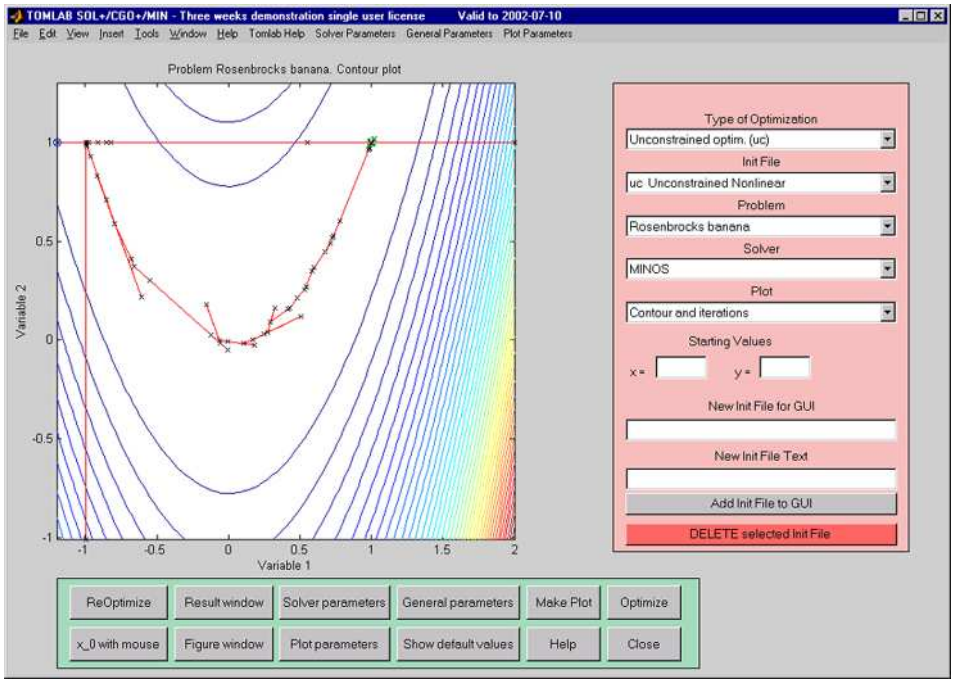
User's Guide for Tomlab v3.2.1[1]

Kenneth Holmström and Anders Göran

Tomlab Optimization

Gäddgatan 9
SE-723 49 Västerås, Sweden

September 2, 2002



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

Contents

1 Introduction to TOMLAB

1.1 What is TOMLAB?

TOMLAB is a general purpose development 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 want 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. An unexperienced user or a student might want some very easy way to solve the problem, and would like to use a menu system or a graphical user interface (GUI). Using a GUI or a menu system also makes it very easy to change parameters influencing the solution process. 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. Furthermore, it is easy to define new problems in the TOMLAB Quick format, and try to solve them using any solver. To access the user problem in the GUI or menu system, routines converting the problem into the TOMLAB Init File format and adding the problems to the GUI data base are available and simple to use. To use TOMLAB in real-time control, the efficient MEX-file interfaces calling fast Fortran solvers are of great importance.

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 two available input formats, the TOMLAB Quick Format (TQ) and the TOMLAB Init File Format (IF).

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 contains information on efficient use of the SOL (Stanford System Optimization Laboratory) solvers.

Section 10 discusses a number of special system features such as derivatives, automatic differentiation, partially separable functions and user supplied parameter information for the function computations.

Section 11 presents the Graphical User Interface (GUI). The GUI gives the user the possibility to set all kinds of solver parameters that influences the optimization process. It can also be used as a code generator, saving the status of the GUI and generating m-file code to run the current problem. There is also an option to retrieve the saved status of the GUI.

Section 12 presents the menu system, *tomMenu*. The menu system implements some, but not all of the functionality of the GUI, but can be useful when running TOMLAB on remote machines over text-only connections.

Section 13 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 called using MEX-file interfaces are not described, but documentation is available for each solver, e.g. the MINOS User's Guide [68].

Section 14 describes the *LDO* (Linear and Discrete Optimization) solvers for linear, quadratic, and discrete optimization problems grouped together in the TOMLAB LDO Toolbox.

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. An array of such structures are also returned if calling the GUI or menu system with an output variable.

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

Appendix D describes in detail how to edit TOMLAB Init Files directly.

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 [51]. The use of TOMLAB for nonlinear programming and parameter estimation is presented in [54], and the use of linear and discrete optimization is discussed in [55]. Global optimization routines are also implemented, one is described in [11].

In all these papers TOMLAB was divided into two toolboxes, the NLPLIB TB and the OPERA TB. This was impractical because of the integration of linear and mixed-integer programming in the GUI and the other menu and driver tools. The first version of the graphical user interface (GUI) is described in [21]. TOMLAB v2.0 was discussed in [52], [49]. and [50]. TOMLAB v3.2 and how to solve practical optimization problems with TOMLAB is discussed in [53].

The use of TOMLAB for costly global optimization with industrial applications is discussed in [12]; costly global optimization with financial applications in [45, 46, 47]. Applications of global optimization for robust control is presented in [30, 31]. The use of TOMLAB for exponential fitting and nonlinear parameter estimation are discussed in e.g. [58, 7, 27, 28, 56, 57].

2 The Design of TOMLAB

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. Likewise, 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.
minlp	12	Constrained mixed-integer nonlinear optimization.
sdp	13	Semi-definite programming.
miqq	14	MIQP with quadratic constraints.
exp	15	Exponential fitting problems.
nts	16	Nonlinear Time Series.

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.
sdp	13	Semi-definite optimization problems.
miqq	14	MIQP + quadratic constraints problems.
exp	15	Exponential fitting problems.
nts	16	Nonlinear time series problems.
mgh	4	More, Garbow, Hillstrom nonlinear least squares problems.
amp	3	AMPL test problems as *nl*-files.
chs*	3	Hock-Schittkowski constrained test problems.
uhs*	1	Hock-Schittkowski unconstrained test problems.
cto*	3	CUTE constrained test problems as *dll*-files.
ctl*	3	CUTE large constrained test problems as *dll*-files.
uto*	1	CUTE unconstrained test problems as *dll*-files.
utl*	1	CUTE large unconstrained test problems as *dll*-files.

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 $\diamond$*_d2c* and $\diamond$*_d2r*. Only the solver *nlpSolve* can utilize the information in $\diamond$*_d2c*.

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. The menu systems and the GUI are using the string matrix returned from the Init File for user selection of which problem to be solved. 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 with TOMLAB

A flow-sheet of the process of optimization in TOMLAB is shown in Figure 1. Normally, a single optimization problem is solved running the menu system or the Graphical User Interface (GUI). When several problems are to be solved, e.g. in algorithmic development, it is inefficient to use an interactive system. This is symbolized with the *Advanced User* box in the figure, which directly runs the *Optimization Driver*. If a problem is specified in the TOMLAB Quick format and not converted to the TOMLAB Init File format , then the GUI and menu systems

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

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

are not available and the user must either call the driver routine or call the solver directly. The direct solver call is possible when running a TOM solver, that takes the problem structure as the input. See Section 3 for a list of the TOM 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.

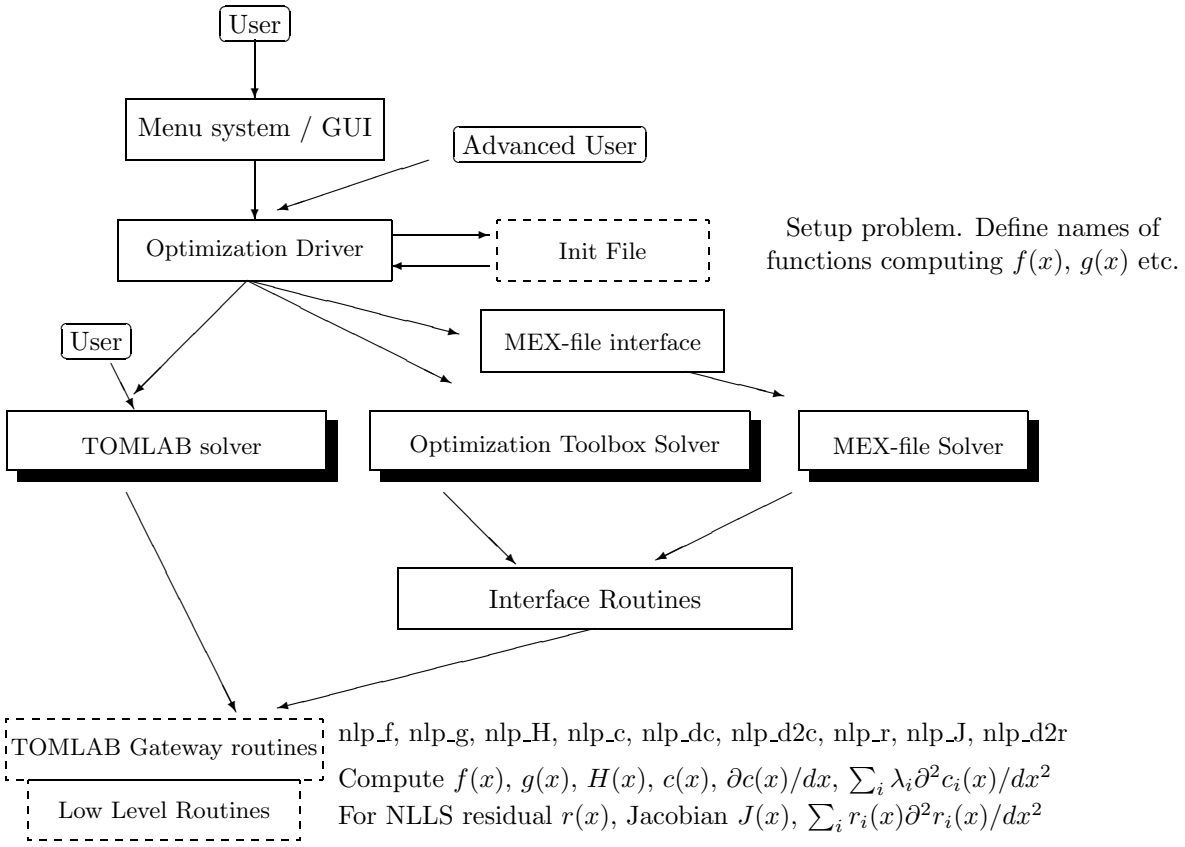


Figure 1: The process of optimization in TOMLAB.

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.USER.f*, *Prob.USER.g*, *Prob.USER.H* etc. It is the task for the Init File (the predefined Init Files all have names of the type ⋄**_prob**)) to set the names of the low level m-files. This is done by a call to the routine *mFiles* with the names as arguments. As an example, see the last part of the code of *con_prob* below.

```
...
...
Prob = mFiles(Prob,'con_f','con_g','con_H','con_c','con_dc','con_d2c');

Prob = conProbSet(Prob, Name, P, ...
          x_0, x_L, x_U, x_min, x_max, f_Low, xName, x_opt, f_opt, ...
          cName, A, b_L, b_U, c_L, c_U, HessPattern, ConsPattern,...
          pSepFunc, uP, uPName);
```

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 *mFiles*. 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.

TOMLAB is using 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.uP*, problem specific information 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:

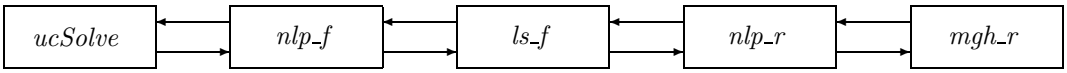


Figure 2: The chain of calls when computing the objective function value in *ucSolve* for a nonlinear least squares problem defined in *mgh_prob*, *mgh_r* and *mgh_J*.

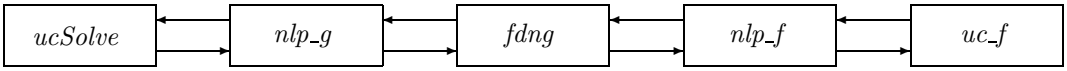


Figure 3: The chain of calls when computing the numerical approximation of the gradient in *ucSolve* for an unconstrained problem defined in *uc_prob* and *uc_f*.

```
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.uP* is described in Section 10.3.

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.

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 $\theta(x_k)$.

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

3 Optimization Problem Types and Solver Routines in TOMLAB

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 Optimization Problem Types Defined in TOMLAB

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

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

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

The TOM routine *ucSolve* includes several of the most popular search step methods for unconstrained optimization. Bound constraints are treated as described in Gill et. al. [34]. 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 are using a subspace minimization technique to handle rank problems, see Lindström [64]. The quasi-Newton algorithms also use safe guarding techniques to avoid rank problem in the updated matrix.

Another TOM 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 [18] and [77], 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

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

where $c, x, x_L, x_U \in \mathbb{R}^n$, $F \in \mathbb{R}^{n\times n}$, $A \in \mathbb{R}^{m_1 \times n}$, and $b_L, b_U \in \mathbb{R}^{m_1}$. 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 [65], 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 v3.2 *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.

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

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

where $x, x_L, x_U \in \mathbb{R}^n$, $f(x) \in \mathbb{R}$, $A \in \mathbb{R}^{m_1 \times n}$, $b_L, b_U \in \mathbb{R}^{m_1}$ and $c_L, c(x), c_U \in \mathbb{R}^{m_2}$. For general constrained nonlinear optimization a sequential quadratic programming (SQP) method by Schittkowski [79] is implemented in the TOM solver *conSolve*. *conSolve* also includes an implementation of the Han-Powell SQP method. There are also a TOM routine *nlpSolve* implementing the Filter SQP by Fletcher and Leyffer presented in [26].

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. In TOMLAB v3.2

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

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

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

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.

The TOM solver *glbSolve* implements the DIRECT algorithm [61], 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 TOM routine *ego* implements the Efficient Global Optimization (EGO) algorithm [63]. 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

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

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

The TOM solver *glcSolve* implements an extended version of the DIRECT algorithm [62], that handles problems with both nonlinear and integer constraints.

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

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

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

The TOM solver *lpSolve* implements an 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 TOM solver *DualSolve*. In TOMLAB v3.2 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 **mixed-integer programming problem** (**mip**) is defined as

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

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: $b_L(i) = b_U(i)$.

Mixed-integer programs are normally solved using the TOM routine *mipSolve* that implements a standard branch-and-bound algorithm, see Nemhauser and Wolsey [69, 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 lpSolve. *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 TOM solver is the cutting plane routine *cutplane*, using Gomory cuts. In TOMLAB Base Module v3.2, both TOM routines are using the linear programming routines in TOMLAB (*lpSolve* and *DualSolve*), to solve relaxed subproblems. In TOMLAB v3.2, *mipSolve* is using 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.

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

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

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

LSQR solves unconstrained sparse **lls** problems. *LSEI* solves the general dense problems. *WNNLS* 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

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

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

The TOM nonlinear least squares solver *clsSolve* treats problems with bound constraints in a similar way as the routine *ucSolve*. This strategy is combined with an an active-set strategy to handle linear equality and inequality constraints [10].

clsSolve includes four optimization methods for nonlinear least squares problems: the Gauss-Newton method, the Al-Baali-Fletcher [4] and the Fletcher-Xu [24] hybrid method, and the Huschens TSSM method [59]. If rank problems occur, the solver is using 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 TOM line search algorithm *LineSearch*, used by the solvers *conSolve*, *clsSolve* and *ucSolve*. It implements a modified version of an algorithm by Fletcher [25, chap. 2]. The line search algorithm uses quadratic and cubic interpolation, see Section 13.2.3.

Another TOM routine, *preSolve*, is running a presolve analysis on a system of linear equalities, linear inequalities and simple bounds. An algorithm by Gondzio [42], somewhat modified, is implemented in *preSolve*. See [10] for a more detailed presentation.

3.2 Solver Routines in TOMLAB

3.2.1 Solvers Available in all Tomlab Versions

TOMLAB includes a large set of optimization solvers. Most of them were originally developed by the Applied Optimization and Modeling group (TOM) [51]. 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 TOM optimization solvers in TOMLAB Base Module v3.2.

Function	Description	Section	Page
ucSolve	Unconstrained optimization with simple bounds on the parameters. Implements Newton, quasi-Newton and conjugate-gradient methods.	13.1.22	135
sTrustr	Constrained convex optimization of partially separable functions, using a structural trust region algorithm.	13.1.20	132
glbSolve	Box-bounded global optimization, using only function values.	13.1.7	104
glbFast	Box-bounded global optimization, using only function values. Fortran MEX implementation of *glbSolve*.	13.1.8	106
glcSolve	Global mixed-integer nonlinear programming, using no derivatives.	13.1.9	108
glcFast	Box-bounded global optimization, using only function values. Fortran MEX implementation of *glcSolve*.	13.1.10	110
glcCluster	Hybrid algorithm for constrained mixed-integer global optimization. Uses a combination of *glcFast* (DIRECT) and a clustering algorithm.	13.1.11	113
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.	13.1.1	90
conSolve	Constrained nonlinear minimization solver using two different sequential quadratic programming methods.	13.1.2	92
nlpSolve	Constrained nonlinear minimization solver using a filter SQP algorithm.	13.1.16	123
lpSolve	Linear programming using a simplex algorithm.	13.1.13	117
DualSolve	Solves a linear program with a dual feasible starting point.	13.1.4	97
qpSolve	Non-convex quadratic programming.	13.1.17	125
mipSolve	Mixed-integer programming using a branch-and-bound algorithm.	13.1.15	121
cutplane	Mixed-integer programming using a cutting plane algorithm.	13.1.3	95
infSolve	Constrained minimax optimization. Reformulates problem and calls any suitable nonlinear solver.	13.1.12	115
slsSolve	Sparse constrained nonlinear least squares. Reformulates problem and calls any suitable sparse nonlinear solver.	13.1.19	130
L1Solve	Constrained L1 optimization. Reformulates problem and calls any suitable nonlinear solver.	13.1.14	119

Additional Fortran solvers in TOMLAB v3.2 are listed in Table 6. They are called using a set of MEX-file interfaces developed in TOMLAB.

Another set of Fortran solvers were developed by the Stanford Optimization Laboratory (SOL). Table 7 lists the SOL optimization solvers available in TOMLAB v3.2, 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.

Table 6: Additional solvers in TOMLAB Base Module v3.2.

Function	Description	Reference	Page
DFZERO	Finding a zero to f(x) in an interval, x is one-dimensional.	[80, 19]	
LSQR	Sparse linear least squares.	[71, 70, 78]	
LSEI	Dense constrained linear least squares		
WNNLS	Nonnegative constrained linear least squares		
QLD	Convex quadratic programming		

Table 7: The SOL optimization solvers in TOMLAB /MINOS v3.2.

Function	Description	Reference	Page
MINOS 5.5	Sparse linear and nonlinear programming with linear and nonlinear constraints.	[68]	
LP-MINOS	A special version of the *MINOS 5.5* MEX-file interface for sparse linear programming.	[68]	
QP-MINOS	A special version of the *MINOS 5.5* MEX-file interface for sparse quadratic programming.	[68]	
LPOPT 1.0-10	Dense linear programming.	[36]	
QPOPT 1.0-10	Non-convex quadratic programming with dense constraint matrix and sparse or dense quadratic matrix.	[36]	

3.2.2 Solvers Available in the TOMLAB /SOL Toolbox

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 8.

Table 8: 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.	[40]	
SNOPT 6.1-1	Large, sparse linear and nonlinear programming with linear and nonlinear constraints.	[39, 37]	
SQOPT 6.1-1	Sparse convex quadratic programming.	[38]	
NLSSOL 5.0-2	Constrained nonlinear least squares. NLSSOL is based on NPSOL. No reference except for general NPSOL reference.	[40]	
LSSOL 1.05-4	Dense linear and quadratic programs (convex), and constrained linear least squares problems.	[35]	

3.2.3 Solvers Available in the TOMLAB/CGO Toolbox

The add-on toolbox Tomlab /CGO solves costly global optimization problems. The solvers are listed in Table 9. 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 9: Additional solvers in Tomlab /CGO.

Function	Description	Reference	Page
rbfSolve	Costly constrained box-bounded optimization using a RBF algorithm.	[12]	127
ego	Costly constrained box-bounded optimization using the Efficient Global Optimization (EGO) algorithm.	[63]	99

3.2.4 Finding Available Solvers

To get a list of all available solvers, including Fortran, C and Matlab Optimization Toolbox solvers, for a certain *solvType* the user just calls the routine *SolverList* with *solvType* as argument. *solvType* should either be a string ('uc', 'con' etc.) or the corresponding *solvType* number, see Table 1. As an example, if wanting a list of all available solvers of *solvType* **con**, then

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

gives the output

```
--------
Solver
--------
nlpSolve
conSolve
sTrustr
constr
minos
npsol
npopt
snopt
fmincon
```

and if *SolverList* is called with no given argument then all available solvers for all different *solvType* are printed. The second output arguments gives the *solvType* for each solver. Note that solvers for a more general problem type may be used to solve the problem. In Table 10 an attempt has been made to classify these relations.

Table 10: 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 User Problems in the TOMLAB Quick or Init File format

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. Two formats are defined, the TOMLAB Quick format (TQ format) and the Init File format (IF format). The TQ 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.

The definition of an advanced general graphical user interface (GUI) and a similar menu system demanded a more complicated format. The solution is the IF format, where groups of problems are collected into sets, each set having an initialization file. Besides defining the problem, a list of all problems in the set is also generated by the initialization file.

In this section follows a more detailed description of the two formats.

4.1 TOMLAB Quick (TQ) Format for User Problems

The TQ 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 TQ format has four steps:

1. Define the problem structure, often called Prob.
2. Define any user supplied function routines.
3. Call the solver or the solver driver routine.
4. 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 11.

Table 11: Routines to create a problem structure in the TQ format.

Matlab call	**probTypes**	**Type of optimization problem**
Prob = qpAssign(...)	2	Quadratic programming.
Prob = conAssign(...)	1,3	Unconstrained and constrained nonlinear optimization.
Prob = clsAssign(...)	4,5,6	Unconstrained and constrained nonlinear least squares.
Prob = mipAssign(...)	7	Mixed-Integer programming.
Prob = lpAssign(...)	8	Linear programming.
Prob = glcAssign(...)	9,10	Box-bounded or mixed-integer constrained global programming.
Prob = probAssign(...)	1,3-6,9-10	General routine, but does not include all possible options.

Step 2 is a call to mFiles.m giving the function names as strings:

```
Prob = mFiles(Prob, ...);
```

This step is only needed when using *probAssign*, the other routines calls mFiles directly.

Step 3, the solver call, is either a direct to the call, e.g. 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, also the TOMLAB Init File format format described in Section 4.2. It may also print the names of the available solvers.

Step 4 could be a call to PrintResult.m:

```
PrintResult(Result);
```

The 4th step could be included in Step 3 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 TQ format: *conDemo.m*, *ucDemo.m*, *qpDemo.m*, *lsDemo.m*, *lpDemo.m*, *mipDemo.m*, *glbDemo.m* and *glcDemo.m*.

4.2 TOMLAB Init File (IF) Format for User Problems

In the IF format one initialization file is defined for each set of user problems. The set could consist of only one problem.

The initialization file should perform two tasks:

- If the input problem number is empty, return a string matrix with the i:th row defining the name of the i:th problem defined in the file.
- If the input problem number is nonempty, return the TOMLAB problem structure defining the corresponding problem number.

To write such a basic routine is very simple. The user could write such a routine and it will function well.

However, some thoughts make it clear that additional functionality is nice to have in such a routine Adding an integer *ask*, to tell if the initialization routine should ask questions, makes the routine either silent, or optionally asking for problem dependent parameters. TOMLAB has a query routine predefined, that is suitable to use: *askparam.m*. Examples of the use of *askparam* is implemented in several of the predefined Init Files in the *testprob* directory: *uc_prob.m*, *qp_prob.m*, *mgh_prob.m*, *ls_prob.m*, *cls_prob.m* and *glb_prob.m*. Most often a problem dimension is asked for, or a certain data series. Sometimes a parameter value.

Adding the problem structure as input makes it possible to override the default parameters, setting some fields beforehand.

The syntax of the initialization file (Init File) is the following (assuming the name of the function is *new_prob.m*):

```
[probList, Prob] = new_prob(P, ask, Prob)
```

where

P	The problem number, either empty, or an integer.
ask	An integer defining if questions should be asked in the Init File. If $ask >= 1$ ask questions in the Init File. If $ask = 0$ use default values. If $ask < 0$ use values defined in Prob.uP if defined or otherwise use defaults. The last options makes it possible to change values before the call. If isempty(ask), then if length(Prob.uP) > 0, $ask = -1$, else $ask = 0$.
Prob	As input, the problem structure is either empty, or some or all of the fields are defined, and overrides the default values in the Init File.
probList	A string matrix, always returned.
Prob	As output, a full definition of the problem structure is returned if a valid problem number P is given as input.

If a group of problems have been defined in the TOMLAB Init File format it is easy to retrieve a problem structure similar to the TOMLAB Quick format format for any of these problems. The general call is

```
Prob = probInit(probFile, probNumber, ask, Prob)
```

where

probFile	Name of the Init File, without file extension.
probNumber	The problem number, an integer.
ask	An integer, defined exactly as in the call to new_prob.m above.
Prob	As input, the problem structure is either empty, or some or all of the fields are defined, and overrides the default values in the Init File.

When a problem is available as one of the problems defined in the TOMLAB Init File format, i.e. as one problem in an Init File, there are four ways to proceed to solve the problem

- Solve the problem using the TOMLAB GUI, *tomGUI*.

  ```
  tomGUI;
  ```

  or

  ```
  global ResultGUI
  tomGUI;  % After the run results are available in ResultGUI
  ...      % Postprocess ResultGUI
  ```

  Only the last problem solved is available in ResultGUI. All solver parameters and other parameters influencing the optimization are possible to change before the call to a suitable solver. Note the code generation facility in *tomGUI*, which creates a m-file and mat-file that solves identically the same problem from the command line. This m-file may, for example, be further extended to solve sequences of problem. This concept is further explained on page 84.

- Solve the problem using the TOMLAB menu program, *tomMenu*. Some, but not all, solver parameters and other parameters influencing the optimization are possible to change before the call to a suitable solver.

  ```
  tomMenu;
  ```

  or

  ```
  Result = tomMenu;  % After the run results are available in Result
  ...      % Postprocess Result.
  ```

  If more than one problem is solved *Result(1)* gives the results for the first problem, *Result(2)* gives the results for the second problem, and so on.

- Pick up the problem structure using *probInit* and make any changes to the problem structure, e.g.

  ```
  probFile   = 'con_prob';
  probNumber = 10;
  ask        = [];
  Prob       = [];
  Prob = probInit(probFile, probNumber, ask, Prob)
  ... % Make changes in the Prob structure
  ```

  The fields in the structure are described in the tables in Appendix A. Then either directly call a solver

  ```
  Result = conSolve(Prob);
  ```

  or call the multi-solver driver routine *tomRun*

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

If increasing the print level to 1, 2 or 3 in the call to the driver routine the call is

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

- Define a *Prob* structure with only the fields you want to change, e.g

```
Prob.optParam.MaxIter = 1000;  % Increase maximal numbers of iterations
Prob.x_0 = [0 1 3]';           % Change initial value of x
% Tell tomRun to use conSolve to solve problem 10 from con_prob.
% Print Level 3 in the call to PrintResult
Result = tomRun('conSolve', 'con_prob', 10, Prob, [], 3);
```

It is very easy to try another solver in TOMLAB, e.g. to see what *nlpSolve* does on this problem just add one line

```
Result = tomRun('nlpSolve', 'con_prob', 10, Prob, [], 3);
```

Note that when solving a sequence of similar problems, the best way is to pick up the problem structure once using *probInit*, and then make a loop, do the changes in the structure, and solve the problem for each change. An example solving problem 10 in *con_prob.m* one hundred times for different starting values in the interval $[100, 100]$

```
probFile   = 'con_prob';
probNumber = 10;
ask        = [];
Prob       = [];
Prob = probInit(probFile, probNumber, ask, Prob)
for i=1:100
    Prob.x_0 = -100 + 200*rand(3,1);
    Result = tomRun('conSolve', Prob, [], 1);
end
```

For each type of the optimization problem there is at least one Init File. All the predefined Init Files with test problems are available in the *testprob* directory. See also the different demonstration files in the *examples* directory that also includes a few examples of how to apply the IF format. In Appendix D detailed descriptions are given on how to copy an Init File into a new Init File, and also adding new problems.

4.3 Create an Init File going from TQ to IF format

If dropping the wish to ask user questions, then a simpler type of initialization file may be created. Using the routines *newInitFile*, *addProb* and *makeInitFile*, it is easy to collect a group of problem structures Prob into a set and put them into the TOMLAB Init File format. The Prob structures are saved in a Matlab mat-file having the same name as the Init File. Having defined a proper file in the TOMLAB Init File format, it is added to the GUI calling the routine *AddProblemFile*.

See the file *makeInitFileDemo* for an example on how to create a new Init File and get it into the GUI. It is recommended that the problems added into the Init File are of the same problem type (*probType*).

If only up to five problem structures are collected to the Init File, then only one call is needed

```
% Assume four structures P1,P3,P3 and P4 are created in TQ format
...

makeInitFile('new_prob',P1,P2,P3,P4);
```

If doing this type of call several times during a session global arrays used by *makeInitFile* must be cleared, and a call to *newInitFile* is needed

```
newInitFile;

% Assume four structures P1,P3,P3 and P4 are created in TQ format
...

makeInitFile('new_prob',P1,P2,P3,P4);
```

It might be easier to collect the problems in a loop. Then *addProb* is usable. Assuming that *conAssign* is used to create the basic problem, the following example shows the principles of making the new Init File:

```
% Make the assignments and calls for the first problem
Prob = conAssign ( .....)
...
newInitFile(Prob);
for i=2:10  % Define problem 2, 3, up to 10

    % Make the assignments necessary for this particular problem i
    Prob.P = i;
    ...

    % Then add the structure when it is complete
    addProb(Prob);
end
makeInitFile('new_prob');  % Create the Init File and save problems
```

Note that for LP, QP and MIP problems there is another alternative to create a file in the Init File format. The routines *lpAssign*, *qpAssign* and *mipAssign* have an option to create Init Files with an arbitrarily number of problems. See the help for the input arguments *setupFile* and *nProblem* in these routines. With this strategy the problems are saved more efficiently with regards to space. One binary mat-file is created for each problem in the Init File. If the number of problems are large, this alternative may be preferable. The Init File creation is more complicated, and described in detail with examples in Section 5.

4.4 Adding an Init File to the GUI Data Base

In order to get a file created in the TOMLAB Init File format accessible to the GUI and menu system, it must be added to the GUI init file data base. The database is stored in the file *TomlabProblem.mat*.

The command *AddProblemFile* adds a new Init File to the data base. The name of the file, a menu text shortly describing the content of the Init File, the problem type, and one additional number *mexType* must be given. *mexType* is always zero for Matlab files. The following example adds the user file *new_prob*:

```
% Assume the name of the problem to add is 'new_prob', of type 'con'

AddProblemFile('new_prob','New user created problems','con',0);
```

Note that in order to run the GUI and the menu system with the newly added problems, they must reside in the current directory, or somewhere in the Matlab PATH. The added Init File is always put first, as the default file, among the files having the same problem type.

If the GUI can not find the files it will crash. Then you must either put the problems in the Matlab PATH, or delete the problems from the GUI data base. A problem is deleted by a call to DeleteProblemFile. You must also know the problem type of the file.

```
% Assume the name of the problem to delete is 'new_prob', of type 'con'

DeleteProblemFile('new_prob','con');
```

If there are many problems to be deleted from the GUI data base, or if other problems occur, there is a command to restore everything back to the original distribution of TOMLAB:

```
CreateTomProb;
```

The user must press ENTER when a question *Overwrite??? (ctrl-c to break)* appears.

Another way to get an Init File into the GUI, and deleted from the GUI, is to use the GUI itself. By filling in the file name in the field *New Init File for GUI*, and a menu text in *New Init File Text*, and after that pressing the button *Add Init File to GUI*, the problem will be added to the GUI problem data base. Another RED button *Delete selected Init File*, will delete the currently selected Init File, if is not part of the original TOMLAB distribution.

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 v2.1. 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. The diary command is used to save screen output and the resulting files are stored with the extension *out*, and having the same name as the test file.

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

$$\begin{aligned} \min_x \quad & f(x) = c^T x \\ s/t \quad & \begin{array}{ccccc} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U \end{array} \end{aligned} \tag{10}$$

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: $b_L(i) = b_U(i)$.

Linear programs are normally solved using the TOM routine *lpSolve* that implements a revised simplex algorithm. It is also possible to use the active-set QP method routine *qpSolve* or a general nonlinear solver like *nlpSolve* or *conSolve*. In TOMLAB v3.2, the SOL routines *MINOS* or *QPOPT* are suitable for linear programming problems. To choose alternative solvers, the multi-driver routine *tomRun* is called.

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

$$\begin{array}{lrcc} \min\limits_{x_1,x_2} & f(x_1,x_2) & = & -7x_1 - 5x_2 \\ s/t & x_1 + 2x_2 & \leq & 6 \\ & 4x_1 + x_2 & \leq & 12 \\ & x_1, x_2 & \geq & 0 \end{array} \tag{11}$$

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 Quick format:

File: tomlab/usersguide/lpTest1.m

```
lpExample;

Prob   = lpAssign(c, A, b_L, b_U, x_L, x_U, x_0, 'lpExample');
Result = lpSolve(Prob);

PrintResult(Result);
```

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 *lpSolve* 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+/MIN - Three weeks demonstration single user license         Valid to 2002-07-10
================================================================================
Problem: No Init File    -  1: lpExample        f_k     -26.571428571428569000
                                              f(x_0)      0.000000000000000000

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

FuncEv    3 GradEv    0 Iter    3
Starting vector x:
x_0:   0.000000   0.000000
Optimal vector x:
x_k:   2.571429   1.714286
Lagrange multipliers v. Vector length 4:
v_k:  1.776357e-015  1.776357e-015 -4.549052e+000 -4.453845e+000
Diff x-x0:
      2.571429e+000  1.714286e+000
Gradient g_k:
g_k: -1.019533e+001 -1.029884e+001
Reduced gradient gPr: :
gPr:  5.329071e-015  0.000000e+000

=== * * * ================================================= * * *
```

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

```
Result = qpSolve(Prob);
PrintResult(Result)
```

Even a nonlinear solver may be used.

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

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

5.1.2 Several Linear Programs

If the user wants to solve more than one LP, or maybe the same LP but under different conditions, then it is possible to define the problems in the TOMLAB Init File format directly. The *lpAssign* has additional functionality and may create an Init File as an option. The same applies for the files *qpAssign* and *mipAssign* for quadratic and mixed-integer programs. The file is then easily added to the GUI data base, and accessible from the GUI and menu system.

Using the same example (11) to illustrate this format gives the Matlab statements

File: tomlab/usersguide/lpTest2.m

```
lpExample;

if exist('lptest') % Remove lptest if it previously exists
   d=which('lptest');
   delete(d);
end

lpAssign(c, A, b_L, b_U, x_L, x_U, x_0, 'lpExample', ...
        'lptest', 1, [], x_min, x_max);

AddProblemFile('lptest','Users Guide LP test problems','lp');

tomRun('lpSolve','lptest',1);
```

In this example more parameters are used in the call to *lpAssign*, which tells *lpAssign* to define a new TOMLAB problem Init File called *lptest*. The last two extra parameters *x_min* and *x_max* defines initial plotting axis and are needed when the user wants to see plots in the GUI and menu programs. Otherwise these parameters are not needed. The default is to use the lower and upper bounds on the variables, if they have finite values.

The *lptest* problem file is included in the GUI data base by the call to *AddProblemFile* and is furthermore set as the default file for LP problems. Calling *tomRun* to solve the first problem in *lptest* will give exactly the same output as in the first example in Section 5.1.1.

In the call to *lpAssign* the parameter after the file name *lptest*, called *nProblem*, controls which of two types of definition files are created (See *help lpAssign*). Setting this parameter as empty or one, as in this case, defines a program structure in the file *lptest* in which the user easily can insert more test problems. Note the comments in the created file, which guides the user in how to define a new problem. There are two places to edit. The name of the new problem must be added to the *probList* string matrix definition on row 17-21, and then the actual problem definition from row 61 and afterwards. The new problem definition can be loaded by execution of a script, by reading from a stored file or the user can explicitly write the Matlab statements to define the problem in the file *lptest.m*. For more information on how to edit an Init File, see the Section D.1. The other type of LP definition file created by *lpAssign* is discussed in detail in Section 5.1.3.

When doing this automatic Init File generation *lpAssign* stores the problem in a mat-file with a name combined by three items: the name of the problem file (*lptest*), the string '_P' and the problem number given as the input parameter next after the problem file name. In this case *lpAssign* defines the file *lptest_P1.mat*.

5.1.3 Large Sets of Linear Programs

It is easy to create an Init File for a large set of test problems. This feature is illustrated by running a test where the test problem is disturbed by random noise. The vector c in the objective function are disturbed, and the new problems are defined and stored in the TOMLAB Init File format. To avoid too much output restrict the large number of test problems to be three.

File: tomlab/usersguide/lpTest3.m

```
LargeNumber=3;
```

```
lpExample;

n=length(c);

if exist('lplarge')      % Remove lplarge if it previously exists
   d=which('lplarge');
   delete(d);
end

for i=1:LargeNumber
    cNew =c + 0.1*(rand(n,1)-0.5); % Add random disturbances to vector c

    if i==1
       % lpAssign defines lplarge.m for LargeNumber testproblems
       % and stores the first problem in lplarge_P1.mat
       k=[1,LargeNumber];
    else
       k=i; % lpAssign stores the ith problem in the lplarge_Pi.mat problem file
    end

    lpAssign(cNew, A, b_L, b_U, x_L, x_U, [], Name,'lplarge', k);
end


% Define lplarge as the default file for LP problems in TOMLAB.

AddProblemFile('lplarge','Large set of randomized LP problems','lp');

runtest('lpSolve',0,'lplarge',1:LargeNumber,0,0,1);
```

Each problem gets a unique name. In the first iteration, $i = 1$, *lpAssign* defines an Init File with three test problems, and defines the first test problem, stored in *lplarge_P1.mat*. In the other iterations *lpAssign* just defines the other mat-files. All together three mat-files are defined: *lplarge_P1.mat*, *lplarge_P2.mat* and *lplarge_P3.mat*.

AddProblemFile adds the new *lplarge* problem as the default LP test problem in TOMLAB. The *runtest* test program utility runs the selected problems, in this case all three defined. The second zero argument is used if the actual solver has several algorithmic options. In this case the zero refers to the default option in *lpSolve*, to use the minimum cost rule as the variable selection rule. The last arguments are used to lower the default output and avoid a pause statement after each problem is solved. The results are shown in the following file listing.

File: tomlab/usersguide/lpTest3.out

```
Solver: lpSolve. Algorithm 0
===== * * * ========================================================== * * *
TOMLAB SOL+/CGO+/MIN - Three weeks demonstration single user license         Valid to 2002-07-10
================================================================================
Problem: lplarge -  1: lptest - 1              f_k     -26.501771581492278000

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

FuncEv    3 GradEv    0 Iter    3
CPU time: 0.016000 sec. Elapsed time: 0.016000 sec.
===== * * * ========================================================== * * *
TOMLAB SOL+/CGO+/MIN - Three weeks demonstration single user license         Valid to 2002-07-10
```

```
================================================================================
Problem: lplarge -  2: lptest - 2              f_k     -26.546357769596234000

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

FuncEv    3 GradEv    0 Iter    3
CPU time: 0.016000 sec. Elapsed time: 0.015000 sec.
===== * * * ========================================================== * * *
TOMLAB SOL+/CGO+/MIN - Three weeks demonstration single user license         Valid to 2002-07-10
================================================================================
Problem: lplarge -  3: lptest - 3              f_k     -26.425877951614162000

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

FuncEv    3 GradEv    0 Iter    3
CPU time: 0.015000 sec. Elapsed time: 0.016000 sec.
```

5.1.4 More on Solving Linear Programs

When the problem is defined in the TOMLAB Init File format, it is then possible to run the graphical user interface *tomGUI*, the menu program *tomMenu*, or the multi-solver driver routine *tomRun* with the necessary arguments. To call the menu system, either type *Result = tomMenu;* or just *tomMenu;* at the Matlab prompt, choose *Linear Programming* and the main menu in Figure 4 will be displayed.

Pushing the *Choice of problem Init File and Problem* button will display the menu *Choice of problem Init File* button in Figure 5.

Selecting the menu *lp Linear Programming* displays the standard menu in Figure 6.

Note that if the test problems described in this chapter has been run, there are two more entries in Figure 5, *lp Large set of randomized LP problems* and *lp Users Guide LP test problems*. If clicking on the button *lp Users Guide LP test problems* the single problem *lpExample* will be selected without any further questions. If clicking on the button *lp Large set of randomized LP problems* the menu in Figure 7 is shown and a selection of one of the three different problems defined is then possible.

After selecting the problem, the menu system returns back to the main LP menu. Selection of optimization solver is done after pushing the *Solver* button. Then pushing the *Solver algorithm* button a particular algorithm choice is possible for some of the solvers. Others have only one choice, and the menu system directly returns, after writing a text in the command window.

The default settings of the optimization parameters are changed selecting *Optimization Parameter Menu* in the main menu. Pushing the *Optimize* button will call the driver routine *tomRun* with the solver selected (or the default one) and the result will be displayed in the Matlab command window. Finally, choose *End* and the menu will disappear.

Instead of using the menu system, the problem could be solved by a direct call to *tomRun* from the Matlab prompt or as a command in an m-file. The most straightforward way of doing it (when the problem is defined in *lpnew_prob.m*) is to give the following call from the Matlab prompt:

```
probNumber = 13; % Assume problem 13 is defined in lpnew_prob
Result = tomRun('lpSolve','lpnew_prob', probNumber);
```

Some other possibilities: Assume that a solution of a problem with the following requirements are wanted:

- Start in the point $(1, 1)$.
- No output printed, neither in the driver routine nor in the solver.

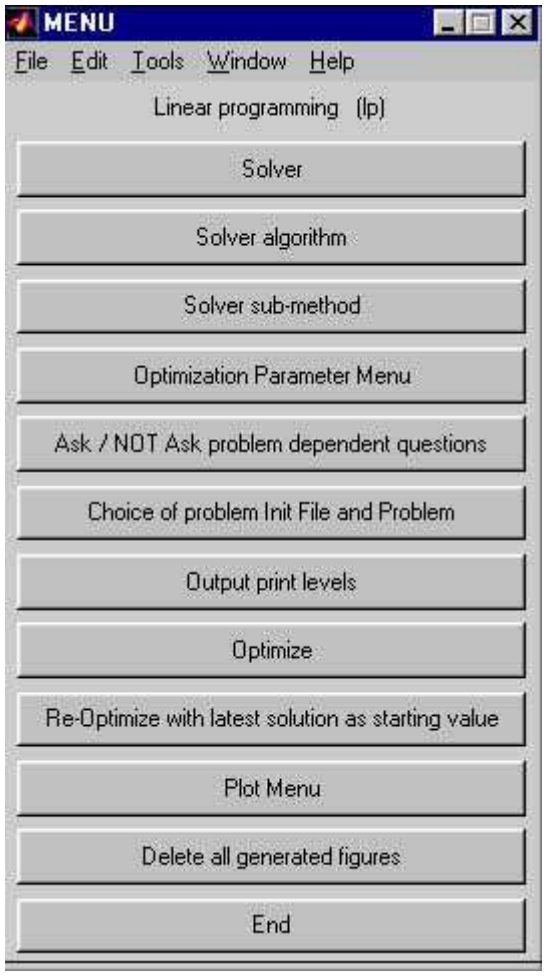


Figure 4: The main menu for Linear Programming in *tomMenu*.

- Use MathWorks Optimization TB v1.5 solver *lp*.

Then the call to *tomRun* should be:

```
Prob           = probInit('lpnew_prob',13); % Use Init File to define Prob
PriLev         = 0;
Prob.x_0       = [1;1];
Prob.PriLevOpt = 0;
Result         = tomRun('lp', Prob, [], PriLev);
```

To have the result of the optimization displayed call the routine *PrintResult*:

```
PrintResult(Result);
```

The following example shows how to call *tomRun* to solve the first problem in *ownlp_prob.m*. Assume the same requirements as the previous problem, but increase print level to get printing of results

```
probFile             = 'ownlp_prob';
Prob                 = probInit(probFile,1);
PriLev               = 2;
Prob.x_0             = [1;1];
Prob.optParam.PriLev = 0;
Result               = tomRun('lp', Prob, [], PriLev);
```

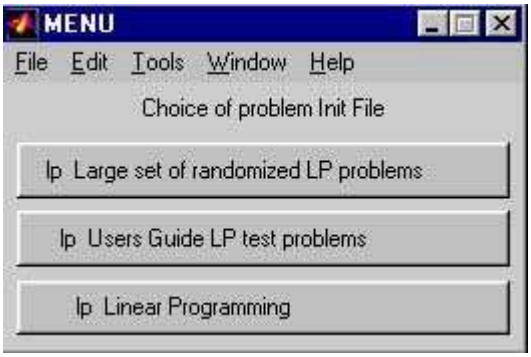


Figure 5: The Init File choice menu in *tomMenu*.

Figure 6: The problem choice menu for Linear Programming in *tomMenu*.

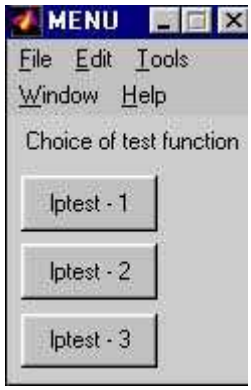


Figure 7: The problem choice menu for the three test problems defined in the User's Guide.

5.2 Quadratic Programming Problems

The general formulation in TOMLAB for a quadratic programming problem is

$$\begin{array}{lll} \min\limits_{x} & f(x) = \frac{1}{2}x^T F x + c^T x & \\ s/t & \begin{array}{ccccc} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U \end{array} & \end{array} \tag{12}$$

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.

Quadratic programs are normally solved with a standard active-set method implemented in *qpSolve*, which 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. It is also possible to choose a general constrained solver or in v3.0 the SOL routine *QPOPT* called using a MEX-file interface. To choose alternative solvers, the multi-driver routine *tomRun* is called.

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

$$\begin{array}{ll} \min\limits_{x} & f(x) = 4x_1^2 + x_1x_2 + 4x_2^2 + 3x_1 - 4x_2 \\ s/t & x_1 + x_2 \leq 5 \\ & x_1 - x_2 = 0 \\ & x_1 \geq 0 \\ & x_2 \geq 0, \end{array} \tag{13}$$

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

File: tomlab/usersguide/qpExample.m

```
Name  = 'QP Example';  % File qpExample.m
F     = [ 8   2       % Matrix F in 1/2 * x' * F * x + c' * x
          2   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 Quick format:

File: tomlab/usersguide/qpTest1.m

```
qpExample;

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

Result = qpSolve(Prob);

PrintResult(Result);
```

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

File: tomlab/usersguide/qpTest1.out

```
===== * * * ========================================================== * * *
TOMLAB SOL+/CGO+/MIN - Three weeks demonstration single user license         Valid to 2002-07-10
================================================================================
Problem: No Init File    -  1: qpExample        f_k      -0.024999999999999994
                                              f(x_0)      0.000000000000000000

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

Iter    4
Starting vector x:
x_0:   0.000000   0.000000
Optimal vector x:
x_k:   0.050000   0.050000
Lagrange multipliers v. Vector length 4:
v_k:  0.000000e+000  0.000000e+000  0.000000e+000  3.500000e+000
Diff x-x0:
      5.000000e-002  5.000000e-002
Gradient g_k:
g_k:  3.500000e+000 -3.500000e+000
Reduced gradient gPr: :
gPr:  2.220446e-015 -1.332268e-015
Eigenvalues of Hessian at x_k
eig:   6.000000  10.000000

=== * * * ================================================ * * *
```

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 *lpSolve* 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 $x_0 = (0, 0)^T$.

5.2.2 Several Quadratic Programs

If the user wants to solve more than one QP, or maybe the same QP but under different conditions, then it is possible to define the problems in the TOMLAB Init File format directly. The *qpAssign* has additional functionality and may create an Init File as an option. The file is then easily added to the GUI data base, and accessible from the GUI and menu system.

Using the same example (13) to illustrate this feature gives

File: tomlab/usersguide/qpTest2.m

```
qpExample;

if exist('qptest.m')   % Remove qptest if it previously exists
   d=which('qptest');
   delete(d);
end
```

```
qpAssign(F, c, A, b_L, b_U, x_L, x_U, x_0, 'qpExample', ...
        'qptest',1,[],x_min,x_max);

AddProblemFile('qptest','Users Guide QP test problems','qp');

tomRun('qpSolve','qptest',1);
```

In this example more parameters are used in the call to *qpAssign*, which tells *qpAssign* to define a new TOMLAB Init File called *qptest*. The last two extra parameters *x_min* and *x_max* defines initial plotting axis and are efficient when the user wants to see plots in the GUI and menu programs. Otherwise these parameters are not needed. The default is to use the lower and upper bounds on the variables, if they have finite values.

The *qptest* problem file is included as a test file by the call to *AddProblemFile* and further set as the default file for QP problems. Calling *tomRun* to solve the first problem in *qptest* will give exactly the same output as in the first example in Section 5.2.1.

In the call to *qpAssign* the parameter after the file name *qptest*, called *nProblem*, controls which of two types of definition files are created (See *help qpAssign*). Setting this parameter as empty or one, as in this case, defines an open structure in *qptest* where the user easily can insert more test problems. Note the comments in the created file, which guides the user in how to define a new problem. There are two places to edit. The name of the new problem must be added to the *probList* string matrix definition on row 17-21, and then the actual problem definition from row 61 and afterwards. The new problem definition can be loaded by execution of a script, by reading from a stored file or the user can explicitly write the Matlab statements to define the problem in the file *qptest.m*. For more information on how to edit a QP Init File, see the Section D.2. The other type of QP definition file created by *qpAssign* is discussed in detail in Section 5.2.3.

When doing this automatic Init File generation *qpAssign* stores the problem in a mat-file with a name combined by three items, the name of the problem file (*qptest*), the string '_P' and the problem number given as the input parameter next after the problem file name. In this case *qpAssign* defines the file *qptest_P1.mat*.

5.2.3 Large Sets of Quadratic Programs

It is easy to create an Init File for a large set of test problems. This feature is illustrated by running a test where the test problem is disturbed by random noise. The matrix F and the vector c in the objective function are disturbed, and the new problems are defined and stored in the TOMLAB Init File format. To avoid too much output restrict the large number of test problems to be three.

File: tomlab/usersguide/qpTest3.m

```
LargeNumber=3;

qpExample;

n=length(c);

if exist('qplarge')   % Remove qplarge if it previously exists
   d=which('qplarge');
   delete(d);
end

for i=1:LargeNumber
    cNew =c + 0.1*(rand(n,1)-0.5);  % Generate random disturbances to vector c
    FNew =F + 0.05*(rand(n,n)-0.5); % Generate random disturbances to matrix F
    FNew =(F + F')/2;               % Make FNew symmetric

    if i==1
       % qpAssign defines qplarge.m for LargeNumber testproblems
       % and stores the first problem in qplarge_P1.mat
```

```
        k=[1,LargeNumber];
    else
        k=i; % qpAssign stores the ith problem in the qplarge_Pi.mat problem file
    end

    qpAssign(FNew, cNew, A, b_L, b_U, x_L, x_U, [], Name,'qplarge', k);
end

% Define qplarge as the default file for QP problems in TOMLAB.

AddProblemFile('qplarge','Large set of randomized QP problems','qp');

runtest('qpSolve',0,'qplarge',1:LargeNumber,0,0,1);
```

Each problem gets a unique name. In the first iteration, $i = 1$, *qpAssign* defines an Init File with three test problems, and defines the first test problem, stored in *qplarge_P1.mat*. In the other iterations *qpAssign* just defines the other mat-files. All together three mat-files are defined: *qplarge_P1.mat*, *qplarge_P2.mat* and *qplarge_P3.mat*.

AddProblemFile adds the new *qplarge* problem as the default QP test problem in TOMLAB. The *runtest* test program utility runs the selected problems, in this case all three defined. The second zero argument is used if the actual solver has several algorithmic options. In this case the zero refers to the default option, and the only option, in *qpSolve*. The last arguments are used to lower the default output and avoid a pause statement after each problem is solved. The results are shown in the following file listing.

File: tomlab/usersguide/qpTest3.out

```
Solver: qpSolve. Algorithm 0
===== * * * ========================================================== * * *
TOMLAB SOL+/CGO+/MIN - Three weeks demonstration single user license         Valid to 2002-07-10
================================================================================
Problem: qplarge -  1: QP Example - 1           f_k      -0.027694057239663620

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

FuncEv    4 GradEv    4 ConstrEv    4 Iter    4
CPU time: 0.031000 sec. Elapsed time: 0.031000 sec.
===== * * * ========================================================== * * *
TOMLAB SOL+/CGO+/MIN - Three weeks demonstration single user license         Valid to 2002-07-10
================================================================================
Problem: qplarge -  2: QP Example - 2           f_k      -0.021808805588788130

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

FuncEv    4 GradEv    4 ConstrEv    4 Iter    4
CPU time: 0.016000 sec. Elapsed time: 0.015000 sec.
===== * * * ========================================================== * * *
TOMLAB SOL+/CGO+/MIN - Three weeks demonstration single user license         Valid to 2002-07-10
================================================================================
Problem: qplarge -  3: QP Example - 3           f_k      -0.023503493112249588

Solver: qpSolve.  EXIT=0.  INFORM=1.
```

```
Active set strategy
Optimal point found
First order multipliers >= 0

FuncEv    4 GradEv    4 ConstrEv    4 Iter    4
CPU time: 0.016000 sec. Elapsed time: 0.015000 sec.
```

5.2.4 Another Direct Approach to a QP Solution

The following example shows yet another way to define and solve the quadratic programming problem (13) by a direct call to the routine *qpSolve*. The approach is define a default *Prob* structure calling *ProbDef*, and then just insert values into the fields.

```
Prob      = ProbDef;
Prob.QP.F = [ 8   2      % Hessian.
              2   8 ];
Prob.QP.c = [ 3  -4 ]';  % Constant vector.
Prob.x_L  = [ 0   0 ]';  % Lower bounds on the variables
Prob.x_U  = [inf inf]';  % Upper bounds on the variables
Prob.x_0  = [ 0   1 ]';  % Starting point
Prob.A    = [ 1   1      % Constraint matrix
              1  -1 ];
Prob.b_L  = [-inf  0 ]'; % Lower bounds on the constraints
Prob.b_U  = [  5   0 ]'; % Upper bounds on the constraints
Result    = qpSolve(Prob);
```

A similar approach is possible when solving all types of problems in TOMLAB.

5.2.5 More on Solving Quadratic Programs

When the problem is defined in the TOMLAB Init File format, it is then possible to run the graphical user interface *tomGUI*, the menu program *tomMenu*, or the multi-solver driver routine *tomRun* with the necessary arguments. To call the menu system, either type *Result = tomMenu;* or just *tomMenu;* at the Matlab prompt, choose *Quadratic Programming* The usage is very similar to the solution of Linear Programs, see the discussion and figures in Section 5.1.4.

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

$$\begin{array}{lrcl} \min\limits_{x} & & c^T x & \\ s/t & 0 \leq & x & \leq 1, \\ & & Ax & = b, \end{array} \tag{14}$$

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 Quick 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
Result                = mipSolve(Prob);

% ------------------------------------------------
% 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
Result                = mipSolve(Prob);

% ------------------------------------------------
% 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
Result          = mipSolve(Prob);

% ------------------------------------------------
% 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
Result          = mipSolve(Prob);
```

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 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 shows 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
Making standard form with   30 variables
Branch and bound. Depth First, then Breadth.

--- Branch & Bound converged!!! ITER =   892

    Optimal Objective function =    -141278.0000000002900000
x:       0       0       1      -0       1       1       1       1       0       1       0       1
         0       0       0       0       1       0       1       0       1       1       0       1
         5       6
B: 0 0 -1 0 -1 -1 -1 -1 1 -1 0 -1 -1 -1 0 0 0 0 -1 0 -1 0 -1 -1 0 0 0 0 0 0
Branch and bound. Depth First, then Breadth. Priority Weighting.

--- Branch & Bound converged!!! ITER =   470

    Optimal Objective function =    -141277.9999999998300000
x:       0       0       1      -0       1       1       1       1       0       1       0       1
         0       0       0       0       1       0       1       0       1       1       0       1
         5       6
B: 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 0 0 0 0 0
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!!! ITER =   138

    Optimal Objective function =    -141278.0000000002900000
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     5     6
B: 0 0 -1 0 -1 -1 -1 -1 1 -1 0 -1 -1 -1 0 0 0 0 -1 0 -1 0 -1 -1 0 0 0 0 0 0
Branch and bound. Depth First, then Breadth. Knapsack Heuristic. Priority Weighting.
    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!!! ITER =    94

    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     5     6
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 0 0 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.

5.3.1 Large Sets of Mixed-Integer Programs

It is easy to setup a problem definition file for a large set of mixed-integer problems. The approach is similar to the one for Linear Programs discussed in Section 5.1.3, the main difference is that *mipAssign* is used instead of *lpAssign*.

5.3.2 More on Solving Mixed-Integer Programs

When the problem is defined in the TOMLAB Init File format, it is then possible to run the graphical user interface *tomGUI*, the menu program *tomMenu*, or the multi-solver driver routine *tomRun* with the necessary arguments. To call the menu system, either type *Result = tomMenu;* or just *tomMenu;* at the Matlab prompt, choose *Mixed-Integer Programming* The usage is very similar to the solution of Linear Programs, see the discussion and figures in Section 5.1.4.

Instead of using the menu system solve the problem by a direct call to *tomRun* from the Matlab prompt or as a command in an m-file. This approach is of interest in an testing environment. The most straightforward way of doing it (when the problem is defined in *mipnew_prob.m*) is to give the following call from the Matlab prompt:

```
probNumber = 7;
Result = tomRun('mipSolve', 'mipnew_prob', probNumber);
```

Assume the problem should be solved with the following requirements:

- No printing output neither in the driver routine nor in the solver.
- Use solver *mipSolve*.

Then the call to *tomRun* should be:

```
Prob           = probInit('mipnew_prob',7);
```

```
PriLev         = 0;
Prob.PriLevOpt = 0;
Result         = tomRun('mipSolve', Prob, [], PriLev);
```

To have the result of the optimization displayed call the routine *PrintResult*:

```
PrintResult(Result);
```

Assume that the problem to be solved is defined in another TOMLAB Init File, say *ownmip_prob.m*, which is not the default Init File. The following example shows how to call *tomRun* to solve the first problem in *ownmip_prob.m*. Assume the same requirements as itemized above.

```
probFile       = 'ownmip_prob';
Prob           = probInit(probFile,1);
PriLev         = 0;
Prob.x_0       = [1;1];
Prob.PriLevOpt = 0;
Result         = tomRun('mipSolve', Prob, [], PriLev);
```

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. TOMLAB is also compatible with MathWorks Optimization TB v2.1. See Appendix E for more information and test examples.

All demonstration examples that are using the Tomlab Quick (TQ) 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 *ucDemo* and *conDemo*. All files that show how to use the Init File format are collected in the directory *usersguide*. The full path to these files are always given in the text. Throughout this section the test problem *Rosenbrock's banana*,

$$\begin{array}{ll} \min\limits_{x} & f(x) = \alpha \left(x_2 - x_1^2\right)^2 + (1 - x_1)^2 \\ s/t & \begin{array}{rcccl} -10 & \leq & x_1 & \leq & 2 \\ -10 & \leq & x_2 & \leq & 2 \end{array} \end{array} \tag{15}$$

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 v2.1, 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. The user must then always use the MathWorks Optimization TB v2.1 type of calls, not the TOMLAB function calls, and access to the GUI, menu and driver routines are not possible.

It is generally recommended to use the TOMLAB format instead, because having defined the files in this format, all MathWorks Optimization TB v2.1 and MathWorks Optimization TB v1.5 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. These files does not depend on if the TQ or IF format are used to solve the problem, in both cases they are identical. The m-files for a constrained IF format example is shown. The files are defined in the directory *usersguide* and described in more detail in Appendix D. In Section 6.2 and onwards similar m-files are used to solve unconstrained optimization using the TQ format.

The most simple way to write the m-file to compute the function value is shown for the example in (15) assuming $\alpha = 100$:

File: tomlab/usersguide/rbbs_f.m

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

function f = crbb_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.uP* is recommended, and for matrices, vectores and structures subfields to the field *Prob.user* and *Prob.userParam* are recommended. See the Section 10.3 on how the to set *User Supplied Problem Parameters* in the Init File format in TOMLAB.

Using the above convention, then the new m-file for the example in (15) 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)

if isempty(Prob.uP)
   alpha = 100;
else
   alpha = Prob.uP(1);
end

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

The value in the field *Prob.uP* is the α value. It is defined before calling the solver, either in a TOMLAB Init File, or directly before the call 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)

if isempty(Prob.uP)
   alpha = 100;
else
   alpha = Prob.uP(1);
end

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 = crbb_H(x, Prob)

if isempty(Prob.uP)
   alpha = 100;
```

```
else
   alpha = Prob.uP(1);
end

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 (15) 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 = crbb_c(x, Prob)

function c = crbb_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 = crbb_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=crbb_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 51 in Appendix C 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 C.

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 Quick format.

6.2 Solving Unconstrained Optimization using the TQ format

The use of the TOMLAB Quick 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 *probAssign* to create a structure in the TOMLAB TQ 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 Quick format (short call)
Prob    = probAssign('uc', x_L, x_U, Name, x_0, fLowBnd);

% Update the Prob structure with the names of files
Prob    = mFiles(Prob,'uc1_f', 'uc1_g', 'uc1_H');

Result  = ucSolve(Prob);

PrintResult(Result);
```

Instead of using *probAssign* and *mFiles*, it is possible to use *conAssign* with a limited number of input parameters. 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 10.6. 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 TQ 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, it is just to set the corresponding gradient function name empty, or reduce the number of parameters in the call to *mFiles*, as the following example (*uc2Demo*) shows:

```
...
% Only give the function. TOMLAB then estimates any derivatives automatically
Prob   = mFiles(Prob,'uc1_f');
...
```

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 (15). 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 version below is slightly modified, doing the call to *conAssign* making the parameter definitions directly. The α value is sent to the user routines using the field *userParam* 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 - compact, slightly modified, version
% ----------------------------------------------------------------------

% 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.userParam.alpha = Steep(i);
    Result(i)            = ucSolve(Prob);
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 define those parameters by first calling the routines *ProbAssign* and *mFiles*, or the routine *conAssign*. In this subsection, examples of two different approaches are given.

First, solve the problem *RB BANANA* in (15) as an unconstrained problem. In this case, define the problem in the files *ucnew_prob*, *ucnew_f*, *ucnew_g* and *ucnew_H* as described in Appendix D.3. 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 = ucSolve(Prob);
PrintResult(Result);
```

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 *ProbAssign*. 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. The call to the routine *mFiles* defines the files that defines the problem.

File: tomlab/usersguide/ucnewSolve2.m

```
oType   = 'uc';        % Problem type.
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    = probAssign(oType, x_L, x_U, 'ucNew',x_0);% Setup Prob structure.
Prob    = mFiles(Prob,'ucnew_f','ucnew_g');        % Problem definition files.
Prob.P  = 18;                                      % Problem number.
Prob.Solver.Alg=2;                                 % Use quasi-Newton BFGS
Prob.uP = 100;                                     % Set alpha parameter

Result = ucSolve(Prob);
PrintResult(Result);
```

Note that the calls to *ProbAssign* and *mFiles* could be replaced with the following call to *conAssign*.

```
Prob = conAssign('ucnew_f','ucnew_g',[],[],[-10;-10], [2;2], [-1.2;1]);
```

6.4 Solving Constrained Optimization using the TQ format

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

$$\begin{array}{lrcccl} \min\limits_{x} & f(x) = exp(x_1)(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1) & & & & \\ & -10 & \leq & x_1 & \leq & 10 \\ & -10 & \leq & x_2 & \leq & 10 \\ s/t & 0 & \leq & x_1 + x_2 & \leq & 0 \\ & 1.5 & \leq & -x_1x_2 + x_1 + x_2 & & \\ & -10 & \leq & x_1x_2 & & \end{array} \quad . \tag{16}$$

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 Quick 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 Quick 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  = conSolve(Prob);
PrintResult(Result);
```

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 Quick 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 TOM solvers

To follow the iterations in the TOM 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 Quick (TQ) 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 Solving Box-Bounded Global Optimization with TQ format

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 Quick format (short call)
Prob   = glcAssign('glb1_f', x_L, x_U, Name);
Result = glbSolve(Prob); % Solve using the default of 200 iterations

PrintResult(Result);
```

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 Quick 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 Quick 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 Defining Global Mixed-Integer Nonlinear Problems with TQ format

To solve global mixed-integer nonlinear programming problems with the TQ 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
setupFile = []; nProblem  = []; IntVars = []; VarWeight = []; KNAPSACK = [];
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, setupFile, nProblem, IntVars, VarWeight, ...
          KNAPSACK, 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 = glcSolve(Prob);
PrintResult(Result,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 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 v2.1. See Appendix E for more information and test examples.

All demonstration examples that are using the Tomlab Quick (TQ) 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 *lsDemo* and *llsDemo*. All files that show how to use the Init File format are collected in the directory *usersguide*. The full path to these files are always given in the text.

8.1 Solving Linear Least Squares Problems using the TQ Format

This section shows examples how to define and solve linear least squares problems using the TOMLAB Quick 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* [35].

```
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 = [0.1 0.5 0.3333 0.25 0.2 0.1667 0.1428 0.125 0.111]';
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 Solving 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* [35].

```
% 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 = [0.1 0.5 0.3333 0.25 0.2 0.1667 0.1428 0.125 0.111]';
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 Solving Nonlinear Least Squares Problems using the TQ Format

This section shows examples how to define and solve nonlinear least squares problems using the TOMLAB Quick 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 12. 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.userParam.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 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.userParam.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 Quick 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.userParam.K = 5;

Result  = clsSolve(Prob);

PrintResult(Result,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.userParam.K = 3.8 + 0.2*i;

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

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

Table 12 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 12: 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 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 [76]. 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 13. 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 13: 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 [72]. Good initial values are extremely important when solving real life exponential fitting problems, because they are so ill-conditioned. Table 14 shows the relevant routines. The best way to define new problems of the predefined exponential type is to edit the *exp_prob.m* Init File as described in Appendix D.9 on page 208.

Table 14: Exponential fitting test problems.

Function	**Description**
exp_ArtP	Generate artificial exponential sum problems.
expInit	Find starting values for the exponential parameters λ.
expSolve	Quick setup and solution of 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.
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.

9 Efficient Use of the SOL Solvers in TOMLAB

This section discusses the use of the Fortran solvers from Stanford System Optimization Laboratory (SOL). In order to use these solvers efficiently, it is recommended to read the corrsponding user guides as well. It is important to do help on the m-files corresponding to each solver as well as the TOMLAB interface routine. The names for *MINOS* solver are *minos.m* and *minosTL.m*, and similar for other solvers.

To learn all the different parameter settings for a solver it is useful to run the GUI, where all parameters are selectable, and all default values are shown. Furthermore there are short help available for the different solver parameters in the drag menus. Even if the user is not using the GUI to solve the particular user problem, it might be useful to run the test problems defined in TOMLAB to learn how to use the SOL solvers in the most efficient way.

9.1 Setting Parameters for the SOL Solvers

TOMLAB is using the structures *Prob.optParam* and *Prob.LineParam* for most parameters that influence the performance of the solvers. The Fortran solvers from Stanford System Optimization Laboratory (SOL) are using many parameters similar to the TOMLAB parameters, but also a large number of other parameters. To handle the use of the SOL solvers, a special field in the *Prob* structure, *Prob.SOL*, is used to send information to the SOL solvers, see Table 41 It is also used to store the information needed for warm starts of the SOL solvers.

The vector *Prob.SOL.optPar* of length 62 holds most of the different parameters that control the performance of the SOL solvers. All parameters have default values. If calling the SOL solver directly, not using TOMLAB, the user should set the values wanted in the *optPar* vector. The rest should have the value -999, which gives the default value used by the solver. If using TOMLAB then some of the elements in *optPar* are set to the current values in the *Prob.optParam* structure. For information on which elements are set and not set see the Table 39. Also, doing *help* on the TOMLAB interface routine gives all elements used, and their names in the *Prob.optParam* structure, if present in this structure. The TOMLAB interface routine always has the name of the routine, with the additional two letters *TL*, e.g. for *MINOS* the TOMLAB interface routine is *minosTL*.

Other important fields to set when using the SOL solvers are the print and summary files that the solvers create. These files are very important to look through if any problems are occuring, to find what the causes might be, and if any warnings or errors are reported from the SOL solvers. To create a print and summary file, one example when running *MINOS* is the following statements

```
Prob.SOL.optPar    = -999*ones(62,1);
Prob.SOL.optPar(1) = 111111;     % Maximal print level
Prob.SOL.PrintFile = 'minos.pri' % Print file called minos.pri
Prob.SOL.SummFile  = 'minos.sum' % Summary file called minos.sum
Prob.SOL.optPar(39)= 0;          % Derivative level. 0 = derivatives not known
Prob.NumDiff       = 6;          % Tell Tomlab that minos is estimating
Prob.ConsDiff      = 6;          % all derivatives
```

Here is added some other settings in the *optPar* vector as well. The choice of derivative level is very important, and should match the choice of *Prob.NumDiff* and *Prob.ConsDiff*, see Section 10.1. If *MINOS* is told that no derivatives are given, then *MINOS* will try to estimate them, and then TOMLAB must not do the same, i.e. *Prob.NumDiff* and *Prob.ConsDiff* must be set to six (internal solver estimation of derivatives). If *MINOS* is told that all derivatives are given, then TOMLAB might estimate them for *MINOS* using any of the five methods possible, or by using automatic differentiation.

9.2 Derivatives for the SOL Solvers

The Fortran solvers from Stanford System Optimization Laboratory (SOL), have some useful special features, which influence the way that input is prepared to the solvers.

When defining the gradient vector and the constraint Jacobian matrix it is often the case that they are only partially known. The SOL solvers give the possibility to mark these elements. They will then be estimated by finite differences.

In TOMLAB the gradient and the constraint Jacobian matrix are defined in two separate routines. If any element is unknown, it is just marked with the standard Matlab element *NaN*. The TOMLAB SOL interface routine will then convert the *NaN* element to the corresponding element used by SOL to mark that the element is unknown.

If any gradient or constraint Jacobian element is infinite, in Matlab set as *Inf* or *-Inf*, this element is converted to a big number, 10^{20}, in the TOMLAB SOL interface.

The following applies to the sparse nonlinear programming solvers *MINOS* and *SNOPT*. When the constraint Jacobian matrix is sparse, then only the nonzero elements should be given. The sparse pattern is given as a sparse matrix *Prob.ConsPattern*. In this matrix nonzero elements are marking nonzero elements in the constraint Jacobian. This pattern is static, i.e. given once before the call to the SOL solver. One problem is that a sparse matrix in Matlab is dynamic, i.e. only the nonzero elements of the matrix are stored. As soon as an element becomes zero, the vector of nonzeros are decreased one element. A gradient element that is normally nonzero might become zero during the optimization. Therefore care must be taken by the interface to return the correct values, because the SOL solvers assume the possible non-zero elements of the constraint Jacobian to be returned in the correct order. If some elements at the end are missing, they are estimated by finite differences.

The TOMLAB interface assumes the following conventions for the constraint Jacobian matrix:

- If the user returns a sparse matrix, and the number of nonzeros are equal to the number of nonzeros in *Prob.ConsPattern*, no checks are done.
- If the user returns a sparse matrix, and the number of nonzeros are not equal to the number of nonzeros in *Prob.ConsPattern*, the interface is matching all elements in the sparse array to find which nonzeros they represent, and returns the correct vector of static nonzeros.
- If the user returns a sparse matrix, and has given no pattern of nonzeros in *Prob.ConsPattern*, i.e. it is an empty array, then the solver and the interface assumes a full, dense matrix and the interface makes a full matrix before returning the elements, column by column, to the solver.
- If the user returns a dense matrix, the interface just feeds all elements, column by column, back to the solver.
- If too few elements are returned, the solver will estimate the rest using finite differences.

When using the dense SOL nonlinear programming solvers, the constraint Jacobian matrix is always assumed to be dense. The interface will convert any sparse matrix returned by the user to a dense, full matrix, and return the elements, column by column, to the solver.

If no derivatives are available, it might be better to use the *Nonderivative linesearch* in *SNOPT*. It is based on safeguarded quadratic interpolation. The default is to use a safeguarded cubic interpolation. To select *Nonderivative linesearch* set the following parameter:

```
Prob.SOL.optPar(40) = 0; % Use Nonderivative instead of Derivative Linesearch
```

9.3 SOL Solver Output on Files

The SOL solvers print different amount of information on ASCII files, one *Print File* with more information, and one *Summary File* with less information. *SNOPT* is using *snoptpri.txt.m* and *snoptsum.txt.m* as default names. *MINOS* is using *minospri.txt.m* and *minossum.txt.m* as default names. Both *SNOPT* and *MINOS* always define these log files in order to write error or warning messages if needed. The following example shows how to set new names, other than the default, for these files.

```
Prob.SOL.PrintFile = 'snoptp.out';   % New name for Print File
Prob.SOL.SummFile  = 'snopts.out';   % New name for Summary File
```

Even if the print level is set as low as possible, i.e. zero, still *SNOPT* and *MINOS* will write some summary and result information in the files. The only way to make *SNOPT* and *MINOS* totally silent, and avoid writing to log files, is by setting both the print level and the print file number to zero, i.e.

```
Prob.SOL.optPar(1) = 0;   % No print out
Prob.SOL.optPar(1) = 0;   % Print File unit 0
```

The *SQOPT* solver by default also defines the two log files as *sqoptpri.txt.m* and *sqoptsum.txt.m*. If the print level is 0 no output will occur, unless some errors are encountered. It is possible to make *SQOPT* totally silent and avoid any opening of files by the following statements.

```
Prob.SOL.PrintFile = '';
Prob.SOL.SummFile  = '';
Prob.SOL.optPar(2) = 0;
Prob.SOL.optPar(3) = 0; % Must be set explicitely to 0 to avoid file creation
```

The important thing is that besides a zero print level, also the file unit number for *SQOPT* is set to zero. The amount of printing is determined by a print level code, which is different for different solvers. See the help and manual for each solver. Some solvers also have two print levels, one major print level and one minor print level. This applies for *SNOPT*, *NPSOL* and *NLSSOL*. There are also different other parameters that influence how much output is written on the files. The following example show how to get maximum output for *SNOPT* on files with user defined names.

```
Prob.SOL.PrintFile = 'sn.p'; % New name for Print File
Prob.SOL.SummFile  = 'sn.s'; % New name for Summary File
Prob.SOL.optPar(1) = 111111; % Major print level, combination of six 0/1
Prob.SOL.optPar(2) = 10;     % Minor print level, 0, 1 or 10. 10 is maximum
Prob.SOL.optPar(5) = 1;      % Print Frequency
Prob.SOL.optPar(6) = 1;      % Summary Frequency
Prob.SOL.optPar(7) = 1;      % Solution yes. 0 = Solution not printed
Prob.SOL.optPar(8) = 1;      % Full options listing, not default
```

The other SOL solvers, *NPSOL*, *NLSSOL*, *LSSOL*, *QPOPT* and *LPOPT*, only define the *Print File* and *Summary File* if the user has defined names for them. See the help for each solver on how to set the correct print level and other parameters.

9.4 Warm Starts for the SOL Solvers

In TOMLAB warm starts for the SOL solvers are automatically handled. The only thing needed is to call the routine *WarmDefSOL* after having solved the problem the first time, as the following principal example shows doing repeated calls to *SNOPT*.

```
...                              % Define first problem
Result = tomRun('snopt',Prob); % Solve problem at t=1
...
for t=2:N
   ...                               % Changes at time t in Prob structure
   Prob      = WarmDefSOL('snopt', Prob, Result(t-1));
   Result(t) = tomRun('snopt',Prob);   % Call tomRun to solve again
   ...                               % Postprocessing
end
```

The *WarmDefSOL* routine are setting the warm start flag *Prob.WarmStart*, as true.

```
Prob.WarmStart = 1;
```

It is also moving subfields on the *Result.SOL* structure into *Prob.SOL* for the next run. For *SNOPT*, *SQOPT* and *MINOS* the following commands are needed.

```
Prob.SOL.xs = Result.SOL.xs
Prob.SOL.hs = Result.SOL.hs
Prob.SOL.nS = Result.SOL.nS
```

For *NPSOL* and the other SOL solvers the commands are

```
Prob.SOL.xs      = Result.SOL.xs
Prob.SOL.iState  = Result.SOL.iState
Prob.SOL.cLamda  = Result.SOL.cLamda
Prob.SOL.H       = Result.SOL.H
```

The fields *SOL.cLamda* and *SOL.H* are not used for *QPOPT*, *LPOPT*, and *LSSOL*. For *NPSOL* and *NLSSOL* the TOMLAB interface is automatically setting the necessary parameter *Hessian Yes* for subsequent runs. However, in order for the first warm start to work the user must set this parameter *before* the first run. The principal calls will be

```
...                              % Define first problem
Prob.SOL.optPar(50) = 1;       % Define Hessian Yes BEFORE first call
Result = tomRun('npsol',Prob); % Solve problem at t=1
...
for t=2:N
   ...
   Prob   = WarmDefSOL('npsol', Prob, Result);
   Result = tomRun('npsol',Prob);
   ...
end
```

Note that for all solvers the new initial value are taken from the field *Prob.SOL.xs* and any user change in the standard initial value *Prob.x_0* is neglected. More advanced handling of backup basis, solutions dumps etc. are possible, see the different user guides.

9.5 Memory Issues for the SOL Solvers

Several users have encountered problems where SOL solvers report insufficient memory on machines where memory should not be an issue. The solvers estimate their internal memory requirements at startup. This estimate is not always large enough so the user might have to specify additional memory. This can be accomplished by

```
Prob.optParam.morememe = 1000; % or even larger if necessary
```

10 Special Notes and Features

In this section is collected several topics of general interest, which enables a more efficient use of TOMLAB. The section about derivatives is particulary important from a practical point of view. It often seems to be the case that either it is nearly impossible or the user has difficulties in coding the derivatives.

10.1 Approximation of Derivatives

Both numerical differentiation and automatic differentiation are possible. There are five ways to compute numerical differentiation. Furthermore, the SOL solvers *MINOS*, *NPSOL*, *NLSSOL* and *SNOPT* includes numerical differentiation.

Numerical differentiation is automatically used for gradient, Jacobian, constraint gradient and Hessian if the user routine is nonpresent.

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 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* or *Prob.HessPattern*. If there are many zeros in the matrix, this leads to significant savings in each iteration of the optimization.

Forward or Backward Difference Approximations

The default way is to use the classical approach with forward or backward differences together with an automatic step size selection procedure. This is handled by the routine *fdng*, which is a direct implementation of the FD algorithm [34, page 343].

The *fdng* routine is using the parameter field *DiffInt*, in the structure optParam, see Table 39, page 183, 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 in 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

If the Spline Toolbox is installed, gradient, Jacobian, constraint gradient and Hessian approximations could be computed in three different ways depending of which of the three routines *csapi*, *csaps* or *spaps* 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

The fifth approximation method is a method by Squire and Trapp [82], which is using complex variables to estimate the derivative of real functions. The method is not particulary 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 gradient 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 [66], 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 ADMAT toolbox. For information of how to get a copy of ADMAT TB, see http://www.tc.cornell.edu/~averma/AD/download.html. Below a short instruction of how to install it is given.

1. Install the ADMAT TB at e.g. d:\Admat\...
2. Change the path commands in ...\tomlab\lib\admatInit.m and execute the file. (If choosing d:\Admat in 1. it should be:)

```
...
...
path(path,'d:\admat');
path(path,'d:\admat\reverse');
path(path,'d:\admat\reverseS');
path(path,'d:\admat\PROBS');
path(path,'d:\admat\ADMIT\ADMIT-1');
...
...
```

3. If not done before, setup location of installed c-compiler by "mex -setup".
4. In directory d:\Admat\ADMIT\ADMIT-1, execute "mex id.c" to form id.dll.
5. **NOTE:** The ADMAT distribution contains a bug in the file ...\admat\admit\ADMAT-1\getJPI.m. Line 40 which originally reads

   ```
   PartInfo.SPJ=spones(m,1);
   ```

   should be changed to

   ```
   PartInfo.SPJ=sparse(m,1);
   ```

ADMAT TB should be initialized by calling *admatInit* before running TOMLAB with automatic differentiation. Note that in order for TOMLAB to be fully compatible with the ADMAT TB, the functions must be defined according to the ADMAT TB 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 *tomMenu*, push 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.AutoDiff* equal to 1 for automatic differentiation or *Prob.NumDiff* to 1, 2, 3, 4 or 5 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 call the Spline Toolbox routines *csapi*, *csaps* and *spaps*, 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 39, 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 with the same meaning as for *Prob.NumDiff* as described above.

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 toolbox ADMAT TB must be installed.

Automatic Differentiation example

```
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.AutoDiff   = 1;             % Use Automatic Differentiation.
Result          = ucSolve(Prob);
```

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          = ucSolve(Prob);
PrintResult(Result,2);

% Change the tolerances used by algorithm FD

Prob.GradTolg   = [1E-5; 1E-6];  % Change the predefined step size
Result          = ucSolve(Prob);

% The change leads to worse accuracy

PrintResult(Result,2);

% 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          = ucSolve(Prob);

% Now the result is perfect, very close to the optimal == 0.

PrintResult(Result,2);

Prob.NumDiff    = 5;             % Use the complex variable technique
```

```
% The advantage is that it is insensitive to the choice of step length

Result          = ucSolve(Prob);

% When it works, like in this case, it gives absolutely perfect result.

PrintResult(Result,2);

pause
```

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. It is only possible to run if the Spline toolbox is installed.

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 the Spline Toolbox routine csapi.
Result          = clsSolve(Prob);
PrintResult(Result,2);
```

10.2 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 C. 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 recent 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.

10.3 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. One field in the *Prob* structure, the field *Prob.uP* is used to store the user supplied problem parameters.

The best way to describe the User Supplied Problem Parameter feature is by an example. Assume a problem with variable dimension. 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 of help. The problem 27 in *cls_prob* is an example of the above:

```
...
...
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 it if the Name of the problem is the same as the one set in *Prob.Name*. If this is true, *uP* is set to found value. When calling *askparam*, if $ask <= 0$, then 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, then *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 to 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 VARProb.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 as 20, the output will be:

```
Current value = 10

Give problem dimension 20
```

Now call *clsSolve* to solve the problem,

```
Result=clsSolve(Prob);
```

which gives the printed output

```
================================================
Iteration no:    0  Func        80.00000000000000000000 Cond 1
================================================
Iteration no:    1  Func         1.25000000000000000000 Cond 1
*** Convergence 2, Projected gradient small ***
```

As a second example assume that the user wants to solve the problem above for all dimensions between 10 and 30. Then the following code snippet will do the work.

```
for dim=10:30
   Prob = [];
   Prob.uP(1) = dim;
   PriLev = 0;

   Result = tomRun('clsSolve', 'cls_prob', 27, Prob, [], PriLev);
end
```

10.4 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* (15) 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.

10.5 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 15 the meaning of different print levels are defined. There is a third print level, defined in *Prob.optParam.PriLev*, that determines how much output is written by the SOL solvers on files. See Section 9.3 for details on this.

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 15: 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 39.

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, see Table 23 page 153.

For matrices there are two routines, *mPrint* and *printmat*. The routine *printmat* 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, *printmat* will first display $A(:, 1:10)$, then $A(:, 11:20)$ and so on.

The routine *printmat* 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.

10.6 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 :

$$\begin{array}{ll} \min\limits_{x} & f(x) = \frac{1}{2}\sum\limits_{1}^{6} r_i(x)^2 \\ s/t & Ax \geq b \\ & x \geq 0 \end{array} \tag{17}$$

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 (17) 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 (17) 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   = sTrustr(Prob);
```

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   = sTrustr(Prob);
```

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*.

10.7 Usage of routines from Optimization Toolbox 1.x

TOMLAB has interfaces to some of the solvers in MathWorks Optimization TB v1.5. If the user has this toolbox and want to run these routines for problems defined in the TOMLAB format, the path to directory *optim1.x* must be placed before the path to the directory *mex*. Edit the file *startup.m* and change *if 0* to *if 1* to add the correct path.

If running Mideva, there are equivalents to these routines. If the user wants to run these routines for problems defined in the TOMLAB format, the path to directory *mideva1.x* must be placed before the path to the directory *mex*. and to the directory *optim1.x* (if in path). In the file *startup.m*, change *if 0* to *if 1* to add the correct path.

10.8 Using Matlab 5.0 or 5.1

Are you are running TOMLAB under Matlab 5.0 or 5.1? If running on PC then the directory *matlab5.1* must be put before the other TOMLAB directories in the Matlab search path.

If running on Unix then the directory *unix5.1* must be put before the other TOMLAB directories in the Matlab search path.

The *matlab5.1* directory contains two routines, *strcmpi* and *xnargin*. The command *strcmpi*, used by some TOMLAB routines, is a Matlab 5.2 command. Therefore, the *matlab5.1* directory routine *strcmpi* is created for 5.0/5.1 users. It simply calls *strcmp* after doing *upper* on the arguments.

A bug in Matlab 5.1 on PC for the *nargin* command makes it necessary to call *nargin* with only non-capitalized letters. The routine *xnargin* in Matlab 5.1 does lower on the arguments in the call to *nargin*, and the *xnargin* routine in the *lib* directory does not do it. On unix systems it is necessary to keep the exact function name.

The *unix5.1* directory contains one routine, *strcmpi*.

10.9 Utility Test Routines

The utility routines listed in Table 16 run tests on a large set of test problems.

Table 16: System test routines.

Function	Description	Section	Page
runtest	Runs all selected problems defined in a problem file for a given solver.	13.2.8	147
systest	Runs big test for each *probType* in TOMLAB.	13.2.10	149

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.

11 The tomGUI Graphical User Interface (GUI)

The Graphical User Interface is started by calling the Matlab m-file *tomlabGUI.m*, i.e. by entering the command *tomlabGUI* at the Matlab prompt. There is a short command *tomGUI.m*. The GUI has five modes; Normal (Result mode), Figure, General parameter mode, Solver parameter mode and Plot parameter mode. At start the GUI is in Normal mode, shown in Figure 8.

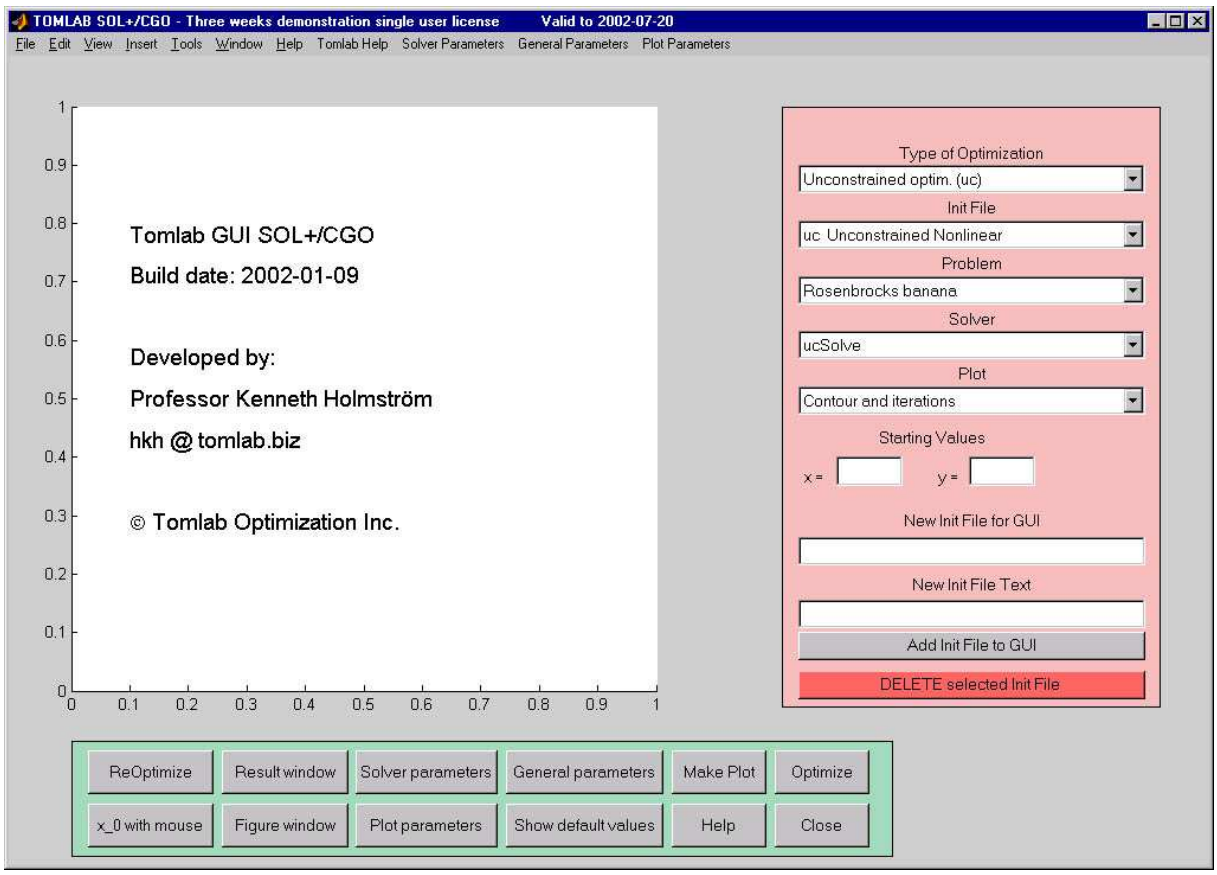


Figure 8: The GUI after startup.

There are one axes area, five menus; *Type of Optimization*, *Init File*, *Problem*, *Solver* and *Plot*, and twelve push buttons; *ReOptimize, x_0 with Mouse, Result window, Figure window, Solver parameters, Plot parameters, General parameters, Show default values, Make Plot, Help, Optimize* and *Close*.

There are two edit controls where it is possible to enter the first two initial values (Starting Values) of the unknown parameters vector. If the problem has more then two dimensions, the rest of the initial values are given in the *General Parameter* mode.

In the axes area plots and information given as text are displayed.

The *Type of Optimization* menu is used to select subject, i.e. which type of problem to be solved. There are currently ten main problem types; unconstrained optimization, quadratic programming, constrained optimization, nonlinear least squares, exponential sum fitting, constrained nonlinear least squares, mixed-integer linear programming, linear programming, global unconstrained optimization and global constrained optimization.

With the selection in the *Init File*, the user makes the choice of which file to get the problem to solve from. In the *Problem* menu, the user selects the actual problem to be solved, among the ones present in the current selected Init File. Presently, there are about 15 to 50 predefined test problems for each problem type. The user can easily define his own problems and try to solve them using any solver, see sections 5, 6 and 8.

The *Solver* menu is used to select solver. It can either be a TOMLAB internal solver, a solver in the Matlab Optimization Toolbox or a general-purpose solver implemented in Fortran or C and ran using a MEX-file interface.

Changing *Type of Optimization* will automatically change the menu entries in the *Init File* menu, the *Problem* menu, and the *Solver* menu.

From the *Plot* menu, the type of plot to be drawn is selected. The different types are contour plot, mesh plot, plot of function values and plot of convergence rate. The contour plot and the mesh plot can be displayed either in the axes area or in a new figure. The plot of function values and convergence rate are always displayed in a new figure. For least squares problems and exponential fitting problems it is possible to plot the residuals, the starting model and the obtained model.

When clicking the *Show default values* button, the default values for every parameter are displayed in the edit controls. The button then shows the text *Hide default values*. If pushing the button again, the parameters will disappear. Before solving a problem, the user can change any of the values. If leaving an edit control empty, the default values are used. If giving a value less than -1, it will normally not be used at all. The default values are used instead. The value -999 indicates missing value, and the default value is always used by the solver. The value -900 indicates both a missing value, and that this quantity is never used by the currently defined solver. Thus it is pointless to set a value for this quantity.

The buttons *General Parameters*, *Solver Parameters* and *Plot Parameters* are described in Section 11.1.

Pushing the *Make Plot* button gives a plot of the current problem. Pushing the *Figure window* button switches back to the last plot made. In the contour plot, known local minima, known local maxima and known saddle points are shown. It is possible to make a contour plot and a mesh plot without first solving the problem. After the problem is solved, a contour plot shows the search direction and trial step lengths for each iteration. A contour plot of the classical Rosenbrock banana function, together with the iteration search steps and with marks for the line search trials displayed, is shown in Figure 9.

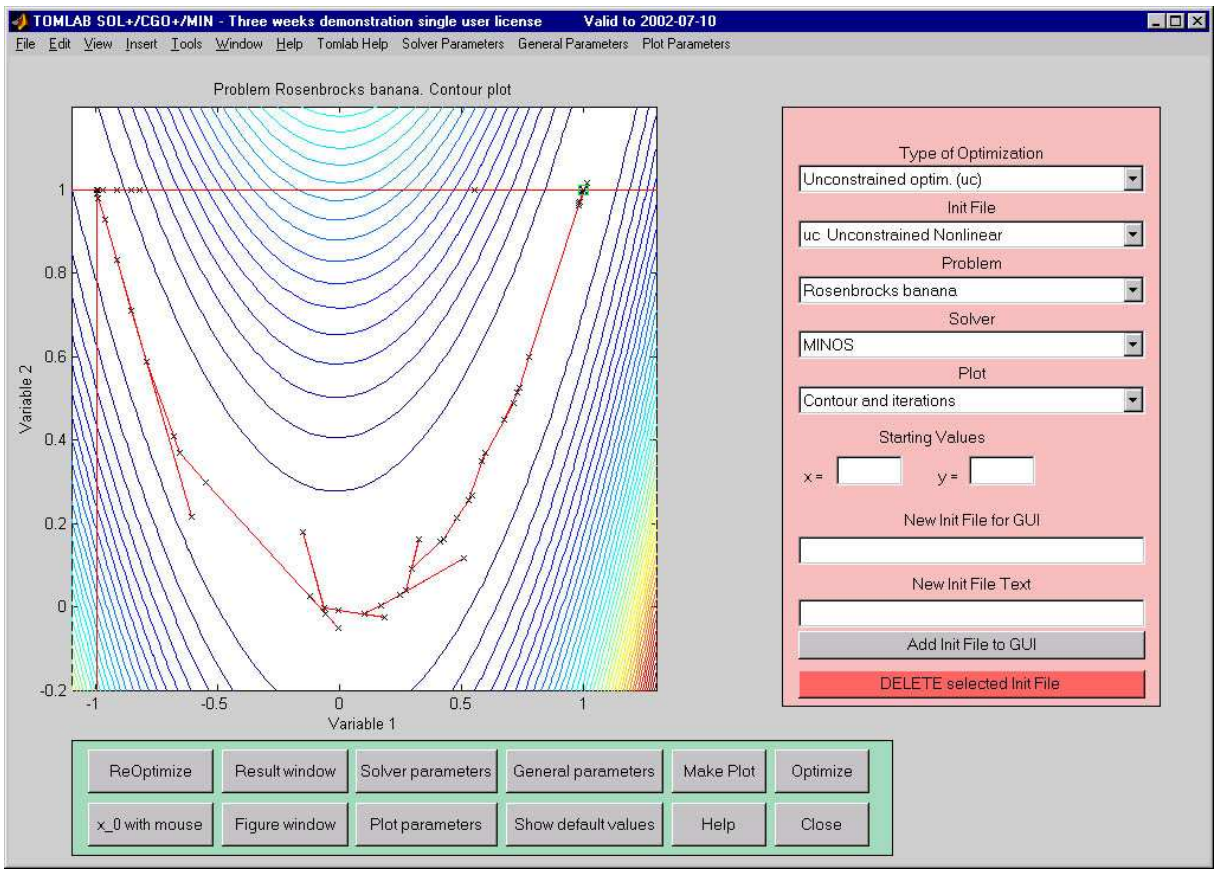


Figure 9: A contour plot with the search directions and marks for the line search trials for each iteration when solving an unconstrained optimization problem.

A contour plot for a constrained problem and a plot of the data is given in Figure 10. In the contour plot, (inequality) constraints are depicted as dots. Starting from the infeasible point $(x_1, x_2) = (-5.0, 2.5)$, the solution

algorithm first finds a point inside the feasible region. The algorithm then iteratively finds new points. For several of the search directions, the full step is too long and violates one of the constraints. Marks show the line search trials. Finally, the algorithm converges to the optimal solution $(x_1^*, x_2^*) = (-9.5474, 1.0474)$.

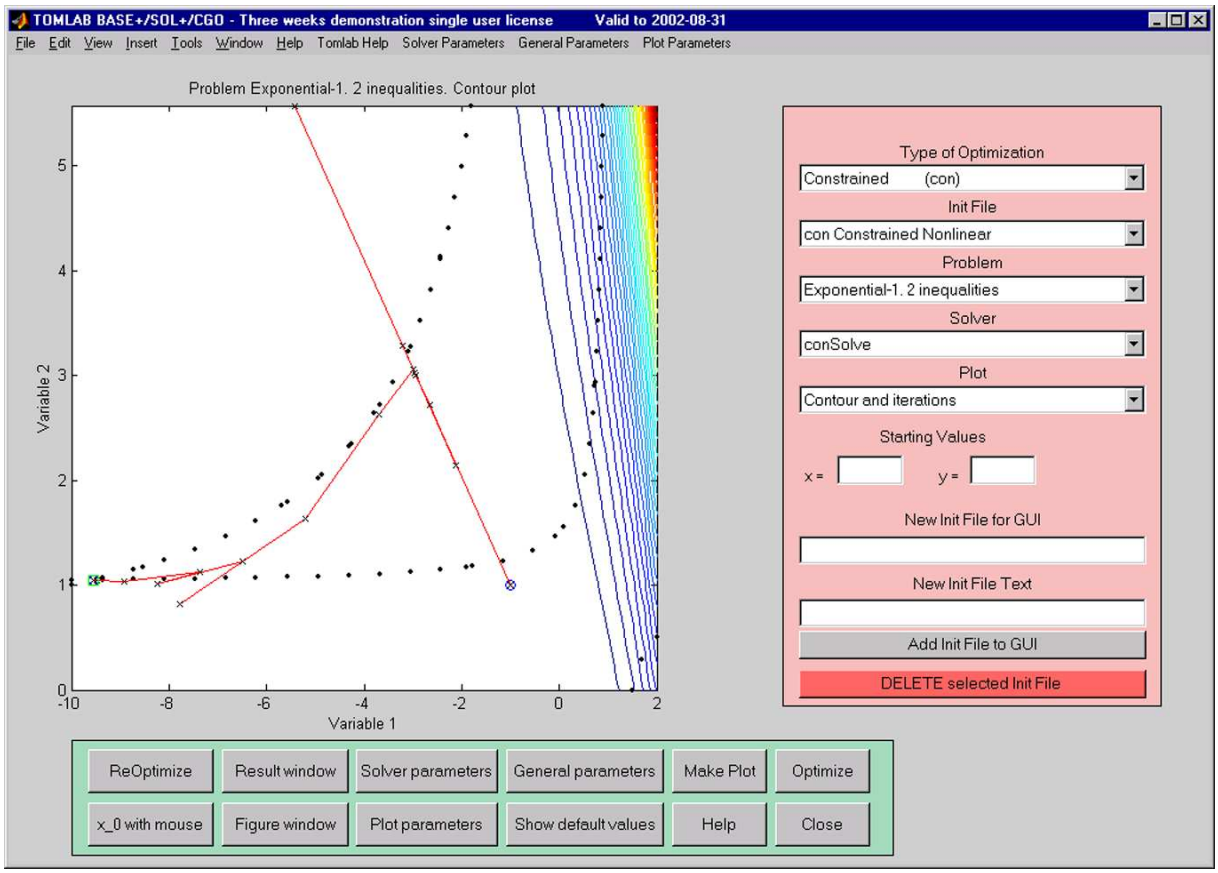


Figure 10: A contour plot for a constrained problem

In Figure 11 is shown a plot of the data and the obtained model for a nonlinear least squares problem, in this case an exponential fitting problem.

For global optimization one option is a contour plot together with the sampled points. This plot is illustrative for how the search procedure is sampling. The points samples cluster around the different local minima. One example is shown in Figure 12, where the blue dots are the sampled points.

The *Help* button gives some information about the current problem, e.g. the number of variables. Note that there are four drag menus on top *TOMLAB Help*, *Solver Parameters*, *General Parameters*, and *Plot Parameters*. Selecting any items in these menus displays a help text in plot window.

When the user has chosen a solver and a problem, he then pushes the *Optimize* button to solve it. When the algorithm has converged, information about the solution procedure are displayed. This information will include the solution found, the function value at the solution, the number of iterations used, the number of function evaluations, the number of gradient evaluations, the number of floating point operations used and the computation time. If no algorithm is selected as in Figure 8, the Run button has the same function as the Plot button.

If a contour plot is displayed in the axes area and the user pushes the button named *x_0 with mouse*, it is possible to select starting point for the current algorithm using the mouse. Pushing the *ReOptimize* button, the current problem is re-optimized with the starting point defined as the solution found in the previous solution attempt.

To close the GUI, push the Close button.

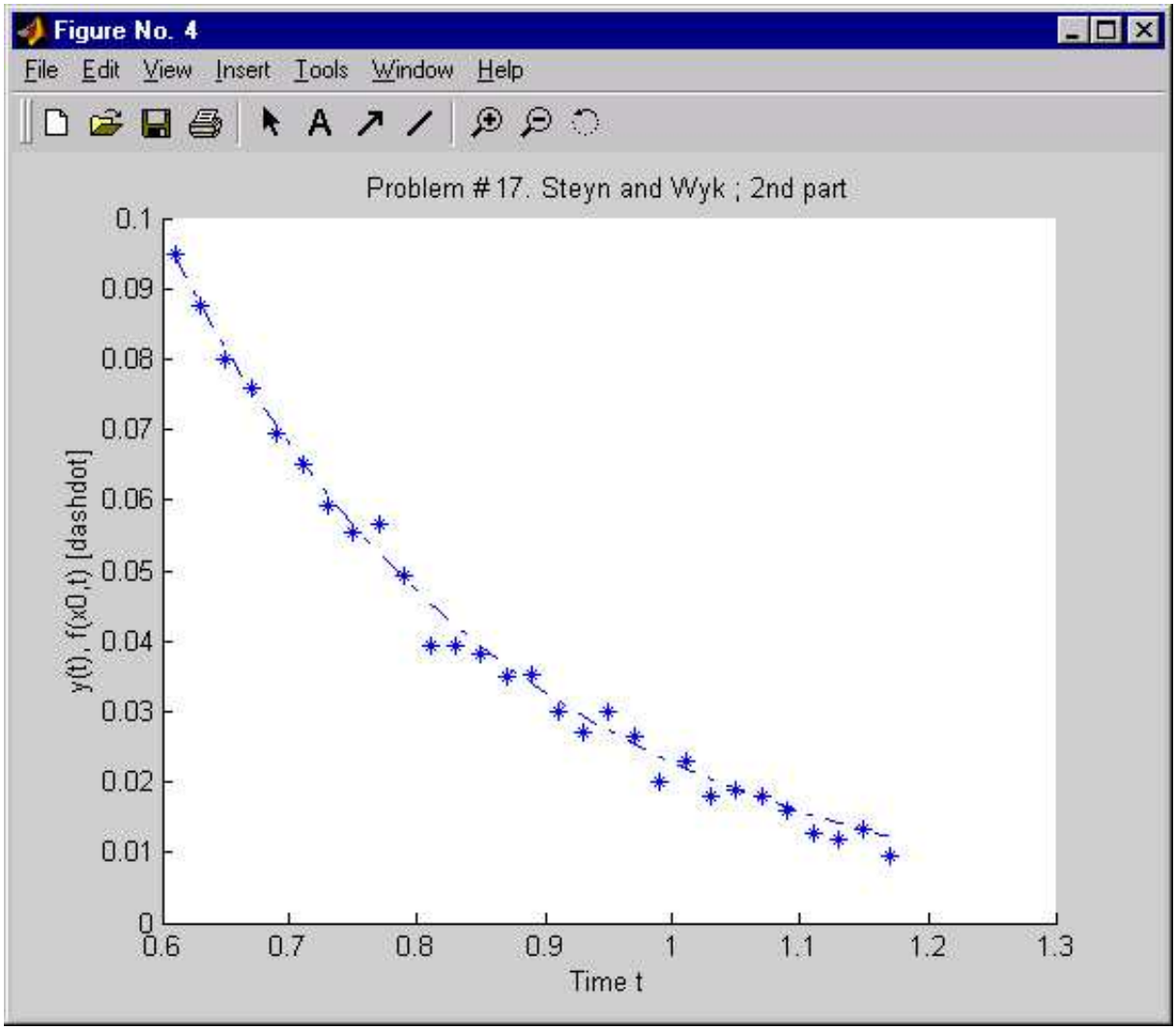


Figure 11: A plot of the data and the model for a exponential sum fitting problem. The figure shows the second part of the data series and the estimated optimal model

11.1 The Input Modes

This section describes the three input modes *General Parameter* mode, *Solver Parameter* mode and *Plot Parameter* mode. When pushing one of these buttons, the GUI will change to the corresponding mode. The axes area is replaced by more edit controls and menus.

11.2 General Parameter Mode

The General Parameter Mode makes it possible to set parameters common for the current *Type of Optimization* given, See Figure 13.

To the left is the maximum number of iterations (*Max iterations*), and limits on major and minor iterations. The last two are used for some solvers. Above each other is the tolerances, e.g. the termination tolerance on the function value (EpsF), the rank test tolerance (EpsR), the termination tolerance on the change in the decision variables (EpsX), and the termination tolerance on the gradient (EpsG). If a solver for constrained optimization handling linear constraints is selected, an edit control for the allowed tolerance on constraint violation (EpsB) is shown. Another similar edit control (EpsC) is shown for solvers handling nonlinear constraints. This edit control sets the allowed termination tolerance on the nonlinear constraint violation.

For problems with more than two decision variables, starting values for decision variable x_3 to x_n are given in the

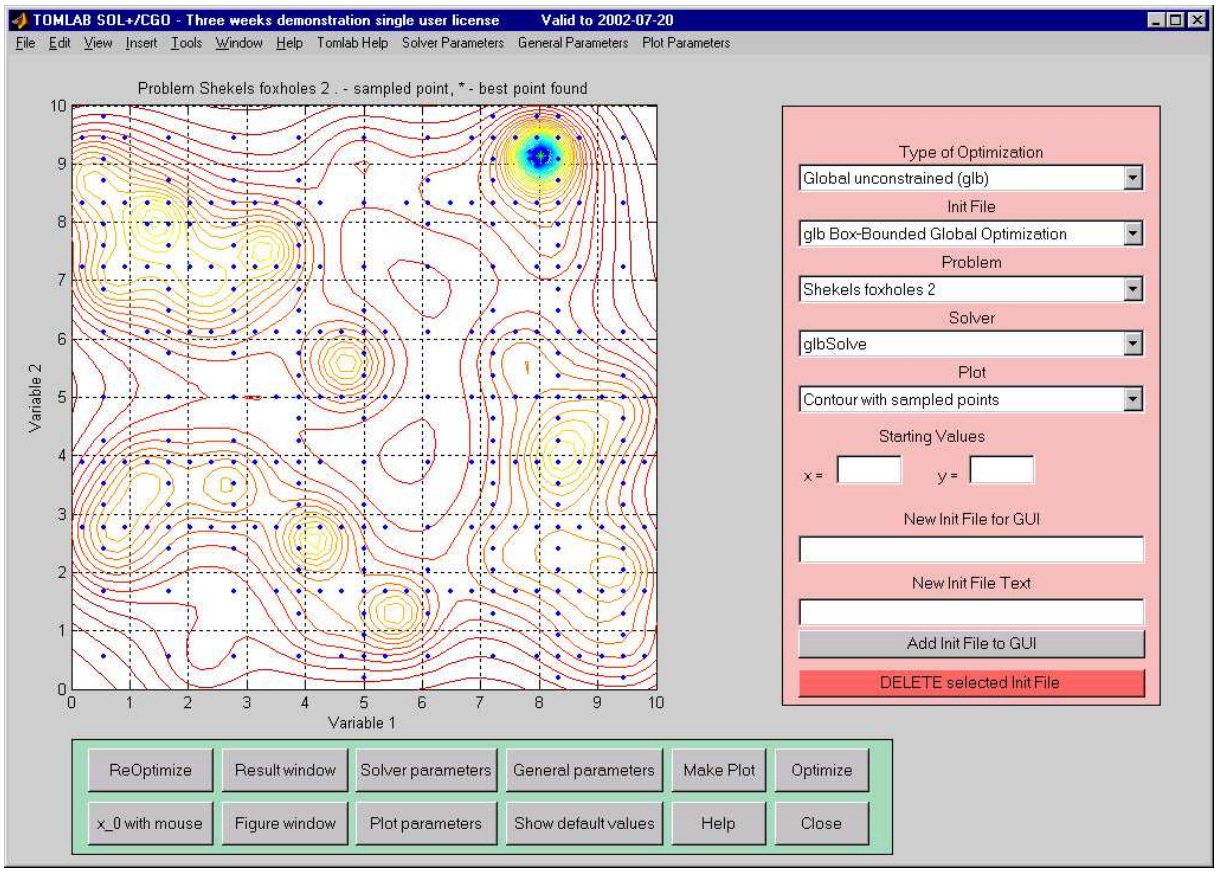


Figure 12: Contour plot with sampled points for the two-dimensional Shekels foxhole problem. The problem has several local optima and is best solved by global optimization methods.

edit control named 'Starting Values x3 - xn'. Starting values for x_1 and x_2 are given in the edit controls named 'Starting Values'.

The first menu selects method to compute first and second derivatives. Except for using an analytical expression, these can be computed either by automatic differentiation using the ADMAT Toolbox, distributed by Arun Verma at http://www.tc.cornell.edu/~averma/AD/download.html, or by five different approaches for numerical differentiation. Three of them requires the Spline Toolbox to be installed.

The *Print Level Driver Routine* menu selects the level of output from the optimization driver after the solver has been called. All this output is printed in the Matlab Command Window. Normally it is enough with the default information given in the GUI result window.

If the *Pause Each Iteration* check box is selected, the TOMLAB internal solvers are using the pause statement to halt after each iteration.

If the check box *Hold Previous Run* is selected, all information about the runs are stored. Making a contour plot, the step and trial step lengths for all solution attempts are drawn. This option is useful, e.g. when comparing the performance of different algorithms or checking how the choice of starting point affects the solution procedure.

For some predefined test problems, it is possible to set parameter values when initializing the problem. These parameters can for example be the size of the problem, the number of residuals or the number of constraints. Questions asking for input of the parameters will appear when selecting the check box named **Ask Questions when defining problem**, otherwise, if the check box is not selected, default values will be used.

When selecting exponential fitting problems, two new menus and a new edit control will appear. The number of exponential terms (*Terms*) in the approximating model is selected, default two. There is a choice whether to solve the weighted least squares fitting problem using an ordinary or separable nonlinear least squares algorithm (*Least Squares Method*). There are four types of residual weighting selectable (*Residual Weights*). The option *y-weighting*, i.e. weighting with the measured data, is often proposed, but default is *No weighting*.

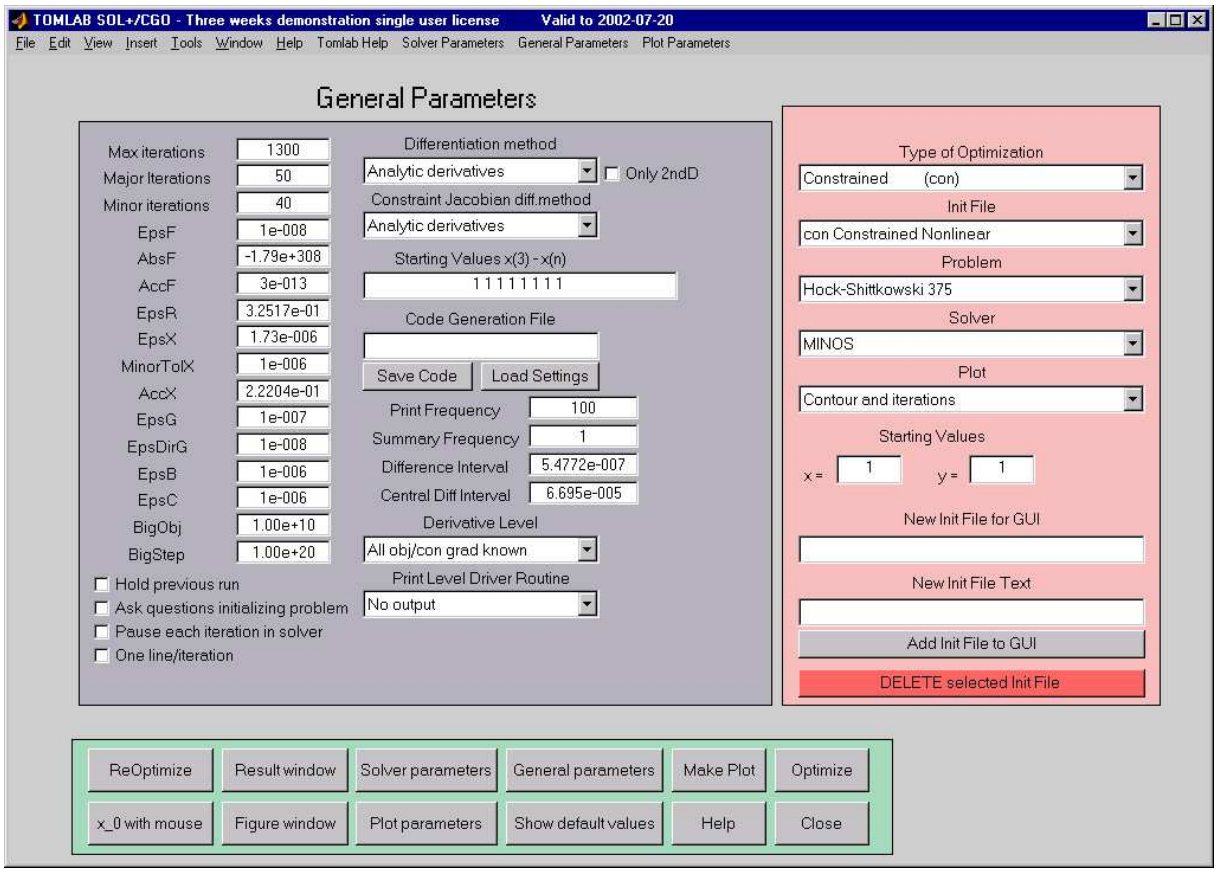


Figure 13: The GUI in General Parameter Mode.

Code Generation

Entering a name in the edit control *Code Generation File* and clicking the *Save Code* button, two files will be generated; one Matlab mat-file and one Matlabm-file. The file name given should not include any extension. For example, entering the name *test* in the edit control, the files *test.mat* and *test.m* will be generated. The files are saved in the current directory. In the mat-file parameters are stored in the *Prob* structure format, but the name of the structure is *Problem*. In the m-file all commands needed to make a stand-alone run without using the GUI are defined. The parameter values are those currently used by the GUI. To run the problem, just issue *test* in the command window. Note that the print level is set very low by default, and often nothing is displayed. It is easy to edit the m-file for different needs.

If entering a name in the *Code Generation File* edit control and clicking the *Load Settings* button, the GUI will read the corresponding mat-file. The mat-file should contain a TOMLAB *Prob* structure with the name *Problem*. This is the type of mat-file generated by the *Save Code* button. The GUI will switch to the *Type of Optimization*, *Init File*, *Problem File* and *Solver* defined in the mat-file. The default values for all parameters will be loaded from the mat-file. This option is useful for retrieving complicated settings for a particular problem and solver.

11.3 Solver Parameter Mode

For the Solver Parameter Mode the parameters areas shown, and possible to set, are dependent on the particular solver selected in the *Solver* menu. Some parameters are common for many of the internal TOMLAB solvers FLow, the best guess on a lower bound for the optimal function value, is used by TOMLAB solver algorithms using the Fletcher line search algorithm [25]. In Figure 14 the Solver Parameter Mode for the MINOS solver is shown. MINOS and SNOPT are the solvers with most parameters to be set. Default values are always defined, and in the picture is shown the default values for MINOS.

Algorithms using a line search approach needs the line search accuracy σ (Sigma) between zero and one. Values

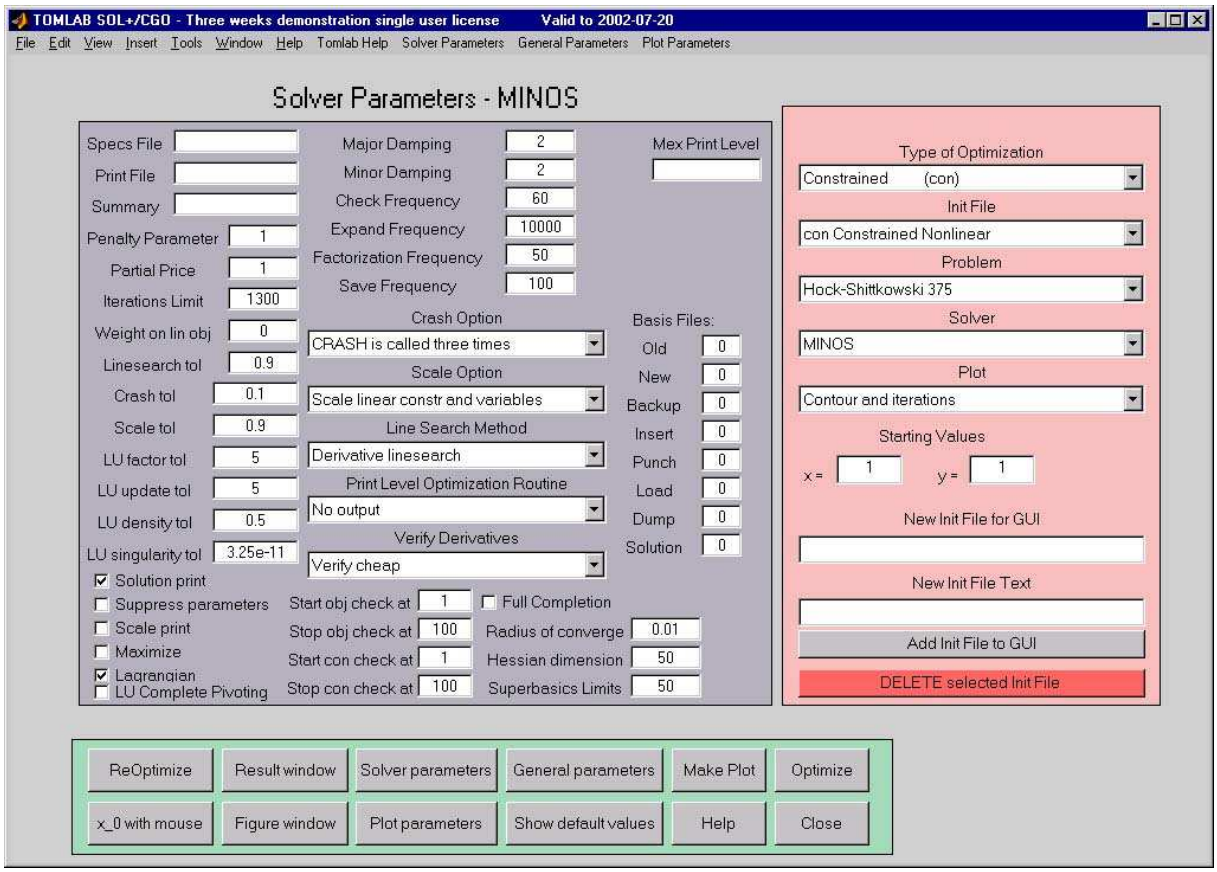


Figure 14: The Solver Parameters for the MINOS solver.

close to one (0.9) gives an inaccurate line search, often recommended. Values close to zero (0.1) gives a more accurate line search, recommended for conjugate gradient methods and sometimes for quasi-Newton methods. Another menu determines if a quadratic or a cubic interpolation shall be used in the line search algorithm.

The *Print Level Optimization Routine* menu is used to select the level of output from the optimization solver. All output printed during the optimization are displayed in the Matlab Command Window.

The menu named *Algorithm* differs between different solvers. Some solvers have only one algorithm alternative, others have several. The menu named *Method* also differs between different solvers, and is sometimes hidden. Using an unconstrained solver, a least squares solver or an exponential fitting solver, the menu selects method to compute the search direction. In the constrained case, the Method menu gives the quadratic programming solver to be used in SQP algorithms.

11.4 Plot Parameter Mode

The Plot Parameter Mode makes it possible to set parameters common for plotting, see Figure 15.

The edit controls named 'Axes' set the axes in the contour plot and the mesh plot.

To make a contour plot or a mesh plot for problems with more than two decision variables, the user selects the two-dimensional subspace to plot. The indices of the decision variables defining the subspace are given in the edit controls called 'Variables At Axis When $n > 2$'. The view for a mesh plot is changed using the edit controls 'Mesh View'.

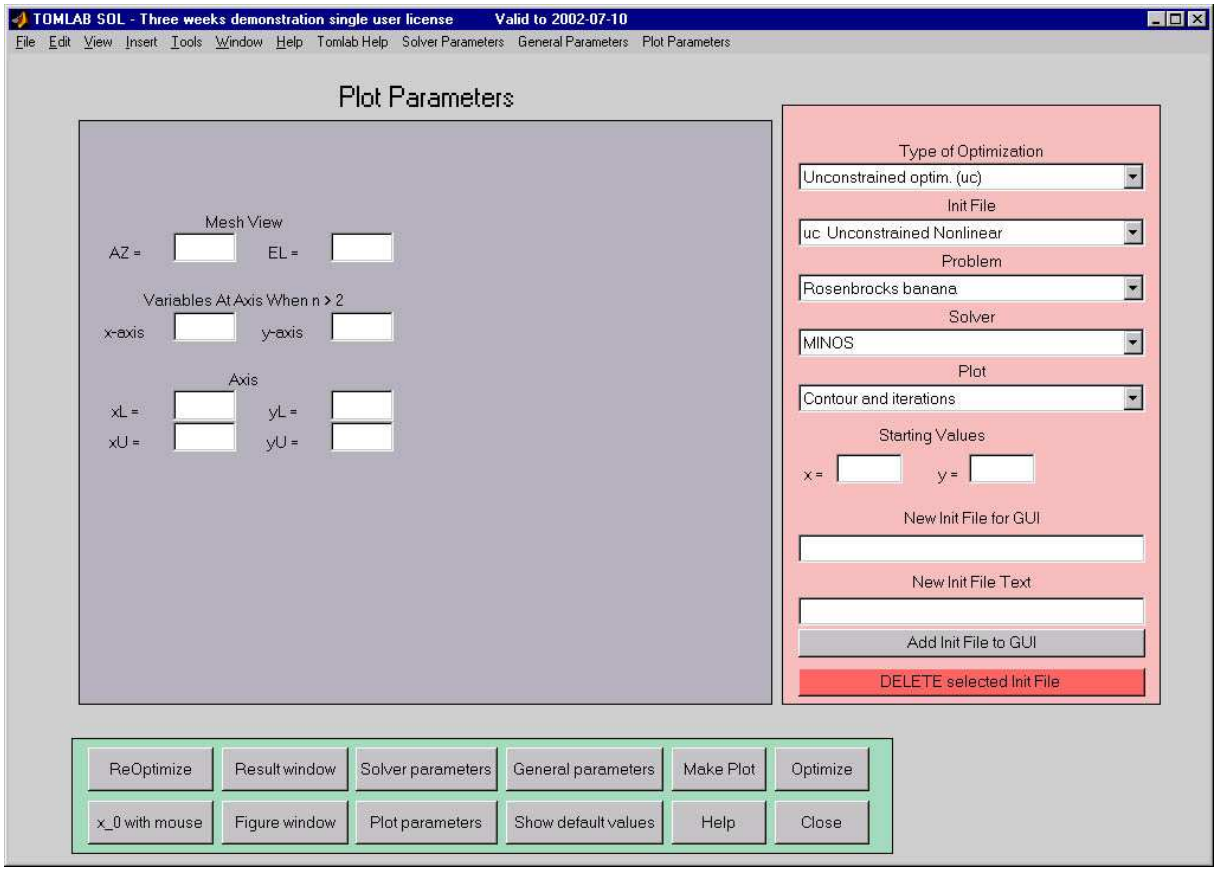


Figure 15: The plot parameters in the Plot Parameter Mode.

12 The Menu Program tomMenu

The general menu program *tomMenu* has much of the functionality of the GUI (Section 11), and is sometimes faster to use. It is also possible to run when not running a window system, e.g. when using telnet to a machine, in which case the GUI is not possible to use. Some specific solver parameter settings are not available in *tomMenu*, as well as the code generation possibility.

Starting the menu system, the first menu, seen in Figure 16, is the selection of the type of optimization problem (*probType*).

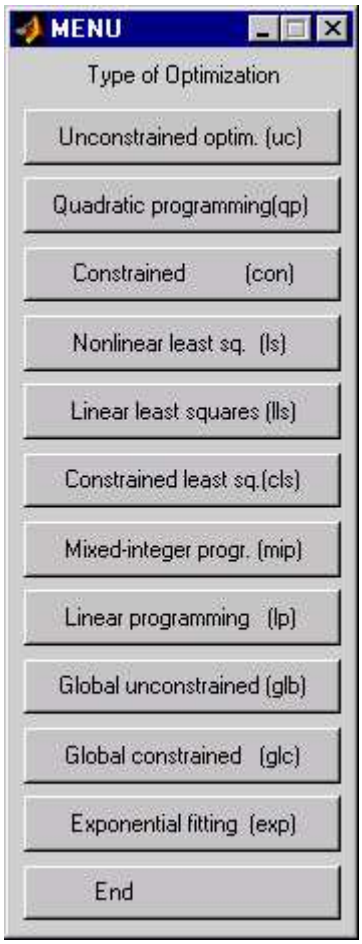


Figure 16: The choice of the Type of Optimization in *tomMenu*.

The *tomMenu* sub-menu for unconstrained optimization is shown in Figure 17. The other sub-menus look similar, with additional items corresponding to options needed for the relevant problem and solver type. In the following of this section, the most important standard menu choices are commented.

The *Choice of Problem Init File and Problem* button selects the problem Init File and the problem to be solved. Correspondingly, the *Solver*, *Solver algorithm* and *Solver sub-method* buttons selects the solver, particular solver algorithm, and other method choice to be used.

From the *Optimization Parameter Menu*, parameters needed for the solution can be changed. The user selects new values or simply uses the default values. The parameters are those stored in the *optParam* structure, see Table 39. The *Output print levels* button selects the level of output to be displayed in the Matlab Command Window during the solution procedure. The *Optimization Parameter Menu* also allows the user to choose the differentiation strategy he wants to use. The *Optimization Parameter Menu* is dependent on the type of optimization problem. A short parameter menu for quadratic programming is shown in Figure 18.

Pushing the *Optimize* button, the relevant routines are called to solve the problem.

When the problem is solved, it is possible to make different types of plots to illustrate the solution procedure.

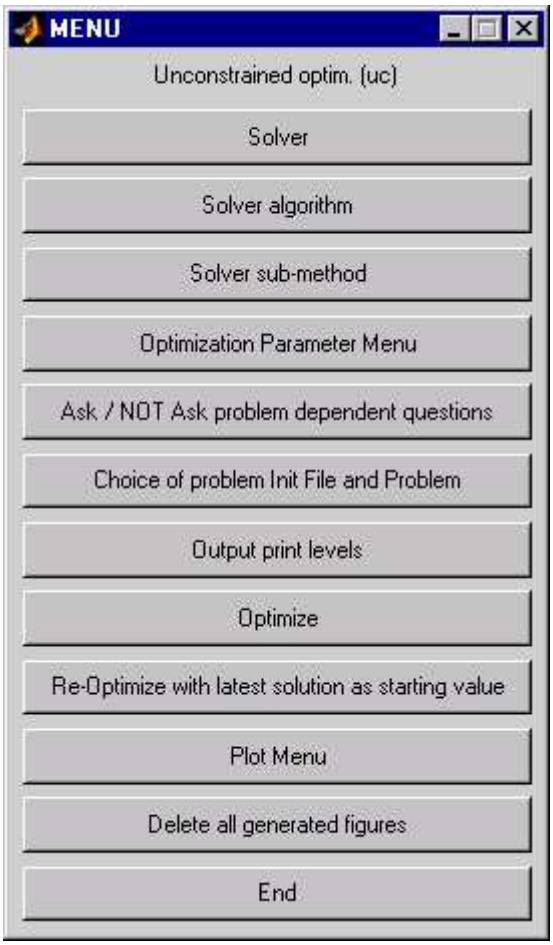


Figure 17: The main menu for unconstrained optimization in *tomMenu*.

Pushing the *Plot Menu* button, a menu choosing type of plot will appear. A overview of the available plotting options are given in connection with the Graphical User Interface described in Section 11.

The menu routine is started by just typing *tomMenu* at the Matlab prompt. In Section 14.2.1 we illustrate how to use the menu system for linear programming problems (*lpMenu*). The menus for nonlinear problems work in a similar way.

Calling *tomMenu*) by typing *Result = tomMenu* will return a structure array containing the *Result* structures of all the runs made. As an example, to display the results from the third run, enter the command *Result(3)*. To display the solution found in the third run, enter the command *Result(3).x_k*. The information stored in the structure are given in Table 46.

The menu program calls the driver routine *tomRun*.

There are some options in the menu programs to display graphical information for the selected problem. For two-dimensional nonlinear unconstrained problems, the menu programs support graphical display of the relevant optimization problem as mesh or contour plots. In the contour plot, the iteration steps are displayed. For higher-dimensional problems, iterations steps are displayed in two-dimensional subspaces. Special plots for nonlinear least squares problems, such as plotting model against data, are available. The plotting utility also includes plot of convergence rate, plot of circles approximating points in the plane for the Circle Fitting Problem etc. The plot facilities are exactly the same as for the GUI. See Section 11 for figures similar to the ones produced running the menu system.

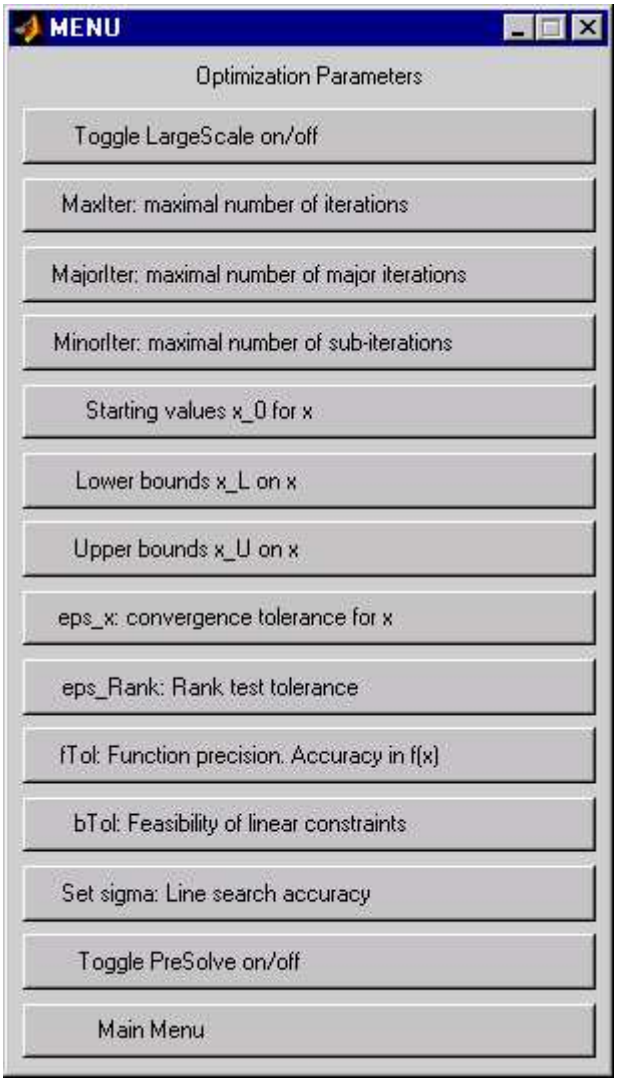


Figure 18: Setting optimization parameters for quadratic programming.

13 The TOMLAB Routines - Detailed Descriptions

13.1 The TOM Solvers

Detailed descriptions of the TOM solvers, driver routines and some utilities are given in the following sections. Also see the M-file help for each solver.

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

13.1.1 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)

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. 4: Gauss-Newton
	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).
	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.
	USER.r	Name of m-file computing the residual vector $r(x)$.
	USER.J	Name of m-file computing the Jacobian matrix $J(x)$.
	PriLevOpt	Print Level.
	LargeScale	Set to 1 to use *sqr2*, a sparse QR method for efficient storage of the Q matrix. Only applicable if *Prob.Solver.Method* = 0 (default).
	optParam	Structure with special fields for optimization parameters, see Table 39. Fields used are: *bTol*, *eps_absf*, *eps_g*, *eps_Rank*, *eps_x*, *IterPrint*, *MaxIter*, *PreSolve*, *size_f*, *size_x*, *xTol*, *wait*, and *QN_InitMatrix*.
	LineParam	Structure with line search parameters, see Table 38.
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.
	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 47.
	bState	State of each linear constraint, described in Table 48.
	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.
		2: Projected gradient small.
		4: Function value close to 0.
		8: Relative function value reduction low for $LowIts = 10$ iterations.
		32: Local minimum with all variables on bounds.
		101: Maximum number of iterations reached.
		102: Function value below given estimate.
		104: *x_k* not feasible, constraint violated.
		104: The residual is empty, no NLLS problem.
	Solver	Solver used.
	SolverAlgorithm	Solver algorithm used.
	Prob	Problem structure used.

Description

The solver *clsSolve* includes four optimization methods for nonlinear least squares problems: the Gauss-Newton method, the Al-Baali-Fletcher [5] and the Fletcher-Xu [24] hybrid method, and the Hushens TSSM method [59]. 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 [25]. Bound constraints are partly treated as described in Gill, Murray and Wright [34]. *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.

13.1.2 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)

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 94. 0,1,2: Schittkowski SQP. 3,4: Han-Powell SQP.
NumDiff	How to obtain derivatives (gradient, Hessian).
ConsDiff	How to obtain the constraint derivative matrix.
AutoDiff	If true, use automatic differentiation.
LineParam	Structure with line search parameters. See Table 38.
optParam	Structure with optimization parameters, see Table 39. Fields used are: *bTol*, *cTol*, *eps_absf*, *eps_g*, *eps_x*, *eps_Rank*, *IterPrint*, *MaxIter*, *QN_InitMatrix*, *size_f*, *size_x*, *xTol* and *wait*.
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.
USER.f	Name of m-file computing the objective function $f(x)$.
USER.g	Name of m-file computing the gradient vector $g(x)$.
USER.H	Name of m-file computing the Hessian matrix $H(x)$.
USER.c	Name of m-file computing the vector of constraint functions $c(x)$.
USER.dc	Name of m-file computing the matrix of constraint normals $\partial c(x)/dx$.
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.
	v_k	Lagrange multipliers.
	c_k	Value of constraints at optimum.
	cJac	Constraint Jacobian at optimum.
	xState	State of each variable, described in Table 47 .
	bState	State of each linear constraint, described in Table 48.
	cState	State of each general constraint.
	x_0	Starting point.
	f_0	Function value at start.
	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. 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 [79]. 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(USER.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*

13.1.3 cutPlane

Purpose
Solve mixed integer linear programming problems (MIP).

cutplane solves problems of the form

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

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)

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_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: *lpSolve* 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. See below for usage instructions.
optParam	Structure with special fields for optimization parameters, see Table 39. 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:	
	Iter	Number of iterations.
	ExitFlag	0: OK.
		1: Maximal number of iterations reached.
		2: Unbounded feasible region.
		3: Rank problems. Can not find any solution point.
		4: No feasible point *x_0* found.
		5: Illegal *x_0*.
	Inform	If $ExitFlag > 0$, $Inform = ExitFlag$.
	QP.B	Optimal active set. See input variable *QP.B*.
	f_0	Function value at start.
	f_k	Function value at optimum.
	g_k	Gradient value at optimum, *c*.
	x_0	Starting point.
	x_k	Optimal point.
	v_k	Lagrange multipliers.
	xState	State of each variable, described in Table 47 .
	Solver	Solver used.
	SolverAlgorithm	Solver algorithm used.
	FuncEv	Number of function evaluations. Equal to *Iter*.
	ConstrEv	Number of constraint evaluations. Equal to *Iter*.
	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 *lpSolve* and *DualSolve* to solve relaxed subproblems. *cutplane* 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.

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

- Scalar value $N \leq n$: variables $x_1, x_2, \ldots, x_N$ are restricted to integer values.
- 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.

Examples

See *exip39*, *exknap*, *expkorv*.

M-files Used

lpSolve.m, *DualSolve.m*

See Also

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

13.1.4 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:	
	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.
	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 39. Fields used are: *MaxIter*, *PriLev*, *wait*, *eps_f*, *eps_Rank* and *xTol*.
	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*.

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	Iter	Number of iterations.
	QP.B	Optimal active set.
	ExitFlag	Exit flag:
		0: OK.
		1: Maximal number of iterations reached. No primal feasible solution found.
		2: Infeasible Dual problem.
		3: No dual feasible starting point found.
		4: Illegal step length due to numerical difficulties. Should not occur.
		5: Too many active variables in initial point.
	f_k	Function value at optimum.
	x_0	Starting point.
	x_k	Optimal primal solution x.
	v_k	Optimal dual parameters. Lagrange multipliers for linear constraints.
	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 [41, 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

lpSolve

13.1.5 ego

Purpose
Solve box-bounded constrained global nonlinear optimization problems.

ego solves problems of the form:

$$\begin{array}{lccccc} \min\limits_{x} & f(x) & & & & \\ \text{subject to} & x_L & \leq & x & \leq & x_U, \quad x_L \text{ and } x_U \text{ finite} \\ & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \end{array}$$

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

$f(x)$ is assumed to be a costly function while the constraints $c(x)$ are assumed to be cheaply computed. If some subset of the constraints, $c_I(x)$, is very costly, create $\hat{f}(x)$ as a penalty function:

$$\hat{f}(x) = f(x) + \beta^T c_I(x), \quad \beta \text{ positive penalties}$$

Calling Syntax
Result=ego(Prob,varargin)

Description of Inputs

Prob		Problem description structure. The following fields are used:
	USER.f	String giving the name of the function to compute the objective function.
	USER.c	String giving the name of the function to compute the nonlinear constraint vector.
	x_L	Lower bounds for each element in x.
	x_U	Upper bounds for each element in x.
	b_L	Lower bounds for the linear constraints.
	b_U	Upper bounds for the linear constraints.
	A	Linear constraint matrix.
	c_L	Lower bounds for the nonlinear constraints.
	c_U	Upper bounds for the nonlinear constraints.
	PriLevOpt	Print level.
	Name	Problem name. Used for safety check when doing warm starts.
	WarmStart	If true (> 0), *ego* reads the output from the last run from the mat-file *cgoSave.mat* and resumes optimization from where the last run ended. *ego* and *rbfSolve* (page 127) uses the same mat-file and can read the output of one another.
	optParam	Structure with fields for optimization parameters. Used fields are:
	IterPrint	Print one line of information each iteration, including the new x tried. Default 1.
	MaxIter	Maximum number of iterations used in the global optimization on the response surface in each step. Default 10000.
	MaxFunc	Maximum number of function evaluations in *ego*. Default 200. If doing a warm start and *MaxFunc* $\leq$ *nFunc* (present # of $f(x)$ calls) then *MaxFunc* = *MaxFunc* +*nFunc*.
	fGoal	Goal for function value, not used if empty.
	eps_f	Relative accuracy for function value. Stop if $\lvert f - fGoal \rvert \leq \lvert fGoal \rvert \cdot eps\_f$, if *fGoal* is nonzero. Stop if $\lvert f - fGoal \rvert \leq eps\_f$, if *fGoal* is zero.
	cTol	Nonlinear constraint tolerance.

Description of Inputs, continued.

Prob Fields used in input argument *Prob*:

CGO Structure (*Prob.CGO*) with parameters concerning global optimization options. The following fields are used:

SCALE 0 - original search space.
1 - transform search space to unit cube (default).

PLOT 0 - no plotting (default).
1 - Plot sampled points.

REPLACE 0 - No replacement.
1 - Large function values are replaced by the median (default).

globalSolver Name of solver used for global optimization on the response surface.

localSolver Name of solver user for local optimization on the response surface.

Percent Strategy to get initial sampled values.
$Percent \geq 100$:
User gives initial points x as a matrix in *CGO.X*.
Each column is one sampled point. The user must supply at least $d+1$ points:
If $d = length(Prob.x)$, then $size(X,1) = d,\ size(X,2) \geq d+1)$ must hold.
CGO.F should be defined as empty, or contain a vector of corresponding $f(x)$ values. Any *CGO.F* value set as NaN will be computed by *rbfSolve*.

$0 < Percent < 100$: Random strategy, the *Percent* value gives the percentage size of an ellipsoid around the so far sampled points that the new points are not allowed in.
Range 1%-50%. Recommended values 10% - 20%.

$Percent = 0$: Initial points are the corner points of the box $x_U - x_L$
Generates too many points if the dimension is high.

$Percent < 0$: Latin hypercube space-filling design.
$|Percent|$ should in principle be the dimension.
The call made is X = daceInit(round(abs(Percent)),Prob.x_L,Prob.x_U);
See the help of *daceInit.m*.

varargin Other arguments sent directly to low level functions.

Description of Outputs

Result Structure with results from optimization.

x_k	Matrix containing the best points as columns.
f_k	Vector with function values corresponding to *x_k*.
Iter	Number of iterations used.
FuncEv	Number of function evaluations.
ExitText	Text string giving information about

cgoSave.mat MATLAB mat-file saved to current directory, used for warmstarts. This file can be read by *rbfSolve* as well.
The file contains the following variables:

Name	Problem name. Checked against the *Prob.Name* field if doing a warmstart.
O	Matrix with sampled points (in original space).
X	Matrix with sampled points (in unit space if SCALE==1)
F	Vector with function values.
F_m	Vector with function values (replaced).
nInit	Number of initial points.

Description

ego implements the algorithm EGO by D. R. Jones, Matthias Schonlau and William J. Welch presented in the paper "Efficient Global Optimization of Expensive Black-Box Functions" [63].

Please note that Jones et al. has a slightly different problem formulation. The TOMLAB version of *ego* treats linear and nonlinear constraints separately.

ego samples points to which a response surface is fitted. The algorithm then balances between sampling new points and minimization on the surface.

ego and *rbfSolve* (page 127) use the same format for saving warm start data. This means that it is possible to try one solver for a certain number of iterations/function evaluations and then do a warm start with the other. Example:

```
>> Prob           = probInit('glc_prob',1);  % Set up problem structure
>> Result_ego     = tomRun('ego',Prob);      % Solve for a while with ego
>> Prob.WarmStart = 1;                       % Indicate a warm start
>> Result_rbf     = tomRun('rbfSolve',Prob); % Warm start with rbfSolve
```

M-files Used

iniSolve.m, *endSolve.m*, *conAssign.m*, *glcAssign.m*

See Also

rbfSolve

References

13.1.6 expSolve

Purpose
Quick solution to exponential fitting problems.

Calling Syntax
Result = expSolve(p, Name, t, y, wType, eType, SepAlg, x_0, Solver)

Description of Inputs

p	Number of exponential terms.
Name	Name of problem.
t	Time steps.
y	Observations (must be same length as *t*).
wType	Weight type: 1 = weight with data, 0 = no weighting.
eType	Exponential function type (default 1), see Table 13, page 64.

Optional parameters:

SepAlg	1 = Use separable least squares (default 0).
x_0	Initial values. If empty, initial value algorithm is used.
Solver	Name of TOMLAB solver to use. Selected depending on license if empty.

Description of Outputs

Result	TOMLAB Result structure as returned by solver selected by input argument *Solver*.
LS	Statistical information about the solution. See Table 50, page 188.

Global Parameters Used

Description
expSolve formulates a **cls** (constrained least squares) problem for exponential fitting applications. The problem is solved with a suitable **cls** solver.

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

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

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

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

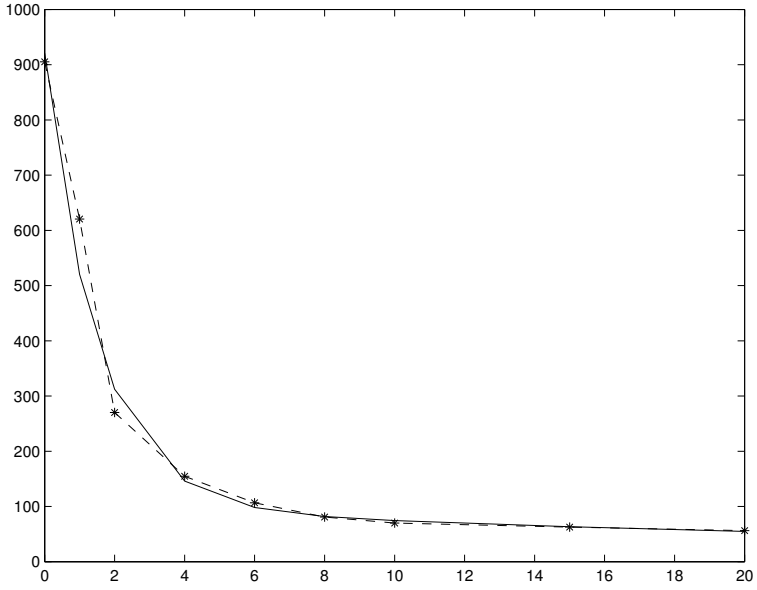
give the following commands:

```
>> p      = 2;                          % Two terms
>> Name   = 'Simple two-term exp fit'; % Problem name, can be anything
>> wType  = 0;                          % No weighting
>> eType  = 1;                          % Exponential model 1

>> Result = expSolve(p,Name,t,y,wType,eType);
>> 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 19: Results of fitting experimental data to two-term exponential model. Solid line: final model, dash-dot: data.



13.1.7 glbSolve

Purpose
Solve box-bounded global optimization problems.

glbSolve solves problems of the form

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

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

Calling Syntax
Result = glbSolve(Prob,varargin)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	optParam	Structure with special fields for optimization parameters, see Table 39. Fields used are: *PriLev*.
	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.
	USER.f	Name of m-file computing the objective function $f(x)$.
	GLOBAL	Special structure field containing:
	iterations	Number of iterations, default 50.
	epsilon	Global/local weight parameter, default 10^{-4}.
	K	The Lipschitz constant. Not used.
	tolerance	Error tolerance parameter. Not used.
	If restart is chosen in the menu system, the following fields in *GLOBAL* 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.
	L	Matrix with all rectangle side lengths in each dimension.
	F	Vector with function values.
	d	Row vector of all different distances, sorted.
	d_min	Row vector of minimum function value for each distance.
varargin	Other parameters directly sent to low level routines.	

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	Iter	Number of iterations.
	FuncEv	Number function evaluations.
	x_k	Matrix with all points giving the function value *f_k*.
	f_k	Function value at optimum.
	GLOBAL	Special structure field containing:
	C	Matrix with all rectangle centerpoints.
	D	Vector with distances from centerpoint to the vertices.
	L	Matrix with all rectangle side lengths in each dimension.
	F	Vector with function values.
	d	Row vector of all different distances, sorted.
	d_min	Row vector of minimum function value for each distance.
	Solver	Solver used.
	SolverAlgorithm	Solver algorithm used.

Description
The global optimization routine *glbSolve* is an implementation of the DIRECT algorithm presented in [61]. 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* which is an implementation of the algorithm GRAHAMHULL in [75, 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*

13.1.8 glbFast

Purpose
Solve box-bounded global optimization problems.

glbFast solves problems of the form

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

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

glbFast is a Fortran MEX implementation of *glbSolve*.

Calling Syntax
Result = glbFast(Prob,varargin)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	optParam	Structure with special fields for optimization parameters, see Table 39. Fields used are: *PriLev*.
	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.
	USER.f	Name of m-file computing the objective function $f(x)$.
	GLOBAL	Special structure field containing:
	iterations	Number of iterations, default 50.
	epsilon	Global/local weight parameter, default 10^{-4}.
	K	The Lipschitz constant. Not used.
	tolerance	Error tolerance parameter. Not used.
	If restart is chosen in the menu system, the following fields in *GLOBAL* 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.
	L	Matrix with all rectangle side lengths in each dimension.
	F	Vector with function values.
	d	Row vector of all different distances, sorted.
	d_min	Row vector of minimum function value for each distance.
varargin	Other parameters directly sent to low level routines.	

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	Iter	Number of iterations.
	FuncEv	Number function evaluations.
	x_k	Matrix with all points giving the function value *f_k*.
	f_k	Function value at optimum.
	GLOBAL	Special structure field containing:
	C	Matrix with all rectangle centerpoints.
	D	Vector with distances from centerpoint to the vertices.
	L	Matrix with all rectangle side lengths in each dimension.
	F	Vector with function values.
	d	Row vector of all different distances, sorted.
	d_min	Row vector of minimum function value for each distance.
	Solver	Solver used.
	SolverAlgorithm	Solver algorithm used.

Description
The global optimization routine *glbFast* is an implementation of the DIRECT algorithm presented in [61]. The algorithm in *glbFast* 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 *glbFast* 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** *glbFast* with the final status of all parameters from the previous run, a so called

warm start Assume that a run has been made with *glbFast* 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 *glbFast* a flag *Prob.WarmStart* should be set to one. Then *glbFast* is using output previously written to the file *glbFastSave.mat* to make the restart. *glbFast* also includes the subfunction *conhull* which is an implementation of the algorithm GRAHAMHULL in [75, 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*.

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

Calling Syntax
Result = glcSolve(Prob,varargin)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	optParam	Structure with special fields for optimization parameters, see Table 39. Fields used are: *PriLev*, *cTol*.
	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.
	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.
	USER.f	Name of m-file computing the objective function $f(x)$.
	USER.c	Name of m-file computing the vector of constraint functions $c(x)$.
	PriLevOpt	Print level.
	WarmStart	If true (> 0), *glcSolve* reads the output from the last run from the mat-file *glcSave.mat*, and continues from the last run.
	MIP	Structure in *Prob*, *Prob.MIP*. Only field used is *Intvars* : set of integer variables.
	GO	Structure in *Prob*, *Prob.GO*. Fields used:
	fEqual	All points with function values within tolerance *fEqual* are considered to be global minima and returned.
	LinWeight	$RateOfChange = LinWeight \cdot \lvert a(i,:) \rvert$ for linear constraints. Balance between linear and nonlinear constraints. Default value 0.1.

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	Iter	Number of iterations.
	FuncEv	Number function evaluations.
	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*.
	GLOBAL	Special structure field containing:
	C	Matrix with all rectangle centerpoints.
	D	Vector with distances from centerpoint to the vertices.
	F	Vector with function values.
	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.
	G	Matrix with constraint values for each point.
	ignoreidx	Rectangles to be ignored in the rectangle selection procedure.
	I_L	$I\_L(i, j)$ is the lower bound for rectangle j in integer dimension $I(i)$.
	I_U	$I\_U(i, j)$ is the upper bound for rectangle j in integer dimension $I(i)$.
	feasible	Flag indicating if a feasible point has been found.
	f_min	Best function value found at a feasible point.
	s_0	s_0 is used as $s(0)$.
	s	$s(j)$ is the sum of observed rates of change for constraint j.
	t	$t(i)$ is the total number of splits along dimension i.
	Solver	Solver used.
	SolverAlgorithm	Solver algorithm used.

Description

The routine *glcSolve* implements an extended version of DIRECT, see [62], 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*

13.1.10 glcFast

Purpose
Solve global mixed-integer nonlinear programming problems.

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

glcFast is a Fortran MEX implementation of *glcSolve*.

Calling Syntax
Result = glcFast(Prob,varargin)

Description of Inputs

Prob Problem description structure. The following fields are used:

Name	Problem name. Used for safety when doing warm starts.
WarmStart	Set to 1 makes *glcFast* read data from *glcFastSave.mat* and resume optimization from where previous run ended. See below for details.
USER.f	Name of m-file computing the objective function $f(x)$.
USER.c	Name of m-file computing the vector of constraint functions $c(x)$.
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.
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.
PriLev	Print level.
optParam	Structure with special fields for optimization parameters, see Table 39 on page 183. Fields used by *glcFast* are: *IterPrint*, *bTol*, *cTol*, *MaxIter*, *MaxFunc*, *EpsGlob*, *fGoal*, *eps_f*, *eps_x*.
GO	Structure with special fields for global optimization parameters. The fields used are:
fEqual	All points with function values within tolerance fEqual are considered to be global minima and returned.
LinWeight	Controls balance between linear and nonlinear constraints. Default 0.1.
MIP	Structure for integer optimization parameters. Currently, the only field used is:
IntVars	Set of integer variables, default empty ([ ]). To make the tree search more efficient, it is recommended to number the integer values as the first variables.

varargin Other parameters directly sent to low level routines.

If restart is chosen in the menu system, the following fields in *GLOBAL* are also used and contains information

	C	Matrix with all rectangle centerpoints.
	D	Vector with distances from centerpoint to the vertices.
	F	Vector with function values.
	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.
	G	Matrix with constraint values for each point.
from the previous run:	*ignoreidx*	Rectangles to be ignored in the rectangle selection procedure.
	I_L	$I\_L(i,j)$ is the lower bound for rectangle j in integer dimension $I(i)$.
	I_U	$I\_U(i,j)$ is the upper bound for rectangle j in integer dimension $I(i)$.
	feasible	Flag indicating if a feasible point has been found.
	f_min	Best function value found at a feasible point.
	s_0	s_0 is used as $s(0)$.
	s	$s(j)$ is the sum of observed rates of change for constraint j.
	t	$t(i)$ is the total number of splits along dimension i.

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	Iter	Number of iterations.
	FuncEv	Number function evaluations.
	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.
	GLOBAL	Special structure field containing:
	C	Matrix with all rectangle centerpoints.
	D	Vector with distances from centerpoint to the vertices.
	F	Vector with function values.
	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.
	G	Matrix with constraint values for each point.
	ignoreidx	Rectangles to be ignored in the rectangle selection procedure.
	I_L	$I\_L(i,j)$ is the lower bound for rectangle j in integer dimension $I(i)$.
	I_U	$I\_U(i,j)$ is the upper bound for rectangle j in integer dimension $I(i)$.
	feasible	Flag indicating if a feasible point has been found.
	f_min	Best function value found at a feasible point.
	s_0	s_0 is used as $s(0)$.
	s	$s(j)$ is the sum of observed rates of change for constraint j.
	t	$t(i)$ is the total number of splits along dimension i.
	Solver	Solver used.
	SolverAlgorithm	Solver algorithm used.

Description

The routine *glcFast* implements an extended version of DIRECT, see [62], that handles problems with both nonlinear and integer constraints. The algorithm in *glcFast* 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 *glcFast* 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** *glcFast* with the final status of all parameters from the previous run, a so called *warm start* Assume that a run has been made with *glcFast* 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 *glcFast* a flag *Prob.WarmStart* should be set to one. Then *glcFast* is using output previously written to the file *glcFastSave.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 glbSolve.m*.

Warnings

A significant portion of *glcFast* 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 behaviour if *glcFast* is started again. To reduce the risk of trouble, do "`clear mex`" if a run has been aborted.

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

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

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.
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.
USER.f	Name of m-file computing the objective function $f(x)$.
USER.c	Name of m-file computing the vector of constraint functions $c(x)$.
PriLevOpt	Print level. 0=silent. 1=warm start info. 2=output each iteration.
Name	Name of the problem. *glcCluster* uses the warmstart capability in *glcFast* and needs the name for security reasons.
optParam	Structure with special fields for optimization parameters, see Table 39. Fields used are: *PriLev*, *cTol*, *IterPrint*, *MaxIter*, *MaxFunc*, *EpsGlob*, *eps_f*, *eps_x*.
MIP	Structure in *Prob*, *Prob.MIP*. Only field used is *Intvars* : set of integer variables.
GO	Structure in *Prob*, *Prob.GO*. Fields used:
maxFunc1	Maximum number of function evaluations in first and second calls to *glcFast*.
maxFunc2	Can also be specified as 2nd and 3rd parameters in call to *glcCluster*.
fEqual	All points with function values within tolerance *fEqual* are considered to be global minima and returned.
LinWeight	$RateOfChange = LinWeight \cdot \lvert a(i,:) \rvert$ for linear constraints. Balance between linear and nonlinear constraints. Default value 0.1.

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	Iter	Number of iterations.
	FuncEv	Number function evaluations.
	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*.
	GLOBAL	Special structure field containing:
	C	Matrix with all rectangle centerpoints.
	D	Vector with distances from centerpoint to the vertices.
	F	Vector with function values.
	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.
	G	Matrix with constraint values for each point.
	ignoreidx	Rectangles to be ignored in the rectangle selection procedure.
	I_L	$I\_L(i,j)$ is the lower bound for rectangle j in integer dimension $I(i)$.
	I_U	$I\_U(i,j)$ is the upper bound for rectangle j in integer dimension $I(i)$.
	feasible	Flag indicating if a feasible point has been found.
	f_min	Best function value found at a feasible point.
	s_0	s_0 is used as $s(0)$.
	s	$s(j)$ is the sum of observed rates of change for constraint j.
	t	$t(i)$ is the total number of splits along dimension i.
	Solver	Solver used.
	SolverAlgorithm	Solver algorithm used.

Description

The routine *glcCluster* implements an extended version of DIRECT, see [62], 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*

13.1.12 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:

$$\begin{array}{lllll} \min\limits_{x} & z & & & \\ & & & & \\ \text{subject to} & x_L & \leq \quad (x_1, x_2, \ldots, x_n)^T & \leq & x_U \\ & -\infty & \leq \quad z & \leq & \infty \\ & b_L & \leq \quad Ax & \leq & b_U \\ & c_L & \leq \quad c(x) & \leq & c_U \\ & -\infty & \leq \quad r(x) - ze & \leq & 0 \end{array}$$

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.

13.1.13 lpSolve

Purpose
Solve general linear programming problems.

lpSolve solves problems of the form

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

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 = lpSolve(Prob)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	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 39. Fields used are: *MaxIter*, *PriLev*, *wait*, *eps_f*, *eps_Rank*, *xTol* and *bTol*.
	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.

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	Iter	Number of iterations.
	ExitFlag	0: OK. 1: Maximal number of iterations reached. 2: Unbounded feasible region. 3: Rank problems. Can not find any solution point. 4: Illegal *x_0*. 5: No feasible point *x_0* found.
	Inform	If $ExitFlag > 0$, $Inform = ExitFlag$.
	QP.B	Optimal active set. See input variable *QP.B*.
	f_0	Function value at start.
	f_k	Function value at optimum.
	g_k	Gradient value at optimum, *c*.
	x_0	Starting point.
	x_k	Optimal point.
	v_k	Lagrange multipliers.
	xState	State of each variable, described in Table 47 .
	Solver	Solver used.
	SolverAlgorithm	Solver algorithm used.
	FuncEv	Number of function evaluations. Equal to *Iter*.
	ConstrEv	Number of constraint evaluations. Equal to *Iter*.
	Prob	Problem structure used.

Description
The routine *lpSolve* implements an active set strategy (Simplex method) for Linear Programming using an addi-

tional 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

13.1.14 L1Solve

Purpose
Find a constrained L1 solution of a function of several variables with the use of any suitable nonlinear TOMLAB solve.

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:

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

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*.

$= 0$	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

$$\begin{array}{lrcccl} \min\limits_{x} & \sum_i (y_i + z_i) & & & & \\ \text{subject to} & x_L & \leq & x & \leq & x_U \\ & 0 & \leq & y & \leq & \infty \\ & 0 & \leq & z & \leq & \infty \\ & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \\ & 0 & \leq & r(x) + y - z & \leq & 0 \end{array}$$

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

13.1.15 mipSolve

Purpose
Solve mixed integer linear programming problems (MIP).

mipSolve solves problems of the form

$$\begin{array}{lccccc} \min\limits_{x} & f(x) & = & c^T x & & \\ s/t & x_L & \leq & x & \leq & x_U \\ & & & Ax & = & b \\ & & & x_j \in \mathbb{N} & \forall j \in \mathrm{I} & \end{array}$$

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

Please note that the linear constraints $Ax = b$ is different from most other TOMLAB solvers. The user must use slack variables to handle inequality constraints. The right hand side b of the linear constraint expression should be given as input argument *Prob.b_U*. *mipSolve* ignores any value supplied in *Prob.b_L*.

Calling Syntax
Result = 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. **NOTE:** ignored by *mipSolve*.
b_U	Upper bounds on the linear constraints. Should be the vector b in the problem formulation.
x_L	Lower bounds on the variables.
x_U	Upper bounds on the variables.
x_0	Starting point.
QP.B	Active set *B_0* at start: $B(i) = 1$: Include variable $x(i)$ 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*.
PriLevOpt	Print level in *lpSolve* 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 IntVars is a scalar, then variables $1, \ldots, IntVars$ are assumed to be integers. 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.
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.

Description of Inputs, continued.

Prob	Problem description structure, continued.	
	optParam	Structure with special fields for optimization parameters, see Table 39. 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/lpSolve: 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:	
	Iter	Number of iterations.
	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.
	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.
	f_0	Function value at start.
	f_k	Function value at optimum.
	g_k	Gradient value at optimum, c.
	x_0	Starting point.
	x_k	Optimal point.
	v_k	Lagrange multipliers, [Constraints + lower + upper bounds].
	xState	State of each variable, described in Table 47, page 188.
	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 [69, chap. 8.2]. *mipSolve* normally uses the linear programming routines *lpSolve* 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 [69, chap. 8.2] and the code in *mipSolve.m*.

Examples

See *exip39*, *exknap*, *expkorv*.

M-files Used

lpSolve.m, *DualSolve.m*, *GetSolver.m*, *tomSolve.m*

See Also

cutplane, *balas*, *SolveLP*, *SolveDLP*

13.1.16 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)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	optParam	Structure with special fields for optimization parameters, see Table 39. Fields used are: *eps_g*, *eps_c*, *eps_x*, *MaxIter*, *wait*, *size_x*, *PriLev*, *method* and *QN_InitMatrix*.
	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.
	USER.f	Name of m-file computing the objective function $f(x)$.
	USER.g	Name of m-file computing the gradient vector $g(x)$.
	USER.H	Name of m-file computing the Hessian matrix $H(x)$.
	USER.c	Name of m-file computing the vector of constraint functions $c(x)$.
	USER.dc	Name of m-file computing the matrix of constraint normals $\partial c(x)/dx$.
	USER.d2c	Name of m-file computing the second derivatives of the constraints, weighted by an input Lagrange vector
varargin	Other parameters directly sent to low level routines.	

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	Iter	Number of iterations.
	ExitFlag	Flag giving exit status.
	ExitFlag	0: Convergence. Small step. Constraints fulfilled.
		1: Infeasible problem?
		2: Maximal number of iterations reached.
	Inform	Type of convergence.
	f_0	Function value at start.
	f_k	Function value at optimum.
	g_k	Gradient value at optimum.
	H_k	Hessian value at optimum.
	x_0	Starting point.
	x_k	Optimal point.
	v_k	Lagrange multipliers.
	c_k	Value of constraints at optimum.
	cJac	Constraint Jacobian at optimum.
	xState	State of each variable, described in Table 47 .
	bState	State of each linear constraint, described in Table 48.
	cState	State of each general constraint.
	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 [26].

M-files Used

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

See Also

conSolve, *sTrustr*

13.1.17 qpSolve

Purpose
Solve general quadratic programming problems.

qpSolve solves problems of the form

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

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)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	optParam	Structure with special fields for optimization parameters, see Table 39. Fields used are: *eps_f*, *eps_Rank*, *MaxIter*, *wait*, *bTol* and *PriLev*.
	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.

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	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.
	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.
	f_0	Function value at start.
	f_k	Function value at optimum.
	g_k	Gradient value at optimum.
	H_k	Hessian value at optimum.
	x_0	Starting point.
	x_k	Optimal point.
	v_k	Lagrange multipliers.
	xState	State of each variable, described in Table 47 .
	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 *lpSolve*. The routine is the standard QP solver used by *nlpSolve*, *sTrustr* and *conSolve*.

M-files Used

ResultDef.m, *lpSolve.m*, *tomSolve.m*, *iniSolve.m*, *endSolve.m*

See Also
qpBiggs, *qpe*, *qplm*, *nlpSolve*, *sTrustr* and *conSolve*

13.1.18 rbfSolve

Purpose
Solve general constrained global optimization problems with costly objective functions.

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

$f(x)$ is assumed to be a costly function while $c(x)$ is assumed to be cheaply computed.

If some subset $c_I(x)$ of the constraints is very costly, create F(x) as a penalty function as $F(x) = f(x) + \beta^T c_I(x)$, β positive penalties.

Calling Syntax
Result = rbfSolve(Prob,varargin)

Description of Inputs

Prob	Problem description structure. The following fields are used:
USER.f	Name of function to compute the objective function.
USER.c	Name of function to compute the nonlinear constraint vector.
x_L	Lower bounds on the variables. Must be finite.
x_U	Upper bounds on the variables. Must be finite.
A	Linear constraint matrix.
b_U	Upper bounds for the linear constraints.
b_L	Lower bounds for the linear constraints.
c_U	Upper bounds for the nonlinear constraints.
c_L	Lower bounds for the nonlinear constraints.
Name	Name of the problem. Used for security when doing warm starts.
WarmStart	Set true (non-zero) to load data from previous run from *cgoSave.mat*.
PriLevOpt	Print Level.
optParam	Structure with optimization parameters. The following fields are used:
IterPrint	Print one line of information each iteration (default 1)
MaxFunc	Maximum number of function evaluations allowed.
MaxIter	Maximum number of iterations.
fGoal	Goal for function value, not used if inf or empty.
eps_f	Relative accuracy for function value.
cTol	Nonlinear constraint tolerance.

Description of Inputs, continued.

Fields used in input argument *Prob*:

CGO Structure (*Prob.CGO*) with parameters concerning global optimization options. The following fields are used:

idea Type of search strategy on the response surface.
1 - cycle of 6 points in target value *fnStar*
2 - cycle of 4 points in *alpha* (default).

rbfType Type of radial basis function: 1 - thin plate spline; 2 - Cubic Spline (default).

SCALE 0 - original search space.
1 - transform search space to unit cube (default).

PLOT 0 - no plotting (default).
1 - Plot sampled points.

REPLACE 0 - No replacement.
1 - Large function values are replaced by the median (default).

globalSolver Name of solver used for global optimization on the RBF surface.

localSolver Name of solver user for local optimization on the RBF surface.

Percent Strategy to get initial sampled values.
$Percent \geq 100$:
User gives initial points x as a matrix in *CGO.X*.
Each column is one sampled point. The user must supply at least $d+1$ points:
If $d = length(Prob.x)$, then $size(X,1) = d,\ size(X,2) \geq d+1)$ must hold.
CGO.F should be defined as empty, or contain a vector of corresponding $f(x)$ values. Any *CGO.F* value set as NaN will be computed by *rbfSolve*.

$0 < Percent < 100$: Random strategy, the *Percent* value gives the percentage size of an ellipsoid around the so far sampled points that the new points are not allowed in. Range 1%-50%. Recommended values 10% - 20%.

$Percent = 0$: Initial points is the corner points of the box Generates too many points if the dimension is high.

$Percent < 0$: Latin hypercube space-filling design. The value of $abs(Percent)$ should in principle be the dimension. The call made is X = daceInit(round(abs(Percent)),Prob.x_L,Prob.x_U); See the help of *daceInit.m*.

varargin Other parameters are sent directly to low level routines.

Description of Outputs

Result Structure with optimization results. The following fields are changed:

x_k	Matrix with the best points as columns.
f_k	The best function value found so far.
Iter	Number of iterations.
FuncEv	Number of function evaluations.
ExitText	Text string with information about the run.

cgoSave.mat To make warm starts possible, *rbfSolve* saves the following information in the file *cgoSave.mat*:

Name	Name of the problem. Used for safety check so that the warm start is not made with data from a different problem.
O	Matrix with sampled points (in original space).
X	Matrix with sampled points (in unit space, if *Prob.CGO.SCALE* ==1).
F	Vector with function values.
F_m	Vector with function values (replaced).
nInit	Number of initial points: $nInit \geq d+1,\ 2^d$ if center points.

Description

rbfSolve implements the Radial Basis Function (RBF) algorithm presented in [12], augmented to handle linear and non-linear constraints. The method is based on [43].

A response surface based on radial basis functions is fitted to a collection of sampled points. The algorithm then balances between minimizing the fitted function and adding new points to the set.

M-files Used

daceInit.m, *iniSolve.m*, *endSolve.m*, *conAssign.m*, *glcAssign.m*

MEX-files Used

tomsol

See Also

ego.m

Warnings

If *rbfSolve* is called but is interrupted by the user, or fails, for example due to incorrect input data, it is necessary to issue one of the following commands:

- clear mex
- clear all
- tomsol(25)

The reason for this is that the MEX-file solver *tomsol* is used by *rbfSolve* for time-critical tasks and allocates memory which must be deallocated.

13.1.19 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:

$$\begin{array}{lccccc} \min\limits_{x} & \frac{1}{2} r(x)^T r(x) & & & & \\ \text{subject to} & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \end{array}$$

where $x, x_L, x_U \in \mathbb{R}^n$, $r(x) \in \mathbb{R}^m$, $A \in \mathbb{R}^{m_1,n}$, $b_L, b_U \in \mathbb{R}^{m_1}$ 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:

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	Linear constraint matrix.
c_L	Upper bounds on the nonlinear constraints.
c_U	Lower bounds on the nonlinear constraints.

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 is *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 the reformulated problem.

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.
cJac	Jacobian of nonlinear constraints at optimum.
x_0	Starting point.
xState	State of variables at optimum.
cState	State of constraints at optimum.
v_k	Lagrange multipliers.
g_k	The gradient vector is calculated as $J\_k^T \cdot r\_k$.

Result.Prob is 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*

13.1.20 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:	
	optParam	Structure with special fields for optimization parameters, see Table 39. 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 40.
	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.
	USER.f	Name of m-file computing the objective function $f(x)$.
	USER.g	Name of m-file computing the gradient vector $g(x)$.
	USER.H	Name of m-file computing the Hessian matrix $H(x)$.
	USER.c	Name of m-file computing the vector of constraint functions $c(x)$.
	USER.dc	Name of m-file computing the matrix of constraint normals $\partial c(x)/dx$.
varargin	Other parameters directly sent to low level routines.	

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	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. 4: Relative function value reduction low for *LowIts* iterations. 8: Too small trust region. 101: Maximum number of iterations reached. 102: Function value below given estimate. 103: Convergence to saddle point (eigenvalues computed).
	f_0	Function value at start.
	f_k	Function value at optimum.
	g_k	Gradient value at optimum.
	H_k	Hessian value at optimum.
	x_0	Starting point.
	x_k	Optimal point.
	v_k	Lagrange multipliers.
	c_k	Value of constraints at optimum.
	cJac	Constraint Jacobian at optimum.
	xState	State of each variable, described in Table 47 .
	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 [18] and [77]. BFGS or DFP is used for the Quasi-Newton update, if the analytical Hessian is not used. *sTrustr* calls *itrr*.

Functions Used

itrr.m

M-files Used

qpSolve.m, *tomSolve.m*, *iniSolve.m*, *endSolve.m*

See Also

conSolve, *nlpSolve*, *clsSolve*, *ittr*

13.1.21 itrr

Purpose
Determine the initial trust region radius.

Calling Syntax
[D_0, f_0, x_0] = itrr(x_0, fS, gS, HS, jMax, iMax, Prob, varargin)

Description of Inputs

x_0	Starting point.
x_L	Lower bounds for x.
x_U	Upper bounds for x.
fS	String with function call sequence. x_k current point.
gS	String with gradient call sequence. x_k current point.
HS	String with Hessian call sequence. x_k current point.
jMax	Number of outer iterations, normally 1.
iMax	Number of inner iterations, normally 5.
Prob	Prob.PartSep.index is the index for the partial function to be analyzed.
varargin	Extra user parameters, passed to f, g and H;

Description of Outputs

D_0	Initial trust region radius.
f_0	Function value at the input starting point x_0.
x_0	Updated starting point, if $jMax > 1$.

Description
The routine *itrr* implements the *initial trust region radius* algorithm as described by Sartenaer in [77]. *itrr* is called by *sTrustr*.

See Also
sTrustr

13.1.22 ucSolve

Purpose
Solve unconstrained nonlinear optimization problems with simple bounds on the variables.

ucSolve solves problems of the form

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

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

Calling Syntax
Result = ucSolve(Prob, varargin)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	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.
	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 38.
	optParam	Structure with special fields for optimization parameters, see Table 39. 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*.
	x_L	Lower bounds on the variables.
	x_U	Upper bounds on the variables.
	x_0	Starting point.
	USER.f	Name of m-file computing the objective function $f(x)$.
	USER.g	Name of m-file computing the gradient vector $g(x)$.
	USER.H	Name of m-file computing the Hessian matrix $H(x)$.
	f_Low	Lower bound on function value.
	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.
	x_0	Starting point.
	f_k	Function value at optimum.
	f_0	Function value at start.
	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.
	xState	State of each variable, described in Table 47.
	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 [64]. 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 [25]. Bound constraints are treated as described in Gill, Murray and Wright [34].

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

13.1.23 pensdp

Purpose
Solve (linear) semi-definite programming problems.

pensdp solves problems of the form

$$\begin{array}{lllll} \min\limits_{x} & f(x) = c^T x & & & \\ s/t & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \\ & Q_i^{(0)} + \sum_{k=1}^{n} Q_k^{(i)} x_k & \preceq & 0, \quad k = 1, 2, \ldots, m_{LMI} & & \end{array} \tag{18}$$

where $x, x_L, x_U \in \mathbb{R}^n$, $A \in \mathbb{R}^{m_l \times n}$, $b_L, b_U \in \mathbb{R}^{m_l}$ and $Q_k^{(i)}$ are sparse or dense symmetric matrices. The matrix sizes may vary between different linear matrix inequalities (*LMI*) but must be the same in each particular constraint.

Calling Syntax
Result = tomRun('pensdp',Prob,...)

Description of Inputs

Prob Problem description structure. The following fields are used:

QP.c	Vector with coefficients for linear objective function.
A	Linear constraints matrix.
b_L	Lower bound for linear constraints.
b_U	Upper bound for linear constraints.
x_L	Lower bound on variables.
x_U	Upper bound on variables.
x_0	Starting point.

PENSDP Structure with special fields for SDP parameters. Fields used are:

LMI Structure array with matrices for the linear matrix inequalities. See *Examples* on page 139 for a discussion of how to set this correctly.

ioptions 8×1 vector with options, defaults in ().
Any element set to a value less than zero will be replaced by a default value, in some cases fetched from standard Tomlab parameters.

ioptions(1)	0/1: use default/user defined values for options.
ioptions(2)	Maximum number of iterations for overall algorithm (50). If not given, *Prob.optParam.MaxIter* is used.
ioptions(3)	Maximum number of iterations in unconstrained optimization (100). If not given, *Prob.optParam.MinorIter* is used.
ioptions(4)	Output level: 0/(1)/2/3 = silent/summary/brief/full. Tomlab parameter: *Prob.PriLevOpt*.
ioptions(5)	(0)/1: Check density of Hessian / Assume dense.
ioptions(6)	(0)/1: (Do not) use linesearch in unconstrained minimization.
ioptions(7)	(0)/1: (Do not) write solution vector to output file.
ioptions(8)	(0)/1: (Do not) write computed multipliers to output file.

foptions 1×7 vector with optimization parameters, defaults in ():

foptions(1)	Scaling factor linear constraints; must be positive. (1.0).
foptions(2)	Restriction for multiplier update; linear constraints (0.7).
foptions(3)	Restriction for multiplier update; matrix constraints (0.1).
foptions(4)	Stopping criterium for overall algorithm (10^{-7}). Tomlab equivalent: *Prob.optParam.eps_f*.
foptions(5)	Lower bound for the penalty parameters (10^{-6}).
foptions(6)	Lower bound for the multipliers (10^{-14}).
foptions(7)	Stopping criterium for unconstrained minimzation (10^{-2}).

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	Iter	Number of iterations.
	ExitFlag	0: OK.
		1: Maximal number of iterations reached.
		2: Unbounded feasible region.
		3: Rank problems. Can not find any solution point.
		4: Illegal *x_0*.
		5: No feasible point *x_0* found.
	Inform	If $ExitFlag > 0$, $Inform = ExitFlag$.
	QP.B	Optimal active set. See input variable *QP.B*.
	f_0	Function value at start.
	f_k	Function value at optimum.
	g_k	Gradient value at optimum, *c*.
	x_0	Starting point.
	x_k	Optimal point.
	v_k	Lagrange multipliers.
	xState	State of each variable, described in Table 47 .
	Solver	Solver used.
	SolverAlgorithm	Solver algorithm used.
	FuncEv	Number of function evaluations. Equal to *Iter*.
	ConstrEv	Number of constraint evaluations. Equal to *Iter*.
	Prob	Problem structure used.

Description

pensdp implements a penalty algorithm based on the PBM method of Ben-Tal and Zibulevsky. It is possible to give input data in three different formats:

- Standard sparse SPDA format
- PENSDP Structural format
- Tomlab Quick format

In all three cases, problem setup is done via *sdpAssign*.

See Also

sdpAssign.m, *sdpa2pen.m*, *sdpDemo.m*, *tomlab/docs/pensdp.pdf*

Examples
Setting the LMI constraints is best described by an example. Assume 3 variables $x = (x_1, x_2, x_3)$ and 2 linear matrix inequalities of sizes 3×3 and 2×2 respectively, here given on block-diagonal form:

$$\left(\begin{array}{ccc|c} 0 & & & \\ & 0 & & \\ & & 0 & \\ \hline & & & \begin{matrix} 0 & \\ & 1 \end{matrix} \end{array}\right) + \left(\begin{array}{ccc|c} 2 & -1 & 0 & \\ & 2 & 0 & \\ & & 2 & \\ \hline & & & \begin{matrix} 1 & \\ & -1 \end{matrix} \end{array}\right) x_1$$
$$+ \left(\begin{array}{ccc|c} 0 & & & \\ & 0 & & \\ & & 0 & \\ \hline & & & \begin{matrix} 3 & \\ & -3 \end{matrix} \end{array}\right) x_2 + \left(\begin{array}{ccc|c} 2 & 0 & -1 & \\ & 2 & 0 & \\ & & 2 & \\ \hline & & & \begin{matrix} 0 & \\ & 0 \end{matrix} \end{array}\right) x_3 \preceq 0$$

The *LMI* structure could then be initialized with the following commands:

```
%  Constraint 1
>> LMI(1).Q0  = [ ];
>> LMI(1,1).Q = [ 2 -1  0 ; ...
                  0  2  0 ; ...
                  0  0  2 ];
>> LMI(1,2).Q = [ ];
>> LMI(1,3).Q = [ 2  0 -1 ; ...
                  0  2  0 ; ...
                  0  0  2 ];

%  Constraint 2, diagonal matrices only
>> LMI(2).Q0  = diag( [0, 1] );
>> LMI(2,1).Q = diag( [1,-1] );
>> LMI(2,2).Q = diag( [3,-3] );
>> LMI(2,3).Q = [ ];

%  Use LMI in call to sdpAssign:
>> Prob=sdpAssign(c,LMI,...)

%  ... or set directly into Prob.PENSDP.LMI field:
>> Prob.PENSDP.LMI = LMI;
```

Some points of interest:

- The *LMI* structure must be of correct size. This is important if a LMI constraint has zero matrices for the highest numbered variables. If the above example had zero coefficient matrices for x_3, the user would have to set `LMI(1,3).Q = []` explicitly, so that the *LMI* structure array is really 2×3. (`LMI(2,3).Q` would automatically become empty in this case, unless set otherwise by the user).
- MATLAB sparse format is allowed and encouraged.
- Only the upper triangular part of each matrix is used (symmetry is assumed).

Input in Sparse SDPA Format is handled by the conversion routine *sdpa2pen*. For example, the problem defined in *tomlab/examples/arch0.dat-s* can be solved using the following statements:

```
>> p = sdpa2pen('arch0.dat-s')
p =
   vars: 174
   fobj: [1x174 double]
   constr: 174
   ci: [1x174 double]
   bi_dim: [1x174 double]
   bi_idx: [1x174 double]
   bi_val: [1x174 double]
   mconstr: 1
   ai_dim: 175
   ai_row: [1x2874 double]
   ai_col: [1x2874 double]
   ai_val: [1x2874 double]
   msizes: 161
   ai_idx: [175x1 double]
   ai_nzs: [175x1 double]
   x0: [1x174 double]
   ioptions: 0
   foptions: []

>> Prob=sdpAssign(p);            % Can call sdpAssign with only 'p' structure
>> Result=tomRun('pensdp',Prob); % Call tomRun to solve problem
```

13.2 Utility Functions in TOMLAB

In the following subsections the driver routine and the utility functions in TOMLAB will be described.

13.2.1 tomRun

Purpose
General multi-solver driver routine for TOMLAB.

Calling Syntax
Result = tomRun(Solver, probFile, probNumber, Prob, ask, PriLev)

Result = tomRun(Solver, Prob, ask, PriLev)

Result = tomRun(Solver, probType, probNumber, ask, PriLev)

Result = tomRun

Result = tomRun(probType)

Description of Inputs

Solver	The name of the solver that should be used to optimize the problem. Default *conSolve*. 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 31 and Table 32.
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 13.2.7 page 146. $PriLev = 0$ 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, default *con_prob.m*.
probNumber	Problem number in *probFile*. $probNumber = 0$ gives a menu in *probFile*.

Description of Outputs

Result	Structure with result from optimization, see Table 46.

Description
The driver routine *tomRun* is called from the command line, or called from the menu routine *tomMenu* or from the graphical user interface *tomGUI* to solve any problem defined in the TOMLAB Quick format or TOMLAB Init File format. If called with less than the required two parameters, a list of available solvers are printed.

M-files Used
xxxRun.m, *xxxRun2.m*, *PrintResult.m*, *inibuild.m*, *probInit.m*, *mkbound.m*, *mexSOL.m*, *mexRun.m*

13.2.2 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.

13.2.3 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 38, 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 38.
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 [25]. 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.

Functions Used
intpol2.m, *intpol3.m*

13.2.4 intpol2

Purpose
Find the minimum of a quadratic approximation of a scalar function in a given interval.

Calling Syntax
alfa = intpol2(x0, f0, g0, x1, f1, a, b, PriLev)

Description of Inputs

x0	Interpolation point x_0.
f0	Function value at x_0.
g0	Derivative value at x_0.
x1	Interpolation point x_1.
f1	Function value at x_1.
a	Lower interval bound.
b	Upper interval bound.
PriLev	Printing level, $Prilev > 3$ gives a lot of output.

Description of Outputs

alfa	The minimum of the interpolated second degree polynomial in the interval $[a, b]$.

Description
In the line search routine *LineSearch* the problem of choosing α in a given interval $[a, b]$ occurs both in the *bracketing phase* and in the *sectioning phase*. If quadratic interpolation are to be used *LineSearch* calls *intpol2* which finds the minimum of a second degree polynomial approximation in the given interval.

See Also
LineSearch, *intpol3*

13.2.5 intpol3

Purpose
Find the minimum of a cubic approximation of a scalar function in a given interval.

Calling Syntax
alfa = intpol3(x0, f0, g0, x1, f1, g1 , a, b, PriLev)

Description of Inputs

x0	Interpolation point x_0.
f0	Function value at x_0.
g0	Derivative value at x_0.
x1	Interpolation point x_1.
f1	Function value at x_1.
g1	Derivative value at x_1.
a	Lower interval bound.
b	Upper interval bound.
PriLev	Printing level, $Prilev > 3$ gives a lot of output.

Description of Outputs

alfa	The minimum of the interpolated third degree polynomial in the interval $[a, b]$.

Description
In the line search routine *LineSearch* the problem of choosing α in a given interval $[a, b]$ occurs both in the *bracketing phase* and in the *sectioning phase*. If cubic interpolation are to be used *LineSearch* calls *intpol3* which finds the minimum of a third degree polynomial approximation in the given interval.

See Also
LineSearch, *intpol2*

13.2.6 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 [42], which is applicable to general linear constrained problems. See [10] 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.

13.2.7 PrintResult

Purpose
Prints the result of an optimization.

Calling Syntax
PrintResult(Result, PriLev)

Description of Inputs

Result	Result structure from optimization.
PriLev	Printing level:
0	Silent.
1	Problem number and name. Function value at the solution and at start. Known optimal function value (if given).
2	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 $Inform$.
3	Jacobian, Hessian or Quasi-Newton Hessian approximation.

13.2.8 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

13.2.9 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

13.2.10 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

14 TOMLAB LDO (Linear and Discrete Optimization)

TOMLAB LDO is a collection of routines in TOMLAB for solving linear and discrete optimization (LDO) problems in operations research and mathematical programming. Included are many routines for special problems in linear programming, network programming, integer programming and dynamic programming.

Note that included in standard TOMLAB are the standard solver *lpSolve* for linear programming, the solver *DualSolve*, used to solve linear programming problems when a dual feasible point is known, and two routines for mixed-integer programming, *mipSolve* and *cutplane*. The aim of the corresponding simpler routines in the LDO collection are mainly teaching. Use the standard TOMLAB routines for production runs.

14.1 Optimization Algorithms and Solvers in TOMLAB LDO

This section describes the LDO routines by giving tables describing most Matlab functions with some comments. All function files are collected in the directory *ldo*.

There is a simple menu program, *simplex*, for linear programming. The routine is a utility to interactively solve LP problems in canonical standard form. When the problem is defined, *simplex* calls the TOMLAB internal LDO solvers *lpsimp1* and *lpsimp2*.

Like the MathWorks, Inc. Optimization Toolbox 1.x, TOMLAB LDO is using a vector with optimization parameters. In Optimization Toolbox, the routine setting up the default values in a vector OPTIONS with 18 parameters is called *foptions*. Our solvers need more parameters, currently 29, and therefore the routine *goptions* is used instead of *foptions*.

In TOMLAB the routine *lpDef* is used to define the *optPar* vector and the routine *optParamDef* the *optParam* structure.

14.1.1 Linear Programming

There are several algorithms implemented for **linear programming**, listed in Table 17. The solver *lpSolve2* is using the same input and output format as the TOMLAB solvers described in Section 2. It is using the optimization parameter structure *optParam* instead of the optimization parameter vector *optPar*.

lpsimp1 and *lpsimp2* are simpler versions of the two basic parts in *lpSolve2* that solves Phase I and Phase II LP problem, respectively. *lpdual* is an early version of the TOMLAB solver *DualSolve*.

lpSolve calls both the routines *Phase1Simplex* and *Phase2Simplex* to solve a general **linear program** (**lp**) defined as

$$\begin{array}{lll} \min\limits_{x} & f(x) = c^T x & \\ & & \\ s/t & \begin{array}{ccccc} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U \end{array} & \end{array} \tag{19}$$

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 implementation of *lpsimp2* is based on the standard revised simplex algorithm as formulated in Goldfarb and Todd [41, page 91] for solving a Phase II LP problem. *lpsimp1* implements a Phase I simplex strategy which formulates a LP problem with artificial variables. This routine is using *lpsimp2* to solve the Phase I problem. The dual simplex method [41, pages 105-106], usable when a dual feasible solution is available instead of a primal feasible, is also implemented (*lpdual*).

Two polynomial algorithms for linear programming are implemented. Karmakar's projective algorithm (*karmark*) is developed from the description in Bazaraa et. al. [6, page 386]. There is a choice of update, either according to Bazaraa or the rule by Goldfarb and Todd [41, chap. 9]. The affine scaling variant of Karmakar's method (*akarmark*) is an implementation of the algorithm in Bazaraa [41, pages 411-413]. As the purification algorithm a modification of the algorithm on page 385 in Bazaraa is used.

The internal linear programming solvers *lpsimp2* and *lpdual* both have three rules for variable selection implemented. Bland's cycling prevention rule is the choice if fear of cycling exists. There are two 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).

Table 17: Solvers for linear programming.

Function	Description	Section	Page
lpSolve2	General solver for linear programming problems. Has two internal routines. Phase1Simplex finds basic feasible solution (bfs) using artificial variables. It calls the other internal routine, Phase2Simplex, which implements the revised simplex algorithm with three selection rules.	13.1.13	117
lpsimp1	The Phase I simplex algorithm. Finds a basic feasible solution (bfs) using artificial variables. Calls *lpsimp2*.	14.4.11	167
lpsimp2	The Phase II revised simplex algorithm with three selection rules.	14.4.12	168
karmark	Karmakar's algorithm. Kanonical form.	14.4.6	164
lpkarma	Solves LP on equality form, by converting and calling *karmark*.	14.4.10	166
lpdual	The dual simplex algorithm.	14.4.9	166
akarmark	Affine scaling variant of Karmarkar's algorithm.	14.4.1	160

14.1.2 Transportation Programming

Transportation problems are solved using an implementation of the transportation simplex method as described in Luenberger [65, chap 5.4] (*TPsimplx*). Three simple algorithms to find a starting basic feasible solution for the transportation problem are included; the northwest corner method (*TPnw*), the minimum cost method (*TPmc*) and Vogel's approximation method (*TPvogel*). The implementation of these algorithms follows the algorithm descriptions in Winston [83, chap. 7.2]. The functions are described in Table 18.

Table 18: Routines for transportation programming.

Function	Description	Section	Page
TPnw	Find initial bfs to TP using the northwest corner method.	14.5.6	176
TPmc	Find initial bfs to TP using the minimum cost method.	14.5.5	175
TPvogel	Find initial bfs to TP using Vogel's approximation method.	14.5.7	176
TPsimplx	Implementation of the transportation simplex algorithm.	14.4.19	172

14.1.3 Network Programming

The implementation of the **network programming** algorithms are based on the forward and reverse star representation technique described in Ahuja et al. [3, pages 35-36]. The following algorithms are currently implemented:

- Search for all reachable nodes in a network using a stack approach (*gsearch*). The implementation is a variation of the Algorithm SEARCH in [2, pages 231-233].
- Search for all reachable nodes in a network using a queue approach (*gsearchq*). The implementation is a variation of the Algorithm SEARCH in [2, pages 231-232].
- Find the minimal spanning tree of an undirected graph (*mintree*) with Kruskal's algorithm described in Ahuja et. al. [3, page 520-521].
- Solve the shortest path problem using Dijkstra's algorithm (*dijkstra*). A direct implementation of the Algorithm DIJKSTRA in [2, pages 250-251].
- Solve the shortest path problem using a label correcting method (*labelcor*). The implementation is based on Algorithm LABEL CORRECTING in [2, page 260].

- Solve the shortest path problem using a modified label correcting method (*modlabel*). The implementation is based on Algorithm MODIFIED LABEL CORRECTING in [2, page 262], including the heuristic rule discussed to improve running time in practice.
- Solve the maximum flow problem using the Ford-Fulkerson augmenting path method (*maxflow*). The implementation is based on the algorithm description in Luenberger [65, pages 144-145].
- Solve the minimum cost network flow problem (MCNFP) using a network simplex algorithm (*NWsimplx*). The implementation is based on Algorithm network simplex in Ahuja et. al. [3, page 415].
- Solve the symmetric traveling salesman problem using Lagrangian relaxation and the subgradient method with the Polyak rule II (*salesman*), an algorithm by Held and Karp [44].

The network programming routines are listed in Table 19.

Table 19: Routines for network programs.

Function	Description	Section	Page
gsearch	Searching all reachable nodes in a network. Stack approach.	14.5.2	174
gsearchq	Searching all reachable nodes in a network. Queue approach.	14.5.3	174
mintree	Finds the minimum spanning tree of an undirected graph.	14.5.4	175
dijkstra	Shortest path using Dijkstra's algorithm.	14.4.3	162
labelcor	Shortest path using a label correcting algorithm.	14.4.8	165
modlabel	Shortest path using a modified label correcting algorithm.	14.4.14	169
maxflow	Solving maximum flow problems using the Ford-Fulkerson augmenting path method.	14.4.13	168
salesman	Symmetric traveling salesman problem (TSP) solver using Lagrangian relaxation and the subgradient method with the Polyak rule II.	14.4.18	171
NWsimplx	Solving minimum cost network flow problems (MCNFP) with a network simplex algorithm.	14.4.15	169

14.1.4 Mixed-Integer Programming

To solve mixed linear inequality integer programs two algorithms are implemented as part of TOMLAB, *mipSolve* and *cutplane*. Described in the Network Programming section 14.1.3 is the *salesman* routine, which is an special type of integer programming problem. The directory *tsplib* contains test problems for the travelling salesman problem. The routine *runtsp* runs any of the 25 predefined problems. *tsplib* reads the actual problem definition and generates the problem.

Balas method for binary integer programs restricted to integer coefficients is implemented in the routine *balas* [48].

14.1.5 Dynamic Programming

Two simple examples of dynamic programming are included. Both examples are from Winston [83, chap. 20]. Forward recursion is used to solve an inventory problem (*dpinvent*) and a knapsack problem (*dpknap*), see Table 20.

Table 20: Routines for dynamic programming.

Function	Description	Section	Page
dpinvent	Forward recursion DP algorithm for the inventory problem.	14.4.4	162
dpknap	Forward recursion DP algorithm for the knapsack problem.	14.4.5	163

14.1.6 Quadratic Programming

Two simple routines for quadratic programming are included,

Table 21: Routines for quadratic programming.

Function	Description	Section	Page
qpe	Solves a **qp** problem, restricted to equality constraints, using a null space method.	14.4.17	171
qplm	Solves a **qp** problem, restricted to equality constraints, using Lagrange's method.	14.4.16	170

14.1.7 Lagrangian Relaxation

The usage of Lagrangian relaxation techniques is exemplified by the routine *ksrelax*, which solves integer linear programs with linear inequality constraints and upper and lower bounds on the variables. The problem is solved by relaxing all but one constraint and hence solving simple knapsack problems as subproblems in each iteration. The algorithm is based on the presentation in Fischer [23], using subgradient iterations and a simple line search rule. Lagrangian relaxation is used by the symmetric travelling salesman solver *salesman*. Also a routine to draw a plot of the relaxed function is included. The Lagrangian relaxation routines are listed in Table 22.

Table 22: Routines for Lagrangian relaxation.

Function	Description	Section	Page
ksrelax	Lagrangian relaxation with knapsack subproblems.	14.4.7	164
urelax	Lagrangian relaxation with knapsack subproblems, plot result.	14.4.20	173

14.1.8 Utility Routines

Table 23 lists the utility routines used in TOMLAB LDO. Some of them are also used by the other routines in TOMLAB.

Table 23: Utility routines.

Function	Description
a2frstar	Convert node-arc A matrix to Forward-Reverse Star Representation.
z2frstar	Convert matrix of arcs (and costs) to Forward-Reverse Star.
cpTransf	Transform general convex programs to other forms.
optParamDef	Define optimization parameters in the TOMLAB format (*optParam*).
lpDef	Define optimization parameters in the Optimization Toolbox 1.x format (*optPar*).
mPrint	Print matrix, format: NAME$(i,:)$ $a(i,1)a(i,2)...a(i,n)$.
printmat	Print matrix with row and column labels.
vPrint	Print vector in rows, format: NAME$(i_1 : i_n)$ $v_{i_1} v_{i_2} ... v_{i_n}$.
xPrint	Print vector x, row by row, with format.
xPrinti	Print integer vector x. Calls *xprint*.
xPrinte	Print integer vector x in exponential format. Calls *xprint*.

14.2 How to Solve Optimization Problems Using TOMLAB LDO

This section describes how to use TOMLAB LDO to solve the different type of problems discussed in Section 14.1

14.2.1 How to Solve Linear Programming Problems

The following example shows how the simple LP problem in (11) can be solved by direct use of the optimization routines *lpsimp1* and *lpsimp2*:

```
A = [ 1  2
      4  1 ];
b = [ 6 12 ]';
c = [-7 -5]';
meq = 0;
optPar = lpDef;
optPar(13) = meq;
[x_0, B_0, optPar, y] = lpsimp1(A, b, optPar);
[x, B, optPar, y] = lpsimp2(A, b, c, optPar, x_0, B_0);
```

For further illustrations of how to solve linear programming problems see the example files listed in Table 24 and Table 25.

Table 24: Test examples for linear programming.

Function	Description
exinled.m	First simple LP example from a course in Operations Research.
excycle	Menu with cycling examples.
excycle1	The Marshall-Suurballe cycling example. Run both the Bland's cycle preventing rule and the default minimum reduced cost rule and compare results.
excycle2	The Kuhn cycling example.
excycle3	The Beale cycling example.
exKleeM	The Klee-Minty example. Shows that the simplex algorithm with Dantzig's rule visits all vertices.
exfl821	Run exercise 8.21 from Fletcher, Practical methods of Optimization. Illustrates redundancy in constraints.
ex412b4s	Wayne Winston example 4.12 B4, using *lpsimp1* and *lpsimp2*.
expertur	Perturbed both right hand side and objective function for Luenberger 3.12-10,11.
ex6rev17	Wayne Winston chapter 6 Review 17. Simple example of calling the dual simplex solver *lpdual*.
ex611a2	Wayne Winston example 6.11 A2. A simple problem solved with the dual simplex solver *lpdual*.

Some test examples are collected in the file *demoLP* and further described in Table 25.

Table 25: Test examples for linear programming running interior point methods.

Function	Description
exww597	Test of *karmark* and *lpsimp2* on Winston example page 597 and Winston 10.6 Problem A1.
exstrang	Test of *karmark* and *lpsimp2* on Strangs' *nutshell* example.
exkarma	Test of *akarmark*.
exKleeM2	Klee-Minty example solved with *lpkarma* and *karmark*.

14.2.2 How to Solve Transportation Programming Problems

The following is a simple example of a transportation problem

$$s = \begin{pmatrix} 5 \\ 25 \\ 25 \end{pmatrix}, d = \begin{pmatrix} 10 \\ 10 \\ 20 \\ 15 \end{pmatrix}, C = \begin{pmatrix} 6 & 2 & -1 & 0 \\ 4 & 2 & 2 & 3 \\ 3 & 1 & 2 & 1 \end{pmatrix}, \tag{20}$$

where s is the supply vector, d is the demand vector and C is the cost matrix. See *TPsimplx* Section 14.4.19. Solving (20) by use of the routine *TPsimplx* is done by:

```
s = [ 5 25 25 ]';
d = [10 10 20 15 ]';
C = [ 6  2 -1  0
      4  2  2  3
      3  1  2  1 ];

[X, B, optPar, y, C] = TPsimplx(s, d, C)
```

When neither starting base nor starting point is given as input argument *TPsimplx* calls *TPvogel* (using Vogel's approximation method) to find an initial basic feasible solution (bfs). To use another method to find an initial bfs, e.g. the northwest corner method, explicitly call the corresponding routine (*TPnw*) before the call to *TPsimplx*:

```
s = [ 5 25 25 ]';
d = [10 10 20 15 ]';
C = [ 6  2 -1  0
      4  2  2  3
      3  1  2  1 ];

[X_0, B_0] = TPnw(s, d)

[X, B, optPar, y, C] = TPsimplx(s, d, C, X_0, B_0)
```

For further illustrations of how to solve transportation programming problems see the example files listed in Table 26.

Table 26: Test examples for transportation programming.

Function	Description
extp_bfs	Test of the three routines that finds initial basic feasible solution to a TP problem, routines *TPnw*, *TPmc* and *TPvogel*.
exlu119	Luenberger TP page 119. Find initial basis with *TPnw*, *TPmc* and *TPvogel* and run *TPsimplx* for each.
exlu119U	Test unbalanced TP on Luenberger TP page 119, slightly modified. Runs *TPsimplx*.
extp	Runs simple TP example. Find initial basic feasible solution and solve with *TPsimplx*.

14.2.3 How to Solve Network Programming Problems

In TOMLAB LDO there are several routines for network programming problems. Here follows an example of how to solve a shortest path problem. Given the network in Figure 20, where the numbers at each arc represent the distance of the arc, find the shortest path from node 1 to all other nodes. Representing the network with the node-arc incidence matrix A and the cost vector c gives:

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & -1 & -1 & 0 & 1 \end{pmatrix}, c = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 4 \\ 2 \\ 4 \\ 1 \\ 3 \end{pmatrix} \tag{21}$$

Representing the network with the *forward and reverse star* technique gives:

$$P = \begin{pmatrix} 1 \\ 3 \\ 4 \\ 6 \\ 8 \\ 9 \end{pmatrix}, Z = \begin{pmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 4 \\ 3 & 2 \\ 3 & 5 \\ 4 & 5 \\ 4 & 3 \\ 5 & 4 \end{pmatrix}, c = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 4 \\ 2 \\ 4 \\ 1 \\ 3 \end{pmatrix}, T = \begin{pmatrix} 1 \\ 4 \\ 2 \\ 7 \\ 3 \\ 8 \\ 5 \\ 6 \end{pmatrix}, R = \begin{pmatrix} 1 \\ 1 \\ 3 \\ 5 \\ 7 \\ 9 \end{pmatrix} \tag{22}$$

See *a2frstar* Section 14.5.1 for an explanation of the notation.

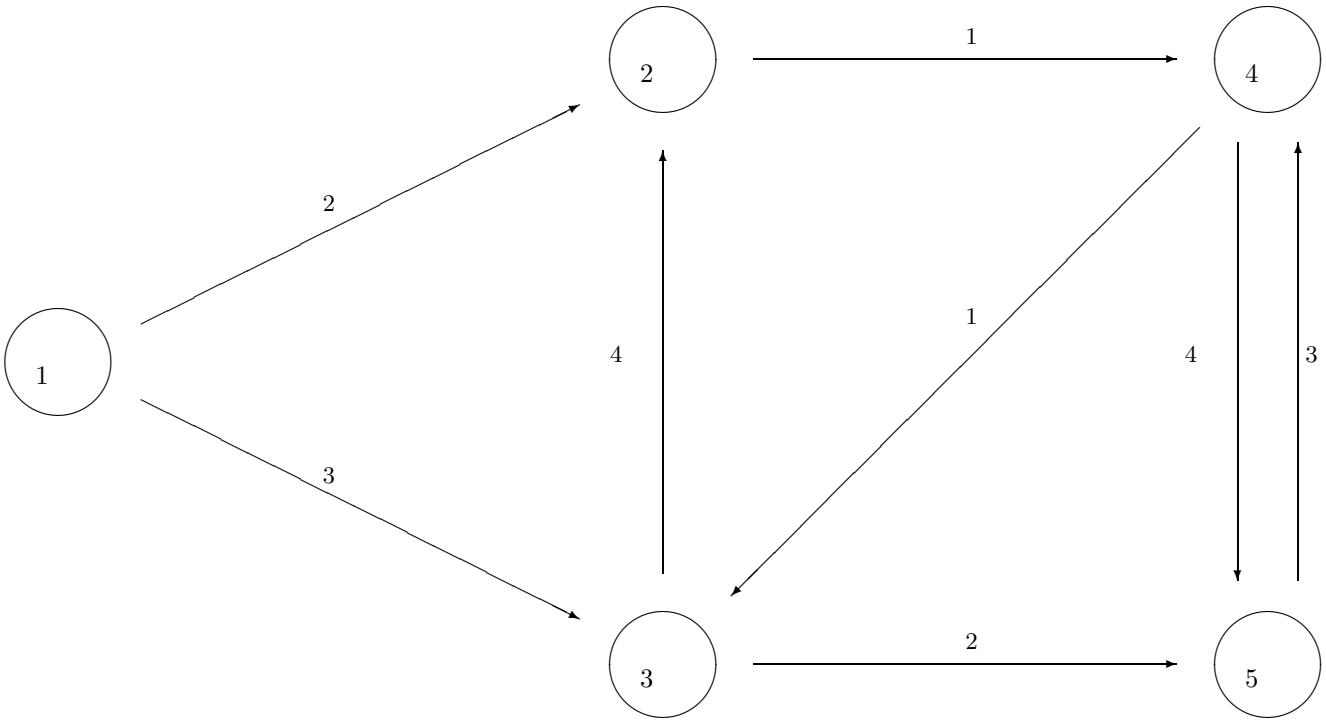


Figure 20: A network example.

Choose *modlabel* to solve this example, see Section 14.4.14, *modlabel* implements a modified label correcting algorithm. First define the incidence matrix A and the cost vector c and call the routine *a2frstar* to convert to a *forward and reverse star* representation (which is used by *modlabel*). Then the actual problem is solved.

```
A = [ 1  1  0  0  0  0  0  0
```

```
     -1  0  1 -1  0  0  0  0
      0 -1  0  1  1  0 -1  0
      0  0 -1  0  0  1  1 -1
      0  0  0  0 -1 -1  0  1 ];

C = [ 2  3  1  4  2  4  1  3 ];

[P Z c T R x_U] = a2frstar(A, C);

[pred dist] = modlabel(1,P,Z,c);
```

For further illustrations of how to solve network programming problems see the example files listed in Table 27.

Table 27: Test examples for network programming.

Function	Description
exgraph	Testing network routines on simple example.
exflow	Testing several maximum flow examples.
pathflow	Pathological test example for maximum flow problems.
exflow31	Test example N31.
exmcnfp	Minimum Cost Network Flow Problem (MCNFP) example from Ahuja et. al.

14.2.4 How to Solve Integer Programming Problems

The routines originally in TOMLAB LDO for solving integer programming problems were *cutplane*, *mipSolve* and *balas*. Now *cutplane* and *mipSolve* are part of the standard Tomlab and are called using the Tomlab format, see Section 5.3. Examples showing how to use the *balas* routine and the other solvers are listed in Table 28. These examples are all part of the demonstration routine *demoMIP.m*.

Table 28: Test examples for integer programming.

Function	Description
expkorv	Test of *cutplane* and *mipSolve* for example PKorv.
exIP39	Test example I39.
exbalas	Test of 0/1 IP (Balas algorithm) on simple example.

14.2.5 How to Solve Dynamic Programming Problems

In this subsection dynamic programming is illustrated with a simple approach to solve a knapsack problem and an inventory problem. The routines *dpknap* (see Section 14.4.5) and *dpinvent* (Section 14.4.4) are used. The knapsack problem (23) is an example from Holmberg [48] and the inventory problem is an example from Winston [83, page 1013].

$$\begin{array}{ll} \max\limits_{u} & f(u) = 7u_1 + u_2 + 4u_3 \\ s/t & 2u_1 + 3u_2 + 2u_3 \leq 4 \\ & 0 \leq u_1 \leq 1 \\ & 0 \leq u_2 \leq 1 \\ & 0 \leq u_3 \leq 2 \\ & u_j \in \mathbb{N}, j = 1, 2, 3 \end{array} \tag{23}$$

Problem (23) will be solved by the following definitions and call:

```
A      = [ 2 3 2 ];
b      =   4;
c      = [ 7 2 4 ];
u_UPP  = [ 1 1 2 ];
PriLev = 0;

[u, f_opt] = dpknap(A, b, c, u_UPP, PriLev);
```

Description of the inventory problem:
A company knows that the demand for its product during each of the next for months will be as follows: month 1, 1 unit; month 2, 3 units; month 3, 2 units; month 4, 4 units. At the beginning of each month, the company must determine how many units should be produced during the current month. During a month in which any units are produced, a setup cost of $3 is incurred. In addition, there is a variable cost of $1 for every unit produced. At the end of each month, a holding cost of 50 cents per unit on hand is incurred. Capacity limitations allow a maximum of 5 units to be produced during each month. The size of the company's warehouse restricts the ending inventory for each month to at most 4 units. The company wants to determine a production schedule that will meet all demands on time and will minimize the sum of production and holding costs during the four months. Assume that 0 units are on hand at the beginning of the first month.

The inventory problem described above will be solved by the following definitions and call:

```
d   = [1 3 2 4]';    % Demand. N = 4;
P_s = 3;             % Setup cost $3 if u > 0
P   = ones(5,1);     % Production cost $1/unit in each time step
I_s = 0;             % Zero setup cost for the Inventory
I   = 0.5*ones(5,1); % Inventory cost $0.5/unit in each time step
x_L = 0;             % lower bound on inventory, x
x_U = 4;             % upper bound on inventory, x
x_LAST = [];         % Find best choice of inventory at end
x_S = 0;             % Empty inventory at start
u_L = [0 0 0 0];     % Minimal amount produced in each time step
u_U = [5 5 5 5];     % Maximal amount produced in each time step
PriLev = 1;

[u, f_opt] = dpinvent(d, P_s, P, I_s, I, u_L, u_U, x_L, x_U, x_S, x_LAST, PriLev);
```

For further illustrations of how to solve dynamic programming problems see the example files listed in Table 29.

Table 29: Test examples for dynamic programming.

Function	Description
exinvent	Test of *dpinvent* on two inventory examples.
exknap	Test of *dpknap* (calls *mipSolve* and *cutplane*) on five knapsack examples.

14.2.6 How to Solve Lagrangian Relaxation Problems

This section shows an example of using Lagrangian relaxation techniques implmented in the routine *ksrelax* to solve an integer programming problem. The problem to be solved, (24), is an example from Fischer [23].

$$\begin{array}{ll} \max\limits_{x} & f(x) = 16x_1 + 10x_2 + 4x_4 \\ s/t & 8x_1 + 2x_2 + x_3 + x_4 \leq 10 \\ & x_1 + x_2 \leq 1 \\ & x_3 + x_4 \leq 1 \\ & x_j \in 0/1, j = 1, 2, 3, 4 \end{array} \tag{24}$$

```
A = [ 8  2  1  4
      1  1  0  0
      0  0  1  1 ];
b = [10  1  1 ]';
c = [16 10  0  4 ]';
r = 1;                % Do not relax the first constraint
x_UPP = [1 1 1 1]';

[x u f_opt optPar] = ksrelax(A, b, c, r, x_UPP);
```

For further illustrations of how to solve Lagrangian Relaxation problems see the example files listed in Table 30.

Table 30: Test examples for Lagrangian Relaxation.

Function	**Description**
exrelax	Test of *ksrelax* on LP example from Fischer -85.
exrelax2	Simple example, runs *ksrelax*.
exIP39rx	Test example I39, relaxed. Calls *urelax* and plot.

14.3 Printing Utilities and Print Levels

This section is written for the part of TOMLAB LDO which is not using the same input/output format and is not designed in the same way as the other routines in TOMLAB. Information about printing utilities and print levels for the other routines could be found in Section 10.5

The amount of printing is determined by setting a print level for each routine. This parameter most often has the name *PriLev*.

Normally the zero level ($PriLev = 0$) corresponds to silent mode with no output. The level one corresponds to a result summary and error messages. Level two gives output every iteration and level three displays vectors and matrices. Higher levels give even more printing of debug type. See the help in the actual routine.

The main driver or menu routine called may have a *PriLev* parameter among its input parameters. The routines called from the main routine normally sets the *PriLev* parameter to *optPar(1)*. The vector *optPar* is set to default values by a call to *goptions*. The user may then change any values before calling the main routine. The elements in *optPar* is described giving the command: *help goptions*. For linear programming there is a special initialization routine, *lpDef*, which calls *goptions* and changes some relevant parameters.

There is a wait flag in *optPar*, *optPar(24)*. If this flag is set, the routines uses the pause statement to avoid the output just flushing by.

The TOMLAB LDO 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, see Table 23.

For matrices there are two routines, *mPrint* and *printmat*. The routine *printmat* 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, *printmat* will first display $A(:, 1:10)$, then $A(:, 11:20)$ and so on.

The routine *printmat* 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.

14.4 Optimization Routines in TOMLAB LDO

In the following subsections the optimization routines in TOMLAB LDOwill be described.

14.4.1 akarmark

Purpose
Solve linear programming problems of the form

$$\begin{array}{lrcl} \min\limits_{x} & f(x) & = & c^T x \\ s/t & Ax & = & b \\ & x & \geq & 0 \end{array}$$

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

Calling Syntax
[x, optPar, y, x_0] = akarmark(A, b, c, optPar, x_0)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
optPar	Optimization parameter vector, see *goptions.m*.
x_0	Starting point.

Description of Outputs

x	Optimal point.
optPar	Optimization parameter vector, see *goptions.m*.
y	Dual parameters.
x_0	Starting point used.

Description
The routine *akarmark* is an implementation of the affine scaling variant of Karmarkar's method as described in Bazaraa [41, pages 411-413]. As the purification algorithm a modified version of the algorithm on page 385 in Bazaraa is used.

Examples
See *exakarma*, *exkarma*, *exkleem2*.

M-files Used
lpDef.m

See Also
lpkarma, *karmark*

14.4.2 balas

Purpose
Solve binary integer linear programming problems.

balas solves problems of the form

$$\begin{array}{lrcll} \min\limits_{x} & f(x) & = & c^T x & \\ s/t & a_i^T x & = & b_i & i = 1, 2, ..., m_{eq} \\ & a_i^T x & \leq & b_i & i = m_{eq}+1, ..., m \\ & x_j \in 0/1 & & & j = 1, 2, ..., n \end{array}$$

where $c \in \mathbb{Z}^n$, $A \in \mathbb{Z}^{m \times n}$ and $b \in \mathbb{Z}^m$.

Calling Syntax
[x, optPar] = balas(A, b, c, optPar)

Description of Inputs

A	Constraint matrix, integer coefficients.
b	Right hand side vector, integer coefficients.
c	Cost vector, integer coefficients.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

x	Optimal point.
optPar	Optimization parameter vector, see *goptions.m*.

Description
The routine *balas* is an implementation of Balas method for binary integer programs restricted to integer coefficients.

Examples
See *exbalas*.

M-files Used
lpDef.m

See Also
mipSolve, *cutplane*

14.4.3 dijkstra

Purpose
Solve the shortest path problem.

Calling Syntax
[pred, dist] = dijkstra(s, P, Z, c)

Description of Inputs

s	The starting node.
p	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.

Description of Outputs

pred	$pred(j)$ is the predecessor of node j.
dist	$dist(j)$ is the shortest distance from node s to node j.

Description
dijkstra is a direct implementation of the algorithm DIJKSTRA in [2, pages 250-251] for solving shortest path problems using Dijkstra's algorithm. Dikstra's algorithm belongs to the class of *label setting* methods which are applicable only to networks with nonnegative arc lengths. For solving shortest path problems with arbitrary arc lengths use the routine *labelcor* or *modlabel* which belongs to the class of *label correcting* methods.

Examples
See *exgraph*, *exflow31*.

See Also
labelcor, *modlabel*

Limitations
dijkstra can only solve problems with nonnegative arc lengths.

14.4.4 dpinvent

Purpose
Solve production/inventory problems of the form

$$\begin{array}{lccccc} \min\limits_{u} & f(u) & = & P_s(t) + P(t)^T u(t) + I(t)^T x(t) & & \\ s/t & u_L & \leq & u(t) & \leq & u_U \\ & x_L & \leq & x(t) & \leq & x_U \\ & 0 & \leq & u(t) & \leq & x(t) + d(t) \\ & u_j \in \mathbb{N} & & j = 1, 2, ..., n & & \\ & x_j \in \mathbb{N} & & j = 1, 2, ..., n & & \end{array}$$

where $x(t) = x(t-1) + u(t) - d(t)$ and $d \in \mathbb{N}^n$.

Calling Syntax
[u, f_opt, exit] = dpinvent(d, P_s, P, I_s, I, u_L, u_U, x_L, x_U, x_S, x_LAST, PriLev)

Description of Inputs

d	Demand vector.
P_s	Production setup cost.
P	Production cost vector.
I_s	Inventory setup cost.
I	Inventory cost vector.
u_L	Minimal amount produced in each time step.
u_U	Maximal amount produced in each time step.
x_L	Lower bound on inventory.
x_U	Upper bound on inventory.
x_S	Inventory state at start.
x_LAST	Inventory state at finish.
PriLev	Printing level: $PriLev = 0$, no output. $PriLev = 1$, output of convergence results. $PriLev > 1$, output of each iteration.

Description of Outputs

u	Optimal control.
f_opt	Optimal function value.
exit	Exit flag.

Description

dpinvent solves production/inventory problems using a forward recursion dynamic programming technique as described in Winston [83, chap. 20].

Examples

See *exinvent*.

14.4.5 dpknap

Purpose

Solve knapsack problems of the form

$$\begin{array}{lccc} \max\limits_{u} & f(u) & = & c^T u \\ s/t & Au & \leq & b \\ & u & \leq & u_U \\ & u_j \in \mathbb{N} & & j = 1, 2, ..., n \end{array}$$

where $A \in \mathbb{N}^n$, $c \in \mathbb{R}^n$ and $b \in \mathbb{N}$

Calling Syntax

[u, f_opt, exit] = dpknap(A, b, c, u_U, PriLev)

Description of Inputs

A	Weigth vector.
b	Knapsack capacity.
c	Benefit vector.
u_U	Upper bounds on *u*.
PriLev	Printing level: $PriLev = 0$, no output. $PriLev = 1$, output of convergence results. $PriLev > 1$, output of each iteration.

Description of Outputs

u	Optimal control.
f_opt	Optimal function value.
exit	Exit flag.

Description

dpknap solves knapsack problems using a forward recursion dynamic programming technique as described in [83, chap. 20]. The Lagrangian relaxation routines *ksrelax* and *urelax* call *dpknap* to solve the knapsack subproblems.

Examples
See *exknap*.

14.4.6 karmark

Purpose
Solve linear programming problems of Karmakar's form

$$\begin{array}{rccc} \min\limits_{x} & f(x) & = & c^T x \\ s/t & Ax & = & 0 \\ & \sum\limits_{j=1}^{n} x_j & = & 1 \\ & x & \geq & 0 \end{array}$$

where $x, c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and the following assumptions hold:

- The point $x^{(0)} = \left(\frac{1}{n}, ..., \frac{1}{n}\right)^T$ is feasible.
- The optimal objective value is zero.

Calling Syntax
[x, optPar] = karmark(A, c, optPar)

Description of Inputs

A	Constraint matrix.
c	Cost vector.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

x	Optimal point.
optPar	Optimization parameter vector, see *goptions.m*.

Description
The routine *karmark* is an implementation of Karmakar's projective algorithm which is of polynomial complexity. The implementation uses the description in Bazaraa [6, page 386]. There is a choice of update, either according to Bazaraa or the rule by Goldfarb and Todd [41, chap. 9]. As the purification algorithm a modified version of the algorithm on page 385 in Bazaraa is used. *karmark* is called by *lpkarma* which transforms linear maximization problems on inequality form into Karmakar's form needed for *karmark*.

Examples
See *exstrang*, *exww597*.

M-files Used
lpDef.m

See Also
lpkarma, *akarmark*

14.4.7 ksrelax

Purpose
Solve integer linear problems of the form

$$\begin{array}{rccc} \max\limits_{x} & f(x) & = & c^T x \\ s/t & Ax & \leq & b \\ & x & \leq & x_U \\ & x_j \in \mathbb{N} & & j = 1, 2, ..., n \end{array}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{N}^{m \times n}$ and $b \in \mathbb{N}^m$.

Calling Syntax
[x_P, u, f_P, optPar] = ksrelax(A, b, c, r, x_U, optPar)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
r	Constraint not to be relaxed.
x_U	Upper bounds on the variables.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

x_P	Primal solution.
u	Lagrangian multipliers.
f_P	Function value at *x_P*.
optPar	Optimization parameter vector, see *goptions.m*.

Description

The routine *ksrelax* uses Lagrangian Relaxation to solve integer linear programming problems with linear inequality constraints and simple bounds on the variables. The problem is solved by relaxing all but one constraint and then solve a simple knapsack problem as a subproblem in each iteration. The algorithm is based on the presentation in Fisher [23], using subgradient iterations and a simple line search rule. LDO also contains a routine *urelax* which plots the result of each iteration.

Examples

See *exrelax*, *exrelax2*.

M-files Used

lpDef.m, *dpknap.m*

See Also

urelax

14.4.8 labelcor

Purpose

Solve the shortest path problem.

Calling Syntax

[pred, dist] = labelcor(s, P, Z, c)

Description of Inputs

s	The starting node.
p	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.

Description of Outputs

pred	$pred(j)$ is the predecessor of node j.
dist	$dist(j)$ is the shortest distance from node s to node j.

Description

The implementation of *labelcor* is based on the algorithm LABEL CORRECTING in [2, page 260] for solving shortest path problems. The algorithm belongs to the class of *label correcting* methods which are applicable to networks with arbitrary arc lengths. *labelcor* requires that the network does not contain any negative directed cycle, i.e. a directed cycle whose arc lengths sum to a negative value.

Examples

See *exgraph*.

See Also

dijkstra, *modlabel*

Limitations

The network must not contain any negative directed cycle.

14.4.9 lpdual

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

lpdual solves problems of the form

$$\begin{array}{lrcll} \min\limits_{x} & f_P(x) & = & c^T x & \\ s/t & a_i^T x & = & b_i & i = 1, 2, ..., m_{eq} \\ & a_i^T x & \leq & b_i & i = m_{eq}+1, ..., m \\ & x & \geq & 0 & \end{array}$$

by rewriting it into standard form and solving the dual problem

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

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

Calling Syntax
[x, y, B, optPar] = lpdual(A, b, c, optPar, B_0, x_0, y_0)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
optPar	Optimization parameter vector, see *goptions.m*.
B_0	Logical vector of length n for basic variables at start.
x_0	Starting point, must be dual feasible.
y_0	Dual parameters (Lagrangian multipliers) at x_0.

Description of Outputs

x	Optimal point.
y	Dual parameters (Lagrangian multipliers) at the solution.
B	Optimal basic set.
optPar	Optimization parameter vector, see *goptions.m*.

Description
When a dual feasible solution is available, the dual simplex method is possible to use. *lpdual* implements this method using the algorithm in [41, 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).

Examples
See *ex611a2*, *ex6rev17*.

M-files Used
lpDef.m

See Also
lpsimp1, *lpsimp2*

14.4.10 lpkarma

Purpose
Solve linear programming problems of the form

$$\begin{array}{lccc} \max\limits_{x} & f(x) & = & c^T x \\ s/t & Ax & \leq & b \\ & x & \geq & 0 \end{array}$$

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

Calling Syntax
[x, y, optPar] = lpkarma(A, b, c, optPar)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

x	Optimal point.
y	Dual solution.
optPar	Optimization parameter vector, see *goptions.m*.

Description
lpkarma converts a linear maximization problem on inequality form into Karmakar's form and calls *karmark* to solve the transformed problem.

Examples
See *exstrang*, *exww597*.

M-files Used
lpDef.m, *karmark.m*

See Also
karmark, *akarmark*

14.4.11 lpsimp1

Purpose
Find a basic feasible solution to linear programming problems.

lpsimp1 finds a basic feasible solution to problems of the form

$$\begin{array}{lrcll} \min\limits_{x} & f(x) & = & c^T x & \\ s/t & a_i^T x & = & b_i & i = 1, 2, ..., m_{eq} \\ & a_i^T x & \leq & b_i & i = m_{eq}+1, ..., m \\ & x & \geq & 0 & \end{array}$$

where $x, c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m, b \geq 0$.

Calling Syntax
[x, B, optPar, y] = lpsimp1(A, b, optPar)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

x	Basic feasible solution.
B	Basic set at the solution x.
optPar	Optimization parameter vector, see *goptions.m*.
y	Lagrange multipliers.

Description
The routine *lpsimp1* implements a Phase I Simplex strategy which formulates a LP problem with artificial variables. Slack variables are added to the inequality constraints and artificial variables are added to the equality constraints. The routine uses *lpsimp2* to solve the Phase I problem.

Examples
See *exinled*, *excycle*, *excycle2*, *exKleeM*, *exfl821*, *ex412b4s*.

M-files Used
lpDef.m, *lpsimp2.m*

See Also
lpsimp2

14.4.12 lpsimp2

Purpose
Solve linear programming problems.

lpsimp2 solves problems of the form

$$\begin{array}{lrcll} \min\limits_{x} & f(x) & = & c^T x & \\ s/t & a_i^T x & = & b_i & i = 1, 2, ..., m_{eq} \\ & a_i^T x & \leq & b_i & i = m_{eq}+1, ..., m \\ & x & \geq & 0 & \end{array}$$

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

Calling Syntax
[x, B, optPar, y] = lpsimp2(A, b, c, optPar, x_0, B_0)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
optPar	Optimization parameter vector, see *goptions.m*.
x_0	Starting point, must be a *basic feasible solution*.
B_0	Logical vector of length n for basic variables at start.

Description of Outputs

x	Optimal point.
B	Optimal basic set.
optPar	Optimization parameter vector, see *goptions.m*.
y	Lagrange multipliers.

Description
The routine *lpsimp2* implements the Phase II standard revised Simplex algorithm as formulated in Goldfarb and Todd [41, page 91]. 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).

Examples
See *exinled*, *excycle*, *excycle1*, *excycle2*, *excycle3*, *exKleeM*, *exfl821*, *ex412b4s*, *expertur*.

M-files Used
lpDef.m

See Also
lpsimp1, *lpdual*

Warnings
No check is done whether the given starting point is feasible or not.

14.4.13 maxflow

Purpose
Solve the maximum flow problem.

Calling Syntax
[max_flow, x] = maxflow(s, t, x_U, P, Z, T, R, PriLev)

Description of Inputs

s	The starting node, the source.
t	The end node, the sink.
P	Pointer vector to start of each node in the matrix Z.
x_U	The capacity on each arc.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
T	Trace vector, points to Z with sorting order Head.
R	Pointer vector in T vector for each node.
PriLev	Printing Level: 0 Silent, 1 Print result (default).

Description of Outputs

max_flow	Maximal flow between node s and node t.
x	The flow on each arc.

Description

maxflow finds the maximum flow between two nodes in a capacitated network using the Ford-Fulkerson augmented path method. The implementation is based on the algorithm description in Luenberger [65, page 144-145].

Examples

See *exflow*, *exflow31*, *pathflow*.

14.4.14 modlabel

Purpose

Solve the shortest path problem.

Calling Syntax

[pred, dist] = modlabel(s, P, Z, c)

Description of Inputs

s	The starting node.
p	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.

Description of Outputs

pred	$pred(j)$ is the predecessor of node j.
dist	$dist(j)$ is the shortest distance from node s to node j.

Description

The implementation of *modlabel* is based on the algorithm MODIFIED LABEL CORRECTING in [2, page 262] with the addition of the heuristic rule discussed to improve running time in practice. The rule says: Add *node* to the beginning of $LIST$ if *node* has been in $LIST$ before, otherwise add *node* at the end of $LIST$. The algorithm belongs to the class of *label correcting* methods which are applicable to networks with arbitrary arc lengths. *modlabel* requires that the network does not contain any negative directed cycle, i.e. a directed cycle whose arc lengths sum to a negative value.

Examples

See *exgraph*.

See Also

dijkstra, *labelcor*

Limitations

The network must not contain any negative directed cycle.

14.4.15 NWsimplx

Purpose

Solve the minimum cost network flow problem.

Calling Syntax
[Z, X, xmax, C, S, my, optPar] = NWsimplx(A, b, c, u, optPar)

Description of Inputs

A	Node-arc incidence matrix. A is $m \times n$.
b	Supply/demand vector of length m.
c	Cost vector of length n.
u	Arc capacity vector of length n.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
X	Optimal flow.
xmax	Upper bound on the flow.
C	Costs related to the arcs in the matrix Z.
S	Arc status at the solution: $S_i = 1$, arc i is in the optimal spanning tree. $S_i = 2$, arc i is in L (variable at lower bound). $S_i = 3$, arc i is in U (variable at upper bound).
my	Lagrangian multipliers at the solution.
optPar	Optimization parameter vector, see *goptions.m*.

Description
The implementation of the network simplex algorithm in *NWsimplx* is based on the algorithm NETWORK SIMPLEX in Ahuja et al. [3, page 415]. *NWsimplx* uses the forward and reverse star representation technique of the network, described in [3, pages 35-36].

Examples
See *exmcnfp*.

M-files Used
lpDef.m, *a2frstar.m*

14.4.16 qplm

Purpose
Solve equality constrained quadratic programming problems.

qplm solves problems of the form

$$\begin{array}{ll} \min\limits_{x} & f(x) = \frac{1}{2}(x)^T Fx + c^T x \\ s/t & Ax = b \end{array}$$

where $x, c \in \mathbb{R}^n$, $F \in \mathbb{R}^{n \times n}$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

Calling Syntax
[x, lambda] = qplm(F, c, A, b)

Description of Inputs

F	Constant matrix, the Hessain.
c	Constant vector.
A	Constraint matrix for the linear constraints.
b	Right hand side vector.

Description of Outputs

x	Optimal point.
lambda	Lagrange multipliers.

Description
The routine *qplm* solves a quadratic programming problem, restricted to equality constraints, using the Lagrange method.

See Also
qpBiggs, *qpSolve*, *qpe*

14.4.17 qpe

Purpose
Solve equality constrained quadratic programming problems.

qpe solves problems of the form

$$\begin{array}{ll} \min\limits_{x} & f(x) = \frac{1}{2}(x)^T Fx + c^T x \\ s/t & Ax = b \end{array}$$

where $x, c \in \mathbb{R}^n$, $F \in \mathbb{R}^{n \times n}$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

Calling Syntax
[x, lambda, QZ, RZ] = qpe(F, c, A, b)

Description of Inputs

F	Constant matrix, the Hessain.
c	Constant vector.
A	Constraint matrix for the linear constraints.
b	Right hand side vector.

Description of Outputs

x	Optimal point.
lambda	Lagrange multipliers.
QZ	The matrix Q in the QR-decomposition of F.
RZ	The matrix R in the QR-decomposition of F.

Description
The routine *qpe* solves a quadratic programming problem, restricted to equality constraints, using a null space method.

See Also
qpBiggs, *qpSolve*, *qplm*

14.4.18 salesman

Purpose
Solve the symmetric travelling salesman problem.

Calling Syntax
[Tour, f_tour, OneTree, f_tree, w_max, my_max, optPar] =
salesman(C, Zin, Zout, my, f_BestTour, optPar)

Description of Inputs

C	Cost matrix of dimension $n \times n$ where $C_{ij} = C_{ji}$ is the cost of arc (i, j). If there are no arc between node i and node j then set $C_{ij} = C_{ji} = \infty$. It must hold that $C_{ii} = NaN$.
Zin	List of arcs forced in.
Zout	List of arcs forced out.
my	Lagrange multipliers.
f_BestTour	Cost (total distance) of a known tour.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

Tour	Arc list of the best tour found.
f_tour	Cost (total distance) of the best tour found.
OneTree	Arc list of the best 1-tree found.
f_tree	Cost of the best 1-tree found.
w_max	Best dual objective.
my_max	Lagrange multipliers at *w_max*.
optPar	Optimization parameter vector, see *goptions.m*.

Description
The routine *salesman* is an implementation of an algorithm by Held and Karp [44] which solves the symmetric travelling salesman problem using Lagrangian Relaxation. The dual problem is solved using a subgradient method

with the step length given by the Polyak rule II. The primal problem is to find a 1-tree. Here the routine *mintree* is called to get a minimum spanning tree. With this method there is no guarantee that an optimal tour is found, i.e. a zero duality gap can not be guaranteed. To ensure convergence, *salesman* could be used as a subroutine in a Branch and Bound algorithm, see *travelng* which calls *salesman*.

Examples
See *ulyss16*.

M-files Used
lpDef.m, *mintree.m*

See Also
travelng

14.4.19 TPsimplx

Purpose
Solve transportation programming problems.

TPsimplx solves problems of the form

$$\begin{array}{lrcll} \min\limits_{x} & f(x) & = & \sum\limits_{i}^{m}\sum\limits_{j}^{n} c_{ij}x_{ij} & \\ s/t & \sum\limits_{j}^{n} x_{ij} & = & s_i & i = 1, 2, ..., m \\ & \sum\limits_{i}^{n} x_{ij} & = & d_j & j = 1, 2, ..., n \\ & x & \geq & 0 & \end{array}$$

where $x, c \in \mathbb{R}^{m\times n}$, $s \in \mathbb{R}^m$ and $d \in \mathbb{R}^n$.

Calling Syntax
[X, B, optPar, y, C] = TPsimplx(s, d, C, X, B, optPar, Penalty)

Description of Inputs

s	Supply vector.
d	Demand vector.
C	The cost matrix of linear objective function coefficients.
X	Basic Feasible Solution matrix.
B	Index (i, j) of basis found.
optPar	Optimization parameter vector, see *goptions.m*.
Penalty	If the problem is unbalanced with $\sum_i^m s_i < \sum_j^n d_j$, a dummy supply point is added with cost vector *Penalty*. If the length of *Penalty* $< n$ then the value of the first element in *Penalty* is used for the whole added cost vector. Default: Computed as $10\max(C_{ij})$.

Description of Outputs

X	Solution matrix.
B	Optimal set. Index (i, j) of the optimal basis found.
optPar	Optimization parameter vector, see *goptions.m*.
y	Lagrange multipliers.
C	The cost matrix, changed if the problem is unbalanced.

Description
The routine *TPsimplx* is an implementation of the Transportation Simplex method described in Luenberger [65, chap 5.4]. In LDO, three routines to find a starting basic feasible solution for the transportation problem are included; the Northwest Corner method (*TPnw*), the Minimum Cost method (*TPmc*) and Vogel's approximation method (*TPvogel*). If calling *TPsimplx* without giving a starting point then Vogel's method is used to find a starting basic feasible solution.

Examples
See *extp_bfs*, *exlu119*, *exlu119U*, *extp*.

M-files Used
TPvogel.m

See Also
TPmc, *TPnw*, *TPvogel*

Warnings
No check is done whether the given starting point is feasible or not.

14.4.20 urelax

Purpose
Solve integer linear problems of the form

$$\begin{array}{lccc} \max\limits_{x} & f(x) & = & c^T x \\ s/t & Ax & \leq & b \\ & x & \leq & x_U \\ & x_j \in \mathbb{N} & & j = 1, 2, ..., n \end{array}$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{N}^{m \times n}$ and $b \in \mathbb{N}^m$.

Calling Syntax
[x_P, u, f_P] = urelax(u_max, A, b, c, r, x_U, optPar)

Description of Inputs

u_max	Upper bounds on u.
A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
r	Constraint not to be relaxed.
x_U	Upper bounds on the variables.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

x_P	Primal solution.
u	Lagrangian multipliers.
f_P	Function value at *x_P*.

Description
The routine *urelax* is a simple example of the use of Lagrangian Relaxation to solve integer linear programming problems. The problem is solved by relaxing all but one constraint and then solve a simple knapsack problem as a subproblem in each iteration. *urelax* plots the result of each iteration. LDO also contains a more sophisticated routine, *ksrelax*, for solving problems of this type.

Examples
See *exip39rx*.

M-files Used
dpknap.m

See Also
ksrelax

14.5 Optimization Subfunction Utilities in TOMLAB LDO

In the following subsections the optimization subfunction utilities in TOMLAB LDO will be described.

14.5.1 a2frstar

Purpose
Convert a node-arc incidence matrix representation of a network to the forward and reverse star data storage representation.

Calling Syntax
[P, Z, c, T, R, u] = a2frstar(A, C, U)

Description of Inputs

A	The node-arc incidence matrix. A is $m \times n$, where m is the number of arcs and n is the number of nodes.
C	Cost for each arc, n-vector.
U	Upper bounds on flow (optional).

Description of Outputs

P	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.
T	Trace vector, points to Z with sorting order Head.
R	Rewerse pointer vector in T for each node.
u	Upper bounds on flow if U is given as input, else infinity.

Description
The routine *a2frstar* converts a node-arc incidence matrix representation of a network to the forward and reverse star data storage representation as described in Ahuja et.al. [3, pages 35-36].

Examples
See *exflow*, *exflow31*, *exgraph*, *pathflow*.

14.5.2 gsearch

Purpose
Find all nodes in a network which is reachable from a given source node.

Calling Syntax
[pred, mark] = gsearch(s, P, Z, c)

Description of Inputs

s	The starting node.
P	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.

Description of Outputs

pred	$pred(j)$ = Predecessor of node j.
mark	If $mark(j) = 1$ the node is reachable from node s.

Description
gsearch is searching for all nodes in a network which is reachable from the given source node s. The implementation is a variation of the Algorithm SEARCH in [2, pages 231-233]. The algorithm uses a depth-first search which means that it creates a path as long as possible and backs up one node to initiate a new probe when it can mark no new nodes from the tip of the path. A stack approach is used where nodes are selected from the front and added to the front.

Examples
See *exgraph*.

See Also
gsearchq

14.5.3 gsearchq

Purpose
Find all nodes in a network which is reachable from a given source node.

Calling Syntax
[pred, mark] = gsearchq(s, P, Z, c)

Description of Inputs

s	The starting node.
P	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.

Description of Outputs

pred	$pred(j)$ = Predecessor of node j.
mark	If $mark(j) = 1$ the node is reachable from node s.

Description
gsearchq is searching for all nodes in a network which is reachable from the given source node s. The implementation is a variation of the Algorithm SEARCH in [2, pages 231-233]. The algorithm uses a breadth-first search which means that it visits the nodes in order of increasing distance from s. The distance being the minimum number of arcs in a directed path from s. A queue approach is used where nodes are selected from the front and added to the rear.

Examples
See *exgraph*.

See Also
gsearch

14.5.4 mintree

Purpose
Find the minimum spanning tree of an undirected graph.

Calling Syntax
[Z_tree, cost] = mintree(C, Zin, Zout)

Description of Inputs

C	Cost matrix of dimension $n \times n$ where $C_{ij} = C_{ji}$ is the cost of arc (i, j). If there are no arc between node i and node j then set $C_{ij} = C_{ji} = \infty$. It must hold that $C_{ii} = NaN$.
Zin	List of arcs which should be forced to be included in *Z_tree*.
Zout	List of arcs which should not be allowed to be included in *Z_tree* (could also be given as NaN in C).

Description of Outputs

Z_tree	List of arcs in the minimum spanning tree.
cost	The total cost.

Description
mintree is an implementation of Kruskal's algorithm for finding a minimal spanning tree of an undirected graph. The implementation follows the algorithm description in [3, page 520-521]. It is possible to give as input, a list of those arcs which should be forced to be included in the tree as well as a list of those arcs which should not be allowed to be included in the tree. *mintree* is called by *salesman*.

14.5.5 TPmc

Purpose
Find a basic feasible solution to the Transportation Problem.

Calling Syntax
[X,B] = TPmc(s, d, C)

Description of Inputs

s	Supply vector of length m.
d	Demand vector of length n.
C	The cost matrix of linear objective function coefficients.

Description of Outputs

X	Basic feasible solution matrix.
B	Index (i, j) of the basis found.

Description

TPmc is an implementation of the Minimum Cost method for finding a basic feasible solution to the transportation problem. The implementation of this algorithm follows the algorithm description in Winston [83, chap. 7.2].

Examples

See *extp_bfs*, *exlu119*, *exlu119U*, *extp*.

See Also

TPnw, *TPvogel*, *TPsimplx*

14.5.6 TPnw

Purpose

Find a basic feasible solution to the Transportation Problem.

Calling Syntax

[X, B] = TPnw(s, d)

Description of Inputs

s	Supply vector of length m.
d	Demand vector of length n.

Description of Outputs

X	Basic feasible solution matrix.
B	Index (i, j) of the basis found.

Description

TPnw is an implementation of the Northwest Corner method for finding a basic feasible solution to the transportation problem. The implementation of this algorithm follows the algorithm description in Winston [83, chap. 7.2].

Examples

See *extp_bfs*, *exlu119*, *exlu119U*, *extp*.

See Also

TPmc, *TPvogel*, *TPsimplx*

14.5.7 TPvogel

Purpose

Find a basic feasible solution to the Transportation Problem.

Calling Syntax

[X, B] = TPvogel(s, d, C, PriLev)

Description of Inputs

s	Supply vector of length m.
d	Demand vector of length n.
C	The cost matrix of linear objective function coefficients.
$PriLev$	If $PriLev > 0$, the matrix X is displayed in each iteration. If $PriLev > 1$, pause in each iteration. Default: $PriLev = 0$.

Description of Outputs

X	Basic feasible solution matrix.
B	Index (i, j) of the basis found.

Description
TPvogel is an implementation of Vogel's method for finding a basic feasible solution to the transportation problem. The implementation of this algorithm follows the algorithm description in Winston [83, chap. 7.2].

Examples
See *extp_bfs*, *exlu119*, *exlu119U*, *extp*.

See Also
TPmc, *TPnw*, *TPsimplx*

14.5.8 z2frstar

Purpose
Convert a table of arcs and corresponding costs in a network to the forward and reverse star data storage representation.

Calling Syntax
[P, Z, c, T, R, u] = z2frstar(Z, C, U)

Description of Inputs

Z	A table with arcs (i, j). Z is $n \times 2$, where n is the number of arcs. The number of nodes m is set equal to the greatest element in Z.
C	Cost for each arc, n-vector.
U	Upper bounds on flow (optional).

Description of Outputs

P	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:, 1)$ Tail. $Z(:, 2)$ Head.
c	Costs related to the arcs in the matrix Z.
T	Trace vector, points to Z with sorting order Head.
R	Rewerse pointer vector in T for each node.
u	Upper bounds on flow if U is given as input, else infinity.

Description
The routine *z2frstar* converts a table of arcs and corresponding costs in a network to the forward and reverse star data storage representation as described in Ahuja et.al. [3, pages 35-36].

A Description of 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 behaviour 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 39, 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 34), 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 version of TOMLAB.

Table 31: Information stored in the problem structure *Prob*, part I. Fields defining sub-structures are defined in Table 33

.

Field	Description
A	Matrix with linear constraints, one constraint per row (dense or sparse).
AutoDiff	If true, use automatic differentiation.
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.
cName	Name of each general constraint.
CHECK	If true, no more check is done by ProbCheck. Set to true (=1) after first call to ProbCheck.
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.
ConsPattern	Matrix with non-zero pattern in the constraint gradient matrix.
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.
HessPattern	Matrix with non-zero pattern in the Hessian matrix.
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.
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).
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 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.

Table 32: Information stored in the problem structure *Prob*, part II. Fields defining sub-structures are defined in Table 33

.

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 SOL solver interface (0).
probFile	Name of m-file in which the problems are defined.
probType	TOMLAB problem type, see Table 1.
SolverDLP	Name of the solver that should solve dual LP sub-problems. Used by SolveDLP.
SolverLP	Name of the Solver that should solve LP sub-problems. Used by SolveLP.
SolverFP	Name of the solver that should solve a phase one LP sub-problem, i.e. finding a feasible point to a convex set. Used by SolveFP.
SolverQP	Name of the solver that should solve QP sub problems. Used by SolveQP.
uP	User supplied parameters for the problem.
uPName	Problem name (*Prob.Name*) connected to the user supplied parameters in *(* uP).
WarmStart	Flag. If > 0 the solver should do a warm start, if the solver supports a warm start.
x_0	Starting point.
x_L	Lower 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.
x_U	Upper bounds on the variables x.
xName	Name of each decision variable in x.

Table 33: 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 34.
LS	Structure with special fields for least squares problems, see Table 35.
MIP	Structure with special fields for mixed-integer programming, see Table 36.
ExpFit	Structure with special fields for exponential fitting problems, see Table 37.
LineParam	Structure with special fields for line search optimization parameters, see Table 38.
optParam	Structure with special fields for optimization parameters, see Table 39.
PartSep	Structure with special fields for partially separable functions, see Table 40.
SOL	Structure with special fields for SOL (Stanford Optimization Laboratory) solvers, see Table 41.
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 13.
USER	Structure with user defined names of the m-files computing the objective, gradient, Hessian etc. See Table 42. These routines are called from the corresponding gateway routine

Table 34: 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 39.
SOL	Structure with special fields for SOL (Stanford Optimizaton Laboratory) solvers, see Table 41.
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 13.

Table 35: 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 36: 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 37: 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 38: 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 39: Information stored in the structure *Prob.optParam*. Default values in parenthesis. Items marked (SOL-n) are corresponding to SOL parameters, and are given different initial values dependent on which solver is used. The number n gives the element number used in the *optParam* parameter vector in *Prob.SOL*.

Field	Description
PriLev	Solver major print level in file output (SOL-1).
PriFreq	Print frequency (SOL-5) in optimization solver.
SummFreq	Summary frequency (SOL-6) in optimization solver.
MinorPriLev	Minor print level in file output (SOL-2) in SOL sub-problem solver.
IterPrint	Flag for one-row-per-iteration printout during optimization (0). If SOL-1 not explicitely set, setting this flag will set SOL-1 to 1 for the SOL solvers.
wait	Flag, if true use pause statements after output in each iteration (0).
MaxFunc	Maximal number of function evaluations.
MaxIter	Maximum number of iterations (SOL-30).
MajorIter	Maximum number of iterations in major problem (SOL-35).
MinorIter	Maximum number of iterations in minor problem (SOL-36).
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 (SOL-10).
eps_dirg	Convergence tolerance for the directed derivative (10^{-8}).
eps_c	Feasibility tolerance for nonlinear constraints (SOL-9).
eps_g	Gradient (or reduced gradient) convergence tolerance (10^{-7}).
eps_Rank	Rank test tolerance (SOL-27).
EpsGlob	Global/local weight parameter in global optimization (10^{-4}).
fTol	Relative accuracy in the computation of the function value (SOL-41).
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 (SOL-11).
cTol	Feasibility tolerance for nonlinear constraints (SOL-9).
MinorTolX	Relative convergence tolerance in parameters x in sub-problem (SOL-12).
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).
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 (3,SOL-39).
GradCheck	$0, 1, 2, 3$ gives increasing level of user-supplied gradient checks (SOL-13).
DiffInt	Difference interval in derivative estimates (SOL-42).
CentralDiff	Central difference interval in derivative estimates (SOL-43).
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 (0.4).
splineTol	Tolerance parameter sent to the SPLINE Toolbox routine *spaps.m* when computing numerical approximations of the derivatives (10^{-3}).
BigStep	Unbounded step size. Used to detect unbounded nonlinear problems. (SOL-45).
BigObj	Unbounded objective value. Used to detect unbounded nonlinear problems. (SOL-46).
CHECK	If true, no more check is done on the structure. Set to true (=1) after first call to *optParamSet*.

Table 40: 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 41: 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 42: Information stored in the structure *Prob.USER*

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.USER.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 43: Information stored in the structure *Prob.DUNDEE*

Field	Description
callback	If 1, use a callback to Matlab to compute $QP.F \cdot x$. 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.PriLevOpt*. Default name "bqpd.txt".
optPar	Vector with optimization parameters. -999 in any element gives default. Length between 0 and 7 allowed. Default values:
optPar(1) (tol)	$1 \cdot 10^{-10}$ Relative accuracy in solution.
optPar(2) (emin)	1.0 1.0: Use cscale (constraint scaling), 0.0 no scaling.
optPar(3) (sgnf)	$5 \cdot 10^{-4}$ Max relative error in two numbers equal in exact arithmetic.
optPar(4) (nrep)	2 Maximum number of refinement steps.
optPar(5) (npiv)	3 No repeat of more than *npiv* steps were taken.
optPar(6) (nres)	2 Maximum number of restarts if unsuccessful.
optPar(7) (nfreq)	500 The maximum interval between refactorizations.

B Description of Result, the optimization 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 46. 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 44. The information stored in the mat-file *glcSave.mat* by the solver *glcSolve* is shown in Table 45.

Table 44: Information stored in the mat-file *glbSave.mat* by the solver *glbSolve*. Used for automatic restarts.

Variable	**Description**
Name	Name of the problem, used as identification.
C	Matrix with all rectangle centerpoints in original coordinates.
F	Vector with function values.
D	Vector with distances from centerpoints to the vertices.
L	Matrix with all rectangle side lengths in each dimension.
d	Row vector of all different distances, sorted.
d_min	Row vector of minimum function value for each distance.
f_min	Best function value found at a feasible point.
E	Computed tolerance, dependent on f_min.
i_min	indices for all best points.

Table 45: Information stored in the mat-file *glcSave.mat* by the solver *glcSolve*. Used for automatic restarts.

Variable	**Description**
Name	Name of the problem, used as identification.
C	Matrix with all rectangle centerpoints in original coordinates.
F	Vector with function values.
T	$T(i)$ is the number of times rectangle i has been trisected.
D	Vector with distances from centerpoint to the vertices.
G	Matrix with constraint values for each point.
I_L	$I\_L(i,j)$ is the lower bound for rectangle j in integer dimension $I(i)$.
I_U	$I\_U(i,j)$ is the upper bound for rectangle j in integer dimension $I(i)$.
s_0	s_0 is used as $s(0)$.
s	$s(j)$ is the sum of observed rates of change for constraint j.
t	$t(i)$ is the total number of splits along dimension i.
ignoreidx	Rectangles to be ignored in the rectangle selection procedure.
feasible	Flag indicating if a feasible point has been found.
Split	$Split(i,j)$ is the number of splits along dimension i of rectangle j.
f_min	Best function value found at a feasible point.

The field *xState* describes the state of each of the variables. In Table 47 the different values are described. The different conditions for linear constraints are defined by the state variable in field *bState*. In Table 48 the different values are described.

Table 46: Information stored in the global Matlab structure *Result*.

Field	Description
Name	Problem name.
P	Problem number.
probType	TOMLAB problem type, according to Table 1, page 10.
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).
FuncEv	Number of function evaluations needed.
GradEv	Number of gradient evaluations needed.
HessEv	Number of Hessian evaluations needed.
ConstrEv	Number of constraint evaluations needed.
ResEv	Number of residual evaluations needed (least squares).
JacEv	Number of Jacobian evaluations needed (least squares).
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.
x_0	Starting point.
f_0	Function value at start i.e. $f(x\_0)$.
x_k	Optimal point.
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.
c_k	Value of constraints at optimum.
cJac	Constraint Jacobian at optimum.
xState	State of each variable, described in Table 47.
bState	State of each linear constraint, described in Table 48.
cState	State of each general constraint, described in Table 49.
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 50.
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 34.
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 41. The routine *WarmDefSOL* moves the relevant fields back to to *Prob.SOL* for the subsequent call.
plotData	Structure with plotting parameters.
Prob	Problem structure, see Table 31 and Table 32. Please note that certain solvers that do reformulations of the problem, such as *L1Solve*, *infSolve* and *slsSolve* return the problem structure of the *reformulated* problem in this field, *not* the original one.

Table 47: 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 48: 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 49: 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 50: 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 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 51 and 52.

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 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 51: 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.
CUTEPATH	The path ending with \cute.
CUTEDLL	Name of CUTE DLL file.
DLLPATH	Full path to the CUTE DLL file.
CUTE_g	Gradient.
CUTE_H	Hessian.
CUTE_Hx	Value of x when computing *CUTE_H*.
CUTE_dc	Constraint normals.
CUTE_Equal	Binary vector, element i equals 1 if constraint i is an equality constraint.
CUTE_Linear	Binary vector, element i equals 1 if constraint i is a linear constraint.
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 52: 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 31 and Table 32.
Result	Result structure, see Table 46.
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.

D Editing Init Files directly

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. Two formats are defined, the TOMLAB Quick format (TQ format) and the Init File format (IF format). The TQ 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.

The definition of an advanced general graphical user interface (GUI) and a similar menu system demanded a more complicated format. The solution is the IF format, where groups of problems are collected into sets, each set having an initialization file. Besides defining the problem, a list of all problems in the set is also generated by the initialization file.

In the following sections detailed descriptions are given on how to edit the Init Files for different types of problems, linear programming, quadratic programming, unconstrained optimization, box-bounded global optimization, global mixed-integer nonlinear programming and constrained optimization.

D.1 Editing New Problems in Linear Programming Init Files

The step wise description below shows how to edit new problems into an existing Init File for LP problems. The example shows how to add one new problem in *lp_prob.m*, the default file for LP problems in TOMLAB. As test example choose the same test example as in Section 5.1, (11), here called *lptest1*.

1. Copy **File:** tomlab/testprob/lp_prob to e.g. *lpnew_prob.m* in a working directory.
2. Edit *lpnew_prob.m*. Add the problem name, *lptest1*, to the menu choice:

```
...
      ,'Winston Ex. 4.12 B4. Max || ||. Rewritten'...
      ,'lptest1'...
       ); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

if isempty(P)
   return;
end
...
```

3. There are twelve problems defined in *lp_prob.m*, making thirteen the new problem number. Define the constraint matrix A, the upper bounds for the constraints b_U, the cost vector c as below. If the constraints would be of equality type then define the lower bounds for the constraints b_L equal to b_U.

```
...
elseif P == 13
   Name  = 'lptest1';
   c     = [-7 -5]';
   A     = [ 1  2
             4  1 ];
   b_U   = [ 6 12 ]';
   x_L   = [ 0  0 ]';
   x_min = [ 0  0 ]';
   x_max = [10 10 ]';
else
   disp('lp_prob: Illegal problem number')
   pause
   Name=[];
end
...
```

4. Note that because the file name is changed, the new name must be substituted. There are three places where the name should be changed, the function definition in the beginning of the file, the *disp* statement above and the place shown in the following text

```
...
if ask==-1 & ~isempty(Prob)
   if isstruct(Prob)
      if ~isempty(Prob.P)
         if P==Prob.P & strcmp(Prob.probFile,'lpnew_prob'), return; end
      end
   end
end
...
```

5. Save the file properly.

6. Run

```
AddProblemFile('lpnew_prob','lp_prob with extra user problems','lp');
```

7. To solve this problem the following statements may be used.

```
Prob   = probInit('lpnew_prob', 13);  % Get Prob structure.
...                                   % Define changes in Prob structure
Result = tomRun('lpSolve', Prob);     % Call lpSolve using driver routine
```

It is also possible to define the optional parameters B, f_min and x_0 as described in the problem definition description in *lp_prob.m*. If B and x_0 are not given, as in this case, a Phase I linear program is solved to an initial feasible solution point.

Now *lpnew_prob.m* is one of the TOMLAB Init Files and all problems in *lpnew_prob.m* are accessible from the GUI, the menu systems, and the driver routines. The edited file is found in **File:** tomlab/usersguide/lpnew_prob .

D.2 Editing New Problems in Quadratic Programming Init Files

The step wise description below shows how to edit new problems into an existing QP Init File. The example shows how to add one new problem in *qp_prob.m*, the default file for QP problems in TOMLAB. The test example is the same as in Section 5.2, problem (13), named *QP EXAMPLE*.

1. Copy **File:** tomlab/testprob/qp_prob to e.g. *qpnew_prob.m* in a working directory.
2. Edit *qpnew_prob.m*. Add the problem name, *QP Example*, to the menu choice:

```
...
     ,'Bazaara IQP 9.29b pg 405. F singular'...
     ,'Bunch and Kaufman Indefinite QP'...
     ,'QP EXAMPLE'...
       ); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

if isempty(P)
   return;
end
...
```

3. There are fourteen problems defined in *qp_prob.m*, making fifteen the new problem number. Add the following in *qpnew_prob.m* after the last already existing problem:

```
...
elseif P==15
   Name='QP EXAMPLE';
   F = [ 8   2          % Hessian
         2   8 ];
   c = [ 3  -4 ]';
   A = [ 1   1          % Constraint matrix
         1  -1 ];
   b_L = [-inf  0  ]';  % Lower bounds on the constraints
   b_U = [  5   0  ]';  % Upper bounds on the 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 parameters
   x_max=[ 6  6 ];      % Plot region parameters
else
   disp('qp_prob: Illegal problem number')
   pause
   Name=[];
end
...
```

4. Note that because the file name is changed, the new name must be substituted. There are three places where the name should be changed, the function definition in the beginning of the file, the *disp* statement above and the place shown in the following text

```
...
if ask==-1 & ~isempty(Prob)
   if isstruct(Prob)
      if ~isempty(Prob.P)
         if P==Prob.P & strcmp(Prob.probFile,'qpnew_prob'), return; end
      end
   end
end
...
```

5. Save the file properly.
6. Run

```
AddProblemFile('qpnew_prob','qp_prob with extra user problems','qp');
```

7. To solve this problem the following statements may be used.

```
Prob   = probInit('qpnew_prob', 15);  % Get Prob structure.
...                                   % Define changes in Prob structure
Result = tomRun('qpSolve', Prob);     % Call qpSolve using driver routine
```

Now *qpnew_prob.m* is one of the TOMLAB Init Files in the GUI data base and all problems in *qpnew_prob.m* are accessible from the GUI, the menu systems, and the driver routines. The edited file is found in **File:** tomlab/usersguide/qpnew_prob .

D.3 Editing New Problems in Unconstrained Optimization Init Files

The step wise description below shows how to edit new problems into an existing unconstrained optimization (UC) Init File. The example shows how to add one new problem in *uc_prob.m*, the default file for UC problems in TOMLAB. As test example the problem (15) in Section 6 is used, The m-file code for the objective function for this problem is given in **File:** tomlab/usersguide/rbb_f.m .

1. Copy **File:** tomlab/testprob/uc_prob to e.g. *ucnew_prob.m* in a working directory. Also copy **File:** tomlab/testprob/uc_f , **File:** tomlab/testprob/uc_g and **File:** tomlab/testprob/uc_H to the working directory, and change the names to *ucnew_f.m*, *ucnew_g.m* and *ucnew_H.m*.

2. Add the problem name to the menu choice in *ucnew_prob.m*:

```
...
      ,'Fletcher Q.2.6'...
      ,'Fletcher Q.3.3'...
      ,'RB BANANA'...
           ); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

if isempty(P)
   return;
end
...
```

3. There are seventeen problems defined in *ucnew_prob.m*, making eighteen the new problem number. Add the following in *ucnew_prob.m* after the last already existing problem (the optional parameters are not necessary to define): Note that the routine *checkuP* is used to check if the user parameter in the *Prob* structure sent to this routine is defined for the correct problem. The routine *askparam* then takes care of setting the proper user parameter dependent on what value the flag *ask* is set to.

```
...
elseif P == 18
   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.
   x_opt = [1 1];       % Known optimal point (optional).
   f_opt = 0;           % Known optimal function value (optional).
   f_min = 0;           % Lower bound on function (optional).
   x_max = [ 1.3  1.3]; % Plot region parameters.
   x_min = [-1.1 -0.2]; % Plot region parameters.
   % The following lines show how to use the advanced user parameter
   % definition facility in TOMLAB.
   uP    = checkuP(Name,Prob);  % Check if given uP is for this problem
   % Ask the following question if flag set to ask questions
   uP(1) = askparam(ask,'Give the nonlinear factor in Rosenbrocks banana: ',...
                    0,[],100,uP);
% CHANGE: elseif P == 18
% CHANGE: Add an elseif entry and the other variable definitions needed
...
```

4. Note that because the file name is changed, the new name must be substituted. There are three places where the name should be changed, the function definition in the beginning of the file, a *disp* statement and most important the place shown in the following text

```
...
if ask==-1 & ~isempty(Prob)
```

```
   if isstruct(Prob)
      if ~isempty(Prob.P)
         if P==Prob.P & strcmp(Prob.probFile,'ucnew_prob'), return; end
      end
   end
end
...
```

5. The names of the m-files are also changed and must be changed on two rows

```
...
% Define the m-files that compute the function value, the gradient vector
% and the Hessian matrix

%MIDEVA
%# call   ucnew_f ucnew_g ucnew_H

Prob=mFiles(Prob,'ucnew_f','ucnew_g','ucnew_H');
...
```

6. The function value, gradient vector and Hessian routine must now be edited. The function names should also be renamed, but Matlab does not care what the name is of the routine, only what the file name is, so it is optional to change the name of each function. Make the following addition in *ucnew_f.m*:

```
...
elseif P == 17  % Fletcher Q.3.3
   f = 0.5*(x(1)^2+x(2)^2)*exp(x(1)^2-x(2)^2);
elseif P == 18  % RB BANANA
   f = Prob.uP(1)*(x(2)-x(1)^2)^2 + (1-x(1))^2;
end
...
```

7. Make the following addition in *ucnew_g.m*:

```
...
elseif P == 17  % Fletcher Q.3.3
   %f = 0.5*(x(1)^2+x(2)^2)*exp(x(1)^2-x(2)^2);
   e = exp(x(1)^2-x(2)^2);
   g = e*[x(1)*(1+x(1)^2+x(2)^2); x(2)*(1-x(1)^2-x(2)^2)];
elseif P == 18  % RB BANANA
   g = Prob.uP(1)*[-4*x(1)*(x(2)-x(1)^2)-2*(1-x(1)); 2*(x(2)-x(1)^2)   ];
end
...
```

8. Make the following addition in *ucnew_H.m*:

```
...
elseif P == 17  % Fletcher Q.3.3
   %f = 0.5*(x(1)^2+x(2)^2)*exp(x(1)^2-x(2)^2);
   %g = e*[x(1)*(1+x(1)^2+x(2)^2); x(2)*(1-x(1)^2-x(2)^2)];
   e = exp(x(1)^2-x(2)^2);
   H = [1+5*x(1)^2+2*x(1)^2*x(2)^2+x(2)^2+2*x(1)^4, ...
        -2*x(1)*x(2)*(x(1)^2+x(2)^2);
        0 , 1-x(1)^2+2*x(1)^2*x(2)^2-5*x(2)^2+2*x(2)^4 ];
   H(2,1)=H(1,2);
   H = e*H;
elseif P == 18  % RB BANANA
```

```
   H = Prob.uP(1)*[ 12*x(1)^2-4*x(2)+2 , -4*x(1);
                    -4*x(1)        ,    2  ];
end
...
```

9. Save all the files properly.

10. Run

```
AddProblemFile('ucnew_prob','uc_prob with extra user problems','uc');
```

11. To solve this problem the following statements may be used.

```
Prob   = probInit('ucnew_prob', 18);  % Get Prob structure.
...                                  % Define changes in Prob structure
Result = tomRun('ucSolve', Prob);    % Call ucSolve using driver routine
```

Now *ucnew_prob.m* is one of the TOMLAB Init Files in the GUI data base and all problems in *ucnew_prob.m* are accessible from the GUI, the menu system, and the driver routines. The edited Init File is found in **File:** tomlab/usersguide/ucnew_prob . In the same directory the function, gradient and Hessian routines are found.

D.4 Editing New Problems in Box-bounded Global Optimization Init Files

Box-bounded global optimization problems are defined in the same way as unconstrained optimization problems. Since no derivative information is used, only the Init File definition file and the routine to compute the objective function value have to be modified. The step wise description below shows how to edit a new problem into an existing Init File, in this case the predefined *glb_prob*, and the objective function routine *glb_f*. As test example use the *Rosenbrock's banana* problem

$$\begin{array}{ll} \min\limits_{x} & f(x) = \alpha \left(x_2 - x_1^2\right)^2 + (1 - x_1)^2 \\ s/t & \begin{array}{rcccl} -2 & \leq & x_1 & \leq & 2 \\ -2 & \leq & x_2 & \leq & 2 \end{array} \end{array} \tag{25}$$

The standard value is $\alpha = 100$. To define (25) as a box-bounded global optimization problem follow the step wise instructions below (for all instructions it is assumed that the files are edited in a text editor). Note that in this example the lower variable bounds are changed to $x_L = (-2, -2)^T$. The reason for that is to speed up the global search for the reader who wants to run this example. It is always important to make the box-bounded region as small as possible.

1. Copy the files *glb_prob.m* and *glb_f.m* to a working directory.

2. Add the problem name to the menu choice in *glb_prob.m*:

```
...
      ,'HGO 468:2'...
      ,'Spiral'...
      ,'RB BANANA'...
           ); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

if isempty(P)
   return;
end
...
```

3. Add the following in *glb_prob.m* after the last already existing problem (the optional parameters are not necessary to define):

```
...
elseif P == 33
   Name='RB BANANA';
   x_L = [-2;-2];     % Lower bounds for x.
   x_U = [ 2; 2];     % Upper bounds for x.
   x_opt = [1 1];     % Known optimal point (optional).
   f_opt = 0;         % Known optimal function value (optional).
   n_global = 1;      % Number of global minima (optional).
   n_local  = 1;      % Number of local minima (optional).
   K = [];            % Lipschitz constant, not used.
   x_max = [ 2  2];   % Plot region parameters.
   x_min = [-2 -2];   % Plot region parameters.
...
```

4. Make the following addition in *glb_f.m*:

```
...
elseif P == 33  % RB BANANA
   f = 100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
end
...
```

5. Save all the files properly.

6. To solve this problem the following statements may be used.

```
Prob   = probInit('glb_prob', 33);    % Get Prob structure.
...                                   % Define changes in Prob structure
Result = tomRun('glbSolve', Prob);    % Call glbSolve using driver routine
```

   Note that the working directory in Matlab must be the directory where the new files have been edited, otherwise the predefined files in TOMLAB with the same names will be used.

D.5 Editing New Problems in Global Mixed-Integer Nonlinear Programming Init Files

To illustrate how to define a global mixed-integer nonlinear programming problem in the Init File format, add constraints to the test problem *Rosenbrock's banana*,

$$\begin{array}{lrcccl} \min\limits_{x} & f(x) = \alpha\left(x_2 - x_1^2\right)^2 + (1-x_1)^2 & & & & \\ & -2 & \leq & x_1 & \leq & 2, \quad x_1 \text{ integer} \\ s/t & -2 & \leq & x_2 & \leq & 2 \\ & & & x_1 - x_2 & \leq & 1 \\ & & & -x_1^2 - x_2 & \leq & 1 \end{array} \tag{26}$$

The standard value is $\alpha = 100$ Note the additional constraint that x_1 must be integer. The third constraint is linear and is therefore defined separately from the nonlinear fourth constraint.

To define (26) as a global mixed-integer nonlinear programming problem follow the step wise instructions below (for all instructions it is assumed that the files are edited in a text editor).

1. Copy the files *glc_prob.m*, *glc_f.m* and *glc_c.m* to the working directory.
2. Modify the files *glc_prob.m* and *glc_f.m* in the same way as described for for the box-bounded case in Section D.4.
3. Extend the problem definition in *glc_prob.m* with the constraint parameters:

```
...
elseif P == 25
   Name='RB BANANA';
   x_L = [-2;-2];     % Lower bounds for x.
   x_U = [ 2; 2];     % Upper bounds for x.
   x_opt = [1 1];     % Known optimal point (optional).
   f_opt = 0;         % Known optimal function value (optional).
   A   = [1 -1];      % Linear constraints matrix.
   b_L = -inf;        % Lower bounds on linear constraints.
   b_U = 1;           % Upper bounds on linear constraints.
   c_L = -inf;        % Lower bounds on nonlinear constraints.
   c_U = 1;           % Upper bounds on nonlinear constraints.
   IntVars = 1;       % Indices for integer constrained variables.
   n_global = 1;      % Number of global minima (optional).
   n_local  = 1;      % Number of local minima (optional).
   K = [];            % Lipschitz constant, not used.
   x_max = [ 2  2];   % Plot region parameters.
   x_min = [-2 -2];   % Plot region parameters.
end
...
```

4. Make the following addition in *glc_c.m*:

```
...
elseif P == 25 % RB BANANA
   cx = -x(1)^2 - x(2);
end
...
```

5. Save all the files properly.
6. To solve this problem the following statements may be used.

```
Prob   = probInit('glc_prob', 25);    % Get Prob structure.
...                                   % Define changes in Prob structure
Result = tomRun('glcSolve', Prob);    % Call glcSolve using driver routine
```

Note that the working directory in Matlab must be the directory where the new files have been edited, otherwise the predefined files in TOMLAB with the same names will be used.

D.6 Editing New Problems in Constrained Optimization Init Files

To illustrate how to define a constrained problem in the Init File format, add two constraints to the *Rosenbrock's banana* problem,

$$\begin{array}{lrcccl} \min\limits_{x} & f(x) = \alpha\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2 & & & & \\ & -10 & \leq & x_1 & \leq & 2 \\ & -10 & \leq & x_2 & \leq & 2 \\ s/t & & & x_1 - x_2 & \leq & 1 \\ & & & -x_1^2 - x_2 & \leq & 1 \end{array} \quad . \tag{27}$$

The standard value is $\alpha = 100$. The first two constraints are simple bounds on the variables. The third constraint is linear and treated separately from the fourth nonlinear inequality constraint.

The problem will be defined by following the step wise instructions below (for all instructions it is assumed that the files are edited in a text editor):

1. Copy the files *con_prob.m*, *con_f.m*, *con_g.m*, *con_H.m*, *con_c.m*, *con_dc.m* and *con_d2c.m* to the working directory.

2. Modify the files *con_prob.m*, *con_f.m*, *con_g.m* and *con_H.m* in the same way as described for for the unconstrained case in Section D.3.

3. Extend the problem definition in *con_prob.m* with the constraint parameters:

```
...
elseif P == 15
   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.
   x_opt = [1 1];         % Known optimal point (optional).
   f_opt = 0;             % Known optimal function value (optional).
   f_min = 0;             % Lower bound on function (optional).
   x_max = [ 1.3  1.3];   % Plot region parameters.
   x_min = [-1.1  -0.2];  % Plot region parameters.

   A   = [1 -1];          % Linear constraints matrix.
   b_L = -inf;            % Lower bounds on linear constraints.
   b_U = 1;               % Upper bounds on linear constraints.
   c_L = -inf;            % Lower bounds on nonlinear constraints.
   c_U = 1;               % Upper bounds on nonlinear constraints.
end
...
```

4. Make the following addition in *con_c.m*:

```
...
elseif P == 15  % RB BANANA
   cx = -x(1)^2 - x(2);
end
...
```

5. Make the following addition in *con_dc.m*:

```
...
elseif P == 15  % RB BANANA
   if init==0
      dc = [-2*x(1) -1];
```

```
   else
      dc = ones(1,2);
   end
end
...
```

6. Make the following addition in *con_d2c.m*:

```
...
elseif P == 15  % RB BANANA
   if init==0
      d2c = [-2 0;0 0]*lam;
   else
      d2c = [1 0; 0 0];
   end
end
...
```

7. Save all the files properly.

8. To solve this problem the following statements may be used.

```
   Prob   = probInit('con_prob', 15);    % Get Prob structure.
   ...                                   % Define changes in Prob structure
   Result = tomRun('conSolve', Prob);    % Call conSolve using driver routine
```

D.7 Creating a New Constrained Optimization Init File

Assume a collection of e.g. constrained problems should be defined in new problem definition files. Also assume the problems have been defined in *con_prob*, *con_f*, *con_g*, *con_H*, *con_c* and *con_dc* as described in Section D.6. Of course it is possible to remove the already existing problems and define the first new problem as number one. The extra modifications needed are:

1. Rename the files to for example *connew_prob*, *connew_f*, *connew_g*, *connew_H*, *connew_c* and *connew_dc*.
2. Make the following modification in the beginning of *connew_prob*:

```
...
if ask==-1 & ~isempty(Prob)
   if isstruct(Prob)
      if ~isempty(Prob.P)
         if P==Prob.P & strcmp(Prob.probFile,'connew_prob'), return; end
      end
   end
end
...
```

3. Make the following modifications at the end of *connew_prob*:

```
...
%MIDEVA
%# call   connew_f connew_g connew_H connew_c connew_dc

Prob=mFiles(Prob,'connew_f','connew_g','connew_H','connew_c','connew_dc');
...
```

4. Save all the files properly.
5. Run

```
AddProblemFile('connew_prob','New Constrained Test Problems','con');
```

   The text *New Constrained ...* is of course possible to change as the user likes. Now *connew_prob.m* is one of the TOMLAB Init Files in the GUI data base and all problems in *connew_prob.m* are accessible from the GUI, the menu system, and the driver routines. This file is also the default file for constrained problems.

D.8 Editing New Problems in Nonlinear Least Squares Init Files

To define (15) as a nonlinear least squares problem follow the step wise instructions below (for all instructions it is assumed that the files are edited in a text editor).

1. Copy the files *ls_prob.m*, *ls_r.m* and *ls_J.m* to the working directory.
2. Add the problem name to the menu choice in *ls_prob.m*:

```
...
...
       ,'Plasmid Stability n=3 (subst.)'...
       ,'Plasmid Stability n=3 (probability)'...
       ,'RB BANANA'...
           ); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

if isempty(P)
   return;
end
...
...
```

3. Add the following in *ls_prob.m* after the last already existing problem (the optional parameters are not necessary to define):

```
...
...
elseif P==10
   Name='RB BANANA';
   y=[0;0];              % r(x) = residual = model psi(t,x) - data y(t)
   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.
   x_opt=[1 1];          % Known optimal point (optional).
   f_opt=0;              % Known optimal function value (optional).
   f_min=0;              % Lower bound on function (optional).
   x_max=[ 1.3  1.3];    % Plot region parameters.
   x_min=[-1.1  -0.2];   % Plot region parameters.
else
   disp('ls_prob: Illegal problem number')
   pause
   Name=[];
end
...
...
```

4. Make the following addition in *ls_r.m*:

```
...
...
   yMod=r;
elseif P==10
   % RB BANANA
   r = [10*(x(2)-x(1)^2);1-x(1)];
end

if Prob.LS.yUse & m==length(r),  r=r-y; end
...
...
```

5. Make the following addition in *ls_J.m*:

```
...
...
elseif P==10
   % RB BANANA
   J = [-20*x(1)  10
           -1      0 ];
end
...
...
```

6. Save all the files properly.

7. To solve this problem the following statements may be used.

```
Prob   = probInit('ls_prob', 10);     % Get Prob structure.
...                                   % Define changes in Prob structure
Result = tomRun('clsSolve', Prob);    % Call clsSolve using driver routine
```

   Note that the working directory in Matlab must be the directory where the new files have been edited, otherwise the predefined files in TOMLAB with the same names will be used.

D.9 Editing New Problems in Exponential Sum Fitting Init Files

The pre-defined exponential sum fitting problems are defined in one problem definition file, *exp_prob.m*. Assume a fit of a sum of exponential terms should be made to the data series

$$t = 10^{-3} \begin{pmatrix} 30 \\ 50 \\ 70 \\ 90 \\ 110 \\ 130 \\ 150 \\ 170 \\ 190 \\ 210 \\ 230 \\ 250 \\ 270 \\ 290 \\ 310 \\ 330 \\ 350 \\ 370 \end{pmatrix}, Y(t) = 10^{-4} \begin{pmatrix} 18299 \\ 15428 \\ 13347 \\ 11466 \\ 10077 \\ 8729 \\ 7382 \\ 6708 \\ 5932 \\ 5352 \\ 4734 \\ 4271 \\ 3744 \\ 3485 \\ 3111 \\ 2950 \\ 2686 \\ 2476 \end{pmatrix}, \tag{28}$$

here named *SW*.

To define (28) as a exponential sum fitting problem follow the step wise instructions below (for all instructions it is assumed that the files are edited in a text editor).

1. Copy the file *exp_prob.m* to the working directory.
2. Add the problem name to the menu choice in *exp_prob.m*:

```
...
           ,'CDFsim5'...
           ,'CDFdata5\~ '...
           ,'SW '...
           ); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

if isempty(P)
   return;
end
...
```

3. Add the following in *exp_prob.m* after the last already existing problem:

```
...
elseif P==52
   Name='SW';
   t=[30:20:370]';   % Time in ms
   y=[18299 15428 13347 11466 10077 8729 7382 6708 5932 5352 4734 4271 ...
         3744 3485 3111 2950 2686 2476]';
   t=t/1000;     % Scale to seconds. Gives lambda*1000, of order 1
   y=y/10000;    % Scale function values. Avoid large alpha
else
   disp('exp_prob: Illegal problem number')
...
```

4. Save the file properly.

5. To solve this problem the following statements may be used.

```
Prob   = probInit('exp_prob', 52);    % Get Prob structure.
...                                   % Define changes in Prob structure
Result = tomRun('clsSolve', Prob);    % Call clsSolve using driver routine
```

   Note that the working directory in Matlab must be the directory where the new files have been edited, otherwise the predefined files in TOMLAB with the same names will be used.

There are five different types of exponential models available in TOMLAB. The type of exponential model is determined by the parameter *Prob.ExpFit.eType*, which is set by defining the parameter *eType* in the problem definition file:

```
...
elseif P==52
   Name='SW';
   t=[30:20:370]';   % Time in ms
   y=[18299 15428 13347 11466 10077 8729 7382 6708 5932 5352 4734 4271 ...
         3744 3485 3111 2950 2686 2476]';
   t=t/1000;     % Scale to seconds. Gives lambda*1000, of order 1
   y=y/10000;  % Scale function values. Avoid large alpha
   eType = 1;
else
   disp('exp_prob: Illegal problem number')
...
```

The above definition of *eType* is not necessary and was made just in illustrative purpose since 1 is the default value of *eType*. See page Section 8.4, page 64 for a description of the exponential models available.

D.10 Creating a New Nonlinear Least Squares Init File

Assume a collection of e.g. nonlinear least squares problems should be defined in new Init Files. Also assume that the problems are defined in *ls_prob*, *ls_r* and *ls_J* as described in Section D.8 It is of course possible to remove the already existing problems and define the first new problem as number one. The extra modifications needed are:

1. Rename the files to for example *lsnew_prob*, *lsnew_r* and *lsnew_J*.
2. Make the following modification in the beginning of *lsnew_prob*:

```
...
...
if ask==-1 & ~isempty(Prob)
   if isstruct(Prob)
      if ~isempty(Prob.P)
         if P==Prob.P & strcmp(Prob.probFile,'lsnew_prob'), return; end
      end
   end
end
...
...
```

3. Make the following modifications at the end of *lsnew_prob*:

```
...
...
%MIDEVA
%# call   ls_f ls_g ls_H lsnew_r lsnew_J

Prob=mFiles(Prob,'ls_f','ls_g','ls_H',[],[],[],'lsnew_r','lsnew_J');
...
...
```

4. Save all the files properly.
5. Run

   ```
   AddProblemFile('lsnew_prob','New Nonlinear Least Squares Test Problems','con');
   ```

   The text *New Nonlinear ...* is of course possible to change as the user likes. Now *lsnew_prob.m* is one of the TOMLAB Init Files in the GUI data base and all problems in *lsnew_prob.m* are accessible from the GUI, the menu systems, and the driver routines. This file is also the default file for nonlinear least squares problems.

The following example illustrate how the *tomRun* driver routine could be used in an efficient way. A simple **for** loop solves all the least squares problems defined in the files *lsnew_prob*, *lsnew_r* and *lsnew_J*, see Section D.10. Several parameters are explicitly set for illustrative purpose, one could otherwise rely on the default values. The function *drv_test* below runs *tomRun* for all problems defined in *lsnew_prob*, and then displays the number of iterations performed. Instead of just printing the number of iterations, the results could be saved for later use in e.g. statistical analysis.

```
function drv_test();

probFile  = 'lsnew_prob';     % Solve problems defined in lsnew_prob.m
probNames = feval(probFile);  % Get a list of all available problems.

ask    = 0;  % Do not ask questions in problem definition.
PriLev = 0;  % No printing output.
```

```
for P = 1:size(probNames,1)
   probNumber = P;

   Prob        = probInit(probFile, P, ask, []);
   Prob.optParam.eps_x    = 1E-7;  % Termination tolerance for X
   Prob.optParam.eps_f    = 1E-9;  % Termination tolerance on F
   Prob.optParam.cTol     = 1E-5;  % Constraint violation
   Prob.optParam.MaxIter  = 200;   % Maximum number of iterations
   Prob.optParam.eps_g    = 1E-5;  % Termination tolerance on gradient
   Prob.optParam.eps_absf = 1E-35; % Absolute convergence tol. in function f.
   Prob.optParam.LineSearch.sigma = 0.5; % Line search accuracy sigma


   fprintf('\n Problem number %d:',P);
   fprintf('   %s',Prob.Name);

   Result = tomRun('clsSolve', Prob, ask, PriLev);
   fprintf('\n Number of iterations:  %d',Result.Iter);

end
```

Another example is on how to solve the exponential sum fitting problem (28)

```
probFile = 'exp_prob';                  % Problem definition file.
P        = 44;                          % Problem number.
Prob     = probInit(probFile, P);       % Setup Prob structure.
Result   = tomRun([], Prob, [], 2);     % Default solver is clsSolve
```

Also see the Section 6.3 on how to make a direct call to an optimization routine.

D.11 Using the Driver Routines

Solve the problem *RB BANANA* (15) defined as an unconstrained problem. Default values will be used for all parameters not explicitly changed. The following calls solve the problem:

```
probFile = 'uc_prob';          % Problem definition file.
P = 18;                        % Problem number, after editing the new problem

Result = tomRun('ucSolve', probFile, P);
```

To display the result of the run call the print routine *PrintResult* with the *Result* structure,

```
PrintResult(Result);
```

which gives the following printing output:

```
=== * * * ================================================ * * *
Problem 18: RB BANANA                          f_k       0.000000000000000001
                                  User given f(x_*)      0.000000000000000000
                                             f(x_0)     24.199999999999996000

Solver: ucSolve.  EXIT=0.  INFORM=2.
Safeguarded BFGS

FuncEv   48 GradEv   40
TOMLAB Global Variable Counters give:
FuncEv   48 GradEv   41 Iter   36
Starting vector x:
x_0:  -1.200000   1.000000
Optimal vector x:
x_k:   1.000000   1.000000
Diff x-x0:
      2.200000e+000 -2.312176e-009
Gradient g_k:
g_k: -4.162202e-009  9.227064e-010
TOMLAB found no active constraints.

=== * * * ================================================ * * *
```

To solve the problem using the routine *fminu* call the driver routine *tomRun* with the solver name as string argument:

```
probFile = 'uc_prob';          % Problem definition file.
P = 18;                        % Problem number.
Prob = probInit(probFile, P);  % Setup Prob structure.

Result = tomRun('fminu', Prob); % Call fminu using driver routine
```

The second example has a more "testing and developing" characteristic. The purpose is to illustrate how the driver routines could be used in an efficient way.

By use of a simple **for** loop solve all the constrained problems defined in the files *connew_prob*, *connew_f*, *connew_g*, *connew_H*, *connew_c* and *connew_dc*, see Section D.10. Several parameter values are set explicitly for illustrative purposes. The function *drv_test* below runs *tomRun* for all problems defined in *connew_prob*, and then displays the number of iterations performed. Instead of just printing the number of iterations, the results could be stored in a structure array for later use in e.g. statistical analysis.

```
function drv_test();

probFile  = 'connew_prob';    % Solve problems defined in connew_prob.m
probNames = feval(probFile);  % Get a list of all available problems.
ask       = 0;                % Do not ask questions in problem definition.
PriLev    = 0;                % No printing output.

for P = 1:size(probNames,1)

   probNumber = P;

   Prob        = probInit(probFile, P, ask, []);

   Prob.optParam.eps_x     = 1E-7;  % Termination tolerance for X
   Prob.optParam.eps_f     = 1E-9;  % Termination tolerance on F
   Prob.optParam.cTol      = 1E-5;  % Constraint violation
   Prob.optParam.MaxIter   = 200;   % Maximum number of iterations
   Prob.optParam.eps_g     = 1E-5;  % Termination tolerance on gradient
   Prob.optParam.LineSearch.sigma = 0.5; % Line search accuracy sigma

   fprintf('\n Problem number %d:',P);
   fprintf('   %s',Prob.Name);

   Result = tomRun(Solver, Prob, ask, PriLev);
   fprintf('\n Number of iterations:  %d',Result.Iter);

end
```

E Interfaces

Some users may have been used to work with MathWorks Optimization Toolbox v2.1 (or v1.5), 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 v2.1, but the solution is obtained by converting the problem into the TOMLAB Quick 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. Interfaces have been made to both MathWorks Optimization TB v2.1 and MathWorks Optimization TB v1.5. 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 v2.1 is not accessible.

E.1 Solver Call Compatible with Optimization Toolbox 2.1

TOMLAB is call compatible with MathWorks Optimization TB v2.1. 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 v2.1 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 53 gives a list of the solvers with compatible interfaces.

Table 53: Call compatible interfaces to MathWorks Optimization TB v2.1.

Function	**Type of problem solved**
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 curvefitting.
lsqlin	Linear least squares.
lsqnonlin	Linear least squares with nonnegative variable constraints.
lsqnonneg	Nonlinear least squares.
quadprog	Quadratic programming.

In Table 54 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.

Table 54: Testroutines for the call compatible interfaces to MathWorks Optimization TB v2.1 present in the *examples* directory in the TOMLAB distribution.

Function	**Type of problem solved**
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 curvefitting.
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.

E.1.1 Solving LP Similar to Optimization Toolbox 2.1

For linear programs the MathWorks Optimization TB v2.1 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 (11) illustrates the basic usage

File: tomlab/usersguide/lpTest4.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/lpTest4.out

```
linprog (MINOS): 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: 1
       algorithm: 'MINOS: MEX-interface to MINOS 5.5 NLP code'
    cgiterations: 0
Function value       -26.57142857142856600000. ExitFlag 1
```

E.1.2 Solving QP Similar to Optimization Toolbox 2.1

For quadratic programs the MathWorks Optimization TB v2.1 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 (13) illustrates the basic usage

File: tomlab/usersguide/qpTest4.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 v2.1 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/qpTest4.out

```
Run TOMLAB quadprog on QP Example

Constraints Ax-b_U  -4.900000e+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:      0.000000e+000
x:                   5.000000e-002  5.000000e-002
Output Structure
       iterations: 1
        algorithm: 'bqpd: MEX-interface to BQPD QP/LP code'
     cgiterations: []
    firstorderopt: []
Function value        -0.02500000000000000100. 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 [15]. and MathWorks Optimization TB v2.1 [17]. The solvers that are directly possible to use, when a problem is generated in the TOMLAB format, are listed in Table 55. 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 v2.1 is instead using a structure.

Table 55: Optimization toolbox routines with a TOMLAB interface.

Function	**Type of problem solved**
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 curvefitting.
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 CUTE Interface

The Constrained and Unconstrained Testing Environment (CUTE) [13, 14] is a well-known software environment for nonlinear programming. The distribution of CUTE includes a test problem data base of nearly 1000 optimization problems, both academic and real-life applications. This data base is often used as a benchmark test in the development of general optimization software.

CUTE stores the problems in the standard input format (SIF) in files with extension *sif*. There are tools to select appropriate problems from the data base. Running CUTE, a SIF decoder creates up to five Fortran files; *elfuns, extern, groups, ranges*, and *settyp*, and one ASCII data file; *outsdif.dat* or *outsdif.d*. The Fortran files are compiled and linked together with the CUTE library and a solver routine. Running the binary executable, the problem is solved using the current solver. During the solution procedure, the ASCII data file *outsdif.dat* or *outsdif.d* is read.

With the CUTE distribution follows a Matlab interface. There are one gateway routine, *ctools.f*, for constrained CUTE problems, and one gateway routine, *utools.f*, for unconstrained problems. These routines are using the Matlab MEX-file interface for communication between Matlab and the compiled Fortran (or C) code. The gateway routine is compiled and linked together with the Fortran files, generated by the SIF decoder, and the Matlab MEX library to make a DLL (Dynamic Link Library) file. At run-time, Matlab calls the DLL, which will read the CUTE ASCII data file for the problem specific information. Also included in the CUTE distribution is a set of Matlab m-files that calls the gateway routine.

For the TOMLAB CUTE interface we assume that the DLLs are already built and stored in any of four predefined directories; *cutedll* for constrained problems, *cutebig* for large constrained problems, *cuteudll* for unconstrained problems, *cuteubig* for large unconstrained problems. The name of the dll is the problem name used by CUTE, e.g. *rosenbr.dll* for the Rosenbrock banana function. The ASCII data file also has a unique name, e.g. *rosenbr.dat*. The CUTE Matlab interface assumes the DLLs to be named ctools.dll and utools.dll (and the data file to be called outsdif.dat on PC). TOMLAB calls the Matlab files in the CUTE distribution, but to solve the name problem, using the m-files *ctools.m* and *utools.m* to make a call to the correct DLL file. The ASCII data file is also copied to a temporary file, with the necessary filename *outsdif.dat*, before executing the DLL.

When using the TOMLAB interface, the user either gets a menu of all DLLs in the CUTE directory chosen, or directly makes a choice of which problem to solve. Pre-compiled DLL files for the CUTE data set will be made available, or the necessary files for the user to build his own DLLs. It is thus possible to run the huge set of CUTE test problems in TOMLAB, using any solver callable from the toolbox.

Table 56 describes the low level test functions and the corresponding problem setup routines needed for the predefined unconstrained and constrained optimization problems from the CUTE data base [13, 14].

Table 56: Test problems from CUTE data base.

Function	Description
ctools	Interface routine to constrained CUTE test problems.
utools	Interface routine to unconstrained CUTE test problems.
cto_prob	Initialization of constrained CUTE test problems.
ctl_prob	Initialization of large constrained CUTE test problems.
cto_f	Compute the objective function $f(x)$ for constrained CUTE test problems.
cto_g	Compute the gradient $g(x)$ for constrained CUTE test problems.
cto_H	Compute the Hessian $H(x)$ of $f(x)$ for constrained CUTE test problems.
cto_c	Compute the vector of constraint functions $c(x)$ for constrained CUTE test problems.
cto_dc	Compute the matrix of constraint normals for constrained CUTE test problems.
cto_d2c	Compute the second part of the second derivative of the Lagrangian function for constrained CUTE test problems.
uto_prob	Initialization of unconstrained CUTE test problems.
utl_prob	Initialization of large unconstrained CUTE test problems.
uto_f	Compute the objective function $f(x)$ for unconstrained CUTE test problems.
uto_g	Compute the gradient $g(x)$ for unconstrained CUTE test problems.
uto_H	Compute the Hessian $H(x)$ of $f(x)$ for unconstrained CUTE test problems.

E.4 The AMPL Interface

Using interfaces between a modeling language and TOMLAB could be of great benefit and improve the possibilities for analysis on a given problem. As a first attempt, a TOMLAB interface to the modeling language AMPL [29] was built. The reason to choose AMPL was that it has a rudimentary Matlab interface written in C [33] that could easily be used.

AMPL is using ASCII files to define a problem. The naming convention is to use the problem name and various extensions, e.g. *rosenbr.mod* and *rosenbr.dat* for the Rosenbrock banana function. These files are normally converted to binary files with the extension *nl*, called *nl*-files. This gives a file *rosenbr.nl* for our example. Then AMPL invokes a solver with two arguments, the problem name, e.g. *rosenbr*, and a string *-AMPL*. The second argument is a flag telling AMPL is the caller. After solving the problem, the solver creates a file with extension *sol*, e.g. *rosenbr.sol*, containing a termination message and the solution it has found.

The current TOMLAB AMPL interface is an interface to the problems defined in the AMPL *nl*-format. TOMLAB assumes the *nl*-files to be stored in directory */tomlab/ampl* or */tomlab/amplsp* (for sparse problems). When using the TOMLAB interface, the user either gets a menu of the *nl*-files found or directly makes a choice of which problem to solve. The initialization routine in TOMLAB for AMPL problems, *amp_prob*, either calls *amplfunc* or *spamfunc*, the two MEX-file interface routines written by Gay [33]. The low level routines *amp_f, amp_g* , etc. calls the same MEX-file interface routines, and dependent on the parameters in the call, the appropriate information is returned.

Note that the design of the AMPL solver interface makes it easy to run the TOMLAB solvers from AMPL using the Matlab Engine interface routines, a possible extension in the future. But indeed, any solver callable from TOMLAB may now solve problems formulated in the AMPL language.

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 [8], AMPL [29], ASCEND [73], GAMS [9, 16] 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 [22].

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 [60, 74]. Nowadays, languages like Matlab has a rapid growth of users. Matlab was originally created [67] as a preprocessor to the standard Fortran subroutine libraries in numerical linear algebra, LINPACK [20] and EISPACK [81] [32], 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. And the MIDEVA system that converts the Matlab m-file code into C++ and compiles it makes the time penalty of using an interpretative system like Matlab much less.

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 *lpSolve*, two algorithms implemented in the solver *linprog* from Optimization Toolbox 2.0 [17] and the Fortran solvers MINOS and QPOPT, available in TOMLAB v3.2, are compared. In all test cases the solvers converge to the same solution. The results are presented in five tables

Table 57, Table 58, Table 59, Table 60 and Table 61. The problem dimensions and all elements in (19) are chosen randomly. Since the simplex algorithm in *linprog* does not return the number of iterations as output, these figures could not be presented. *lpSolve* 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 *lpSolve* 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 57 show that problems with about 200 variables and 150 inequality constraints are solved by *lpSolve* fast and efficient. When comparing elapsed computational time for 20 problems, it is clear that *lpSolve* is much faster then the corresponding simplex algorithm implemented in the *linprog* solver. In fact *lpSolve*, with the minimum cost selection rule, is more than five times faster, a remarkable difference. *lpSolve* 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 [17]. There is a penalty about a factor of three to choose Bland's rule to prevent cycling in *lpSolve*. The solvers written in Fortran, MINOS and QPOPT, of course run much faster, but the iteration count show that *lpSolve* 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.

Table 57: 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 *lpSolve*, 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

In Table 58 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 *lpSolve* and the Fortran solvers are not so striking for these lower dimensional problems. *lpSolve* 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*.

Table 58: 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 *lpSolve*, 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

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 61. In that test *lpSolve* 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 62, 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 v3.2 is very important.

Table 59: 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 *lpSolve*, 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 60: 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 *lpSolve*, 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 61: 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 *lpSolve*, 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 62: 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 *lpSolve*, 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] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows. 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.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice-Hall Inc., Kanpur and Cambridge, 1993.

[4] M. Al-Baali and R. Fletcher. Variational methods for non-linear least squares. *J. Oper. Res. Soc.*, 36:405–421, 1985.

[5] M. Al-Baali and R. Fletcher. An efficient line search for nonlinear least-squares. *Journal of Optimization Theory and Applications*, 48:359–377, 1986.

[6] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear Programming and Network Flows*. John Wiley and Sons, New York, 2nd edition, 1990.

[7] Jordan M. Berg and K. Holmström. On Parameter Estimation Using Level Sets. *SIAM Journal on Control and Optimization*, 37(5):1372–1393, 1999.

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

[9] 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.

[10] M. Björkman. Nonlinear Least Squares with Inequality Constraints. Bachelor Thesis, Department of Mathematics and Physics, Mälardalen University, Sweden, 1998. Supervised by K. Holmström.

[11] M. Björkman and K. Holmström. Global Optimization Using the DIRECT Algorithm in Matlab. *Advanced Modeling and Optimization*, 1(2):17–37, 1999.

[12] M. Björkman and K. Holmström. Global Optimization of Costly Nonconvex Functions Using Radial Basis Functions. *Optimization and Engineering*, 1(4):373–397, 2000.

[13] I. Bongartz, A. R. Conn, N. I. M. Gould, and P. L. Toint. CUTE: Constrained and Unconstrained Testing Environment. *ACM Transactions on Mathematical Software*, 21(1):123–160, 1995.

[14] I. Bongartz, A. R. Conn, Nick Gould, and Ph. L. Toint. CUTE: Constrained and Unconstrained Testing Environment. Technical report, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, September 2 1997.

[15] Mary Ann Branch and Andy Grace. *Optimization Toolbox User's Guide*. 24 Prime Park Way, Natick, MA 01760-1500, 1996.

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

[17] 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).

[18] 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.

[19] 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.

[20] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. LINPACK User's Guide. *SIAM*, 1979.

[21] E. Dotzauer and K. Holmström. The TOMLAB Graphical User Interface for Nonlinear Programming. *Advanced Modeling and Optimization*, 1(2):9–16, 1999.

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

[23] Marshall L. Fisher. An Application Oriented Guide to Lagrangian Relaxation. *Interfaces 15:2*, pages 10–21, March-April 1985.

[24] R. Fletcher and C. Xu. Hybrid methods for nonlinear least squares. *IMA Journal of Numerical Analysis*, 7:371–389, 1987.

[25] Roger Fletcher. *Practical Methods of Optimization*. John Wiley and Sons, New York, 2nd edition, 1987.

[26] Roger Fletcher and Sven Leyffer. Nonlinear programming without a penalty function. Technical Report NA/171, University of Dundee, 22 September 1997.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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, 2002. Accepted.

[32] 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.

[33] David M. Gay. Hooking your solver to AMPL. Technical report, Bell Laboratories, Lucent Technologies, Murray Hill, NJ 07974, 1997.

[34] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, London, 1982.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] Jacek Gondzio. Presolve analysis of linear programs prior to applying an interior point method. *INFORMS Journal on Computing*, 9(1):73–91, 1997.

[43] Hans-Martin Gutmann. A radial basis function method for global optimization. *Journal of Global Optimization*, 19:201–227, 2001.

[44] Michael Held and Richard M. Karp. The Traveling-Salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1:6–25, 1971.

[45] 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.

[46] 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.

[47] 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.

[48] Kaj Holmberg. Heltalsprogrammering och dynamisk programmering och flöden i nätverk och kombinatorisk optimering. Technical report, Division of Optimization Theory, Linköping University, Linköping, Sweden, 1988-1993.

[49] K. Holmström. New Optimization Algorithms and Software. *Theory of Stochastic Processes*, 5(21)(1-2):55–63, 1999.

[50] 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.

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

[52] 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.

[53] 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.

[54] K. Holmström and M. Björkman. The TOMLAB NLPLIB Toolbox for Nonlinear Programming. *Advanced Modeling and Optimization*, 1(1):70–86, 1999.

[55] 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.

[56] 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.

[57] K. Holmström, T. Fomina, and Michael Saunders. Parameter Estimation for Inorganic Chemical Equilibria by Least Squares and Minimax Models. *Optimization and Engineering*, 3, 2002. Submitted.

[58] 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.

[59] 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.

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

[61] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, October 1993.

[62] Donald R. Jones. DIRECT. *Encyclopedia of Optimization*, 2001.

[63] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive Black-Box functions. *Journal of Global Optimization*, 13:455–492, 1998.

[64] P. Lindström. *Algorithms for Nonlinear Least Squares - Particularly Problems with Constraints*. PhD thesis, Inst. of Information Processing, University of Umeå, Sweden, 1983.

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

[66] 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.

[67] C. B. Moler. MATLAB — An Interactive Matrix Laboratory. Technical Report 369, Department of Mathematics and Statistics, University of New Mexico, 1980.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

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

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

[76] Axel Ruhe and Per-Åke Wedin. Algorithms for Separable Nonlinear Least Squares Problems. *SIAM Review*, 22(3):318–337, 1980.

[77] 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.

[78] M. A. Saunders. Solution of sparse rectangular systems using LSQR and CRAIG. *BIT*, 35:588–604, 1995.

[79] 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.

[80] L. F. Shampine and H. A. Watts. Fzero, a root-solving code. Technical Report Report SC-TM-70-631, Sandia Laboratories, September 1970.

[81] 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.

[82] William Squire and George Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Review*, 40(1):100–112, March 1998.

[83] Wayne L. Winston. *Operations Research: Applications and Algorithms*. International Thomson Publishing, Duxbury Press, Belmont, California, 3rd edition, 1994.