constrained optimization, which uses a structural trust region algorithm combined with an initial trust region radius algorithm (*itrr*). The feasible region defined by the constraints must be convex. The code is based on the algorithms in [13] and [67]. BFGS or DFP is used for the Quasi-Newton update, if the analytical Hessian is not used. *sTrustr* calls internal routine *itrr*.

M-files Used
qpSolve.m, *tomSolve.m*, *iniSolve.m*, *endSolve.m*

See Also
conSolve, *nlpSolve*, *clsSolve*

11.1.27 Tfmin

Purpose
Minimize function of one variable. Find miniumum x in [x_L, x_U] for function Func within tolerance xTol. Solves using Brents minimization algorithm. Reference: "Computer Methods for Mathematical Computations", Forsythe, Malcolm, and Moler, Prentice-Hall, 1976.

Calling Syntax
[x, nFunc] = Tfmin(Func, x_L, x_U, xTol, Prob)

Description of Inputs

Variable	Description
Func	Function of x to be minimized. Func must be defined as: f = Func(x) if no 5th argument Prob is given or f = Func(x, Prob) if 5th argument Prob is given.
x_L	Lower bound on x.
x_U	Upper bound on x.
xTol	Tolerance on accuracy of minimum.

Prob	Structure (or any Matlab variable) sent to Func. If many parameters are to be sent to Func set them in Prob as a structure. Example for parameters a and b:
	
	Prob.user.a = a;
	Prob.user.b = b;
	[x, nFunc] = Tfmin('myFunc',0,1,1E-5,Prob);
	
	In myFunc:
	
	function f = myFunc(x, Prob)
	a = Prob.user.a;
	b = Prob.user.b;
	f = "matlab expression dependent of x, a and b";

Description of Outputs

Variable	Description
x	Solution.
nFunc	Number of calls to Func.

11.1.28 Tfzero

Purpose
Tfzero, TOMLAB fzero routine.

Find a zero for $f(x)$ in the interval $[x_L, x_U]$. Tfzero searches for a zero of a function $f(x)$ between the given scalar values x_L and x_U until the width of the interval (xLow, xUpp) has collapsed to within a tolerance specified by the stopping criterion, $abs(xLow - xUpp) <= 2. * (RelErr * abs(xLow) + AbsErr)$. The method used is an efficient combination of bisection and the secant rule and is due to T. J. Dekker.

Calling Syntax
[xLow, xUpp, ExitFlag] = Tfzero(x_L, x_U, Prob, x_0, RelErr, AbsErr)

Description of Inputs

Variable	Description
x_L	Lower limit on the zero x to f(x).
x_U	Upper limit on the zero x to f(x).
Prob	Structure, sent to Matlab routine ZeroFunc. The function name should be set in Prob.FUNCS.f0. Only the function will be used, not the gradient.
x_0	An initial guess on the zero to f(x). If empty, x_0 is set as the middle point in [x_L, x_U].
RelErr	Relative error tolerance, default 1E-7.
AbsErr	Absolute error tolerance, default 1E-14.

Description of Outputs

Variable	Description
xLow	Lower limit on the zero x to f(x).
xUpp	Upper limit on the zero x to f(x).
ExitFlag	Status flag 1,2,3,4,5. 1: xLow is within the requested tolerance of a zero. The interval (xLow, xUpp) collapsed to the requested tolerance, the function changes sign in (xLow, xUpp), and f(x) decreased in magnitude as (xLow, xUpp) collapsed. 2: f(xLow) = 0. However, the interval (xLow, xUpp) may not have collapsed to the requested tolerance. 3: xLow may be near a singular point of f(x). The interval (xLow, xUpp) collapsed to the requested tolerance and the function changes sign in (xLow, xUpp), but f(x) increased in magnitude as (xLow, xUpp) collapsed, i.e. $abs(f(xLow)) > max(abs(f(xLow - IN)), abs(f(xUpp - IN)))$. 4: No change in sign of f(x) was found although the interval (xLow, xUpp) collapsed to the requested tolerance. The user must examine this case and decide whether xLow is near a local minimum of f(x), or xLow is near a zero of even multiplicity, or neither of these.

Variable	Description
	5: Too many (> 500) function evaluations used.

11.1.29 ucSolve

Purpose
Solve unconstrained nonlinear optimization problems with simple bounds on the variables.

ucSolve solves problems of the form

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

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

Calling Syntax
Result = ucSolve(Prob, varargin)

Description of Inputs

Prob Problem description structure. The following fields are used:

x_L	Lower bounds on the variables.
x_U	Upper bounds on the variables.
x_0	Starting point.
FUNCS.f	Name of m-file computing the objective function $f(x)$.
FUNCS.g	Name of m-file computing the gradient vector $g(x)$.
FUNCS.H	Name of m-file computing the Hessian matrix $H(x)$.
f_Low	Lower bound on function value.
Solver.Alg	Solver algorithm to be run: 0: Gives default, either Newton or BFGS. 1: Newton with subspace minimization, using SVD. 2: Safeguarded BFGS with inverse Hessian updates (standard). 3: Safeguarded BFGS with Hessian updates. 4: Safeguarded DFP with inverse Hessian updates. 5: Safeguarded DFP with Hessian updates. 6: Fletcher-Reeves CG. 7: Polak-Ribiere CG. 8: Fletcher conjugate descent CG-method.
Solver.Method	Method used to solve equation system: 0: SVD (default). 1: LU-decomposition. 2: LU-decomposition with pivoting. 3: Matlab built in QR. 4: Matlab inversion. 5: Explicit inverse.

Prob Problem description structure. The following fields are used:, continued

Solver.Method	Restart or not for C-G method: 0: Use restart in CG-method each n:th step. 1: Use restart in CG-method each n:th step.
LineParam	Structure with line search parameters, see routine *LineSearch* and Table 140.
optParam	Structure with special fields for optimization parameters, see Table 141. Fields used are: *eps_absf*, *eps_f*, *eps_g*, *eps_x*, *eps_Rank*, *MaxIter*, *wait*, *size_x*, *xTol*, *size_f*, *LineSearch*, *LineAlg*, *xTol*, *IterPrint* and *QN_InitMatrix*.
PriLevOpt	Print level.

varargin Other parameters directly sent to low level routines.

Description of Outputs

Result Structure with result from optimization. The following fields are changed:

x_k	Optimal point.
f_k	Function value at optimum.
g_k	Gradient value at optimum.
H_k	Hessian value at optimum.
B_k	Quasi-Newton approximation of the Hessian at optimum.
v_k	Lagrange multipliers.
x_0	Starting point.
f_0	Function value at start.
xState	State of each variable, described in Table 150.
Iter	Number of iterations.
ExitFlag	0 if convergence to local min. Otherwise errors.
Inform	Binary code telling type of convergence: 1: Iteration points are close. 2: Projected gradient small. 4: Relative function value reduction low for *LowIts* iterations. 101: Maximum number of iterations reached. 102: Function value below given estimate. 104: Convergence to a saddle point.
Solver	Solver used.
SolverAlgorithm	Solver algorithm used.
Prob	Problem structure used.

Description

The solver *ucSolve* includes several of the most popular search step methods for unconstrained optimization. The

search step methods included in *ucSolve* are: the Newton method, the quasi-Newton BFGS and DFP methods, the Fletcher-Reeves and Polak-Ribiere conjugate-gradient method, and the Fletcher conjugate descent method. The quasi-Newton methods may either update the inverse Hessian (standard) or the Hessian itself. The Newton method and the quasi-Newton methods updating the Hessian are using a subspace minimization technique to handle rank problems, see Lindström [53]. The quasi-Newton algorithms also use safe guarding techniques to avoid rank problem in the updated matrix. The line search algorithm in the routine *LineSearch* is a modified version of an algorithm by Fletcher [20]. Bound constraints are treated as described in Gill, Murray and Wright [28].

The accuracy in the line search is critical for the performance of quasi-Newton BFGS and DFP methods and for the CG methods. If the accuracy parameter *Prob.LineParam.sigma* is set to the default value 0.9, *ucSolve* changes it automatically according to:

Prob.Solver.Alg	*Prob.LineParam.sigma*
4,5 (DFP)	0.2
6,7,8 (CG)	0.01

M-files Used
ResultDef.m, *LineSearch.m*, *iniSolve.m*, *tomSolve.m*, *endSolve.m*

See Also
clsSolve

11.1.30 Additional solvers

Documentation for the following solvers is only available at http://tomopt.com and in the m-file help.

- goalSolve - For sparse multi-objective goal attainment problems, with linear and nonlinear constraints.
- Tlsqr - Solves large, sparse linear least squares problem, as well as unsymmetric linear systems.
- lsei - For linearly constrained least squares problem with both equality and inequality constraints.
- Tnnls - Also for linearly constrained least squares problem with both equality and inequality constraints.
- qld - For convex quadratic programming problem.

12 TOMLAB Utility Functions

In the following subsections the driver routine and the utility functions in TOMLAB will be described.

12.1 tomRun

Purpose
General multi-solver driver routine for TOMLAB.

Calling Syntax

Result = tomRun(Solver, Prob, PriLev, ask)	Call *Solver* on the problem defined in structure *Prob*
Result = tomRun(Solver, probFile, probNumber, Prob, PriLev, ask)	Call *Solver* on problem *probNumber* in Init File *probFile.m*
Result = tomRun(Solver, probType, probNumber, PriLev, ask)	Call *Solver* on problem number *probNumber* in the default Init File for problem type *probType*
tomRun(probType)	Display all solvers for *probType*
tomRun	Display all available solvers for all problem types

Description of Inputs

Solver The name of the solver that should be used to optimize the problem. If the solver may run several different optimization algorithms, then the values of *Prob.Solver.Alg* and *Prob.optParam.Method* determines which algorithm and method to be used.

Prob Problem description structure, see Table 134, page 220.

ask Flag if questions should be asked during problem definition.
- $ask < 0$ Use values in uP if defined or defaults.
- $ask = 0$ Use defaults.
- $ask \geq 1$ Ask questions in *probFile*.
- $ask = [\,]$ If $uP = [\,]$, $ask = -1$, else $ask = 0$.

PriLev Print level when displaying the result of the optimization in the routine *PrintResult*. See Section 12.12 page 199.

$PriLev = 0$	No output.
$PriLev = 1$	Final result, shorter version.
$PriLev = 2$	Final result.
$PriLev = 3$	Full results.

The printing level in the optimization solver is controlled by setting the parameter *Prob.PriLevOpt*.

probFile User problem Init File.

probNumber Problem number in *probFile*. $probNumber = 0$ gives a menu in *probFile*.

Description of Outputs

Result Structure with result from optimization, see Table 149.

Description

The driver routine *tomRun* is called from the command line. If called with less than the required two parameters, a list of available solvers are printed.

M-files Used

PrintResult.m, *probInit.m*, *mkbound.m*

12.2 addPwLinFun

Purpose
Adds piecewise linear function to a TOMLAB MIP problem.

Calling Syntax
There are two ways to call addPwLinFun:

Syntax 1: function Prob = addPwLinFun(Prob, 1, type, var, funVar, point, slope, a, fa)

Syntax 2: function Prob = addPwLinFun(Prob, 2, type, var, funVar, firstSlope, point, value, lastSlope)

Description of Inputs

Prob	The problem to add the function to.
input	Flag indicating syntax used.
type	A string telling whether to construct a general MIP problem or to construct an MIP problem only solvable by CPLEX. Possible values: 'mip', 'cplex'.
var	The number of the variable on which the piecewise linear function depends. Must exist in the problem already.
funVar	The number of the variable which will be equal to the piecewise linear function. Must exist in the problem already.
firstSlope	Syntax 2 only. The slope of the piecewise linear function left of the first point, point(1).
point	An array of break points. Must be sorted. If two values occur twice, there is a step at that point. Length r.
slope	Syntax 1 only. An array of the slopes of the segments. slope(i) is the slope between point(i-1) and point(i). slope(1) is the slope of the function left of point(1). slope(r+1) is the slope of the function right of point(r). If points(i-1) == points(i), slope(i) is the height of the step.
value	Syntax 2 only. The values of the piecewise linear function at the points given in point. f(point(i)) = value(i). If point(i-1) == point(i), value(i-1) is the right limit of the value at the point, and value(i) is the left limit of the value at the point.
lastSlope	Syntax 2 only. The slope of the piecewise linear function right of the last point, point(r).
a, fa	Syntax 1 only. The value of the piecewise linear function at point a is equal to fa. f(a) = fa, that is.

Description of Outputs

Prob	The new problem structure with the piecewise linear function added. New variables and linear constraints added. (MIP problem)

Description

This function will make one already existing variable of the problem to be constrained equal to a piecewise linear function of another already existing variable in the problem. The independent variable must be bounded in both directions.

The variable constrained to be equal to a piecewise linear function can be used like any other variable; in constraints or the objective function.

Depending on how many segments the function consists of, a number of new variables and constraints are added to the problem.

Increasing the upper bound (x_U) or decreasing the lower bound (x_L) of the independent variable after calling this function will ruin the piecewise linear function.

If the problem is to be solved by CPLEX, set type = 'cplex' to enhance performance. Otherwise, let type = 'mip'. NOTICE! You can not solve a problem with another solver than CPLEX if type = 'cplex'.

12.3 binbin2lin

Purpose
Adds constraints when modeling with binary variables which is the product of two other variables.

Calling Syntax
Prob = binbin2lin(Prob, idx4, idx1, idx2, idx3)

Description of Inputs

Prob	Problem structure to be converted.
idx4	Indices for b4 variables.
idx1	Indices for b1 variables.
idx2	Indices for b2 variables.
idx3	Indices for b3 variables.

Description of Outputs

Prob	Problem structure with added constraints.

Description

$b4 = b1 * b2$. The problem should be built with the extra variable b4 in place of the b1*b2 products. The indices of the unique product variables are needed to convert the problem properly.

Three inequalities are added to the problem:

$b4 <= b1$
$b4 <= b2$
$b4 >= b1 + b2 - 1$

By adding this b4 will always be the product of b1 and b2.

The routine also handles products of three binary variables. $b4 = b1 * b2 * b3$. The following constraints are then added:

$b4 <= b1$
$b4 <= b2$
$b4 <= b3$
$b4 >= b1 + b2 + b3 - 1$

12.4 bincont2lin

Purpose
Adds constraints when modeling with binary variables which are multiplied by integer or continuous variables. This is the most efficient way to get rid off quadratic objectives or constraints.

Calling Syntax
Prob = bincont2lin(Prob, idx_prod, idx_bin, idx_cont)

Description of Inputs

Prob	Problem structure to be converted.
idx_prod	Indices for product variables.
idx_bin	Indices for binary variables.
idx_cont	Indices for continuous/integer variables.

Description of Outputs

Prob	Problem structure with added constraints.

Description

$prod = bin * cont$. The problem should be built with the extra variables prod in place of the $bin * cont$ products. The indices of the unique product variables are needed to convert the problem properly.

Three inequalities are added to the problem:

$prod <= cont$
$prod >= cont - x_U * (1 - bin)$
$prod <= x_U * bin$

By adding this prod will always equal $bin * cont$.

12.5 checkFuncs

Purpose
TOMLAB routine for verifying user supplied routines. The routine could be used for general debugging.

Calling Syntax
exitFlag = checkFuncs(Prob, Solver, PriLev)

Description of Inputs

Prob	Problem structure created with assign routine.
Solver	Solver that will be used. For example 'knitro' (default).
PriLev	0 - suppress warnings (info), 1 - full printing (default).

Description of Outputs

exitFlag	0 if no errors.

12.6 checkDerivs

Purpose
TOMLAB routine for verifying derivatives of user supplied routines.

Calling Syntax
[exitFlag,output] = checkDerivs(Prob, x_k, PriLev, ObjDerLev, ConsDerLev, AbsTol)

Description of Inputs

Prob	Problem structure created with assign routine.
x_k	Point the check derivatives for. Default x_0 or $(x_L + x_U)/2$. x_L and x_U have to be within 1e5.
PriLev	Print Level, default 1. (0-1 valid).
ObjDerLev	Depth for objective derivative check, 1 - checks gradient, 2 checks gradient and Hessian. Default 2 or level of derivatives supplied.
ConsDerLev	Depth for constraint derivative check, 1 - checks Jacobian, 2 checks Jacobian and 2nd part of the Hessian to the Lagrangian function. Default 2 or level of derivatives supplied.
AbsTol	Absolute tolerance for errors. Default [1e-5 1e-3 1e-4 1e-3 1e-4] (g H dc d2c J).

Description of Outputs

exitFlag	If exitFlag $= 0$ a problem exist. See output for more information. Binary indicates where problem is: 01011. $1 + 2 + 8 = 13$. Problems everywhere but 'dc', 'J'. 11111 = 'J' 'd2c' 'dc' 'H' 'g'.
output	Structure containing analysis information. g,H,dc,d2c,J Structure with results. minErr: The smallest error. avgErr: The average error. maxErr: The largest error. idx: Index for elements with errors. exitFlag: 1 if problem with the function.

See Also
runtest

12.7 cpTransf

Purpose
Transform general convex programs on the form

$$\begin{array}{lccccc} \min\limits_{x} & f(x) & & & & \\ s/t & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \end{array}$$

where $x, x_L, x_U \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b_L, b_U \in \mathbb{R}^m$, to other forms.

Calling Syntax
[AA, bb, meq] = cpTransf(Prob, TransfType, makeEQ, LowInf)

Description of Inputs

Prob	Problem description structure. The following fields are used:
	QP.c Constant vector c in $c^T x$.
	A Constraint matrix for linear constraints.
	b_L Lower bounds on the linear constraints.
	b_U Upper bounds on the linear constraints.
	x_L Lower bounds on the variables.
	x_U Upper bounds on the variables.
TransfType	Type of transformation, see the description below.
MakeEQ	Flag, if set true, make standard form (all equalities).
LowInf	Variables equal to $-Inf$ or variables $< LowInf$ are set to $LowInf$ before transforming the problem. Default -10^{-4}. $\|LowInf\|$ are limit if upper bound variables are to be used.

Description of Outputs

AA	The expanded linear constraint matrix.
bb	The expanded upper bounds for the linear constraints.
meq	The first *meq* equations are equalities.

Description
If $TransType = 1$ the program is transformed into the form

$$\begin{array}{lccc} \min\limits_{x} & f(x - x_L) & & \\ s/t & AA(x - x_L) & \leq & bb \\ & x - x_L & \geq & 0 \end{array}$$

where the first *meq* constraints are equalities. Translate back with (fixed variables do not change their values):

```
x(~x_L==x_U) = (x-x_L) + x_L(~x_L==x_U)
```

If $TransType = 2$ the program is transformed into the form

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

where the first meq constraints are equalities.

If $TransType = 3$ the program is transformed into the form

$$\begin{array}{lccc} \min\limits_{x} & f(x) & & \\ s/t & AAx & \leq & bb \\ & x & \geq & x_L \end{array}$$

where the first meq constraints are equalities.

12.8 estBestHessian

Purpose
estBestHessian estimates the best Hessian. Result.x_k(:,1) will be used for the estimation. The best step-size is estimated by TOMLAB. If the gradient is given it will be used. The analytical hessian is returned if given.

Calling Syntax
[g_k, H_k] = estBestHessian(Result);

Description of Inputs

Result	Problem structure to be converted.

Description of Outputs

g_k	The gradient at Result.x_k(:,1).
H_k	Hessian of the objective at Result.x_k(:,1).

12.9 lls2qp

Purpose
Converts an lls problem to a new problem based on the formula below. Only the objective function is affected. The problem can be of any type with an LLS objective.

$$\begin{gathered}\min_x \quad f(x) = \tfrac{1}{2}||Cx - d|| = \\ 0.5 * (y - Cx)' * (y - Cx) = 0.5 * (y'y - 2y'Cx + x'C'Cx) = \\ 0.5 * y'y + 0.5 * x'Fx + c'x \\ \\ F = C'C \\ c' = -y'C \\ const = \tfrac{1}{2}y'y\end{gathered} \tag{21}$$

Calling Syntax
qpProb = lls2qp(Prob, IntVars)

Description of Inputs

Prob.LS.C	The linear matrix in $0.5 * \|\|y - Cx\|\|$.
Prob.LS.y	The constant vector in $0.5 * \|\|y - Cx\|\|$.

Description of Outputs

qpProb	The converted problem.

Description
If the problem is a linear least squares problem a qp problem is created. The new problem may have integer variables. Create the problem with llsAssign then use this routine.

If the problem has nonlinear constraints an nlp is created. The new problem may have integer variables. Create the problem with conAssign or minlpAssign, the set the fields Prob.LS.C and Prob.LS.y

12.10 LineSearch

Purpose

LineSearch solves line search problems of the form

$$\min_{0<\alpha_{\min}\leq\alpha\leq\alpha_{\max}} f(x^{(k)}+\alpha p)$$

where $x, p \in \mathbb{R}^n$.

Calling Syntax

Result = LineSearch(f, g, x, p, f_0, g_0, LineParam, alphaMax, pType, PriLev, varargin)

Description of Inputs

f	Name of m-file computing the objective function $f(x)$.
g	Name of m-file computing the gradient vector $g(x)$.
x	Current iterate x.
p	Search direction p.
f_0	Function value at $\alpha = 0$.
g_0	Gradient at $\alpha = 0$, the directed derivative at the present point.
LineParam	Structure with line search parameters 140, the following fields are used: *LineAlg* Type of line search algorithm, 0 = quadratic interpolation, 1 = cubic interpolation. *fLowBnd* Lower bound on the function value at optimum. *sigma InitStepLength rho tau1 tau2 tau3 eps1 eps2* see Table 140.
alphaMax	Maximal value of step length α.
pType	Type of problem: 0 Normal problem. 1 Nonlinear least squares. 2 Constrained nonlinear least squares. 3 Merit function minimization. 4 Penalty function minimization.
PriLev	Printing level: $PriLev > 0$ Writes a lot of output in *LineSearch*. $PriLev > 3$ Writes a lot of output in *intpol2* and *intpol3*.
varargin	Other parameters directly sent to low level routines.

Description of Outputs

Result Result structure with fields:

alpha	Optimal line search step α.
f_alpha	Optimal function value at line search step α.
g_alpha	Optimal gradient value at line search step α.
alphaVec	Vector of trial step length values.
r_k	Residual vector if Least Squares problem, otherwise empty.
J_k	Jacobian matrix if Least Squares problem, otherwise empty.
f_k	Function value at $x + \alpha p$.
g_k	Gradient value at $x + \alpha p$.
c_k	Constraint value at $x + \alpha p$.
dc_k	Constraint gradient value at $x + \alpha p$.

Description

The function *LineSearch* together with the routines *intpol2* and *intpol3* implements a modified version of a line search algorithm by Fletcher [20]. The algorithm is based on the Wolfe-Powell conditions and therefore the availability of first order derivatives is an obvious demand. It is also assumed that the user is able to supply a lower bound f_{Low} on $f(\alpha)$. More precisely it is assumed that the user is prepared to accept any value of $f(\alpha)$ for which $f(\alpha) \leq f_{Low}$. For example in a nonlinear least squares problem $f_{Low} = 0$ would be appropriate.

LineSearch consists of two parts, the *bracketing phase* and the *sectioning phase*. In the bracketing phase the iterates $\alpha^{(k)}$ moves out in an increasingly large jumps until either $f \leq f_{Low}$ is detected or a bracket on an interval of acceptable points is located. The sectioning phase generates a sequence of brackets $\left[a^{(k)}, b^{(k)}\right]$ whose lengths tend to zero. Each iteration pick a new point $\alpha^{(k)}$ in $\left[a^{(k)}, b^{(k)}\right]$ by minimizing a quadratic or a cubic polynomial which interpolates $f\left(a^{(k)}\right)$, $f'\left(a^{(k)}\right)$, $f\left(b^{(k)}\right)$ and $f'\left(b^{(k)}\right)$ if it is known. The sectioning phase terminates when $\alpha^{(k)}$ is an acceptable point.

12.11 preSolve

Purpose
Simplify the structure of the constraints and the variable bounds in a linear constrained program.

Calling Syntax
Prob = preSolve(Prob)

Description of Inputs

Prob Problem description structure. The following fields are used:
- *A* Constraint matrix for linear constraints.
- *b_L* Lower bounds on the linear constraints.
- *b_U* Upper bounds on the linear constraints.
- *x_L* Lower bounds on the variables.
- *x_U* Upper bounds on the variables.

Description of Outputs

Prob Problem description structure. The following fields are changed:
- *A* Constraint matrix for linear constraints.
- *b_L* Lower bounds on the linear constraints, set to NaN for redundant constraints.
- *b_U* Upper bounds on the linear constraints, set to NaN for redundant constraints.
- *x_L* Lower bounds on the variables.
- *x_U* Upper bounds on the variables.

Description
The routine *preSolve* is an implementation of those presolve analysis techniques described by Gondzio in [36], which is applicable to general linear constrained problems. See [7] for a more detailed presentation.

preSolve consists of the two routines *clean* and *mksp*. They are called in the sequence *clean*, *mksp*, *clean*. The second call to *clean* is skipped if the *mksp* routine could not remove a single nonzero entry from A.

clean consists of two routines, *r_rw_sng* that removes singleton rows and *el_cnsts* that improves variable bounds and uses them to eliminate redundant and forcing constraints. Both *r_rw_sng* and *el_cnsts* check if empty rows appear and eliminate them if so. That is handled by the routine *emptyrow*. In *clean* the calls to *r_rw_sng* and *el_cnsts* are repeated (in given order) until no further reduction is obtained.

Note that rows are actually not deleted or removed, instead *preSolve* indicates that constraint i is redundant by setting $b\_L(i) = b\_U(i) = NaN$ and leaves to the calling routine to decide what to do with those constraints.

12.12 PrintResult

Purpose
Prints the result of an optimization.

Calling Syntax
PrintResult(Result, PriLev)

Description of Inputs

Result Result structure from optimization.

PriLev Printing level: (default 3)

0 Silent.

1 Problem number and name.
Function value at the solution and at start.
Known optimal function value (if given).

2 As *PriLev* =1 and:

Optimal point x and starting point x_0.
Number of evaluations of the function, gradient etc.
Lagrange multipliers, both returned and TOMLAB estimate.
Distance from start to solution.
The residual, gradient and projected gradient. (*)
$ExitFlag$ and $Inform$.

(*) The calculation and output of these fields is controlled by *Result.Prob.PrintLM*.

3 As *PriLev* =2 and:

Jacobian, Hessian or Quasi-Newton Hessian approximation.

Global Parameters Used

To avoid too many variables, constraints and residuals in the output, three global variables are limiting the number printed:

MAX_x	Maximum number of variables
MAX_c	Maximum number of constraints
MAX_r	Maximum number of residuals in least squares problems

Example: To increase the number of variables printed by *PrintResult* to 50, do

```
global MAX_x
MAX_x = 50;
PrintResult(Result);
```

12.13 runtest

Purpose
Run all selected problems defined in a problem file for a given solver.

Calling Syntax
runtest(Solver, SolverAlg, probFile, probNumbs, PriLevOpt, wait, PriLev)

Description of Inputs

Solver	Name of solver, default *conSolve*.
SolverAlg	A vector of numbers defining which of the *Solver* algorithms to try. For each element in *SolverAlg*, all *probNumbs* are solved. Leave empty, or set 0 if to use the default algorithm.
probFile	Problem definition file. *probFile* is by default set to *con_prob* if *Solver* is *conSolve*, *uc_prob* if *Solver* is *ucSolve* and so on.
probNumbs	A vector with problem numbers to run. If empty, run all problems in *probFile*.
PriLevOpt	Printing level in *Solver*. Default 2, short information from each iteration.
wait	Set *wait* to 1 if pause after each problem. Default 1.
PriLev	Printing level in *PrintResult*. Default 5, full information.

M-files Used
SolverList.m

See Also
systest

12.14 SolverList

Purpose
Prints the available solvers for a certain *solvType*.

Calling Syntax
[SolvList, SolvTypeList, SolvDriver] = SolverList(solvType)

Description of Inputs

solvType	Either a string 'uc', 'con' etc. or the corresponding *solvType* number. See Table 1.

Description of Outputs

SolvList	String matrix with the names of the solvers for the given *solvType*.
SolvTypeList	Integer vector with the *solvType* for each of the solvers.
SolvDriver	String matrix with the names of the driver routine for each different *solvType*.

Description
The routine *SolverList* prints all available solvers for a given *solvType*, including Fortran, C and Matlab Optimization Toolbox solvers. If *solvType* is not specified then *SolverList* lists all available solvers for all different *solvType*. The input argument could either be a string such as 'uc', 'con' etc. or a number corresponding to the type of solver, see Table 1.

Examples
See Section 3.

M-files Used
SolverList.m

12.15 StatLS

Purpose
Compute parameter statistics for least squares problems.

Calling Syntax
LS = StatLS(x_k, r_k, J_k);

Description of Inputs

x_k	Optimal parameter vector, length n.
r_k	Residual vector, length m.
J_k	Jacobian matrix, length m by n.

Description of Outputs

Structure LS with fields:

SSQ	Sum of squares: $r_k' * r_k$
Covar	Covariance matrix: Inverse of $J' * diag(1./(r_k' * r_k)) * J$
sigma2	Estimate squared standard deviation of problem, SSQ / Degrees of freedom, i.e. SSQ/(m-n)
Corr	Correlation matrix: Normalized Covariance matrix $Cov./(CovDiag * CovDiag')$, where $CovDiag = sqrt(diag(Cov))$
StdDev	Estimated standard deviation in parameters: $CovDiag * sqrt(sigma2)$
x	=x_k, the input x
ConfLim	95 % Confidence limit (roughly) assuming normal distribution of errors $ConfLim = 2 * LS.StdDev$
CoeffVar	The coefficients of variation of estimates: $StdDev./x_k$

12.16 systest

Purpose
Run big test to check for bugs in TOMLAB.

Calling Syntax
systest(solvTypes, PriLevOpt, PriLev, wait)

Description of Inputs

solvTypes	A vector of numbers defining which *solvType* to test.
PriLevOpt	Printing level in the solver. Default 2, short information from each iteration.
wait	Set *wait* to 1 if pause after each problem. Default 1.
PriLev	Printing level in *PrintResult*. Default 5, full information.

See Also
runtest

13 Approximation of Derivatives

This section about derivatives is particularly 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.

For information about supplying partial derivatives see the *Prob* parameter *CheckNaN* in Appendix A.

Options Summary

Observe that the usage depends on which solver is used. Clearly, if a global solver, such as *glbFast* is used, derivatives are not used at all and the following sections do not apply. Some solvers use only first order information, while others may require second order information. See 'help iniSolve' and the TOMLAB interface for each solver (e.g. *snoptTL*) for information on derivative level needed. 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* options 11-15 require that the patterns (*Prob.HessPattern*, *Prob.JacPattern* and *Prob.ConsPattern*) are properly set. They should also be set for options 1-5 but are not required. These are most easily set by calling the corresponding user routines with 2 random set of variables and combining the sparse matrices. The functions *estConsPattern*, *estJacPattern* and *estHessPattern* automates this process with 3 safe trials.

If *Prob.LargeScale* is set to 1 and the respective user routines are not given, then *estConsPattern*, *estJacPattern* and *estHessPattern* are automatically executed before the solution process is started.

If first order information is provided the user should set minus (-) in front of the option that they want. For example if the objective function and gradient are given *Prob.NumDiff* = -1 will make sure that only the Hessian is estimated by numerical differentiation.

Table 130 describes the differentiation options available in TOMLAB.

Table 131 shows the flags that are included in the callbacks to the user routines for objective, constraints and other functions. These flags should be used to optimize the performance of numerical differentiation. For example if *FDVar* = [1 4], then only the constraints that depend on decision variable number 1 and 4 need to be calculated.

The following applies to the different user supplied functions:

User c: If Prob.rows nonempty, then only these constraints need to be computed, others could be set as 0. If it is easier to exclude the constraints to compute from dependencies, then Prob.FDVar says which variables are changed, i.e. to compute numerical derivatives for. Only constraints that depend on the variable indices given in FDVar need to be computed. If FDVar == 0, nothing is known, and all constraints must be computed.

User dc: If Prob.rows nonempty, then only these constraint Jacobian rows need to be computed, others could be set as 0. If Prob.cols nonempty, only these columns in the constraint Jacobian need to be computed, others could be set as 0. Only if Prob.cols is empty and Prob.FDVar(1) > 0 could Prob.FDVar be used to simplify the computations.

User f: If Prob.FDVar(1) > 0, then only variables in Prob.FDVar are changed since the last call with Prob.FDVar == 0. If knowledge about the functional parts that are dependent on the non-changed variables are saved the last time Prob.FDVar == 0, it may be utilized to speed up the computations.

User g: If Prob.FDVar(1) > 0, the variables in Prob.FDVar are changed. They influence the variables in Prob.cols (if nonempty), which means that g(Prob.cols) are the elements accessed to obtain the numerical Hessian.

User r: If Prob.rows nonempty, then only these residuals need to be computed, others could be set as 0. If it is easier to exclude the residuals to compute from dependencies, then Prob.FDVar says which variables are changed,

i.e. to compute numerical derivatives for. Only residuals that depend on the variable indices given in FDVar need to be computed, the rest could be set as 0. If FDVar == 0, nothing is known, and all residuals must be computed.

Table 130: The differentiation options in TOMLAB.

Flag	Value	Comments
Prob.ADObj	1	MAD used for first order information (gradient is automatically calculated with floating point precision). Some functions are not supported by MAD.
	-1	MAD used for second order information (Hessian), i.e. the gradient needs to be specified. Users should make sure that MAD variables are allocated properly in their files and that they are not overwritten.
Prob.ADCons	1	MAD used for first order information (Jacobian of the constraints is automatically calculated with floating point precision). Some functions are not supported by MAD. Users should make sure that MAD variables are allocated properly in their files and that they are not overwritten.
	-1	MAD used for second order information (d2c, the second part of the Hessian to the Lagrangian), i.e. the Jacobian needs to be specified. Users should make sure that MAD variables are allocated properly in their files and that they are not overwritten.
Prob.NumDiff	1	*fndg* calculates the gradient. *FDJac* calculates the Jacobian of the residuals. If needed the Hessian is estimated by *FDHess*. **Default in** TOMLAB .
	-1	Same as option 1 but the gradient is not estimated, as it is given. A negative sign has the same functionality for all options.
	11	Same as option 1 but a function *findpatt* is called from *iniSolve* when using this option. *Prob.HessIx* and *Prob.JacIx* are set to improve the performance of the differentiation routines. These are only for internal use.
	-11	Same as option 11 but the gradient is not estimated, as it is given. A negative sign has the same functionality for all options.
	2	Matlab standard splines (no additional toolboxes needed). *Prob.optParam.CentralDiff* is used by this routine. *fndg2* calculates the gradient. *FDJac2* calculates the Jacobian of the residuals. If needed the Hessian is estimated by *FDHess2*.
	12	Same as option 2 but *Prob.HessIx* and *Prob.JacIx* are set for improved performance.
	3	*csaps* will be used (Splines Toolbox needed). *Prob.optParam.splineSmooth* is used by this routine. *fndg2* calculates the gradient. *FDJac2* calculates the Jacobian of the residuals. If needed the Hessian is estimated by *FDHess2*.
	13	Same as option 3 but *Prob.HessIx* and *Prob.JacIx* are set for improved performance.
	4	*spaps* will be used (Splines Toolbox needed). *Prob.optParam.splineTol* is used by this routine. If *Prob.optParam.SplineTol* < 0 then *csapi* is used instead. *fndg2* calculates the gradient. *FDJac2* calculates the Jacobian of the residuals. If needed the Hessian is estimated by *FDHess2*.
	14	Same as option 4 but *Prob.HessIx* and *Prob.JacIx* are set for improved performance.

Table 130: The differentiation options in TOMLAB, continued.

Flag	Value	Comments
	5	A routine using complex variables. *fndg3* calculates the gradient. *FDJac3* calculates the Jacobian of the residuals. This option is not available for solvers requiring second order information.
	15	Same as option 5 but *Prob.JacIx* is set for improved performance.
	6	The derivatives are estimated by the solver (only available for some options).
Prob.ConsDiff	1	*FDJac* calculates the Jacobian of the constraints. If needed the nonlinear constraint Hessian is estimated by *FDcHess*.
	11	Same as option 1 but a function *findpatt* is called from *iniSolve* when using this option. *Prob.ConIx* is set to improve the performance of the differentiation routines. This is only for internal use.
	2	Matlab standard splines (no additional toolboxes needed). *Prob.optParam.CentralDiff* is used by this routine. *FDJac2* calculates the Jacobian of the constraints. If needed the nonlinear constraint Hessian is estimated by *FDcHess*.
	12	Same as option 2 but *Prob.ConIx* is set for improved performance.
	3	*csaps* will be used (Splines Toolbox needed). *Prob.optParam.splineSmooth* is used by this routine. *FDJac2* calculates the Jacobian of the constraints. If needed the nonlinear constraint Hessian is estimated by *FDcHess*.
	13	Same as option 3 but *Prob.ConIx* is set for improved performance.
	4	*spaps* will be used (Splines Toolbox needed). *Prob.optParam.splineTol* is used by this routine. If *Prob.optParam.SplineTol* < 0 then *csapi* is used instead. *FDJac2* calculates the Jacobian of the constraints. If needed the nonlinear constraint Hessian is estimated by *FDcHess*.
	14	Same as option 4 but *Prob.ConIx* is set for improved performance.
	5	A routine using complex variables. *FDJac3* calculates the Jacobian of the residuals. This option is not available for solvers requiring second order information.
	15	Same as option 5 but *Prob.ConIx* is set for improved performance.
	6	The derivatives are estimated by the solver.

Table 131: Callback flags in TOMLAB.

Flag	Value	Comments
Prob.FDVar	0 or vector	The variables being perturbed in the callback for a numerical differentiation routine. The user should make sure that unnecessary calculation are not made.
Prob.rows	0 or vector	The rows in the user computed vector/matrix that will be accessed, and needs to be set. If empty, no information is available, and all elements need to be set.
Prob.cols	0 or vector	The columns in the user computed matrix that will be accessed, and needs to be set. If empty, no information is available, and all elements need to be set.

Both numerical differentiation and automatic differentiation are possible. There are six ways to compute numerical

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

Numerical differentiation is automatically used for gradient, Jacobian, constraint Jacobian and Hessian if a user routine is not present.

Especially for large problems it is important to tell the system which values are nonzero, if estimating the Jacobian, the constraint Jacobian, or the Hessian matrix. Define a sparse 0-1 matrix, with ones for the nonzero elements. This matrix is set as input in the *Prob* structure using the fields *Prob.JacPattern*, *Prob.ConsPattern*, *Prob.HessPattern* or *Prob.d2LPattern*. If there are many zeros in the matrix, this leads to significant savings in each iteration of the optimization. It is possible to use the TOMLAB estimation routines mentioned above to set these.

A variable *Prob.FDVar* is a dynamically set field that indicates which decision variables are being perturbed during a call to the user's routines. For example if FDVar = [1,3,5], the variables with indices 1, 3 and 5 are being used for an estimation. When the Jacobian (for example) is being calculated by repeated calls to the constraints (or residuals) the user can make sure that unnecessary calculations are avoided.

```
FDVar = Prob.FDVar;
if FDVar == 0 | FDVar == 4
    % Calculate some constraints
end
    % The constraint will only be calculated if FDVar == 0
    % which means that all are required, or if FDVar == 4, i.e.
    % only if decision variable number 4 is being perturbed.
```

To allow efficiently estimation of the derivatives in the five different options, it is possible to obtain the indices needed for the constraints (*Prob.ConIx*), Jacobian (*Prob.JacIx*) and Hessian (*Prob.HessIx*) evaluations. A function called *findpatt* is called by *iniSolve* automatically if a 1 is added before the number assigned for the differentiation method. The same functionality illustrated below applies for *Prob.ConsDiff*.

```
Prob.NumDiff = 11;  % Use index information for NumDiff = 1
Prob.NumDiff = 12;  % Use index information for NumDiff = 2
  ...
Prob.NumDiff = 15;  % Use index information for NumDiff = 5
```

If a set of problems with identical indices for the differentiation routines are optimized in the sequence the following code can be used. This avoids re-generating the indices needed:

```
Prob.ConIx  = findpatt(Prob.ConsPattern);
Prob.JacIx  = findpatt(Prob.JacPattern);
Prob.HessIx = findpatt(Prob.HessPattern);

Prob.NumDiff = 1;  % NumDiff = 11 used automatically
```

Forward or Backward Difference Approximations

Prob.NumDiff = 1 (11) or *Prob.ConsDiff* = 1 (11). **Default in** TOMLAB .

The default way is to use the classical approach with forward or backward differences together with an optional automatic step size selection procedure. Set *Prob.GradTolg = -1* to run the procedure. The differentiation is handled by the routine *fdng*, which is a direct implementation of the FD algorithm [28, page 343].

The *fdng* routine is using the parameter field *DiffInt*, in the structure optParam, see Table 141, page 229, as the numerical step size. The user could either change this field or set the field *Prob.GradTolg*. The field *Prob.GradTolg* may either be a scalar value or a vector of step sizes of the same length as the number of unknown parameters x. The advantage is that individual step sizes can be used, in the case of very different properties of the variables or very different scales. If the field *Prob.GradTolg* is defined as a *negative number*, the *fdng* routine is estimating a suitable step size for each of the unknown parameters. This is a costly operation in terms of function evaluations, and is normally not needed on well-scaled problems.

Similar to the *fdng*, there are two routines *FDJac* and *FDHess*. *FDJac* numerically estimates the Jacobian for nonlinear least squares problems or the constraint Jacobian in constrained nonlinear problems. *FDJac* checks if the field *Prob.GradTolJ* is defined, with the same action as *fdng*. *FDHess* estimates the Hessian matrix in nonlinear problems and checks for the definition of the field *Prob.GradTolH*. Both routines use field *Prob.optParam.DiffInt* as the default tolerance if the other field is empty. Note that *FDHess* has no automatic step size estimation. The implementation in *fdng*, *FDJac* and *FDHess* avoids taking steps outside the lower and upper bounds on the decision variables. This feature is important if going outside the bounds makes the function undefined.

Splines

Matlab splines is selected by setting *Prob.NumDiff* or *Prob.ConsDiff* = 2 (12). *csaps* will be used if *Prob.NumDiff* or *Prob.ConsDiff* is set to 3 (13), *spaps* if set to 4 (14) and finally *csapi* if set to 4 (14) with *Prob.optParam.SplineTol* < 0.

The first spline option can be used without the Spline Toolbox installed. If the Spline Toolbox is installed, gradient, Jacobian, constraint Jacobian and Hessian approximations could be computed in three different ways depending on which of the three routines, *csaps*, *spaps* or *csapi* the user choose to use.

The routines *fdng2*, *FDJac2* and *FDHess2* implements the gradient estimation procedure for the different approximations needed. All routines use the tolerance in the field *Prob.optParam.CentralDiff* as the numerical step length. The basic principle is central differences, taking a small step in both positive and negative direction.

Complex Variables

Prob.NumDiff = 5 (15) or *Prob.ConsDiff* = 5 (15).

The fifth approximation method is a method by Squire and Trapp [72], which is using complex variables to estimate the derivative of real functions. The method is not particularly sensitive to the choice of step length, as long as it is very small. The routine *fdng3* implements the complex variable method for the estimation of the gradient and *FDJac3* the similar procedure to estimate the Jacobian matrix or the constraint Jacobian matrix. The tolerance is hard coded as $1E-20$. There are some pitfalls in using Matlab code for this strategy. In the paper by Martins et. al [55], important hints are given about how to implement the functions in Matlab. They were essential in getting the predefined TOMLAB examples to work, and the user is advised to read this paper before attempting to make new code and using this differentiation strategy. However, the insensitivity of the numerical step size might make it worthwhile, if there are difficulties in the accuracy with traditional gradient estimation methods.

Automatic Differentiation

Automatic differentiation is performed by use of the MAD toolbox. MAD is a TOMLAB toolbox which is documented in a separate manual, see http://tomopt.com.

MAD should be initialized by calling *madinitglobals* before running TOMLAB with automatic differentiation. Note that in order for TOMLAB to be fully compatible with the MAD, the functions must be defined according to the MAD requirements and restrictions. Some of the predefined test problems in TOMLAB do not fulfill those requirements.

In the Graphical User Interface, the differentiation strategy selection is made from the *Differentiation Method* menu reachable in the *General Parameters* mode. Setting the *Only 2ndD* click-box, only unknown second derivatives are estimated. This is accomplished by changing the sign of *Prob.NumDiff* to negative to signal that first order derivatives are only estimated if the gradient routine is empty, not otherwise. The method to obtain derivatives for the constraint Jacobian is selected in the *Constraint Jacobian diff. method* menu in the *General Parameters* mode.

When running the menu program *tomRemote/tomMenu*, push/select the *How to compute derivatives* button in the *Optimization Parameter Menu*.

To choose differentiation strategy when running the driver routines or directly calling the actual solver set *Prob.ADObj* equal to -1 or 1 for automatic differentiation or *Prob.NumDiff* to 1 (11), 2 (12), 3 (13), 4 (14) or 5 (15) for numerical differentiation, before calling drivers or solvers. Note that $Prob.NumDiff = 1$ will run the *fdng* routine. $Prob.NumDiff = 2, 3, 4$ will make the *fdng2* routine use standard Matlab splines or call the Spline Toolbox routines *csaps*, *spaps*, and *csapi* respectively. The *csaps* routine needs a smoothness parameter and the *spaps* routine needs a tolerance parameter. Default values for these parameters are set in the structure optParam, see Table 141, fields *splineSmooth* and *splineTol*. The user should be aware of that there is no guarantee that the default values of *splineSmooth* and *splineTol* are the best for a particular problem. They work on the predefined examples in TOMLAB. To use the built in numerical differentiation in the SOL solvers *MINOS*, *NPSOL*, *NLSSOL* and *SNOPT*, set $Prob.NumDiff = 6$. Note that the *DERIVATIVE LEVEL* SOL parameter must be set properly to tell the SOL solvers which derivatives are known or not known. There is a field *DerLevel* in *Prob.optParam* that is used by TOMLAB to send this information to the solver. To select the method to obtain derivatives for the constraint Jacobian the field *Prob.ConsDiff* is set to 1-6 (11-15, 16 not valid) with the same meaning as for *Prob.NumDiff* as described above. *Prob.ADCons* is the equivalent variable for automatic differentiation of the nonlinear constraints.

Here follows some examples of the use of approximative derivatives when solving problems with *ucSolve* and *clsSolve*. The examples are part of the TOMLAB distribution in the file *diffDemo* in directory *examples*.

To be able to use automatic differentiation the MAD toolbox must be installed.

Automatic Differentiation example

```
madinitglobals;                  % Initiate MAD variables
probFile        = 'uc_prob';     % Name of Init File
P               = 1;             % Problem number
Prob            = probInit(probFile, P);
Prob.Solver.Alg = 2;             % Use the safeguarded standard BFGS
Prob.ADObj      = 1;             % Use Automatic Differentiation.
Result          = tomRun('ucSolve', Prob, 2);
```

FD example

```
% Finite differentiation using the FD algorithm

probFile        = 'uc_prob';     % Name of Init File
P               = 1;             % Problem number
```

```
Prob            = probInit(probFile, P); Prob.Solver.Alg = 2;
Prob.NumDiff    = 1;             % Use the fdng routine with the FD algorithm.
Result          = tomRun('ucSolve', Prob, 2);

% Change the tolerances used by algorithm FD

Prob.GradTolg   = [1E-5; 1E-6];  % Change the predefined step size
Result          = tomRun('ucSolve', Prob, 1);

% The change leads to worse accuracy

% Instead let an algorithm determine the best possible GradTolg
% It needs some extra f(x) evaluations, but the result is much better.

Prob.GradTolg         = -1;  % A negative number demands that the step length
                             % of the algorithm is to be used at the initial point

% Avoid setting GradTolg empty, then instead Prob.optParam.DiffInt is used.

Result          = tomRun('ucSolve', Prob, 1);

% Now the result is perfect, very close to the optimal == 0.

Prob.NumDiff    = 5;             % Use the complex variable technique

% The advantage is that it is insensitive to the choice of step length

Result          = tomRun('ucSolve', Prob, 1);

% When it works, like in this case, it gives absolutely perfect result.
```

Increasing the tolerances used as step sizes for the individual variables leads to a worse solution being found, but no less function evaluations until convergence. Using the automatic step size selection method gives very good results. The complex variable method gives absolutely perfect results, the optimum is found with very high accuracy.

The following example illustrates the use of spline function to approximate derivatives.

Spline example

```
probFile        = 'ls_prob';     % Name of Init File
P               = 1;             % Problem number
Prob            = probInit(probFile, P);
Prob.Solver.Alg = 0;             % Use the default algorithm in clsSolve
Prob.NumDiff    = 2;             % Use Matlab spline .
Result          = tomRun('clsSolve', Prob, 2);
```

FD example with patterns

```
% Finite differentiation using the FD algorithm
%
% This example illustrates how to set nonzeros in HessPattern
% to tell TOMLAB which elements to estimate in the Hessian.
% All elements in Hessian with corresponding zeros in HessPattern are
% set to 0, and no numerical estimate is needed.
%
% This saves very much time for large problems
% In a similar way, Prob.ConsPattern is set for the constraint gradient
% matrix for the nonlinear constraints, and Prob.JacPattern for the
% Jacobian matrix in nonlinear least squares problems.

probFile                = 'con_prob';
P                       = 12;
Prob                    = probInit(probFile, P);
Prob.Solver.Alg         = 1;
Prob.HessPattern        = sparse([ones(6,6), zeros(6,6);zeros(6,12)]);

% Note that if setting Prob.NumDiff = 1, also the gradient would be
% estimated with numerical differences, which is not recommended.
% Estimating two levels of derivatives is an ill-conditioned process.
% Setting Prob.NumDiff = -1, only the Hessian is estimated

% Use qpSolve in base module for QP subproblems
Prob.SolverQP           = 'qpSolve';

Prob.NumDiff            = -1;  % Use the fdng routine with the FD algorithm.
Result                  = tomRun('nlpSolve',Prob,1);

% Setting Prob.NumDiff = -11 makes Tomlab analyze HessPattern
% and use a faster algorithm for large-scale problems
% In this small-scale case it is no advantage,
% it just takes more CPU-time.

Prob.NumDiff            = -11;  % Use the fdng routine with the FD algorithm.
Result                  = tomRun('nlpSolve',Prob,1);

% Run the same problem estimating Hessian with Matlab Spline
% Needs more gradient calculations because it is principle
% smooth central differences to obtain the numerical Hessian.
% If the problem is noisy, then this method is recommended.

Prob.NumDiff            = -2;  % Matlab spline
Result                  = tomRun('nlpSolve',Prob,1);
```

14 Special Notes and Features

In this section several topics of general interest, which enables a more efficient use of TOMLAB are collected.

14.1 Speed and Solution of Optimization Subproblems

It is often the case that the full solution of an optimization problem involves the solution of subtasks, which themselves are optimization problems. In fact, most general solvers are constructed that way, they solve well-defined linear or quadratic subproblems as part of the main algorithm. TOMLAB has a standard way of calling a subsolver with the use of the driver routine *tomSolve*. The syntax is similar to the syntax of *tomRun*. Calling *QPOPT* to solve a QP sub problem is made with the call

```
Result = tomRun('qpopt', Prob);
```

The big advantage is that *tomSolve* handles the global variables with a stack strategy, see Appendix D. Therefore it is possible to run any level of recursive calls with the TOMLAB TOM solvers, that all run in Matlab. Even if care has been taken in the MEX-file interfaces to avoid global variable and memory conflicts, there seem to be some internal memory conflicts occurring when calling recursively the same MEX-file solver. Luckily, because TOMLAB has several solver options, it should be possible to use different solvers. In one two-stage optimization, a control problem, even four solvers were used. *glcSolve* was used to find a good initial value for the main optimization and *SNOPT* was used to find the exact solution. In each iteration several small optimization problems had to be solved. Here *glbSolve* was used to find a good initial point close to the global optimum, and *MINOS* then iterated until good accuracy was found.

The general TOM solvers *clsSolve*, *conSolve*, *cutplane*, *mipSolve*, *nlpSolve*, *qpSolve* and *sTrustr* have all been designed so it shall be possible to change the subproblem solver. For example to solve the QP subproblems in *conSolve* there are several alternatives, *QPOPT*, *qpSolve* or even *SNOPT*. If using the BFGS update in *conSolve*, which guarantees that the subproblems are convex, then furthermore *QLD* or *SQOPT* could be used. The QP, LP, FP (feasible point) and DLP (dual LP) subproblems have special treatment. A routine *CreateProbQP* creates the *Prob* structure for the subproblem. The routine checks on the fields *Prob.QP.SOL* and *Prob.QP.optParam* and move these to the usual places *Prob.SOL* and *Prob.optParam* for the subproblem. Knowing this, the user may send his own choices of these two important subfields as input to *conSolve* and the other solvers mentioned. The choice of the subsolver is made by giving the name of the wanted subsolver as the string placed in *Prob.SolverQP* for QP subproblems and similar for the other subproblems. Note that the time consuming call to *CreateProbQP* is only done once in the solver, and after that only the fields necessary to be changed are altered in each iteration.

Note that if the user needs to cut CPU time as much possible, one way to save some time is to call *tomSolve* instead of *tomRun*. But no checks are made on the structure *Prob*, and such tricks should only be made at the production stage, when the code is known to be error free.

Another way to cut down CPU time for a nonlinear problem is to set

```
Prob.LargeScale = 1;
```

even if the problem is not that large (but time consuming). TOMLAB will then not collect information from iterations, and not try to estimate the search steps made. This information is only used for plotting, and is mostly not needed. Note that this change might lead to other choices of default solvers, because TOMLAB thinks the problem is large and sparse. So the default choices might have to be overridden.

14.2 User Supplied Problem Parameters

If a problem is dependent on a few parameters, it is convenient to avoid recoding for each change of these parameters, or to define one problem for each of the different parameter choices. The user supplied problem parameters gives the user an easy way to change the creation of a problem. A field in the *Prob* structure, the field *Prob.user* is used to store the user supplied problem parameters. The field *Prob.uP* could be used when using Init Files.

Prob.uP, if set, is a numerical matrix or vector (recommended) with numbers. Its primary use is defining sizes and other vitals parameters, like initial value choices, in problems defined as Init Files. *Prob.uP* may not be a Matlab struct.

If $ask == 1$ is set, it is easy to change the values of *Prob.uP* in the test problems interactively. However, if setting *Prob.uP* directly (needed for batch runs) it is more difficult.

Three different ways to set *Prob.uP* are described below. Following these examples will get the problem setup correctly.

The first option assumes that the Prob structure is defined

```
Prob        = ProbDef;
Prob.Name   = 'Exponential problem (pseudorand)';
Prob.uPName = Prob.Name; % Special field used to identify the uP parameters
Prob.P      = 14;
Prob.uP     = [60, 180]; % Increase the problem size to n=60, m=180
Prob        = probInit('ls_prob',14,-1,Prob);
```

The advantage is that other fields may also be set, however they could also be set after the *probInit* call.

The second way avoid defining the full Prob structure. The field *probFile* must then also be initialized.

```
Prob          = [];
Prob.Name     = 'Exponential problem (pseudorand)';
Prob.uPName   = Prob.Name; % Special field used to identify the uP parameters
Prob.probFile = [];
Prob.P        = 14;
Prob.uP       = [60, 180]; % Increase the problem size to n=60, m=180
Prob          = probInit('ls_prob',14,-1,Prob);
```

In the third and final option the *uP* parameters could be sent directly as the 4th input to *probInit*, instead of the problem structure Prob.

```
Prob          = probInit('ls_prob',14,-1,[60, 180]);
```

Note: One must not change *Prob.uP* after executing *probInit*. Doing this will normally result in a crash in some routine, e.g. *ls_r* or *ls_J* or in erroneous results.

The best way to describe the User Supplied Problem Parameter feature inside Init Files is by an example. Assume that a problem with variable dimension needs to be created. If the user wants to change the dimension of the problem during the initialization of the problem, i.e. in the call to the Init File, the routine *askparam* is helpful. Problem 27 in *cls_prob* is an example of the this:

```
...
...
elseif P==27
   Name='RELN';
   % n problem variables, n >= 1 , default n = 10
   uP    = checkuP(Name,Prob);
   n     = askparam(ask, 'Give problem dimension ', 1, [], 10, uP);
   uP(1) = n;
   y     = zeros(n,1);
   x_0   = zeros(n,1);
   x_opt = 3.5*ones(n,1);
...
...
```

The routine *checkuP* is checking if the input Prob structure has the field *Prob.uP* defined, and if the Name of the problem is the same as the one set in *Prob.Name*. If this is true, *uP* is set to supplied value. When calling *askparam*, if $ask <= 0$, the dimension n is set to the default value 10 if *uP* is empty, otherwise to the value of *uP*. If $ask > 0$ is set by the user, *askparam* will ask the question *Give problem dimension* and set the value given by user. At the end of the Init File, the field *Prob.uP* is assigned the value of *uP(1)*.

Using the routine *checkuP*, called after the Name variable is assigned, and the general question asking routine *askparam*, it is easy to add the feature of user supplied problem parameters to any user problem. Type *help askparam* for information about the parameters sent to *askparam*.

To send any amount of other information to the low-level user routines, it is recommended to use sub fields of *Prob.user* as described in Section 2.4.

In the other problem definition files, *cls_r* and *cls_J* in this example, the parameter(s) are "unpacked" and can be used e.g. in the definition of the Jacobian.

```
...
...
elseif P==27
   % 'RELN'
   n = Prob.uP(1);
...
...
```

If questions should be asked during the setup of the problem, in the Init File, the user must set the integer *ask* positive in the call to *probInit*. See the example below:

```
ask=1;
Prob = probInit('cls_prob',27,ask);
```

The system will now ask for the problem dimension, and assuming the choice of the dimension is 20, the output would be:

```
Current value = 10

Give problem dimension 20
```

Now call *clsSolve* to solve the problem,

```
Result=tomRun('clsSolve', Prob, 1);
```

As a second example assume that the user wants to solve the problem above for all dimensions between 10 and 30. The following code snippet will do the job.

```
for dim=10:30
   Prob = [];
   Prob.uP(1) = dim;
   PriLev = 0;
   Result = tomRun('clsSolve', 'cls_prob', 27, Prob, PriLev);
end
```

14.3 User Given Stationary Point

Known stationary points could be defined in the problem definition files. It is also possible for the user to define the type of stationary point (minimum, saddle or maximum). When defining the problem *RB BANANA* (17) in the previous sections, *x_opt* was set as $(1, 1)$ in the problem definition files. If it is known that this point is a minimum point the definition of *x_opt* could be extended to

```
   x_opt = [1 1 StatPntType];      % Known optimal point (optional).
```

where *StatPntType* equals 0, 1, or 2 depending on the type of the stationary point (minimum, saddle or maximum). In this case set *StatPntType* to 0 since $(1, 1)$ is a minimum point. The extension becomes

```
   x_opt = [1 1 0];      % Known optimal point (optional).
```

If there is more than one known stationary point, the points are defined as rows in a matrix with the values of *StatPntType* as the last column. Assume that $(-1, -1)$ is a saddle point, $(1, -2)$ is a minimum point and $(-3, 5)$ is a maximum point for a certain problem. The definition of *x_opt* could then look like

```
   x_opt = [ -1 -1  1
              1 -2  0
             -3  5  2 ];
```

Note that it is not necessary to define *x_opt*. If *x_opt* is defined it is not necessary to define *StatPntType* if all given points are minimum points.

14.4 Print Levels and Printing Utilities

The amount of printing is determined by setting different print levels for different parts of the TOMLAB system. The parameter is locally most often called *PriLev*. There are two main print levels. One that determines the output printing of the driver or menu routines, set in the input structure as *Prob.PriLev*. The other printing level, defined in *Prob.PriLevOpt*, determines the output in the TOM solvers and for the SOL solvers, the output in the Matlab part of the MEX file interface. In Table 132 the meaning of different print levels are defined.

The utility routine *PrintResult* prints the results of an optimization given the *Result* structure. The amount of printing is determined by a second input argument *PriLev*. The driver routine *tomRun* also makes a call to

PrintResult after the optimization, and if the input parameter *PriLev* is greater than zero, the result will be the same as calling *PrintResult* afterwards.

PrintResult is using the global variables, *MAX_c*, *MAX_x* and *MAX_r* to limit the lengths of arrays displayed. All Matlab routines in the SOL interfaces are also using these global variables. The global variables get default values by a call to *tomlabInit*, or if empty is set to default values by the different routines using them. The following example show the lines needed to change the default values.

```
global MAX_c  MAX_x MAX_r
MAX_x = 100;
MAX_c =  50;
MAX_r = 200;
```

This code could either be executed at the command line, or in any main routine or script that the user defines.

Table 132: Print level in the TOM solvers, *Prob.PriLevOpt*

Value	Description
< 0	Totally silent.
0	Error messages and warnings.
1	Final results including convergence test results and minor warnings.
2	Each iteration, short output.
3	Each iteration, more output.
4	Line search or QP information.
5	Hessian output, final output in solver.

There is a wait flag field in *optParam*, *optParam.wait*. If this flag is set true, most of the TOM solvers uses the pause statement to avoid the output just flushing by. The user must press *RETURN* to continue execution. The fields in *optParam* is described in Table 141.

The TOM solvers routines print large amounts of output if high values for the *PriLev* parameter is set. To make the output look better and save space, several printing utilities have been developed.

For matrices there are two routines, *mPrint* and *PrintMatrix*. The routine *PrintMatrix* prints a matrix with row and column labels. The default is to print the row and column number. The standard row label is eight characters long. The supplied matrix name is printed on the first row, the column label row, if the length of the name is at most eight characters. Otherwise the name is printed on a separate row.

The standard column label is seven characters long, which is the minimum space an element will occupy in the print out. On a 80 column screen, then it is possible to print a maximum of ten elements per row. Independent on the number of rows in the matrix, *PrintMatrix* will first display $A(:, 1:10)$, then $A(:, 11:20)$ and so on.

The routine *PrintMatrix* tries to be intelligent and avoid decimals when the matrix elements are integers. It determines the maximal positive and minimal negative number to find out if more than the default space is needed. If any element has an absolute value below 10^{-5} (avoiding exact zeros) or if the maximal elements are too big, a switch is made to exponential format. The exponential format uses ten characters, displaying two decimals and therefore seven matrix elements are possible to display on each row.

For large matrices, especially integer matrices, the user might prefer the routine *mPrint*. With this routine a more dense output is possible. All elements in a matrix row is displayed (over several output rows) before next matrix

row is printed. A row label with the name of the matrix and the row number is displayed to the left using the Matlab style of syntax.

The default in *mPrint* is to eight characters per element, with two decimals. However, it is easy to change the format and the number of elements displayed. For a binary matrix it is possible to display 36 matrix columns in one 80 column row.

14.5 Partially Separable Functions

The routine *sTrustr* implements a structured trust region algorithm for partially separable functions (*psf*). A definition of a *psf* is given below and an illustrative example of how such a function is defined and used in TOMLAB.

f is partially separable if $f(x) = \sum_{i}^{M} f_i(x)$, where, for each $i \in \{1, ..., M\}$ there exists a subspace $\mathbb{N}_i \neq 0$ such that, for all $w \in \mathbb{N}_i$ and for all $x \in \mathbb{X}$, it holds that $f_i(x + w) = f_i(x)$. $\mathbb{X}$ is the closed convex subset of $\mathbb{R}^n$ defined by the constraints.

Consider the problem *DAS 2* in **File:** tomlab/testprob/con_prob :

$$\begin{array}{ll} \min\limits_{x} & f(x) = \frac{1}{2}\sum\limits_{1}^{6} r_i(x)^2 \\ s/t & Ax \geq b \\ & x \geq 0 \end{array} \tag{22}$$

where

$$r = \begin{pmatrix} \frac{\sqrt{11}}{6}x_1 - \frac{3}{\sqrt{11}} \\ \frac{x_2-3}{\sqrt{2}} \\ \sqrt{0.0775}\cdot x_3 + \frac{0.5}{\sqrt{0.0775}} \\ \frac{\frac{x_4}{3}-3}{\sqrt{2}} \\ \frac{-5}{6}x_1 + 0.6x_3 \\ 0.75x_3 + \frac{2}{3}x_4 \end{pmatrix}, A = \begin{pmatrix} -1 & -2 & -1 & -1 \\ -3 & -1 & -2 & 1 \\ 0 & 1 & 4 & 0 \end{pmatrix}, b = \begin{pmatrix} -5 \\ -4 \\ 1.5 \end{pmatrix}.$$

The objective function in (22) is partially separable according to the definition above and the constraints are linear and therefore they define a convex set. *DAS 2* is defined as the constrained problem 14 in *con_prob*, *con_f*, *con_g* etc. as an illustrative example of how to define a problem with a partially separable objective function. Note the definition of *pSepFunc* in *con_prob*.

One way to solve problem (22) with *sTrustr* is to define the following statements:

```
probFile = 'con_prob';              % Name of Init File
P        = 14;                      % Problem number in con_prob
Prob     = probInit(probFile, P);   % Define a problem structure

Result   = tomRun('sTrustr', Prob, 0);
```

The sequence of statements are similar to normal use of TOMLAB. The only thing that triggers the use of the partial separability is the definition of the variable *Prob.PartSep.pSepFunc*. To solve the same problem, and avoiding the use of *psf*, the following statements could be used:

```
probFile = 'con_prob';              % Name of Init File
```

```
P        = 14;                       % Problem number in con_prob
Prob     = probInit(probFile, P);    % Define a problem structure

Prob.PartSep.pSepFunc = 0;           % Redefining number of separable functions

Result   = tomRun('sTrustr', Prob, 0);
```

Another alternative is to set *Prob.PartSep* as empty before the call to *sTrustr*. This example, slightly expanded, is included in the distribution as *psfDemo* in directory *examples*.

14.6 Utility Test Routines

The utility routines listed in Table 133 run tests on a large set of test problems.

Table 133: System test routines.

Function	Description	Section	Page
runtest	Runs all selected problems defined in a problem file for a given solver.	12.13	201
systest	Runs big test for each *probType* in TOMLAB.	12.16	204

The *runtest* routine may also be useful for a user running a large set of optimization problems, if the user does not need to send special information in the *Prob* structure for each problem.

A *Prob* - the Input Problem Structure

The Input Problem Structure, here referred to as *Prob*, is one of the most central aspects of working with TOMLAB. It contains numerous fields and substructures that influence the behavior and performance of the solvers.

There are default values for everything that is possible to set defaults for, and all routines are written in a way that makes it possible for the user to just set an input argument empty ([ ]) and get the default.

TOMLAB is using the structure variable *optParam*, see Table 141, for the optimization parameters controlling the execution of the optimization solvers.

Subproblems

Many algorithms need sub-problems solved as part of the main algorithm. For example, in SQP algorithms for general nonlinear programs, QP problems are solved as sub-problems in each iteration. As QP solver any solver, even a general NLP solver, may be used. To send parameter information to the QP subsolver, the fields *Prob.optParam*, *Prob.Solver* and *Prob.SOL* could be put as subfields to the *Prob.QP* field (see Table 136), i.e. as fields *Prob.QP.optParam*, *Prob.QP.Solver*. The field *Prob.QP.optParam* need not have all subfields, the missing ones are filled with default values for the particular QP solver.

The same *Prob.QP* subfield is used for the other types of subproblems recognized, i.e. Phase 1 feasibility problems, LP and dual LP problems. Note that the fields *Prob.SolverQP*, *Prob.SolverFP Prob.SolverLP* and *Prob.SolverDLP* are set to the name of the solver that should solve the subproblem. If the field is left empty, a suitable default solver is used, dependent on the license for TOMLAB.

Table 134: Information stored in the problem structure *Prob*. Fields defining sub-structures are defined in Table 135

Field	Description
Tomlab	TOMLAB Version number.
A	Matrix with linear constraints, one constraint per row (dense or sparse).
ADObj	Automatic differentiation flag. If 1, -1 MAD is used to obtain gradient and Hessian, respectively.
ADCons	Automatic differentiation flag. If 1, -1 MAD is used to obtain the constraint Jacobian and nonlinear constraint Hessian, respectively.
b_L	Lower bounds on the linear constraints.
b_U	Upper bounds on the linear constraints.
c_L	Lower bounds on the general constraints.
c_U	Upper bounds on the general constraints.
CHECK	If true, no more check is done by ProbCheck. Set to true (=1) after first call to ProbCheck.

Table 134: Information stored in the problem structure *Prob*, continued.

Field	Description
CheckNaN	If *Prob.CheckNaN* $= 0$, *nlp_d2c*, *nlp_H*, *nlp_d2r* checks for NaN elements and estimates the corresponding derivatives numerically. If *Prob.CheckNaN* > 0, the same applies for *nlp_dc*, *nlp_g*, *nlp_J*. Off-diagonal elements in symmetric Hessians should both be set as NaN. fdng, fdng2, fdng3, only estimate NaN elements in gradient, if gradient vector is input.
cName	Name of each general constraint.
cols	The columns in the user computed matrix that will be accessed, and needs to be set.
ConIx	A vector with the sequence of calls required to compute the numerical constraint Jacobian efficiently. See *findpatt* for more information.
ConsDiff	Numerical approximation of the constraint derivatives. If set to 1, the classical approach with forward or backward differences together with automatic step selection will be used. If set to 2, 3 or 4 the spline routines *csapi*, *csaps* or *spaps* in the SPLINE Toolbox will be used. If set to 5, derivatives will be estimated by use of complex variables. For the SOL solvers, the value 6 gives the internal derivative approximation.
ConsIdx	Internally used to speed up the SOL solver computations. Used to send linear index from multiple subscripts for nonzero pattern in constraint Jacobian.
ConsPattern	Matrix with non-zero pattern in the constraint gradient matrix.
d2LPattern	Sparsity pattern of the Hessian of the Lagrangian function.
f_Low	Lower bound on optimal function value.
f_opt	Objective function value $f(x^*)$ corresponding to the points given in *x_opt*.
GradTolg	Size of step length to estimate first order derivatives in the gradient vector. If this field is empty, *optParam.DiffInt* is used instead.
GradTolH	Size of step length to estimate the Hessian matrix. If this field is empty , *optParam.DiffInt* is used instead.
GradTolJ	Size of step length to estimate the Jacobian matrix or the constraint gradient matrix. If this field is empty, *optParam.DiffInt* is used instead.
HessIx	A vector with the sequence of calls required to compute the numerical Jacobian efficiently. See *findpatt* for more information.
HessPattern	Matrix with non-zero pattern in the Hessian matrix.

Table 134: Information stored in the problem structure *Prob*, continued.

Field	Description
JacIx	A vector with the sequence of calls required to compute the numerical Jacobian efficiently. See *findpatt* for more information.
JacPattern	Matrix with non-zero pattern in the Jacobian matrix.
LargeScale	Flag if the problem is large scale. If this flag is set no collection of search steps are made. Also, for some solvers, *LargeScale* chooses between dense (=0) or sparse (=1) versions of the solver. This flag also controls several other features in TOMLAB such as estimation of patterns.
MaxCPU	Maximum execution time in seconds for the solver. The feature is available for a limited number of solvers.
MENU	Flag used to tell if a menu, the GUI or a driver is calling, to avoid unnecessary checks of the fields in *Prob* (0).
Mode	Indicates whether the user should return function values and/or derivatives. Is best used when the user computes both function values and derivatives at the same time to save CPU time. 0 = Assign function values. 1 = Assign known derivatives, unknown derivative values set as -11111 (=missing value). 2 = Assign function and known derivatives, unknown derivatives set as -11111.
nState	Indicates the first and last calls to the user routines to compute function and derivatives. Used by the SOL solvers. 1 First call. 0 Other calls before the last call. $2 + Inform$ Last call, see the Inform parameter for the solver used.
N	Problem dimension (number of variables).
mLin	Number of linear constraints.
mNonlin	Number of nonlinear constraints.
Name	Problem name.
NumDiff	Numerical approximation of the derivatives of the objective function. If set to 1, the classical approach with forward or backward differences together with automatic step selection will be used. If set to 2, 3 or 4 either the standard Matlab spline function or the spline routines *csapi*, *csaps* or *spaps* in the SPLINE Toolbox will be used. If set to 5, derivatives will be estimated by use of complex variables. For the some solvers, the value 6 gives the internal derivative approximation.

Table 134: Information stored in the problem structure *Prob*, continued.

Field	Description
P	Problem number (1).
plotLine	Flag if to do a plot of the line search problem.
PriLev	Print level in the driver routines (0).
PriLevOpt	Print level in the TOM solver and the Matlab part of the solver interface (0).
PrintLM	Flag: controls whether or not *PrintResult* should calculate Lagrange multipliers and reduced (projected) gradient after a solver has been run. Since *PrintResult* operates on a *Result* structure, and not on *Prob*, this flag is accessed as *Result.Prob.PrintLM*, but of course it is possible to set it before solving the problem.
probFile	Name of m-file in which the problems are defined.
probType	TOMLAB problem type, see Table 1.
rows	The rows in the user computed vector/matrix that will be accessed, and needs to be set.
simType	A flag indicating when the TOMLAB simulation format is used. The objective and constraints are calculated at the same function. The gradient and Jacobian are also calculated in the same function.
smallA	If 1 then small elements in the linear constraints are removed. The elements have to be smaller than $eps * max(max(abs(Prob.A)))$.
SolverDLP	Name of the solver that should solve dual LP sub-problems.
SolverLP	Name of the Solver that should solve LP sub-problems.
SolverFP	Name of the solver that should solve a phase one LP sub-problem, i.e. finding a feasible point to a convex set.
SolverQP	Name of the solver that should solve QP sub problems.
uP	User supplied parameters for the problem (Init File Format). Use *Prob.user* for extra parameters.
uPName	Problem name (*Prob.Name*) connected to the user supplied parameters in *(* uP).
user	User supplied parameters for the problem. Should be stored in subfields, for example *Prob.user.aParam*.
WarmStart	For solver with support for warmstarts, *WarmStart* > 0 indicates that the solver should do a warm start.

Table 134: Information stored in the problem structure *Prob*, continued.

Field	Description
x_0	Starting point.
x_L	Lower bounds on the variables x.
x_U	Upper bounds on the variables x.
x_min	Lower bounds on plot region.
x_max	Upper bounds on plot region.
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*.
xName	Name of each decision variable in x.
Warning	If 1 (default), some warnings may be issued.

Table 135: The fields defining sub-structures in the problem structure *Prob*. Default values are in all tables given in parenthesis at the end of each item.

Field	Description
QP	Structure with special fields for linear and quadratic problems, see Table 136.
LS	Structure with special fields for least squares problems, see Table 137.
MIP	Structure with special fields for mixed-integer programming, see Table 138.
GO	Structure with special fields for global optimization.
CGO	Structure with special fields for costly global optimization.
ExpFit	Structure with special fields for exponential fitting problems, see Table 139.
NTS	Structure with special fields for nonlinear time series.
LineParam	Structure with special fields for line search optimization parameters, see Table 140.
optParam	Structure with special fields for optimization parameters, see Table 141.
PartSep	Structure with special fields for partially separable functions, see Table 142.
SOL	Structure with special fields for SOL (Stanford Optimization Laboratory) solvers, see Table 143.
Solver	Structure with fields *Name*, *Alg* and *Method*. *Name* is the name of the solver. *Alg* is the solver algorithm to be used. *Method* is the solver sub-method technique. See the solver descriptions Section 11.
FUNCS	Structure with user defined names of the m-files computing the objective, gradient, Hessian etc. See Table 144. These routines are called from the corresponding gateway routine
DUNDEE	Structure with special fields for the MINLP solvers (University of Dundee).
PENOPT	Structure with special fields for the PENOPT solvers.

Table 136: Information stored in the structure *Prob.QP*. The three last sub-fields, always part of the *Prob* subfields, could optionally be put here to give information to a subproblem QP, LP, dual LP or feasible point (Phase 1) solver.

Field	Description
F	Constant matrix F in $\frac{1}{2}x'Fx + c'x$
c	Cost vector c in $\frac{1}{2}x'Fx + c'x$
B	Logical vector of the same length as the number of variables. A zero corresponds to a variable in the basis.
y	Dual parameters y.
Q	Orthogonal matrix Q in QR-decomposition.
R	Upper triangular matrix R in QR-decomposition.
E	Pivoting matrix E in QR-decomposition. Stored sparse.
Ascale	Flag if to scale the A matrix, the linear constraints.
DualLimit	Stop limit on the dual objective.
UseHot	True if to use a crash basis (hot start).
HotFile	If nonempty and UseCrash true, read basis from this file.
HotFreq	How often to save a crash basis.
HotN	The number of crash basis files.
optParam	Structure with special fields for optimization parameters, see Table 141.
SOL	Structure with special fields for SOL (Stanford Optimization Laboratory) solvers, see Table 143.
Solver	Structure with fields *Name*, *Alg* and *Method*. *Name* is the name of the solver. *Alg* is the solver algorithm to be used. *Method* is the solver sub-method technique. See the solver descriptions Section 11.

Table 137: Information stored in the structure *Prob.LS*

Field	Description
weightType	Weighting type: *0* No weighting. *1* Weight with data in y. If $y(t) = 0$, the weighting is 0, i.e. deleting this residual element. *2* Weight with weight vector or matrix in *weightY*. If *weightY* is a vector then weighting by $weigthY.*r$ (element wise multiplication). If *weightY* is a matrix then weighting by $weigthY * r$ (matrix multiplication). *3* *nlp_r* calls the routine *weightY* (must be a string with the routine name) to compute the residuals.
weightY	Either empty, a vector, a matrix or a string, see *weightType*.
t	Time vector t.
y	Vector or matrix with observations $y(t)$.
E	Fixed matrix, in linear least squares problem $E * x - y$.
yUse	If $yUse = 0$ compute residual as $f(x,t) - y(t)$ (default), otherwise $y(t)$ should be treated separately by the solver and the residual routines just return $f(x,t)$.
SepAlg	If $SepAlg = 1$, use separable non linear least squares formulation (default 0).

Table 138: Information stored in the structure *Prob.MIP*

Field	Description
IntVars	Which variables are integer valued
VarWeight	Priority vector for each variable.
fIP	Function value for point defined in *xIP*. Gives an upper bound on the IP value wanted. Makes it possible to cut branches and avoid node computations
xIP	The point x giving the function value *fIP*.
PI	See the Tomlab /Xpress User's Guide.
SC	See the Tomlab /Xpress User's Guide.
SI	See the Tomlab /Xpress User's Guide.
sos1	See the Tomlab /Xpress User's Guide.
sos2	See the Tomlab /Xpress User's Guide.
xpcontrol	See the Tomlab/ Xpress User's Guide.
callback	See the Tomlab /Xpress User's Guide.
KNAPSACK	See the Tomlab /Xpress User's Guide.

Table 139: Information stored in the structure *Prob.ExpFit*. Default values in parenthesis.

Field	**Description**
p	Number of exponential terms (2).
wType	Weighting type (1).
eType	Type of exponential terms (1).
infCR	Information criteria for selection of best number of terms (0).
dType	Differentiation formula (0).
geoType	Type of equation (0).
qType	Length q of partial sums (0).
sigType	Sign to use in $(P \pm \sqrt{Q})/D$ in *exp_geo* for $p = 3, 4$ (0).
lambda	Vector of dimension p, intensities.
alpha	Vector of dimension p, weights.
beta	Vector of dimension p, weights in generalized exponential models.
x0Type	Type of starting value algorithm.
sumType	Type of exponential sum.

Table 140: Information stored in the structure *Prob.LineParam*

Field	**Description**
LineAlg	Line search algorithm. 0 = quadratic interpolation, 1 = cubic interpolation, 2 = curvilinear quadratic interpolation (not robust), 3 = curvilinear cubic interpolation (not robust) ($LineAlg = 1$).
sigma	Line search accuracy; $0 < sigma < 1$. $sigma = 0.9$ inaccurate line search. $sigma = 0.1$ accurate line search (0.9).
InitStepLength	Initial length of step (1.0).
MaxIter	Maximum number of line search iterations.
fLowBnd	Lower bound on optimal function value. Used in the line search by Fletcher, m-file *LineSearch* ($= -realmax$).
rho	Determines the ρ line (0.01).
tau1	Determines how fast step grows in phase 1 (9).
tau2	How near end point of $[a, b]$ (0.1).
tau3	Choice in $[a, b]$ phase 2 (0.5).
eps1	Minimal length for interval $[a, b]$ (10^{-7}).
eps2	Minimal reduction ($100 \times \epsilon$).

Table 141: Information stored in the structure *Prob.optParam*. Default values in parenthesis. The values are selectively used in Base Module solvers. Refer to individual solver documentation for more information.

Field	Description
PriLev	Solver major print level in file output.
PriFreq	Print frequency in optimization solver.
SummFreq	Summary frequency in optimization solver.
MinorPriLev	Minor print level in file output in sub-problem solver.
IterPrint	Flag for one-row-per-iteration printout during optimization (0).
wait	Flag, if true use pause statements after output in each iteration (0).
MaxFunc	Maximal number of function evaluations.
MaxIter	Maximum number of iterations.
MajorIter	Maximum number of iterations in major problem.
MinorIter	Maximum number of iterations in minor problem.
eps_f	Relative convergence tolerance in f (10^{-8}).
eps_absf	Absolute convergence tolerance for the function value ($-realmax$).
eps_x	Relative convergence tolerance in parameter solution x.
eps_dirg	Convergence tolerance for the directed derivative (10^{-8}).
eps_c	Feasibility tolerance for nonlinear constraints.
eps_g	Gradient (or reduced gradient) convergence tolerance (10^{-7}).
eps_Rank	Rank test tolerance.
EpsGlob	Global/local weight parameter in global optimization (10^{-4}).
fTol	Relative accuracy in the computation of the function value.
xTol	If $x \in [x\_L, x\_L + bTol]$ or $[x\_U - bTol, x\_U]$, fix x on bound ($100 * \epsilon = 2.2204 \cdot 10^{-13}$).
bTol	Feasibility tolerance for linear constraints.
cTol	Feasibility tolerance for nonlinear constraints.
MinorTolX	Relative convergence tolerance in parameters x in sub-problem.
size_x	Size at optimum for the variables x, used in the convergence tests (1). Only changed if scale very different, $x >> 1$.
size_f	Size at optimum for the function f, used in the convergence tests (1). Only changed if scale very different, $f >> 1$.
size_c	Size at optimum for the constraints c, used in the convergence tests (1). Only changed if scale very different, $c >> 1$.
PreSolve	Flag if presolve analysis is to be applied on linear constraints (0).

Table 141: Information stored in the structure *Prob.optParam*, continued.

Field	Description
DerLevel	Derivative Level, knowledge about nonlinear derivatives: 0 = Some components of the objective gradient are unknown and some components of the constraint gradient are unknown, 1 = The objective gradient is known but some or all components of the constraint gradient are unknown, 2 = All constraint gradients are known but some or all components of the objective gradient are unknown, 3 = All objective and constraint gradients are known.
GradCheck	$0, 1, 2, 3$ gives increasing level of user-supplied gradient checks.
DiffInt	Difference interval in derivative estimates.
CentralDiff	Central difference interval in derivative estimates.
QN_InitMatrix	Initial matrix for Quasi-Newton, may be set by the user. When *QN_InitMatrix* is empty, the identity matrix is used.
splineSmooth	Smoothness parameter sent to the SPLINE Toolbox routine *csaps.m* when computing numerical approximations of the derivatives (-1 default). 0 means least squares straight line fit. 1 means natural (variational) cubic spline interpolant. The transition range in [0,1] is small, suggested tries are 0.2 or 0.4. < 0 lets the routine csaps make the choice (default)
splineTol	Tolerance parameter sent to the SPLINE Toolbox routine *spaps.m* when computing numerical approximations of the derivatives (10^{-3}). Should be set in the order of the noise level.
BigStep	Unbounded step size. Used to detect unbounded nonlinear problems.
BigObj	Unbounded objective value. Used to detect unbounded nonlinear problems.
CHECK	If true, no more check is done on the structure. Set to true (=1) after first call to *optParamSet*.

Table 142: Information stored in the structure *Prob.PartSep*

Field	Description
pSepFunc	Number of partially separable functions.
index	Index for the partially separable function to compute, i.e. if $i = index$, compute $f_i(x)$. If $index = 0$, compute the sum of all, i.e. $f(x) = \sum_{i=1}^{M} f_i(x)$.

Table 143: Information stored in the structure *Prob.SOL*

Field	Description
SpecsFile	If nonempty gives the name of a file which is written in the SOL SPECS file format.
PrintFile	If nonempty gives the name of a file which the SOL solver should print information on. The amount printed is dependent on the print level in *Prob.SOL.optPar(1)*.
SummFile	If nonempty gives the name of a file which the SOL solver prints summary information on.
xs	Vector with solution x and slack variables s. Used for warm starts.
hs	Vector with basis information in the SOL sparse solver format. Used for warm starts.
nS	Number of superbasics. Used for warm starts.
hElastic	Elastic variable information in *SQOPT*.
iState	Vector with basis information in the SOL dense solver format. Used for warm starts.
cLamda	Vector with Lagrange multiplier information in the SOL dense solver format. Used for warm starts.
H	Cholesky factor of Hessian Approximation. Either in natural order (*Hessian Yes*) or reordered (*Hessian No*). Used for warm starts using natural order with *NPSOL* and *NLSSOL*.
callback	For large dense or nearly dense quadratic problems (*probType* == **qp**) it is more efficient to use a callback function from the MEX routine to compute the matrix-vector product $F \cdot x$. Then *Prob.QP.F* is never copied into the MEX solver. This option applies to *SQOPT* only.
optPar	Vector with *optParN* elements with parameter information for SOL solvers. Initialized to missing value, -999. The elements used are described in the help for each solver. If running TOMLAB format, also see the help of the TOMLAB solver interface routine whos name always has the letters TL added, e.g. *minosTL*.
optParN	Number of elements in *optPar*, defined as 62.

Table 144: Information stored in the structure *Prob.FUNCS*

Field	**Description**
f	Name of m-file computing the objective function $f(x)$.
g	Name of m-file computing the gradient vector $g(x)$. If *Prob.FUNCS.g* is empty then numerical derivatives will be used.
H	Name of m-file computing the Hessian matrix $H(x)$.
c	Name of m-file computing the vector of constraint functions $c(x)$.
dc	Name of m-file computing the matrix of constraint normals $\partial c(x)/dx$.
d2c	Name of m-file computing the 2nd part of 2nd derivative matrix of the Lagrangian function, $\sum_i \lambda_i \partial^2 c(x)/dx^2$.
r	Name of m-file computing the residual vector $r(x)$.
J	Name of m-file computing the Jacobian matrix $J(x)$.
d2r	Name of m-file computing the 2nd part of the Hessian for nonlinear least squares problem, i.e. $\sum_{i=1}^{m} r_i(x) \frac{\partial^2 r_i(x)}{\partial x_j \partial x_k}$.

Table 145: Information stored in the structure *Prob.DUNDEE*

Field	Description
callback	If 1, use a callback to Matlab to compute $QP.F \cdot x$ in *BQPD* and *miqpBB* . Faster when F is large and nearly dense. Avoids copying the matrix to the MEX solvers.
kmax	Maximum dimension of the reduced space (k), default equal to dimension of problem. Set to 0 if solving an LP problem.
mlp	Maximum number of levels of recursion.
mode	Mode of operation, default set as $2 * Prob.WarmStart$.
x	Solution (warmstart).
k	Dimension of reduced space (warmstart).
e	Steepest-edge normalization coefficient (warmstart).
ls	Indices of active constraints, first $n - k$. (warmstart).
lp	List of pointers to recursion information in *ls* (warmstart).
peq	Pointer to end of equality constraint indices in *ls* (warmstart).
PrintFile	Name of print file. Amount/print type is determined by *Prob.DUNDEE.optPar(1)*.
optPar	Vector with optimization parameters. Described in Table 146.

Table 146: *Prob.DUNDEE.optPar* values used by TOMLAB /MINLP solvers . −999 in any element gives default value.

				Used by:			
Index	**Name**	**Default**	**Description**	BQPD	miqpBB	filterSQP	minlpBB
1	iprint	0	Print level in DUNDEE solvers.	O	O	O	O
2	tol	10^{-10}	Relative accuracy in BQPD solution.	O	O	O	O
3	emin	1	1/0: Use/do not use constraint scaling in BQPD	O	O	O	O
4	sgnf	$5 \cdot 10^{-4}$	Maximum relative error in two numbers equal in exact arithmetic.	O	O	O	O
5	nrep	2	Maximum number of refinement steps.	O	O	O	O
6	npiv	3	No repeat of more than *npiv* steps were taken.	O	O	O	O
7	nres	2	Maximum number of restarts if unsuccessful.	O	O	O	O
8	nfreq	500	Maximum interval between refactorizations.	O	O	O	O
9	ubd	100	Constraint violation parameter I	-	-	O	O
10	tt	0.125	Constraint violation parameter II	-	-	O	O
11	NLP_eps	10^{-6}	Relative tolerance for NLP solutions	-	-	O	O
12	epsilon	10^{-6}	Accuracy for x tests	O	O	O	O
13	MIopttol	10^{-4}	Accuracy for f tests	-	O	-	O
14	fIP	10^{20}	Upper bound on the IP value wanted.	-	O	-	O
15	timing	0	1/0: Use/do not use timing	-	O	-	-
16	max_time	4000	Maximum time (sec's) allowed for the run	-	O	-	-
17	branchtype	1	Branching strategy $(1, 2, 3)$	-	O	-	-
18	ifsFirst	0	Exit when first IP solution is found	-	O	-	-
19	infty	10^{20}	Large value used to represent infinity	O	O	O	O
20	Nonlin	0	Treat all constraints as nonlinear if 1	-	-	O	O

B *Result* - the Output Result Structure

The results of the optimization attempts are stored in a structure array named *Result*. The currently defined fields in the structure are shown in Table 149. The use of structure arrays make advanced result presentation and statistics possible. Results from many runs may be collected in an array of structures, making postprocessing on all results easy.

When running global optimization, output results are also stored in *mat*-files, to enable fast restart (warm start) of the solver. It is seldom the case that one knows that the solver actually converged for a particular problem. Therefore one does restarts until the optimum does not change, and one is satisfied with the results. The information stored in the mat-file *glbSave.mat* by the solver *glbSolve* is shown in Table 147. The information stored in the mat-file *glcSave.mat* by the solver *glcSolve* is shown in Table 148. Different information is stored when using glbFast and glcFast, see the solver reference.

Table 147: Information stored in the mat-file *glbSave.mat* by the solver *glbSolve*. Used for automatic restarts.

Variable	**Description**
C	Matrix with all rectangle centerpoints, in [0,1]-space.
D	Vector with distances from centerpoint to the vertices.
DMin	Row vector of minimum function value for each distance.
DSort	Row vector of all different distances, sorted.
E	Computed tolerance in rectangle selection.
F	Vector with function values.
L	Matrix with all rectangle side lengths in each dimension.
Name	Name of the problem. Used for security if doing warm start.
glbfMin	Best function value found at a feasible point.
iMin	The index in D which has lowest function value, i.e. the rectangle which minimizes $(F - glbfMin + E)./D$ where $E = max(EpsGlob * abs(glbfMin), 1E-8)$.

Table 148: Information stored in the mat-file *glcSave.mat* by the solver *glcSolve*. Used for automatic restarts.

Variable	**Description**
C	Matrix with all rectangle centerpoints.
D	Vector with distances from centerpoint to the vertices.
F	Vector with function values.
G	Matrix with constraint values for each point.
Name	Name of the problem. Used for security if doing warm start.
Split	$Split(i, j)$ is the number of splits along dimension i of rectangle j.
T	$T(i)$ is the number of times rectangle i has been trisected.
fMinEQ	sum(abs(infeasibilities)) for minimum points, 0 if no equalities.
fMinIdx	Indices of the currently best points.
feasible	Flag indicating if a feasible point has been found.
glcf_min	Best function value found at a feasible point.
iL	$iL(i, j)$ is the lower bound for rectangle j in integer dimension $I(i)$.
iU	$iU(i, j)$ is the upper bound for rectangle j in integer dimension $I(i)$.
ignoreidx	Rectangles to be ignored in the rectangle selection procedure.
s	$s(j)$ is the sum of observed rates of change for constraint j.
s_0	s_0 is used as $s(0)$.
t	$t(i)$ is the total number of splits along dimension i.

Table 149: Information stored in the optimization result structure *Result*.

Field	Description
Name	Problem name.
P	Problem number.
probType	TOMLAB problem type, according to Table 1, page 11.
Solver	Solver used.
SolverAlgorithm	Solver algorithm used.
solvType	TOMLAB 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.
v_k	Lagrange multipliers for constraints on variables, linear and nonlinear constraints.
r_k	Residual vector at optimum.
J_k	Jacobian matrix at optimum.

Table 149: Information stored in the optimization result structure *Result*, continued.

Field	Description
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 150.
bState	State of each linear constraint, described in Table 151.
cState	State of each general constraint, described in Table 152.
p_dx	Matrix where each column is a search direction.
alphaV	Matrix where row i stores the step lengths tried for the i:th iteration.
x_min	Lowest x-values in optimization. Used for plotting.
x_max	Highest x-values in optimization. Used for plotting.
LS	Structure with statistical information for least squares problems, see Table 153.
F_X	F_X is a global matrix with rows: [iter_no f(x)].
SepLS	General result variable with fields z and Jz. Used when running separable nonlinear least squares problems.
QP	Structure with special fields for QP problems. Used for warm starts, see Table 136.
SOL	Structure with some of the fields in the *Prob.SOL* structure, the ones needed to do a warm start of a SOL solver, see Table 143. The routine *WarmDefSOL* moves the relevant fields back to *Prob.SOL* for the subsequent call.
DUNDEE	Structure with special result fields from TOMLAB /MINLP solvers.
plotData	Structure with plotting parameters.
Prob	Problem structure, see Table 134. Please note that certain solvers that do reformulations of the problem, e.g. *L1Solve*, *infSolve* and *slsSolve*, return the *Prob* structure of the *reformulated* problem in this field, *not* the original one.

The field *xState* describes the state of each of the variables. In Table 150 the different values are described. The different conditions for linear constraints are defined by the state variable in field *bState*. In Table 151 the different values are described.

Table 150: 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 151: 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 152: 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.

Table 153: Information stored in the structure *Result.LS*.

Field	Description
SSQ	$r_k^T \cdot r_k$.
Covar	Covariance matrix (inverse of $J_k^T \cdot J_k$).
sigma2	Estimate of squared standard deviation.
Corr	Correlation matrix (normalized covariance matrix).
StdDev	Estimated standard deviation in parameters.
x	The optimal point x_k.
ConfLim	95% confidence limit (roughly) assuming normal distribution of errors.
CoeffVar	Coefficients of variation of estimates.

C *TomSym* - the Modeling Engine

This appendix describes the details and functions of relevance to the user when modeling optimization and optimal control problems with TomSym. It is possible to solve the modeling problems, either with *ezsolve* or by converting the problem to a standard TOMLAB *Prob* struct with *sym2prob*. The latter provides for the option of controlling all solver settings per the TOMLAB standard (see the solver manuals for more information). The procedure for using MAD (Matlab Automatic Differentiation) is also described in the sections below.

C.1 Main functions

The following Matlab functions are used directly by the user when modeling problems for processing by the TOMLAB solvers.

C.1.1 tom — Generate a tomSym symbol.

```
x = tom creates a scalar tomSym symbol with an automatic name.
x = tom(label) creates a scalar symbol with the provided name.
x = tom(label,m,n) creates a m-by-n matrix symbol.
x = tom([],m,n) creates a matrix symbol with an automatic name.
x = tom(label,m,n,'int') creates an integer matrix symbol.
x = tom(label,m,n,'symmetric') creates a symmetric matrix symbol.
```

Because constructs like "`x = tom('x')`" are very common, there is the shorthand notation "toms x".

See also: toms, tomSym

C.1.2 toms — Create tomSym objects.

Toms is a shorthand notation, possibly replacing several calls to 'tom'

A symbol is created in the current workspace for each name listed. If a size is specified on the format "NxM" where N and M are integers, then all subsequent symbols will get that size. If the size specification ends with an exclamation point (as in "3x4!") then a symbolic array of concatenated scalar symbols is created instead of one matrix symbol.

The flags "integer" (or "int") and "symmetric" are recognized. If a flag is encountered, then all subsequent symbols will get the properties of that flag.

-integer: The variable is constrained to integer values, resulting in a

```
mixed-integer problem (which requires a compatible solver.)
```

-symmetric: The variable is symmetric, so that x' == x. This requires that

```
the dimensions be square. An NxN symmetric matrix only contains
N*(N+1)/2 unknowns, and the resulting symbolic object uses the
setSymmetric function.
```

Examples:

```
  toms x y z
is equivalent to
  x = tom('x');
  y = tom('z');
  z = tom('z');

  toms 2x3 Q 3x3 -integer R -symmetric S
is equivalent to
  Q = tom('Q', 2, 3);
  R = tom('R', 3, 3, 'integer');
  S = tom('S', 3, 3, 'integer', 'symmetric')

  toms 3x1! v
is equivalent to
  v1 = tom('v1');
  v2 = tom('v2');
  v3 = tom('v3');
  v = [v1;v2;v3];
```

In the last example, with the exclamation point, the result is a vector containing scalar symbols. This works differently from the matrix symbols used in the previous examples. Mathematically this v is equivalent to "toms 3x1 v", but the auto-generated code will be different. Expressions such as `v(1)+sin(v(2))*v(3)` will be more efficient, while expressions such as A*v will be less efficient.

Note: While the "toms" shorthand is very convenient to use prototyping code, it is recommended to only use the longhand "tom" notation for production code. The reason is that Matlab's internal compiler tries to guess whether a statement like "`x(1)`" is an index into a vector or a call to a function. Since it does not realize that the call to "toms" creates new variables is will make the wrong guess if that function (named "x" in this example) is on the Matlab path when the script is loaded into memory. This will cause strange and unexpected results.

See also: tom, tomSym, setSymmetric

C.1.3 tomSym/tomSym — Class constructor

NOTE: Use toms or tom to create tomSym symbols.

```
p = tomSym(a,m,n,arguments...) creates a tomSym using the operator a.
p = tomSym(const) creates a tomSym constant with the value const.
p = tomSym(struct) converts a struct into a tomSym, if the struct has the
required fields.
```

See also: toms, tom, tomSym/funcs, tomSym/tomSym

C.1.4 ezsolve — Solve a tomSym optimization problem.

`[solution, result] = ezsolve(f,c)` returns the solution to the minimization problem that is defined by the objective function f and the constraints c. The result structure from tomRun is provided in a second output argument.

Ezsolve can also be used to find least-square solutions to equations. If options.norm is set to 'L2', then f can be a set of equations. (If there are equations both in f and c, then the ones in c are considered as strict, and will be solved to tolerances, while the ones in f are solved in a least-square sense. If f is a vector, then the L2 norm of f is minimized.

Ezsolve is meant to be as simple to use as possible. It automatically determines the problem type and finds a suitable solver.

The returned solution is a struct, where the fields represent the unknown variables. This struct can be used by subs to convert a tomSym to a numeric value.

`s = ezsolve(f,c,x0)` uses the initial guess x0. The input argument x0 can be a struct containing fields names as the unknown symbols, for example a previously returned solution. Alternatively, x0 can be a cell array of simple tomSym equation that can be converted to a struct using tom2struct.

`s = ezsolve(f,c,x0,name)` sets the problem name.

`s = ezsolve(f,c,x0,OPTIONS)` where OPTIONS is a structure sets solver options. The options structure can have the following fields.

```
OPTIONS.name    - The name of the problem
OPTIONS.type    - The problem type, e.g. 'lp', 'qp', 'con', ...
OPTIONS.solver  - The solver to use, e.g. 'snopt', 'knitro', ...
OPTIONS.prilev  - The ezsolve and tomRun print level (default 1)
OPTIONS.use_d2c - (boolean) true = compute symbolic d2c
OPTIONS.use_H   - (boolean) true = compute symbolic H
```

See also: tomDiagnose, sym2prob, tomRun

C.1.5 sym2prob — Compile symbolic function/constraints into a Prob struct.

`Prob = sym2prob(type,f,c)` creates a Prob structure, suitable for tomRun, with the objective of minimizing f, subject to c, with respect to all symbols that they contain.

The problem type can be (among others):

```
'lp'    - Linear programming
'qp'    - Quadratic programming
'con'   - Nonlinear programming
'qpcon' - Quadratic problem with nonlinear constraints
'minlp' - Mixed integer nonlinear programming
'sdp'   - Linear semidefinite programming
'bmi'   - Bilinear semidefinite programming
```

`Prob = sym2prob(f,c)` calls tomDiagnose to attempt to guess the problem type, and prints a warning.

The objective f must be a tomSym symbolic object, while the constraints list c should be a cell array.

If the objective f is a vector or matrix, then a least-square problem is solved, minimizing 0.5*`sum(vec(f))` (Half the sum-of-squares of the elements of f). If f is an equality on the form lhs == rhs, then the sum-of-squares of the difference (rhs-lhs) is minimized.

`sym2prob(type,f,c,x0)` supplies an initial guess for one or more of the unknowns. The guess x0 should be a struct, where each field is named as a symbol and contains a numeric array of the correct size.

`sym2prob(type,f,c,x0,OPTIONS)` where OPTIONS is a structure sets options. The options structure can have the following fields.

```
OPTIONS.name    - The name of the problem
OPTIONS.use_d2c - (boolean) true = compute symbolic d2c
OPTIONS.use_H   - (boolean) true = compute symbolic H
```

Linear and box constraints will be automatically detected, if they are formulated using simple addition and multiplication. For example,

```
-3*(x+2) <= 4+x
```

is automatically converted to

```
x >= -2.5
```

For nonlinear problems, sym2prob will generate temporary m-files representing the nonlinear functions and their derivatives (These files are usually small and harmless, and many modern operating systems automatically clean up old temporary files). In order to remove these files when they are no longer needed, it is recommended to run `tomCleanup(Prob)` after the problem has been solved.

```
Overloaded methods:
   tomSym/sym2prob
```

C.1.6 getSolution — Extract variables from a solution retuned by tomRun.

`s = getSolution(solution)` returns a struct where each field corresponds to one of the symbols used in the tomSym problem formulation.

C.1.7 tomDiagnose — Determine the type for of tomSym optimization problem.

`type = tomDiagnose(f,c)` returns a text string, defining the problem type that is represented by the objective function f and the constraints c.

Some possible return values:

```
'lp'    - Linear programming
'qp'    - Quadratic programming
'cls'   - Least squares with linear constraints
'nls'   - Nonlinear least squares
'con'   - Nonlinear programming
'qpcon' - Quadratic problem with nonlinear constraints
'minlp' - Mixed integer nonlinear programming
'sdp'   - Linear semidefinite programming
'bmi'   - Bilinear semidefinite programming
```

The objective f must be a tomSym symbolic object, while the constraints list c should be a cell array of tomSym objects.

C.1.8 tomCleanup — Remove any temporary files created for a tomSym problem.

`tomCleanup(Prob)` removes any temporary files (created by sym2prob) that are referenced in the problem structure Prob. This should be done after a problem has been solved, to avoid leaving garbage in the system's "temp" directory (although most modern operating system will clean up these files eventually anyway.)

WARNING: IF THE AUTOGENERATED FILES HAVE BEEN EDITED, THEN RUNNING

```
TOMCLEANUP WILL CAUSE ALL CHANGES TO BE LOST.
```

`tomCleanup('all')` will remove all files and directories that are judged to be tomSym files from the system's "temp" directory. All files that contain a certain "marker" in the filename will be removed. This might be useful if a large number of temporary files were created accidentally, such as when running sym2prob inside a for-loop without calling `tomCleanup(Prob)` at the end of the loop.

Note: Under normal circumstances there is never a need to run

```
tomCleanup('all'). The operating system should remove old temporary
files as required.
```

After running tomCleanup, all Prob structures that reference the deleted files will be useless.

See also: sym2prob

C.2 Using MAD

Some models require the use of alternative method for partial derivatives. If TomSym cannot be used, one alternative is to use MAD.

C.2.1 madWrap — Compute a Jacobian using MAD.

`J = madWrap(FUN,N,...)` uses MAD to call `FUN(...)` and returns the Jacobian matrix with respect to the N:th input argument.

FUN must be the name of an existing function. N must be an integer between one and `nargin(FUN)`.

`J = madWrap(M,FUN,N,...)` computes the Jacobian matrix of the Mth output argument, instead of the first one.

J will be a Jacobian matrix on the form that is used by tomSym.

C.3 Sub function details

Advanced users may need to understand how and what some sub functions execute in TomSym. This section describes how to add functionality to TomSym and some more details "under the hood".

C.3.1 ifThenElse — Smoothened if/then/else.

`y = ifThenElse(cmp1, op, cmp2, yTrue, yFalse, s)` replicates the behavior of the C-language construction y = (condition ? yTrue : yFalse) but with a smoothing sigmoid function, scaled by s. (Setting s≤0 gives discontinuous "standard" if-then-else.)

The operand string op must be the name of one of the comparison operators ('eq', 'lt', 'le', etc.)

Example: The absolute value of x is `ifThenElse(x,'gt',0,x,-x)`

```
Overloaded methods:
   tomSym/ifThenElse
```

C.3.2 tomSym/derivative

The symbolic derivative of a tomSym object.

`df_dx = derivative(f,x)` computes the derivative of a tomSym object f with respect to a symbol x.

`d2f_dxdy = derivative(f,x,y,..)` computes the second (third, ...) derivative of f with respect to the symbols x, y, ...

Both f and x can be matrices.

The symbol x can either be a tomSym symbol, or a concatenation of tomSym symbols.

There exist several conventions for how the elements are arranged in the derivative of a matrix. This function uses the convention that `vec(df)` = df_dx*`vec(dx)`. This means that if `size(f)` = [m n] and `size(x)` = [p q] then `size(df_dx)` = [m*n, p*q]. Thus, the derivative of a matrix or vector with respect to a scalar is a column vector, the derivative of a scalar is a row vector, and the derivative of any matrix with respect to itself is an identity matrix.

Examples:

- If f and x are vectors, then `J = derivative(f,x)` computes the Jacobian matrix.
- If f is scalar and x is a vector, then `H = derivative(f,x,x)` computes the Hessian matrix.

For functions that tomSym/derivative is unfamiliar with, it assumes that there also exists a derivative function for each argument. For example, when computing the derivative of a function "userfun", it is assumed that "userfunJ1" gives the derivative of userfun with respect to its first input argument.

Reference: Brookes, M., "The Matrix Reference Manual"

`http://www.ee.ic.ac.uk/hp/staff/dmb/matrix/intro.html`

C.3.3 ppderivative — The derivative of a piecewise polynomial.

`dpp = ppderivative(pp)` returns a piecewise polynomial which is the derivative of pp, except at points where pp is discontinuous.

`dpp = ppderivative(pp,n)` returns the nth derivative. A negative n results in the antiderivative (integral) of pp.

C.3.4 tomSym/mcode

Generate m-code from a tomSym object.

`[code, data, header] = mcode(f)` generates m-code, representing an object f.

All constants are moved to a data object, which needs to be supplied in the function call.

The returned function header lists all symbols alphabetically, and the data input last.

D Global Variables and Recursive Calls

The use of globally defined variables in TOMLAB is well motivated, for example to avoid unnecessary evaluations, storage of sparse patterns, internal communication, computation of elapsed CPU time etc. The global variables used in TOMLAB are listed in Table 154 and 155.

Even though global variables is efficient to use in many cases, it will be trouble with recursive algorithms and recursive calls. Therefore, the routines *globalSave* and *globalGet* have been defined. The *globalSave* routine saves all global variables in a structure *glbSave(depth)* and then initialize all of them as empty. By using the depth variable, an arbitrarily number of recursions are possible. The other routine *globalGet* retrieves all global variables in the structure *glbSave(depth)*.

For solving some kinds of problems it could be suitable or even necessary to apply algorithms which is based on a recursive approach. A common case occurs when an optimization solver calls another solver to solve a subproblem. For example, the EGO algorithm (implemented in the routine *ego*) solves an unconstrained (**uc**) and a box-bounded global optimization problem (**glb**) in each iteration. To avoid that the global variables are not re-initialized or given new values by the underlying procedure TOMLAB saves the global variables in the workspace before the underlying procedure is called. Directly after the call to the underlying procedure the global variables are restored.

To illustrate the idea, the following code would be a possible part of the *ego* code, where the routines *globalSave* and *globalGet* are called.

```
...
...
   global GlobalLevel
   if isempty(GlobalLevel)
      GlobalLevel=1;
   else
      GlobalLevel=GlobalLevel+1;
   end
   Level=GlobalLevel
   globalSave(Level);

   EGOResult = glbSolve(EGOProb);

   globalGet(Level);
   GlobalLevel=GlobalLevel-1;
...
...
   Level=GlobalLevel
   globalSave(Level);

   [DACEResult] = ucSolve(DACEProb);

   globalGet(1);
   globalGet(Level);
   GlobalLevel=GlobalLevel-1;
...
...
```

In most cases the user does not need to define the above statements and instead use the special driver routine *tomSolve* that does the above global variable checks and savings and calls the solver in between. In the actual implementation of the *ego* solver the above code is simplified to the following:

```
...
...
   EGOResult = tomSolve('glbSolve',EGOProb);
...
...
   DACEResult = tomSolve('ucSolve',DACEProb);
...
...
```

This safely handles the global variables and is the recommended way for users in need of recursive optimization solutions.

Table 154: The global variables used in TOMLAB

Name	**Description**
MAXCOLS	Number of screen columns. Default 120.
MAXMENU	Number of menu items showed on one screen. Default 50.
MAX_c	Maximum number of constraints to be printed.
MAX_x	Maximum number of variables to be printed.
MAX_r	Maximum number of residuals to be printed.
n_f	Counter for the number of function evaluations.
n_g	Counter for the number of gradient evaluations.
n_H	Counter for the number of Hessian evaluations.
n_c	Counter for the number of constraint evaluations.
n_dc	Counter for the number of constraint normal evaluations.
n_d2c	Counter for the number of evaluations of the 2nd part of 2nd derivative matrix of the Lagrangian function.
n_r	Counter for the number of residual evaluations.
n_J	Counter for the number of Jacobian evaluations.
n_d2r	Counter for the number of evaluations of the 2nd part of the Hessian for a nonlinear least squares problem .
NLP_x	Value of x when computing *NLP_f*.
NLP_f	Function value.
NLP_xg	Value of x when computing *NLP_g*.
NLP_g	Gradient value.
NLP_xH	Value of x when computing *NLP_H*.
NLP_H	Hessian value.
NLP_xc	Value of x when computing *NLP_c*.
NLP_c	Constraints value.
NLP_pSepFunc	Number of partially separable functions.
NLP_pSepIndex	Index for the separated function computed.

Table 155: The global variables used in TOMLAB

Name	Description
US_A	Problem dependent information sent between user routines. The user is recommended to always use this variable.
LS_A	Problem dependent information sent from residual routine to Jacobian routine.
LS_x	Value of x when computing *LS_r*
LS_r	Residual value.
LS_xJ	Value of x when computing *LS_J*
LS_J	Jacobian value.
SEP_z	Separated variables z.
SEP_Jz	Jacobian for separated variables z.
wNLLS	Weighting of least squares residuals (internal variable in *nlp_r* and *nlp_J*).
alphaV	Vector with all step lengths α for each iteration.
BUILDP	Flag.
F_X	Matrix with function values.
pLen	Number of iterations so far.
p_dx	Matrix with all search directions.
X_max	The biggest x-values for all iterations.
X_min	The smallest x-values for all iterations.
X_NEW	Last x point in line search. Possible new x_k.
X_OLD	Last known base point x_k
probType	Defines the type of optimization problem.
solvType	Defines the solver type.
answer	Used by the GUI for user control options.
instruction	Used by the GUI for user control options.
question	Used by the GUI for user control options.
plotData	Structure with plotting parameters.
Prob	Problem structure, see Table 134.
Result	Result structure, see Table 149.
runNumber	Vector index when *Result* is an array of structures.
TIME0	Used to compute CPU time and real time elapsed.
TIME1	Used to compute CPU time and real time elapsed
cJPI	Used to store sparsity pattern for the constraint Jacobian when automatic differentiation is used.
HPI	Used to store sparsity pattern for the Hessian when automatic differentiation is used.
JPI	Used to store sparsity pattern for the Jacobian when automatic differentiation is used.
glbSave	Used to save global variables in recursive calls to TOMLAB.

E External Interfaces

Some users may have been used to work with MathWorks Optimization Toolbox, or have code written for use with these toolboxes. For that reason TOMLAB contains interfaces to simplify the transfer of code to TOMLAB. There are two ways in which the MathWorks Optimization Toolbox may be used in TOMLAB. One way is to use the same type of call to the main solvers as in MathWorks Optimization TB, but the solution is obtained by converting the problem into the TOMLAB format and calling a TOMLAB solver. The other way is to formulate the problem in any of the TOMLAB formats, but when solving the problem calling the driver routine with the name of the Optimization Toolbox solver. Which way to use is determined by setting *if 0* or *if 1* in *startup.m* in the addpath for the variable *OPTIM*. If setting *if 1* then the TOMLAB versions are put first and MathWorks Optimization TB is not accessible.

E.1 Solver Call Compatible with Optimization Toolbox

TOMLAB is call compatible with MathWorks Optimization TB. This means that the same syntax could be used, but the solver is a TOMLAB solver instead. TOMLAB normally adds one extra input, the *Prob* structure, and one extra output argument, the *Result* structure. Both extra parameters are optional, but if the user are adding extra input arguments in his call to the MathWorks Optimization TB solver, to use the TOMLAB equivalents, the extra input must be shifted one step right, and the variable *Prob* be put first among the extra arguments. Table 156 gives a list of the solvers with compatible interfaces.

Table 156: Call compatible interfaces to MathWorks Optimization TB.

Function	**Type of problem solved**
bintprog	Binary programming.
fmincon	Constrained minimization.
fminsearch	Unconstrained minimization using Nelder-Mead type simplex search method.
fminunc	Unconstrained minimization using gradient search.
linprog	Linear programming.
lsqcurvefit	Nonlinear least squares curve fitting.
lsqlin	Linear least squares.
lsqnonlin	Linear least squares with nonnegative variable constraints.
lsqnonneg	Nonlinear least squares.
quadprog	Quadratic programming.

In Table 157 a list is given with the demonstration files available in the directory *examples* that exemplify the usage of the call compatible interfaces. In the next sections the usage of some of the solvers are further discussed and exemplified.

E.1.1 Solving LP Similar to Optimization Toolbox

For linear programs the MathWorks Optimization TB solver is *linprog*. The TOMLAB *linprog* solver adds one extra input argument, the *Prob* structure, and one extra output argument, the *Result* structure. Both extra parameters are optional, but means that the additional functionality of the TOMLAB LP solver is accessible.

An example of the use of the TOMLAB *linprog* solver to solve test problem (13) illustrates the basic usage

Table 157: Testroutines for the call compatible interfaces to MathWorks Optimization TB present in the *examples* directory in the TOMLAB distribution.

Function	Type of problem solved
testbintprog	Test of binary programming.
testfmincon	Test of constrained minimization.
testfminsearch	Test of unconstrained minimization using a Nelder-Mead type simplex search method.
testfminunc	Test of unconstrained minimization using gradient search.
testlinprog	Test of linear programming.
testlsqcurvefit	Test of nonlinear least squares curve fitting.
testlsqlin	Test of linear least squares.
testlsqnonlin	Test of linear least squares with nonnegative variable constraints.
testlsqnonneg	Test of nonlinear least squares.
testquadprog	Test of quadratic programming.

File: tomlab/usersguide/lpTest2.m

```
lpExample;

% linprog needs linear inequalities and equalities to be given separately
% If the problem has both linear inequalities (only upper bounded)
% and equalities we can easily detect which ones doing the following calls

ix = b_L==b_U;
E  = find(ix);
I  = find(~ix);

[x, fVal, ExitFlag, Out, Lambda] = linprog(c, A(I,:),b_U(I),...
    A(E,:), b_U(E), x_L, x_U, x_0);

% If the problem has linear inequalites with different lower and upper bounds
% the problem can be transformed using the TOMLAB routine cpTransf.
% See the example file tomlab\examples\testlinprog.m for an example.

fprintf('\n');
fprintf('\n');
disp('Run TOMLAB linprog on LP Example');
fprintf('\n');
xprinte(A*x-b_U,          'Constraints Ax-b_U ');
xprinte(Lambda.lower,     'Lambda.lower:      ');
xprinte(Lambda.upper,     'Lambda.upper:      ');
xprinte(Lambda.eqlin,     'Lambda.eqlin:      ');
xprinte(Lambda.ineqlin,   'Lambda.ineqlin:    ');
xprinte(x,                'x:                 ');
format compact
```

```
disp('Output Structure')
disp(Out)
fprintf('Function value %30.20f. ExitFlag %d\n',fVal,ExitFlag);
```

The results from this test show the same results as previous runs in Section 5, because the same solver is called.

File: tomlab/usersguide/lpTest2.out

```
linprog (CPLEX): Optimization terminated successfully


Run TOMLAB linprog on LP Example

Constraints Ax-b_U   0.000000e+000  0.000000e+000
Lambda.lower:        0.000000e+000  0.000000e+000
Lambda.upper:        0.000000e+000  0.000000e+000
Lambda.eqlin:
Lambda.ineqlin:     -1.857143e+000 -1.285714e+000
x:                   2.571429e+000  1.714286e+000
Output Structure
      iterations: 2
       algorithm: 'CPLEX: CPLEX Dual Simplex LP solver'
    cgiterations: 0
Function value       -26.57142857142857300000. ExitFlag 1
```

E.1.2 Solving QP Similar to Optimization Toolbox

For quadratic programs the MathWorks Optimization TB solver is *quadprog*. The TOMLAB *quadprog* solver adds one extra input argument, the *Prob* structure, and one extra output argument, the *Result* structure. Both extra parameters are optional, but means that the additional functionality of the TOMLAB QP solver is accessible.

An example of the use of the TOMLAB *quadprog* solver to solve test problem (15) illustrates the basic usage

File: tomlab/usersguide/qpTest2.m

```
qpExample;

% quadprog needs linear equalities and equalities to be given separately
% If the problem has both linear inequalities (only upper bounded)
% and equalities we can easily detect which ones doing the following calls

ix = b_L==b_U;
E  = find(ix);
I  = find(~ix);

[x, fVal, ExitFlag, Out, Lambda] = quadprog(F, c, A(I,:),b_U(I),...
    A(E,:), b_U(E), x_L, x_U, x_0);
```

```
% If A has linear inequalites with different lower and upper bounds
% the problem can be transformed using the TOMLAB routine cpTransf.
% See the example file tomlab\examples\testquadprog.m for an example.

fprintf('\n');
fprintf('\n');
disp('Run TOMLAB quadprog on QP Example');
fprintf('\n');
xprinte(A*x-b_U,          'Constraints Ax-b_U ');
xprinte(Lambda.lower,     'Lambda.lower:      ');
xprinte(Lambda.upper,     'Lambda.upper:      ');
xprinte(Lambda.eqlin,     'Lambda.eqlin:      ');
xprinte(Lambda.ineqlin,   'Lambda.ineqlin:    ');
xprinte(x,                'x:                 ');
format compact
disp('Output Structure')
disp(Out)
fprintf('Function value %30.20f. ExitFlag %d\n',fVal,ExitFlag);
```

The restricted problem formulation in MathWorks Optimization TB sometimes makes it necessary to transform the problem. See the comments in the above example and the test problem file tomlab/examples/testquadprog.m . The results from this test show the same results as previous runs

File: tomlab/usersguide/qpTest2.out

```
Run TOMLAB quadprog on QP Example

Constraints Ax-b_U  -4.888889e+000  0.000000e+000
Lambda.lower:        0.000000e+000  0.000000e+000
Lambda.upper:        0.000000e+000  0.000000e+000
Lambda.eqlin:       -3.500000e+000
Lambda.ineqlin:     -5.102800e-016
x:                   5.555556e-002  5.555556e-002
Output Structure
       iterations: 1
        algorithm: 'qpopt: QPOPT 1.0 QP/LP code'
     cgiterations: []
    firstorderopt: []
Function value        -0.02777777777777779000. ExitFlag 1
```

E.2 The Matlab Optimization Toolbox Interface

Included in TOMLAB is an interface to a number of the solvers in the MathWorks Optimization TB v1.5 [10]. and MathWorks Optimization TB [12]. The solvers that are directly possible to use, when a problem is generated in the TOMLAB format, are listed in Table 158. The user must of course have a valid license. The TOMLAB

interface routines are *opt15Run* and *opt20Run*, but the user does not need to call these directly, but can use the standard multi-solver driver interface routine *tomRun*.

Several low-level interface routines have been written. For example, the *constr* solver needs both the objective function and the vector of constraint functions in the same call, which *nlp_fc* supplies. Also the gradient vector and the matrix of constraint normals should be supplied in one call. These parameters are returned by the routine *nlp_gdc*.

MathWorks Optimization TB v1.5 is using a parameter vector OPTIONS of length 18, that the routine *foptions* is setting up the default values for. MathWorks Optimization TB is instead using a structure.

Table 158: Optimization toolbox routines with a TOMLAB interface.

Function	**Type of problem solved**
bintprog	Binary programming.
fmincon	Constrained minimization.
fminsearch	Unconstrained minimization using Nelder-Mead type simplex search method.
fminunc	Unconstrained minimization using gradient search.
linprog	Linear programming.
lsqcurvefit	Nonlinear least squares curve fitting.
lsqlin	Linear least squares.
lsqnonlin	Linear least squares with nonnegative variable constraints.
lsqnonneg	Nonlinear least squares.
quadprog	Quadratic programming.
constr	Constrained minimization.
fmins	Unconstrained minimization using Nelder-Mead type simplex search method.
fminu	Unconstrained minimization using gradient search.
leastsq	Nonlinear least squares.
lp	Linear programming.
qp	Quadratic programming.

E.3 The AMPL Interface

The AMPL interface is described in a separate manual. Enter help amplAssign in MATLAB to see the functionality.

F Motivation and Background to TOMLAB

Many scientists and engineers are using Matlab as a modeling and analysis tool, but for the solution of optimization problems, the support is weak. That was one motive for starting the development of TOMLAB;

To solve optimization problems, traditionally the user has been forced to write a Fortran code that calls some standard solver written as a Fortran subroutine. For nonlinear problems, the user must also write subroutines computing the objective function value and the vector of constraint function values. The needed derivatives are either explicitly coded, computed by using numerical differences or derived using automatic differentiation techniques.

In recent years several modeling languages are developed, like AIMMS [5], AMPL [24], ASCEND [62], GAMS [6, 11] 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 [18].

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 [51, 64]. Nowadays, languages like Matlab has a rapid growth of users. Matlab was originally created [56] as a preprocessor to the standard Fortran subroutine libraries in numerical linear algebra, LINPACK [17] and EISPACK [71] [27], much the same idea as the modeling languages discussed above.

Matlab of today is an advanced and powerful tool, with graphics, animation and advanced menu design possibilities integrated with the mathematics. The Matlab language has made the development of toolboxes possible, which serves as a direct extension to the language itself. Using Matlab as an environment for solving optimization problems offers much more possibilities for analysis than just the pure solution of the problem. The increased quality of the Matlab MEX-file interfaces makes it possible to run Fortran and C-programs on both PC and Unix systems.

The concept of TOMLAB is to integrate all different systems, getting access to the best of all worlds. TOMLAB should be seen as a complement to existing model languages, for the user needing more power and flexibility than given by a modeling system.

G Performance Tests on Linear Programming Solvers

We have made tests to compare the efficiency of different solvers on medium size LP problems. The solver *lpSimplex*, two algorithms implemented in the solver *linprog* from Optimization Toolbox 2.0 [12] and the Fortran solvers MINOS and QPOPT, available in TOMLAB v4.0, are compared. In all test cases the solvers converge to the same solution. The results are presented in five tables

Table 159, Table 160, Table 161, Table 162 and Table 163. The problem dimensions and all elements in (6) are chosen randomly. Since the simplex algorithm in *linprog* does not return the number of iterations as output, these figures could not be presented. *lpSimplex* has been run with two selection rules; Bland's cycling prevention rule and the minimum cost rule. The minimum cost rule is the obvious choice, because *lpSimplex* handles most cycling cases without problems, and also tests on cycling, and switches to Bland's rule in case of emergency (does not seem to occur). But it was interesting to see how much slower Bland's rule was.

The results in Table 159 show that problems with about 200 variables and 150 inequality constraints are solved by *lpSimplex* fast and efficient. When comparing elapsed computational time for 20 problems, it is clear that *lpSimplex* is much faster then the corresponding simplex algorithm implemented in the *linprog* solver. In fact *lpSimplex*, with the minimum cost selection rule, is more than five times faster, a remarkable difference. *lpSimplex* is also more than twice as fast as the other algorithm implemented in *linprog*, a primal-dual interior-point method aimed for large-scale problems [12]. There is a penalty about a factor of three to choose Bland's rule to prevent cycling in *lpSimplex*. The solvers written in Fortran, MINOS and QPOPT, of course run much faster, but the iteration count show that *lpSimplex* converges as fast as QPOPT and slightly better than MINOS. The speed-up is a factor of 35 when running QPOPT using the MEX-file interface.

In Table 160 a similar test is shown, running 20 problems with about 100 variables and 50 inequality constraints. The picture is the same, but the time difference, a factor of five, between *lpSimplex* and the Fortran solvers are not so striking for these lower dimensional problems. *lpSimplex* is now more than nine times faster than the simplex algorithm in *linprog* and twice as fast as the primal-dual interior-point method in *linprog*.

A similar test on larger dense problems, running 20 problems with about 500 variables and 240 inequality constraints, shows no benefit in using the primal-dual interior-point method in *linprog*, see Table 163. In that test *lpSimplex* is more than five times faster, and 15 times faster than the simplex algorithm in *linprog*. Still it is about 35 times faster to use the MEX-file interfaces.

In conclusion, looking at the summary for all tables collected in Table 164, for dense problems the LP solvers in Optimization Toolbox offers no advantage compared to the TOMLAB solvers. It is clear that if speed is critical, the availability of Fortran solvers callable from Matlab using the MEX-file interfaces in TOMLAB v4.0 is very important.

Table 159: Computational results on randomly generated medium size LP problems for four different routines. *Iter* is the number of iterations and $Time$ is the elapsed time in seconds on a Dell Latitude CPi 266XT running Matlab 5.3. The *lpS* solver is the TOMLAB $lpSimplex$, and it is run with both Bland's selection rule (iterations It_b, time T_b) and with the minimum cost selection rule (iterations It_m, time T_m). The *linprog* solver in the Optimization Toolbox 2.0 implements two different algorithms, a medium-scale simplex algorithm (time T_m) and a large-scale primal-dual interior-point method (iterations It_l, time T_l). The number of variables, n, the number of inequality constraints, m, the objective function coefficients, the linear matrix and the right hand side are chosen randomly. The last row shows the mean value of each column.

n	m	*lpS*	*lpS*	$Minos$	*qpopt*	*linprog*	*lpS*	*lpS*	$Minos$	*qpopt*	*linprog*	*linprog*
		It_b	It_m	$Iter$	$Iter$	It_l	T_b	T_m	$Time$	$Time$	T_m	T_l
128	32	37	12	10	11	16	1.05	0.61	0.33	0.31	9.06	1.14
129	60	8	10	10	9	17	0.63	0.59	0.24	0.21	9.20	2.07
125	45	8	9	16	7	14	0.57	0.59	0.35	0.32	8.20	1.34
81	65	27	5	7	4	12	1.30	0.54	0.23	0.21	3.51	1.38
102	40	25	9	12	8	12	1.00	0.60	0.39	0.33	5.26	1.01
96	33	13	7	6	8	11	0.65	0.41	0.34	0.32	4.72	0.84
110	61	29	10	9	9	15	1.38	0.66	0.25	0.33	6.34	1.73
113	27	25	8	161	8	10	0.87	0.50	0.41	0.34	6.72	0.77
127	58	16	9	13	8	14	0.91	0.58	0.26	0.34	8.58	1.82
85	58	10	7	7	7	14	0.68	0.59	0.25	0.21	3.70	1.45
103	31	15	7	9	6	12	0.69	0.52	0.35	0.33	5.39	0.87
101	41	22	9	11	9	11	0.87	0.56	0.36	0.22	5.20	0.98
83	41	9	6	7	7	12	0.54	0.36	0.38	0.33	3.55	0.98
118	39	28	9	8	8	13	0.89	0.57	0.36	0.34	7.23	1.14
92	33	13	8	8	7	12	0.63	0.53	0.23	0.33	4.33	0.90
110	46	21	7	15	6	13	0.81	0.46	0.25	0.34	6.37	1.26
82	65	25	6	6	5	15	1.21	0.51	0.38	0.22	3.41	1.63
104	29	6	6	10	4	11	0.47	0.36	0.23	0.34	5.52	0.85
83	48	28	8	10	10	13	1.13	0.50	0.24	0.35	3.53	1.15
90	50	8	4	4	3	11	0.44	0.35	0.24	0.23	4.13	1.18
103	45	19	8	17	7	13	0.84	0.52	0.30	0.30	5.70	1.23

Table 160: Computational results on randomly generated medium size LP problems for four different routines. *Iter* is the number of iterations and *Time* is the elapsed time in seconds on a Dell Latitude CPi 266XT running Matlab 5.3. The *lpS* solver is the TOMLAB *lpSimplex*, and it is run with both Bland's selection rule (iterations It_b, time T_b) and with the minimum cost selection rule (iterations It_m, time T_m). The *linprog* solver in the Optimization Toolbox 2.0 implements two different algorithms, a medium-scale simplex algorithm (time T_m) and a large-scale primal-dual interior-point method (iterations It_l, time T_l). The number of variables, n, the number of inequality constraints, m, the objective function coefficients, the linear matrix and the right hand side are chosen randomly. The last row shows the mean value of each column.

n	m	*lpS*	*lpS*	*Minos*	*qpopt*	*linprog*	*lpS*	*lpS*	*Minos*	*qpopt*	*linprog*	*linprog*
		It_b	It_m	*Iter*	*Iter*	It_l	T_b	T_m	*Time*	*Time*	T_m	T_l
228	132	32	10	17	12	22	3.41	1.56	0.49	0.39	38.66	11.51
191	164	20	9	9	10	18	3.12	1.85	0.49	0.26	24.91	12.50
212	155	63	16	30	16	19	7.90	2.76	0.54	0.41	33.36	12.57
185	158	53	25	16	16	18	6.86	4.00	0.38	0.43	23.88	11.29
222	168	35	12	0	12	21	5.38	2.56	0.64	0.42	40.13	17.78
207	162	10	8	6	7	21	1.91	1.69	0.51	0.27	33.74	15.66
229	130	42	12	21	19	21	4.31	1.81	0.42	0.44	44.53	11.69
213	136	56	6	21	6	19	6.02	1.19	0.51	0.39	36.54	11.07
227	146	95	19	33	20	23	10.91	2.94	0.45	0.45	44.84	15.82
192	150	25	6	13	5	16	3.22	1.26	0.53	0.27	27.07	10.79
195	155	12	8	9	7	22	2.19	1.76	0.52	0.39	27.40	14.76
221	160	30	12	10	11	22	4.66	2.41	0.59	0.43	36.95	18.00
183	144	61	9	9	10	20	7.08	1.62	0.37	0.39	22.34	11.22
200	165	19	10	0	14	19	3.27	2.22	0.61	0.42	27.94	14.43
199	137	16	6	7	5	19	2.04	1.04	0.48	0.39	28.67	9.90
188	154	18	8	9	7	17	2.59	1.57	0.53	0.39	25.19	10.81
202	159	25	13	0	11	17	3.82	2.50	0.60	0.44	30.28	12.37
223	155	103	16	20	17	24	12.50	2.95	0.56	0.44	39.54	18.06
196	121	17	7	16	6	18	1.81	1.08	0.37	0.40	27.59	7.94
202	133	47	10	12	12	20	4.71	1.34	0.38	0.41	30.03	10.09
206	149	39	11	13	11	20	4.89	2.01	0.50	0.39	32.18	12.91

Table 161: Computational results on randomly generated medium size LP problems for four different routines. *Iter* is the number of iterations and *Time* is the elapsed time in seconds on a Dell Latitude CPi 266XT running Matlab 5.3. The *lpS* solver is the TOMLAB *lpSimplex*, and it is run with both Bland's selection rule (iterations It_b, time T_b) and with the minimum cost selection rule (iterations It_m, time T_m). The *linprog* solver in the Optimization Toolbox 2.0 implements two different algorithms, a medium-scale simplex algorithm (time T_m) and a large-scale primal-dual interior-point method (iterations It_l, time T_l). The number of variables, n, the number of inequality constraints, m, the objective function coefficients, the linear matrix and the right hand side are chosen randomly. The last row shows the mean value of each column.

n	m	*lpS*	*lpS*	*Minos*	*qpopt*	*linprog*	*lpS*	*lpS*	*Minos*	*qpopt*	*linprog*	*linprog*
		It_b	It_m	*Iter*	*Iter*	It_l	T_b	T_m	*Time*	*Time*	T_m	T_l
328	192	174	26	33	34	24	34.73	6.59	0.70	0.76	121.57	50.52
326	212	65	10	24	12	20	14.67	3.28	0.82	0.57	116.00	49.87
325	185	15	15	33	15	33	4.19	4.31	0.78	0.55	112.43	63.63
327	186	21	11	14	13	26	4.49	2.86	0.75	0.55	112.95	49.85
327	192	22	6	8	6	19	5.01	1.92	0.73	0.48	113.05	40.58
285	181	9	7	11	7	21	2.33	1.98	0.64	0.44	80.13	30.33
323	219	24	10	15	11	22	6.44	3.39	0.88	0.56	110.42	59.27
284	201	45	10	10	9	24	9.46	3.21	0.71	0.35	81.13	44.80
285	199	22	9	14	8	21	4.85	2.62	0.71	0.33	78.64	39.07
296	228	33	11	10	13	23	9.00	3.78	0.77	0.39	89.67	59.23
310	185	28	14	19	16	25	5.62	3.30	0.73	0.54	96.93	43.75
311	219	23	12	12	17	22	6.53	4.13	0.78	0.60	97.05	53.90
280	206	58	23	28	17	20	12.20	5.80	0.76	0.40	75.66	38.22
319	204	17	11	11	12	23	4.41	3.45	0.64	0.54	106.16	52.84
287	202	8	6	6	5	17	2.43	1.79	0.75	0.34	78.26	32.93
328	202	44	9	11	10	18	9.32	2.72	0.76	0.53	117.09	41.86
307	213	85	12	34	12	30	19.35	3.97	0.86	0.51	98.97	70.47
285	199	29	11	11	9	24	6.43	3.27	0.71	0.47	78.32	44.30
315	194	22	10	8	9	20	5.14	3.00	0.73	0.52	102.28	41.73
310	181	38	6	7	5	22	6.95	1.80	0.71	0.46	96.99	36.93
308	200	39	11	16	12	23	8.68	3.36	0.75	0.50	98.18	47.20

Table 162: Computational results on randomly generated medium size LP problems for four different routines. *Iter* is the number of iterations and *Time* is the elapsed time in seconds on a Dell Latitude CPi 266XT running Matlab 5.3. The *lpS* solver is the TOMLAB *lpSimplex*, and it is run with both Bland's selection rule (iterations It_b, time T_b) and with the minimum cost selection rule (iterations It_m, time T_m). The *linprog* solver in the Optimization Toolbox 2.0 implements two different algorithms, a medium-scale simplex algorithm (time T_m) and a large-scale primal-dual interior-point method (iterations It_l, time T_l). The number of variables, n, the number of inequality constraints, m, the objective function coefficients, the linear matrix and the right hand side are chosen randomly. The last row shows the mean value of each column.

n	m	*lpS*	*lpS*	*Minos*	*qpopt*	*linprog*	*lpS*	*lpS*	*Minos*	*qpopt*	*linprog*	*linprog*
		It_b	It_m	*Iter*	*Iter*	It_l	T_b	T_m	*Time*	*Time*	T_m	T_l
428	232	8	6	7	5	24	3.02	2.47	0.97	0.57	248.88	90.83
421	234	22	5	11	4	22	7.54	2.64	0.86	0.54	232.29	84.15
397	242	19	9	8	10	26	7.13	4.30	0.93	0.52	196.02	101.09
388	226	30	10	11	10	24	9.19	3.80	0.89	0.51	187.35	78.37
381	248	23	6	11	5	29	8.28	3.31	0.99	0.54	176.07	109.18
402	228	80	16	28	22	25	22.21	5.94	1.03	0.86	207.52	84.60
383	241	41	7	10	7	22	13.30	3.79	0.93	0.57	180.90	83.62
421	236	94	21	19	15	34	27.94	7.80	1.06	0.80	234.26	131.09
402	253	23	8	8	7	22	8.58	4.01	0.89	0.62	206.50	95.63
395	260	24	8	8	7	23	8.95	3.95	0.94	0.48	197.14	100.85
404	224	73	7	13	6	21	20.85	3.11	0.83	0.47	208.55	70.67
393	267	44	11	15	9	25	16.64	5.86	1.09	0.65	192.59	116.73
393	247	15	8	9	7	19	5.56	3.67	0.86	0.63	191.53	77.74
384	245	79	14	27	20	25	24.59	6.10	1.08	0.79	185.63	97.19
385	254	75	9	16	9	21	25.06	5.30	1.06	0.67	177.95	88.69
409	226	58	8	9	8	23	15.76	3.56	0.82	0.63	210.86	78.32
410	263	38	15	20	19	29	14.66	7.27	0.98	0.74	214.83	130.13
403	250	117	12	27	20	20	36.56	5.35	1.06	0.87	201.18	81.53
426	238	15	4	5	3	20	5.20	2.05	0.99	0.44	239.71	80.46
409	250	57	10	13	10	24	19.00	5.01	1.21	0.72	210.15	101.34
402	243	47	10	14	10	24	15.00	4.46	0.98	0.63	204.99	94.11

Table 163: Computational results on randomly generated medium size LP problems for four different routines. *Iter* is the number of iterations and *Time* is the elapsed time in seconds on a Dell Latitude CPi 266XT running Matlab 5.3. The *lpS* solver is the TOMLAB *lpSimplex*, and it is run with both Bland's selection rule (iterations It_b, time T_b) and with the minimum cost selection rule (iterations It_m, time T_m). The *linprog* solver in the Optimization Toolbox 2.0 implements two different algorithms, a medium-scale simplex algorithm (time T_m) and a large-scale primal-dual interior-point method (iterations It_l, time T_l). The number of variables, n, the number of inequality constraints, m, the objective function coefficients, the linear matrix and the right hand side are chosen randomly. The last row shows the mean value of each column.

n	m	*lpS*	*lpS*	*Minos*	*qpopt*	*linprog*	*lpS*	*lpS*	*Minos*	*qpopt*	*linprog*	*linprog*
		It_b	It_m	*Iter*	*Iter*	It_l	T_b	T_m	*Time*	*Time*	T_m	T_l
528	232	35	7	7	6	28	12.33	3.50	1.28	0.86	453.03	124.19
482	252	33	9	7	8	25	12.02	4.26	1.00	0.71	346.37	120.24
503	251	72	15	38	17	35	25.45	6.79	1.49	1.01	387.91	170.35
507	259	142	18	46	27	28	50.68	8.55	1.43	1.33	397.67	147.41
487	240	48	17	33	19	26	16.69	7.02	1.29	1.03	346.64	114.96
506	251	46	8	11	8	24	16.92	4.19	1.13	0.78	394.38	119.71
504	256	35	9	16	8	36	14.73	4.97	1.26	0.81	395.37	183.20
489	255	36	28	27	28	26	14.39	11.87	1.32	1.30	355.66	129.45
514	228	9	4	4	3	32	3.24	1.80	1.05	0.51	399.44	133.82
524	245	64	11	27	14	28	21.99	5.34	1.26	1.00	439.31	135.32
506	255	112	22	28	23	23	40.12	10.07	1.12	1.21	385.12	117.49
497	224	50	11	14	12	31	15.51	4.57	1.11	0.86	362.38	121.94
482	249	27	16	17	20	30	10.24	6.75	1.15	1.08	339.27	138.16
485	249	18	6	21	5	20	6.36	2.87	1.35	0.55	340.35	95.15
509	223	84	22	35	17	35	23.51	7.55	1.17	1.04	390.88	142.31
506	224	38	12	11	14	33	11.89	4.65	1.09	0.94	383.13	132.21
511	241	115	10	36	9	26	36.51	4.32	1.29	0.69	390.78	122.23
497	230	78	23	43	12	26	23.60	8.27	1.29	0.75	362.08	109.30
514	226	84	21	42	26	31	25.10	7.90	1.57	1.47	407.94	126.53
511	268	59	10	30	9	28	24.74	5.76	1.43	0.94	385.56	161.65
503	243	59	14	25	14	29	20.30	6.05	1.26	0.94	383.16	132.28

Table 164: Computational results on randomly generated medium size LP problems for four different routines. *Iter* is the number of iterations and *Time* is the elapsed time in seconds on a Dell Latitude CPi 266XT running Matlab 5.3. The *lpS* solver is the TOMLAB *lpSimplex*, and it is run with both Bland's selection rule (iterations It_b, time T_b) and with the minimum cost selection rule (iterations It_m, time T_m). The *linprog* solver in the Optimization Toolbox 2.0 implements two different algorithms, a medium-scale simplex algorithm (time T_m) and a large-scale primal-dual interior-point method (iterations It_l, time T_l). The number of variables, n, the number of inequality constraints, m, the objective function coefficients, the linear matrix and the right hand side are chosen randomly. Each row presents the mean of a test of 20 test problems with mean sizes shown in the first two columns.

n	m	*lpS*	*lpS*	*Minos*	*qpopt*	*linprog*	*lpS*	*lpS*	*Minos*	*qpopt*	*linprog*	*linprog*
		It_b	It_m	*Iter*	*Iter*	It_l	T_b	T_m	*Time*	*Time*	T_m	T_l
103	45	19	8	17	7	13	0.84	0.52	0.30	0.30	5.70	1.23
206	149	39	11	13	11	20	4.89	2.01	0.50	0.39	32.18	12.91
308	200	39	11	16	12	23	8.68	3.36	0.75	0.50	98.18	47.20
402	243	47	10	14	10	24	15.00	4.46	0.98	0.63	204.99	94.11
503	243	59	14	25	14	29	20.30	6.05	1.26	0.94	383.16	132.28

References

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

[2] M. Al-Baali and R. Fletcher. Variational methods for non-linear least squares. *J. Oper. Res. Soc.*, 36:405–421, 1985.

[3] M. Al-Baali and R. Fletcher. An efficient line search for nonlinear least-squares. *Journal of Optimization Theory and Applications*, 48:359–377, 1986.

[4] Jordan M. Berg and K. Holmström. On Parameter Estimation Using Level Sets. *SIAM Journal on Control and Optimization*, 37(5):1372–1393, 1999.

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

[6] 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.

[7] M. Björkman. Nonlinear Least Squares with Inequality Constraints. Bachelor Thesis, Department of Mathematics and Physics, Mälardalen University, Sweden, 1998. Supervised by K. Holmström.

[8] M. Björkman and K. Holmström. Global Optimization Using the DIRECT Algorithm in Matlab. *Advanced Modeling and Optimization*, 1(2):17–37, 1999.

[9] M. Björkman and K. Holmström. Global Optimization of Costly Nonconvex Functions Using Radial Basis Functions. *Optimization and Engineering*, 1(4):373–397, 2000.

[10] Mary Ann Branch and Andy Grace. *Optimization Toolbox User's Guide*. 24 Prime Park Way, Natick, MA 01760-1500, 1996.

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

[12] Thomas Coleman, Mary Ann Branch, and Andy Grace. *Optimization Toolbox User's Guide*. 24 Prime Park Way, Natick, MA 01760-1500, 1999. Third Printing Revised for Version 2 (Release 11).

[13] A. R. Conn, N. I. M. Gould, A. Sartenaer, and P. L. Toint. Convergence properties of minimization algorithms for convex constraints using a structured trust region. *SIAM Journal on Scientific and Statistical Computing*, 6(4):1059–1086, 1996.

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

[15] T. J. Dekker. Finding a zero by means of successive linear interpolation. In B. Dejon and P. Henrici, editors, *Constructive Aspects of the Fundamental Theorem of Algebra*, New York, 1969. John Wiley.

[16] Matthias Schonlau Donald R. Jones and William J. Welch. Efficient global optimization of expensive Black-Box functions. 1998.

[17] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. LINPACK User's Guide. *SIAM*, 1979.

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

[19] R. Fletcher and C. Xu. Hybrid methods for nonlinear least squares. *IMA Journal of Numerical Analysis*, 7:371–389, 1987.

[20] Roger Fletcher. *Practical Methods of Optimization*. John Wiley and Sons, New York, 2nd edition, 1987.

[21] Roger Fletcher and Sven Leyffer. Nonlinear programming without a penalty function. Technical Report NA/171, University of Dundee, 22 September 1997.

[22] V. N. Fomin, K. Holmström, and T. Fomina. Least squares and Minimax methods for inorganic chemical equilibrium analysis. Research Report 2000-2, ISSN-1404-4978, Department of Mathematics and Physics, Mälardalen University, Sweden, 2000.

[23] T. Fomina, K. Holmström, and V. B. Melas. Nonlinear parameter estimation for inorganic chemical equilibrium analysis. Research Report 2000-3, ISSN-1404-4978, Department of Mathematics and Physics, Mälardalen University, Sweden, 2000.

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

[25] C. M. Fransson, B. Lennartson, T. Wik, and K. Holmström. Multi Criteria Controller Optimization for Uncertain MIMO Systems Using Nonconvex Global Optimization. In *Proceedings of the 40th Conference on Decision and Control*, Orlando, FL, USA, December 2001.

[26] C. M. Fransson, B. Lennartson, T. Wik, K. Holmström, M. Saunders, and P.-O. Gutmann. Global Controller Optimization Using Horowitz Bounds. In *Proceedings of the 15th IFAC Conference*, Barcelona, Spain, 21th-26th July, 2002.

[27] B. S. Garbow, J. M. Boyle, J. J. Dongara, and C. B. Moler. Matrix Eigensystem Routines-EISPACK Guide Extension. In *Lecture Notes in Computer Science*. Springer Verlag, New York, 1977.

[28] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, London, 1982.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] D. Goldfarb and M. J. Todd. Linear 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.

[36] Jacek Gondzio. Presolve analysis of linear programs prior to applying an interior point method. *INFORMS Journal on Computing*, 9(1):73–91, 1997.

[37] T. Hellström and K. Holmström. Parameter Tuning in Trading Algorithms using ASTA. In Y. S. Abu-Mostafa, B. LeBaron, A. W. Lo, and A. S. Weigend, editors, *Computational Finance (CF99) – Abstracts of the Sixth International Conference, Leonard N. Stern School of Business, January 1999*, Leonard N. Stern School of Business, New York University, 1999. Department of Statistics and Operations Research.

[38] T. Hellström and K. Holmström. Parameter Tuning in Trading Algorithms using ASTA. In Y. S. Abu-Mostafa, B. LeBaron, A. W. Lo, and A. S. Weigend, editors, *Computational Finance 1999*, Cambridge, MA, 1999. MIT Press.

[39] T. Hellström and K. Holmström. Global Optimization of Costly Nonconvex Functions, with Financial Applications. *Theory of Stochastic Processes*, 7(23)(1-2):121–141, 2001.

[40] K. Holmström. New Optimization Algorithms and Software. *Theory of Stochastic Processes*, 5(21)(1-2):55–63, 1999.

[41] K. Holmström. Solving applied optimization problems using TOMLAB. In G. Osipenko, editor, *Proceedings from MATHTOOLS '99, the 2nd International Conference on Tools for Mathematical Modelling*, pages 90–98, St.Petersburg, Russia, 1999. St.Petersburg State Technical University.

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

[43] K. Holmström. The TOMLAB v2.0 Optimization Environment. In E. Dotzauer, M. Björkman, and K. Holmström, editors, *Sixth Meeting of the Nordic Section of the Mathematical Programming Society. Proceedings*, Opuscula 49, ISSN 1400-5468, Västerås, 1999. Mälardalen University, Sweden.

[44] K. Holmström. Practical Optimization with the Tomlab Environment. In T. A. Hauge, B. Lie, R. Ergon, M. D. Diez, G.-O. Kaasa, A. Dale, B. Glemmestad, and A Mjaavatten, editors, *Proceedings of the 42nd SIMS Conference*, pages 89–108, Porsgrunn, Norway, 2001. Telemark University College, Faculty of Technology.

[45] K. Holmström and M. Björkman. The TOMLAB NLPLIB Toolbox for Nonlinear Programming. *Advanced Modeling and Optimization*, 1(1):70–86, 1999.

[46] K. Holmström, M. Björkman, and E. Dotzauer. The TOMLAB OPERA Toolbox for Linear and Discrete Optimization. *Advanced Modeling and Optimization*, 1(2):1–8, 1999.

[47] K. Holmström and T. Fomina. Computer Simulation for Inorganic Chemical Equilibrium Analysis. In S.M. Ermakov, Yu. N. Kashtanov, and V.B. Melas, editors, *Proceedings of the 4th St.Petersburg Workshop on Simulation*, pages 261–266, St.Petersburg, Russia, 2001. NII Chemistry St. Peterburg University Publishers.

[48] K. Holmström, T. Fomina, and Michael Saunders. Parameter Estimation for Inorganic Chemical Equilibria by Least Squares and Minimax Models. *Optimization and Engineering*, 4, 2003. Submitted.

[49] K. Holmström and J. Petersson. A Review of the Parameter Estimation Problem of Fitting Positive Exponential Sums to Empirical Data. *Applied Mathematics and Computations*, 126(1):31–61, 2002.

[50] J. Huschens. On the use of product structure in secant methods for nonlinear least squares problems. *SIAM Journal on Optimization*, 4(1):108–129, 1994.

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

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

[53] P. Lindström. *Algorithms for Nonlinear Least Squares - Particularly Problems with Constraints*. PhD thesis, Inst. of Information Processing, University of Umeå, Sweden, 1983.

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

[55] J. R. R. A. Martins, I. M. Kroo, and J. J. Alonso. An automated method for sensitivity analysis using complex variables. In *38th Aerospace Sciences Meeting and Exhibit, January 10-13, 2000, Reno, NV*, AIAA-2000-0689, pages 1–12, 1801 Alexander Bell Drive, Suite 500, Reston, Va. 22091, 2000. American Institute of Aeronautics and Astronautics.

[56] C. B. Moler. MATLAB — An Interactive Matrix Laboratory. Technical Report 369, Department of Mathematics and Statistics, University of New Mexico, 1980.

[57] 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.

[58] 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.

[59] C. C. Paige and M. A. Saunders. Algorithm 583 LSQR: Sparse linear equations and sparse least squares. *ACM Trans. Math. Software*, 8:195–209, 1982.

[60] C. C. Paige and M. A. Saunders. LSQR. An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Software*, 8:43–71, 1982.

[61] J. Petersson. *Algorithms for Fitting Two Classes of Exponential Sums to Empirical Data*. Licentiate Thesis, ISSN 1400-5468, Opuscula ISRN HEV-BIB-OP–35–SE, Division of Optimization and Systems Theory, Royal Institute of Technology, Stockholm, Mälardalen University, Sweden, December 4, 1998.

[62] 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.

[63] J.D. Pintér. *Global Optimization in Action (Continuous and Lipschitz Optimization: Algorithms, Implementations and Applications)*. Kluwer Academic Publishers, Dordrecht / Boston / London., See http://www.wkap.nl/prod/b/0-7923-3757-3, 1996.

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

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

[66] Axel Ruhe and Per-Åke Wedin. Algorithms for Separable Nonlinear Least Squares Problems. *SIAM Review*, 22(3):318–337, 1980.

[67] A. Sartenaer. Automatic determination of an initial trust region in nonlinear programming. Technical Report 95/4, Department of Mathematics, Facultés Universitaires ND de la Paix, Bruxelles, Belgium, 1995.

[68] M. A. Saunders. Solution of sparse rectangular systems using LSQR and CRAIG. *BIT*, 35:588–604, 1995.

[69] K. Schittkowski. On the Convergence of a Sequential Quadratic Programming Method with an Augmented Lagrangian Line Search Function. Technical report, Systems Optimization laboratory, Stanford University, Stanford, CA, 1982.

[70] L. F. Shampine and H. A. Watts. Fzero, a root-solving code. Technical Report Report SC-TM-70-631, Sandia Laboratories, September 1970.

[71] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines - EISPACK Guide Lecture Notes in Computer Science*. Springer-Verlag, New York, 2nd edition, 1976.

[72] William Squire and George Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Review*, 40(1):100–112, March 1998.