User’s Guide for Tomlab v3.1

Kenneth Holmström

Tomlab Optimization

Gäddgatan 9
SE-723 49 Västerås, Sweden

May 8, 2002

# Contents

1 Introduction 7

2 The Design of TOMLAB 9

2.1 Low Level Routines 13

3 Optimization Problems and Solver Routines in TOMLAB 16

4 Define User Problems in the TOMLAB Quick or Init File format 22

4.1 TOMLAB Quick (TQ) format for user problems 22

4.2 TOMLAB Init File (IF) format for user problems 23

4.3 Create an Init File going from TQ to IF format 25

4.4 Adding an Init File to the GUI data base 26

5 Solving Linear, Quadratic and Integer Programming Problems 28

5.1 Linear Programming Problems 28

5.1.1 A Quick Linear Programming Solution 29

5.1.2 Several Linear Programs 30

5.1.3 Large Sets of Linear Programs 30

5.1.4 More on Solving Linear Programs 32

5.2 Quadratic Programming Problems 34

5.2.1 A Quick Quadratic Programming solution 35

5.2.2 Several Quadratic Programs 36

5.2.3 Large Sets of Quadratic Programs 37

5.2.4 Another Direct Approach to a QP Solution 39

5.2.5 More on Solving Quadratic Programs 39

5.3 Mixed-Integer Programming Problems 39

5.3.1 Large Sets of Mixed-Integer Programs 43

5.3.2 More on Solving Mixed-Integer Programs 43

6 Solving Unconstrained and Constrained optimization Problems 46

6.1 Defining the Problem in Matlab m-files 46

6.1.1 Communication between user routines 49

6.2 Solving Unconstrained Optimization using the TQ format 50

6.3 Direct Call to an Optimization Routine 51

6.4 Solving Constrained Optimization using the TQ format 52

6.5 Efficient use of the TOM solvers 54

7 Efficient use of the SOL solvers in TOMLAB 55

7.1 Setting Parameters for the SOL solvers 55

7.2 Derivatives for the SOL solvers 55

7.3 SOL solver output on files 56

7.4 Warm Starts for the SOL solvers 57
Chapter 8: Solving Global Optimization Problems
  8.1 Solving Box-Bounded Global Optimization with TQ format
  8.2 Defining Global Mixed-Integer Nonlinear Problems with TQ format

Chapter 9: Least Squares and Parameter Estimation Problems
  9.1 Solving Linear Least Squares Problems using the TQ Format
  9.2 Solving Linear Least Squares Problems using the SOL Solver LSSOL
  9.3 Solving Nonlinear Least Squares Problems using the TQ Format
  9.4 Fitting Sums of Exponentials to Empirical Data

Chapter 10: Special Notes and Features
  10.1 Approximation of Derivatives
  10.2 Speed and Solution of Optimization Subproblems
  10.3 User Supplied Problem Parameters
  10.4 User Given Stationary Point
  10.5 Print Levels and Printing Utilities
  10.6 Partially Separable Functions
  10.7 Usage of routines from Optimization Toolbox 1.x
  10.8 Using Matlab 5.0 or 5.1
  10.9 Utility Test Routines

Chapter 11: The tomGUI Graphical User Interface (GUI)
  11.1 The input modes
  11.2 General Parameter Mode
  11.3 Solver Parameter Mode
  11.4 Plot Parameter Mode

Chapter 12: The Menu Program tomMenu

Chapter 13: The TOMLAB Routines - Detailed Descriptions
  13.1 Optimization Routines in TOMLAB - The TOM solvers
13.1.14 itrr .................................................. 110
13.1.15 ucSolve ............................................. 111
13.2 Utility Functions in TOMLAB .................................... 113
13.2.1 tomRun ............................................... 113
13.2.2 opTransf ............................................. 114
13.2.3 LineSearch ............................................. 115
13.2.4 intpol2 ................................................ 116
13.2.5 intpol3 ................................................ 116
13.2.6 preSolve ............................................... 117
13.2.7 PrintResult ............................................. 117
13.2.8 runtest ................................................ 118
13.2.9 SolverList ............................................. 118
13.2.10 systest ............................................... 119

14 TOMLAB LDO ........................................... 121
14.1 Optimization Algorithms and Solvers in TOMLAB LDO ........................................... 121
14.1.1 Linear Programming ........................................... 121
14.1.2 Transportation Programming ........................................... 122
14.1.3 Network Programming ........................................... 122
14.1.4 Mixed-Integer Programming ........................................... 123
14.1.5 Dynamic Programming ........................................... 123
14.1.6 Quadratic Programming ........................................... 124
14.1.7 Lagrangian Relaxation ........................................... 124
14.1.8 Utility Routines ........................................... 124
14.2 How to Solve Optimization Problems Using TOMLAB LDO ........................................... 125
14.2.1 How to Solve Linear Programming Problems ........................................... 125
14.2.2 How to Solve Transportation Programming Problems ........................................... 126
14.2.3 How to Solve Network Programming Problems ........................................... 126
14.2.4 How to Solve Integer Programming Problems ........................................... 128
14.2.5 How to Solve Dynamic Programming Problems ........................................... 128
14.2.6 How to Solve Lagrangian Relaxation Problems ........................................... 129
14.3 Printing Utilities and Print Levels ........................................... 131
14.4 Optimization Routines in TOMLAB LDO ........................................... 131
14.4.1 akarmark .............................................. 131
14.4.2 balas .................................................. 132
14.4.3 dijkstra ................................................ 133
14.4.4 dpinvent ............................................... 133
14.4.5 dpknapsack ............................................. 134
14.4.6 karmark ............................................... 135
14.4.7 krsrelax ............................................... 135
14.4.8 labelcor ............................................... 136
14.4.9 lpdual ................................................ 137
E.2 The Matlab Optimization Toolbox Interface ........................................ 187
E.3 The CUTE Interface ........................................................................ 187
E.4 The AMPL Interface ........................................................................ 188

F Motivation and Background to TOMLAB .............................................. 191

G Performance Tests on Linear Programming Solvers ............................. 192
1 Introduction

This user’s guide describes the TOMLAB v3.0 optimization environment. 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 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 problems and many more are easily solved 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.

The basic design of TOMLAB is presented in Section 2. The different types of optimization problems that TOMLAB can treat are defined. Section 2.1 discusses how TOMLAB does the calls to the low-level routines computing functions, gradients, constraints etc. TOMLAB has an additional interface level, called the gateway routines, that handles non-defined routines, the way to estimate derivatives, keeps track of the iteration steps, avoids unnecessary function computations when values are already computed etc. The organization of the predefined test problems is also discussed.

Section 3 gives a list of the most important solvers and gives a mathematical formulation of the optimization problems treated by TOMLAB. TOMLAB has two different input formats, the TOMLAB Quick format (TQ) and the TOMLAB Init File format (IF). In Section 4 these two formats are described. There is a simple way to get the user problems from the TQ to the IF format, with some minor loss of functionality. This feature enables the use of the GUI and menu system for the user problems specified in the TQ format. Appendix D describes in detail how to edit files in the TOMLAB Init File format in order to define new problems.

In the chapters 5, 6, 8 and 9 the definition and solution of each class of problem are discussed, illustrated with test examples that are part of the TOMLAB distribution.

Special system features which helps the solution process are discussed in Section 10. Some features discussed are derivatives, automatic differentiation, partially separable functions and user supplied parameter information for the function computations.

The Graphical User Interface (GUI) is presented in Section 11. The GUI gives the user the possibility to set all kinds of solver parameters that influence 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.

The menu system, tomMenu, is presented in Section 12. The most important parameters are possible to set, but not all parameters for the solvers called using MEX-file interfaces.

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 [53].

All routines in the basic TOMLAB are integrated together in a standardized way, and callable from GUI, the different menu systems, and the different driver routines. TOMLAB also includes a collection of optimization solvers that are single routines. They are all solvers for linear, quadratic and discrete optimization and collected together in the TOMLAB LDO (Linear and Discrete Optimization) toolbox. Included in TOMLAB LDO are also test routines for all solvers. Chapter 14 describes the problems possible to solve and the different solvers and utilities.

In the Appendices, 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.

The aim is that TOMLAB should provide interfaces to all relevant optimization software in the areas that TOMLAB covers. By use of the TOMLAB MEX-file interfaces it is possible to call general-purpose solvers implemented in Fortran or C. In Appendix E the current set of interfaces are described. For algorithm testing the interface to the CUTE test problem data base is of interest. And for the user having problems defined in the AMPL modeling language, the AMPL interface might be of interest.

In Appendix C the global variables used in TOMLAB is discussed. There are routines to save important global variables enabling recursive calls of any depth.

Appendix F gives some motivation for the development of TOMLAB.

TOMLAB is also call compatible with the Math Works Optimization TB v2.1. If the user wants to use this toolbox, the directory optim must be removed from the path.

TOMLAB has been discussed in several papers and at several conferences. The main paper on TOMLAB v1.0 is [40]. The use of TOMLAB for nonlinear programming and parameter estimation is presented in [42], and the use of linear and discrete optimization is discussed in [43]. Global optimization routines are also implemented, one is described in [10]. 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 [20]. TOMLAB v2.0 was discussed in [41].
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.

Normally, when solving an optimization problem, a direct call to a solver is made with a long list of parameters in the call. TOMLAB solves the problem in two steps. First the problem is defined and stored in a Matlab structure. Then the solver is called with one 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 a call to a driver routine with the problem structure as input. The driver routine retrieves the relevant information from the structure and calls to solver. The structure idea is also used for the solver output. All information are stored in one result structure. Then the driver routine can add any information not supplied in the normal solver output, thereby creating a uniform output format.

TOMLAB v3.0 consists of about 95000 lines of m-file code in more than 530 files with algorithms, utilities and predefined problems. This motivates a well-defined naming convention and design.

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 treated in TOMLAB.

<table>
<thead>
<tr>
<th>probType</th>
<th>Number</th>
<th>Type of optimization problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>uc</td>
<td>1</td>
<td>Unconstrained optimization (incl. bound constraints).</td>
</tr>
<tr>
<td>qp</td>
<td>2</td>
<td>Quadratic programming.</td>
</tr>
<tr>
<td>con</td>
<td>3</td>
<td>Constrained nonlinear optimization.</td>
</tr>
<tr>
<td>ls</td>
<td>4</td>
<td>Nonlinear least squares problems (incl. bound constraints).</td>
</tr>
<tr>
<td>lls</td>
<td>5</td>
<td>Linear least squares problems.</td>
</tr>
<tr>
<td>cls</td>
<td>6</td>
<td>Constrained nonlinear least squares problems.</td>
</tr>
<tr>
<td>mip</td>
<td>7</td>
<td>Mixed-Integer programming.</td>
</tr>
<tr>
<td>lp</td>
<td>8</td>
<td>Linear programming.</td>
</tr>
<tr>
<td>glb</td>
<td>9</td>
<td>Box-bounded global optimization.</td>
</tr>
<tr>
<td>glc</td>
<td>10</td>
<td>Global mixed-integer nonlinear programming.</td>
</tr>
<tr>
<td>exp</td>
<td>11</td>
<td>Exponential fitting problems.</td>
</tr>
</tbody>
</table>

Note that the actual numbers used for probType might change. To create code independent on the actual numbers use the text abbreviation for the type of optimization problem. One utility routine checkType is defined to help keep track of the different types. Calling the routine checkType with an empty input string returns the type of optimization as a string as output

```matlab
isType = checkType([],3);
```

In this case the variable isType will contain the string 'con'. To find out if a certain probType is the wanted one use the same routine

```matlab
probType = 3;
if checkType('con', probType);
```
disp('The problem solved is a constrained optimization problem')
end

To find out the current probType number used by the system for a certain type of optimization problem, when knowing the string name, call checkType with the string name as first argument, e.g.

```matlab
probType = checkType('con');
fprintf('probType number for constrained optimization is %d\n',probType);
end
```

Define probSet to be a set of defined optimization problems to be solved. Each probSet belongs to a certain class of optimization problems of type probType. Each probSet is physically stored in one 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

<table>
<thead>
<tr>
<th>probSet</th>
<th>probType</th>
<th>Description of test problem set</th>
</tr>
</thead>
<tbody>
<tr>
<td>uc</td>
<td>1</td>
<td>Unconstrained test problems.</td>
</tr>
<tr>
<td>qp</td>
<td>2</td>
<td>Quadratic programming test problems.</td>
</tr>
<tr>
<td>con</td>
<td>3</td>
<td>Constrained test problems.</td>
</tr>
<tr>
<td>ls</td>
<td>4</td>
<td>Nonlinear least squares test problems.</td>
</tr>
<tr>
<td>lls</td>
<td>5</td>
<td>Linear least squares problems.</td>
</tr>
<tr>
<td>cls</td>
<td>6</td>
<td>Linear constrained nonlinear least squares problems.</td>
</tr>
<tr>
<td>mip</td>
<td>7</td>
<td>Mixed-integer programming problems.</td>
</tr>
<tr>
<td>lp</td>
<td>8</td>
<td>Linear programming problems.</td>
</tr>
<tr>
<td>glb</td>
<td>9</td>
<td>Box-bounded global optimization test problems.</td>
</tr>
<tr>
<td>glc</td>
<td>10</td>
<td>Global MINLP test problems.</td>
</tr>
<tr>
<td>exp</td>
<td>11</td>
<td>Exponential fitting problems.</td>
</tr>
<tr>
<td>mgh</td>
<td>4</td>
<td>More, Garbow, Hillstrom nonlinear least squares problems.</td>
</tr>
<tr>
<td>amp</td>
<td>3</td>
<td>AMPL test problems as nl-files.</td>
</tr>
<tr>
<td>chs*</td>
<td>3</td>
<td>Hock-Schittkowski constrained test problems.</td>
</tr>
<tr>
<td>uhs*</td>
<td>1</td>
<td>Hock-Schittkowski unconstrained test problems.</td>
</tr>
<tr>
<td>cto*</td>
<td>3</td>
<td>CUTE constrained test problems as dll-files.</td>
</tr>
<tr>
<td>ctl*</td>
<td>3</td>
<td>CUTE large constrained test problems as dll-files.</td>
</tr>
<tr>
<td>uto*</td>
<td>1</td>
<td>CUTE unconstrained test problems as dll-files.</td>
</tr>
<tr>
<td>utl*</td>
<td>1</td>
<td>CUTE large unconstrained test problems as dll-files.</td>
</tr>
</tbody>
</table>

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 using 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 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.

The Interface Routines in Figure 1 are used to convert computational results to the form expected by different solvers.

A set of Matlab m-files are needed to implement the chain of function calls for all solver types and problem sets, i.e. for the menu systems, driver routines etc. The names of the predefined Init File, that does the problem setup, and the low level routines are constructed as two parts. The first part being the abbreviation of the relevant probSet, see Table 2, and the second part denotes the computed task, shown in Table 3. The user normally does not have to define the more complicated functions ◦ d2c and ◦ d2r. Only the solver nlpSolve can utilize the information in ◦ d2c. An example, illustrating the constrained nonlinear programming case (solveType = con, probSet = con) is shown in Figure 2.
Figure 1: The process of optimization in TOMLAB.
Either choose pre-defined problem from menu or silent setup

Define names of functions: con_f, con_g, con_H, con_c, con_dc, con_d2c

MEX: minos, npsol or npopt

Interface: nlp_f, nlp_gdc, nlp_fg, nlp_cdcS, nlp_cdc

E.g.: Gateway Routine nlp_f computes f(x) by the call: f = feval(Prob.USER.f, x, Prob);

In the routine set as Prob.USER.f, e.g. con_f, the user computes the function value f(x)

Figure 2: Solution of constrained nonlinear problems in TOMLAB.
Table 3: Names for predefined Init Files and low level m-files in TOMLAB.

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

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.

### 2.1 Low Level Routines

Define the low level routines as the routines that compute the objective function value, the gradient vector, the Hessian matrix (second derivative matrix), the residual vector (for NLLS problems), the Jacobian matrix (for NLLS problems), the vector of constraint functions, the matrix of constraint normals (the constraint Jacobian) and 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 3 and 4. Figure 3 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 4 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.

![Figure 3](ucSolve↩nlp_f↩ls_f↩nlp_r↩mgh_r)

Figure 3: 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 4](ucSolve↩nlp_g↩fdng↩nlp_f↩uc_f)

Figure 4: 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.

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. It is recommended to use subfields of Prob.user. It the user wants to send some variables a, B and C, then, after creating the Prob structure, these extra variables are added to the structure:

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

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

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

A more detailed description of how to define new problems is given in sections 5, 6 and 9. 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.

To conclude, the system design is flexible and easy to expand in many different ways.
Table 4: Generally constrained nonlinear (con) test problems.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>con_prob</td>
<td>Init File. Does the initialization of the con test problems.</td>
</tr>
<tr>
<td>con_f</td>
<td>Compute the objective function $f(x)$ for con test problems.</td>
</tr>
<tr>
<td>con_g</td>
<td>Compute the gradient $g(x)$ for con test problems.</td>
</tr>
<tr>
<td>con_H</td>
<td>Compute the Hessian matrix $H(x)$ of $f(x)$ for con test problems.</td>
</tr>
<tr>
<td>con_c</td>
<td>Compute the constraint residuals $c(x)$ for con test problems.</td>
</tr>
<tr>
<td>con_dc</td>
<td>Compute the derivative of the constraint residuals for con test problems.</td>
</tr>
<tr>
<td>con_d2c</td>
<td>Compute the 2nd part of 2nd derivative matrix of the Lagrangian function, $\sum_i \lambda_i \partial^2 c_i(x)/dx^2$ for con test problems.</td>
</tr>
<tr>
<td>con_fm</td>
<td>Compute merit function $\theta(x_k)$.</td>
</tr>
<tr>
<td>con_gm</td>
<td>Compute gradient of merit function $\theta(x_k)$.</td>
</tr>
</tbody>
</table>
3 Optimization Problems and Solver Routines in TOMLAB

TOMLAB includes a large set of optimization solvers. Most of them were originally developed by the Applied Optimization and Modeling group (TOM) [40]. 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 TOMLAB. The solver for unconstrained optimization, ucSolve, the constrained nonlinear least squares solver clsSolve, and the constrained solver conSolve, are all implementing a set of algorithms.

Table 5: The TOM optimization solvers in TOMLAB v3.0.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ucSolve</td>
<td>Unconstrained optimization with simple bounds on the parameters. Implements Newton, quasi-Newton and conjugate-gradient methods.</td>
<td>13.1.15</td>
<td>111</td>
</tr>
<tr>
<td>sTrustr</td>
<td>Constrained convex optimization of partially separable functions, using a structural trust region algorithm.</td>
<td>13.1.13</td>
<td>109</td>
</tr>
<tr>
<td>glbSolve</td>
<td>Box-bounded global optimization, using only function values.</td>
<td>13.1.5</td>
<td>97</td>
</tr>
<tr>
<td>glbFast</td>
<td>Box-bounded global optimization, using only function values. Fortran MEX implementation of glbSolve.</td>
<td>13.1.6</td>
<td>98</td>
</tr>
<tr>
<td>glcSolve</td>
<td>Global mixed-integer nonlinear programming, using no derivatives.</td>
<td>13.1.7</td>
<td>100</td>
</tr>
<tr>
<td>glcFast</td>
<td>Box-bounded global optimization, using only function values. Fortran MEX implementation of glcSolve.</td>
<td>13.1.8</td>
<td>101</td>
</tr>
<tr>
<td>clsSolve</td>
<td>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.</td>
<td>13.1.1</td>
<td>90</td>
</tr>
<tr>
<td>conSolve</td>
<td>Constrained nonlinear minimization solver using two different sequential quadratic programming methods.</td>
<td>13.1.2</td>
<td>91</td>
</tr>
<tr>
<td>nlpSolve</td>
<td>Constrained nonlinear minimization solver using a filter SQP algorithm.</td>
<td>13.1.11</td>
<td>106</td>
</tr>
<tr>
<td>lpSolve</td>
<td>Linear programming using a simplex algorithm.</td>
<td>13.1.9</td>
<td>103</td>
</tr>
<tr>
<td>DualSolve</td>
<td>Solves a linear program with a dual feasible starting point.</td>
<td>13.1.4</td>
<td>96</td>
</tr>
<tr>
<td>qpSolve</td>
<td>Non-convex quadratic programming.</td>
<td>13.1.12</td>
<td>107</td>
</tr>
<tr>
<td>mipSolve</td>
<td>Mixed-integer programming using a branch-and-bound algorithm.</td>
<td>13.1.10</td>
<td>104</td>
</tr>
<tr>
<td>cutplane</td>
<td>Mixed-integer programming using a cutting plane algorithm.</td>
<td>13.1.10</td>
<td>104</td>
</tr>
</tbody>
</table>

Additional Fortran solvers in TOMLAB v3.0 are listed in Table 6. They are called using a set of MEX-file interfaces developed in TOMLAB.

Table 6: Additional solvers in TOMLAB /MINI v3.0 and TOMLAB v3.0.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Reference</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFZERO</td>
<td>Finding a zero to f(x) in an interval, x is one-dimensional.</td>
<td>[65, 18]</td>
<td></td>
</tr>
<tr>
<td>LSQR</td>
<td>Sparse linear least squares.</td>
<td>[56, 55, 63]</td>
<td></td>
</tr>
<tr>
<td>LEI</td>
<td>Dense constrained linear least squares</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WNNLS</td>
<td>Nonnegative constrained linear least squares</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QLD</td>
<td>Convex quadratic programming</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Another set of Fortran solvers were developed by the Stanford Optimization Laboratory (SOL). Table 7 lists the SOL optimization solvers available in TOMLAB v3.0, 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. Additional SOL solvers, listed in Table 8, are available as an extension toolbox to TOMLAB v3.0.

The add-on toolbox Tomlab /CGO solves costly global optimization problems. The solvers are listed in Table 9.
Table 7: The SOL optimization solvers in TOMLAB v3.0.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Reference</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MINOS 5.5</td>
<td>Sparse linear and nonlinear programming with linear and nonlinear constraints.</td>
<td>[53]</td>
<td></td>
</tr>
<tr>
<td>LP-MINOS</td>
<td>A special version of the MINOS 5.5 MEX-file interface for sparse linear programming.</td>
<td>[53]</td>
<td></td>
</tr>
<tr>
<td>QP-MINOS</td>
<td>A special version of the MINOS 5.5 MEX-file interface for sparse quadratic programming.</td>
<td>[53]</td>
<td></td>
</tr>
<tr>
<td>LPOPT 1.0-10</td>
<td>Dense linear programming.</td>
<td>[31]</td>
<td></td>
</tr>
<tr>
<td>QPOPT 1.0-10</td>
<td>Non-convex quadratic programming with dense constraint matrix and sparse or dense quadratic matrix.</td>
<td>[31]</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: The optimization solvers in the TOMLAB /SOL extension to TOMLAB v3.0.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Reference</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPSOL 5.02</td>
<td>Dense linear and nonlinear programming with linear and nonlinear constraints.</td>
<td>[35]</td>
<td></td>
</tr>
<tr>
<td>SNOPT 6.1-1</td>
<td>Large, sparse linear and nonlinear programming with linear and nonlinear constraints.</td>
<td>[34, 32]</td>
<td></td>
</tr>
<tr>
<td>SQOPT 6.1-1</td>
<td>Sparse convex quadratic programming.</td>
<td>[33]</td>
<td></td>
</tr>
<tr>
<td>NLSSOL 5.0-2</td>
<td>Constrained nonlinear least squares. NLSSOL is based on NPSOL. No reference except for general NPSOL reference.</td>
<td>[35]</td>
<td></td>
</tr>
<tr>
<td>LSSOL 1.05-4</td>
<td>Dense linear and quadratic programs (convex), and constrained linear least squares problems.</td>
<td>[30]</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Reference</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbfSolve</td>
<td>Costly constrained box-bounded optimization using a RBF algorithm.</td>
<td>[11]</td>
<td></td>
</tr>
<tr>
<td>ego</td>
<td>Costly constrained box-bounded optimization using the Efficient Global Optimization (EGO) algorithm.</td>
<td>[48]</td>
<td></td>
</tr>
</tbody>
</table>

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

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

gives the output

```
--------
Solver
--------
nlpSolve
conSolve
sTrustr
cstr
--------
```
and if , is called with no given argument then all available solvers for all different are printed. The second output arguments gives the 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).

<table>
<thead>
<tr>
<th>probType</th>
<th>uc</th>
<th>qp</th>
<th>con</th>
<th>ls</th>
<th>lls</th>
<th>cls</th>
<th>mip</th>
<th>lp</th>
<th>glb</th>
<th>glc</th>
</tr>
</thead>
<tbody>
<tr>
<td>uc</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>qp</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>con</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>ls</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>lls</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>cls</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>mip</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>lp</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(x)</td>
<td></td>
</tr>
<tr>
<td>glb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(x)</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>glc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(x)</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

The TOM routine saves unconstrained optimization problems with simple bounds on the parameters (uc), i.e. solves the problem

\[
\begin{align*}
\min_x & \quad f(x) \\
\text{s.t.} & \quad x_L \leq x \leq x_U,
\end{align*}
\]

where \(x, x_L, x_U \in \mathbb{R}^n\) and \(f(x) \in \mathbb{R}\). represents several of the most popular search step methods for unconstrained optimization. Bound constraints are treated as described in Gill et al. [29]. The search step methods for unconstrained optimization included in 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 [49]. 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 , combined with an initial trust region radius algorithm. The code is based on the algorithms in [17] and [62], 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 TOM solver implements an algorithm for box-bounded global optimization (glb), i.e. problems of the form (1) that have finite simple bounds on all the variables,

\[
\begin{align*}
\min_x & \quad f(x) \\
\text{s.t.} & \quad -\infty < x_L \leq x \leq x_U < \infty,
\end{align*}
\]

where \(x, x_L, x_U \in \mathbb{R}^n\), \(f(x) \in \mathbb{R}\).
glbSolve implements the DIRECT algorithm [46], 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 [48]. 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

$$\min_x f(x)$$

$$s/t \begin{array}{l}
-\infty < x_L \leq x \leq x_U < \infty \\
b_L \leq Ax \leq b_U \\
c_L \leq c(x) \leq c_U,
\end{array} \quad \forall j \in I,$$

where $x,x_L,x_U \in \mathbb{R}^n$, $f(x) \in \mathbb{R}$, $A \in \mathbb{R}^{m \times n}$, $b_L,b_U \in \mathbb{R}^m$ and $c_L,c(x),c_U \in \mathbb{R}^m$. 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 [47], that handles problems with both nonlinear and integer constraints.

The constrained nonlinear optimization problem (con) is defined as

$$\min_x f(x)$$

$$s/t \begin{array}{l}
x_L \leq x \leq x_U \\
b_L \leq Ax \leq b_U \\
c_L \leq c(x) \leq c_U
\end{array}$$

where $x,x_L,x_U \in \mathbb{R}^n$, $f(x) \in \mathbb{R}$, $A \in \mathbb{R}^{m \times n}$, $b_L,b_U \in \mathbb{R}^m$ and $c_L,c(x),c_U \in \mathbb{R}^m$. For general constrained nonlinear optimization a sequential quadratic programming (SQP) method by Schittkowski [64] 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 [25].

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 v3.0 MINOS solves constrained nonlinear programs. The TOMLAB /SOL extension gives an additional set of general solvers for dense or sparse problems.

The TOM solver lpSolve implements an simplex algorithm for the general linear program (lp) defined as

$$\min_x f(x) = c^T x$$

$$s/t \begin{array}{l}
x_L \leq x \leq x_U \\
b_L \leq Ax \leq b_U
\end{array}$$

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

When a dual feasible point is known in (5) it is efficient to use the dual simplex algorithm implemented in the TOM solver DualSolve. In TOMLAB v3.0 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.

A quadratic program (qp) is defined as

$$\min_x f(x) = \frac{1}{2} x^T F x + c^T x$$

$$s/t \begin{array}{l}
x_L \leq x \leq x_U \\
b_L \leq Ax \leq b_U
\end{array}$$

where $c,x,x_L,x_U \in \mathbb{R}^n$, $F \in \mathbb{R}^{n \times n}$, $A \in \mathbb{R}^{m \times n}$, and $b_L,b_U \in \mathbb{R}^m$. 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 [50], implemented in the TOM routine \textit{qpSolve}. \textit{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 v3.0 \textsf{MINOS} in the general or the QP-version (\textsf{QP-MINOS}), or the dense QP solver \textsf{QPOPT} may be used. In the TOMLAB /SOL extension the \textsf{SQOPT} solver is suitable for both dense and large, sparse convex QP and \textsf{SNOPT} works fine for dense or sparse nonconvex QP.

The \textbf{mixed-integer programming problem (mip)} is defined as

\[
\begin{align*}
\text{min}_x & \quad f(x) = c^T x \\
\text{s.t} & \quad x_L \leq x \leq x_U,
\end{align*}
\tag{7}
\]

where \(c, x, x_L, x_U \in \mathbb{R}^n\), \(A \in \mathbb{R}^{m \times n}\), and \(b_L, b_U \in \mathbb{R}^m\). The variables \(x \in \mathbb{I}\), the index subset of \(1, \ldots, n\) 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 \textit{mipSolve} that implements a standard branch-and-bound algorithm, see Nemhauser and Wolsey [54, chap. 8]. When dual feasible solutions are available, \textit{mipSolve} uses the TOMLAB dual simplex solver \textit{DualSolve} to solve subproblems, which is significantly faster than using an ordinary linear programming solver, like the TOMLAB lpSolve. \textit{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 used. Another TOM solver is the cutting plane routine \textit{cutplane}, using Gomory cuts. In TOMLAB /MINI v3.0, both TOM routines are using the linear programming routines in TOMLAB (\textit{lpSolve} and \textit{DualSolve}), to solve relaxed subproblems. In TOMLAB v3.0, \textit{mipSolve} is using the LP version of \textit{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 \textbf{linear least squares problem (lls)} is defined as

\[
\begin{align*}
\text{min}_x & \quad f(x) = \frac{1}{2}||Cx - d|| \\
\text{s.t} & \quad x_L \leq x \leq x_U,
\end{align*}
\tag{8}
\]

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

\textit{LSQR} solves unconstrained sparse problems. \textit{LSEI} solves the general dense problems. \textit{WNLS} is a fast and robust replacement for the Matlab mls. The general least squares solver \textit{clsSolve} may also be used. In the TOMLAB /NPSOL or TOMLAB /SOL extension the \textit{LSSOL} solver is suitable for dense linear least squares problems.

The \textbf{constrained nonlinear least squares problem (cls)} is defined as

\[
\begin{align*}
\text{min}_x & \quad f(x) = \frac{1}{2}r(x)^T r(x) \\
\text{s.t} & \quad x_L \leq x \leq x_U,
\end{align*}
\tag{9}
\]

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 \textit{clsSolve} treats problems with bound constraints in a similar way as the routine \textit{ncSolve}. This strategy is combined with an an active-set strategy to handle linear equality and inequality constraints [9].

\textit{clsSolve} includes four optimization methods for nonlinear least squares problems: the Gauss-Newton method, the Al-Baali-Fletcher [4] and the Fletcher-Xu [23] hybrid method, and the Huschens TSSM method [44]. 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 \textit{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 [24, 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 [37], somewhat modified, is implemented in *preSolve*. See [9] for a more detailed presentation.
4 Define 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>
<thead>
<tr>
<th>Matlab call</th>
<th>probTypes</th>
<th>Type of optimization problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prob = qpAssign( ... )</td>
<td>2</td>
<td>Quadratic programming.</td>
</tr>
<tr>
<td>Prob = conAssign( ... )</td>
<td>1,3</td>
<td>Unconstrained and constrained nonlinear optimization.</td>
</tr>
<tr>
<td>Prob = clsAssign( ... )</td>
<td>4,5,6</td>
<td>Unconstrained and constrained nonlinear least squares.</td>
</tr>
<tr>
<td>Prob = mipAssign( ... )</td>
<td>7</td>
<td>Mixed-Integer programming.</td>
</tr>
<tr>
<td>Prob = lpAssign( ... )</td>
<td>8</td>
<td>Linear programming.</td>
</tr>
<tr>
<td>Prob = glcAssign( ... )</td>
<td>9,10</td>
<td>Box-bounded or mixed-integer constrained global programming.</td>
</tr>
<tr>
<td>Prob = probAssign( ... )</td>
<td>1,3-6,9-10</td>
<td>General routine, but does not include all possible options.</td>
</tr>
</tbody>
</table>

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

\[
\text{Prob} = \text{mFiles}(	ext{Prob}, \ldots);
\]

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:

\[
\text{Result} = \text{conSolve}(	ext{Prob});
\]

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

\[
\text{Result} = \text{tomRun('conSolve', Prob)};
\]
Note that `tomRun` handles several input formats, also the TOMLAB Init File 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 `gqcDemo.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 \geq 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 in 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.

- 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 a 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]

```matlab
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 then edit the new Init file to include 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

```matlab
% 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

```matlab
newInitFile;
```

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

```matlab
% 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:

```matlab
% 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 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.

```matlab
% 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 Math Works 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 run 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

$$\min_x f(x) = c^T x$$

s.t. \[ x_L \leq x \leq x_U, \] \hspace{1cm} (10)

where \( c, x, x_L, x_U \in \mathbb{R}^n, A \in \mathbb{R}^{m \times n}, \) and \( b_L, b_U \in \mathbb{R}^m. \) 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 v3.0, 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

$$\min_{x_1, x_2} f(x_1, x_2) = -7x_1 - 5x_2$$

s.t. \[ x_1 + 2x_2 \leq 6 \]
\[ 4x_1 + x_2 \leq 12 \]
\[ x_1, x_2 \geq 0 \] \hspace{1cm} (11)

named LP Example.

The following statements define this problem in Matlab

File: tomlab/usersguide/lpExample.m

```matlab
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 = [];
```

28
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

```matlab
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

```plaintext
===== * * * =================================================================== * * * 
TOMLAB SOL - Three weeks Demonstration single user license Valid to 2001-04-20
===================================================================================== 
Problem: Empty Init File - 1: User Problem 1 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+011 -1.029884e+011
Projected 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,

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

Even a nonlinear solver may be used.

```matlab
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 \texttt{lpAssign} has additional functionality and may create an Init File as an option. The same applies for the files \texttt{qpAssign} and \texttt{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

\textbf{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', ...
'\texttt{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 \texttt{lpAssign}, which tells \texttt{lpAssign} to define a new TOMLAB problem Init File called \texttt{lptest}. The last two extra parameters \texttt{x min} and \texttt{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 \texttt{lptest} problem file is included in the GUI data base by the call to \texttt{AddProblemFile} and is furthermore set as the default file for LP problems. Calling \texttt{tomRun} to solve the first problem in \texttt{qptest} will give exactly the same output as in the first example in Section 5.1.1.

In the call to \texttt{lpAssign} the parameter after the file name \texttt{lptest}, called \texttt{nProblem}, controls which of two types of definition files are created (See \texttt{help lpAssign}). Setting this parameter as empty or one, as in this case, defines a program structure in the file \texttt{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 \texttt{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 \texttt{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 \texttt{lpAssign} is discussed in detail in Section 5.1.3.

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

\textbf{File: tomlab/usersguide/lpTest3.m}

```
LargeNumber=3;
```

30
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, \( i = 1 \), \( \text{lpAssign} \) defines an Init File with three test problems, and defines the first test problem, stored in \( \text{lplarge}_P1.mat \). In the other iterations \( \text{lpAssign} \) just defines the other mat-files. All together three mat-files are defined: \( \text{lplarge}_P1.mat, \text{lplarge}_P2.mat \) and \( \text{lplarge}_P3.mat \).

\( \text{AddProblemFile} \) adds the new \( \text{lplarge} \) problem as the default LP test problem in TOMLAB. The \( \text{runtest} \) 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 \( \text{lpSolve} \), 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 - Three weeks Demonstration single user license Valid to 2001-04-20
====================================================================
Problem: lplarge - 1: lptest - 1 f_k -26.472717166557445000
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 - Three weeks Demonstration single user license Valid to 2001-04-20
Problem: lplarge - 2: lptest - 2  
Solver: lpSolve. EXIT=0.  
Simplex method. Minimum reduced cost.  
Optimal solution found

FuncEv 3 GradEv 0 Iter 3  
CPU time: 0.031000 sec. Elapsed time: 0.032000 sec.

Problem: lplarge - 3: lptest - 3  
Solver: lpSolve. EXIT=0.  
Simplex method. Minimum reduced cost.  
Optimal solution found

FuncEv 3 GradEv 0 Iter 3  
CPU time: 0.032000 sec. Elapsed time: 0.031000 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 5 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 6.

Selecting the menu lp Linear Programming displays the standard menu in Figure 7.

Note that if the test problems described in this chapter has been run, there are two more entries in Figure 6, 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 8 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 5: The main menu for Linear Programming in tomMenu.

- Use Math Works Optimization TB v1.5 solver lp.

Then the call to tomRun should be:

```matlab
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:

```matlab
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

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

The general formulation in TOMLAB for a quadratic programming problem is

\[
\min_x \ f(x) = \frac{1}{2} x^T F x + c^T x
\]
\[
s/t \ \begin{array}{r}
x_L \leq x \leq x_U, \\
b_L \leq A x \leq b_U
\end{array}
\]

(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 \texttt{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 \texttt{QPOPT} called using a MEX-file interface. To choose alternative solvers, the multi-driver routine \texttt{tomRun} is called.

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

\[
\min_x \ f(x) = 4x_1^2 + x_1 x_2 + 4x_2^2 + 3x_1 - 4x_2
\]
\[
s/t \ \begin{array}{l}
x_1 + x_2 \leq 5 \\
x_1 - x_2 = 0 \\
x_1 \geq 0 \\
x_2 \geq 0
\end{array}
\]

(13)
named \texttt{QP Example}. The following statements define this problem in Matlab

File: \texttt{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

```matlab
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

```plaintext
===== * * * =================================================================== * * *
TOMLAB SOL - Three weeks Demonstration single user license Valid to 2001-04-20
===================================================================================== 
Problem: Empty Init File - 1: User Problem 1  
  f_k  -0.024999999999999994  
  f(x_0)  0.000000000000000000
```

Figure 7: The problem choice menu for Linear Programming in tomMenu.
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
Projected 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 qpprob_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);

% 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 - Three weeks Demonstration single user license Valid to 2001-04-20
==============================================================================
Problem: qplarge - 1: QP Example - 1 f_k -0.023445557772307585

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 - Three weeks Demonstration single user license Valid to 2001-04-20
==============================================================================
Problem: qplarge - 2: QP Example - 2 f_k -0.024242314364138099

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: 0.032000 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 \texttt{qpSolve}. The approach is define a default \texttt{Prob} structure calling \texttt{ProbDef}, and then just
insert values into the fields.

\begin{verbatim}
Prob = ProbDef;
Prob.QP.F = [ 8 2 ]; % Hessian.
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
Prob.b_L = [ -inf 0 ]; % Lower bounds on the constraints
Prob.b_U = [ 5 0 ]; % Upper bounds on the constraints
Result = qpSolve(Prob);
\end{verbatim}

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
\texttt{tomGUI}, the menu program \texttt{tomMenu}, or the multi-solver driver routine \texttt{tomRun} with the necessary arguments. To
call the menu system, either type \texttt{Result = tomMenu;} or just \texttt{tomMenu;} at the Matlab prompt, choose \textit{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 \textit{Weingartner 1}, which has 28 binary variables
and two knapsacks. The problem is defined

\begin{equation}
\min_x \ c^T x \\
0 \leq s/t \leq 1, \quad \text{s.t.} \quad Ax = b,
\end{equation}

where $b = (600,600)^T$, 
\begin{verbatim}
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 )^T
\end{verbatim}
and the $A$ matrix is
\[
\begin{bmatrix}
45 & 0 & 85 & 150 & 65 & 95 & 30 & 0 & 170 & 0 & 40 & 25 & 20 & 0 & 0 & 25 & 0 & 0 & 25 & 0 \\
165 & 0 & 85 & 0 & 0 & 0 & 0 & 0 & 100 \\
30 & 20 & 125 & 5 & 80 & 25 & 35 & 73 & 12 & 15 & 40 & 5 & 10 & 12 & 10 & 9 & 0 & 20 \\
60 & 40 & 50 & 36 & 49 & 40 & 19 & 150
\end{bmatrix}
\]

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 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,m);
fprintf('Making standard form with %d variables
',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 \texttt{mipAssign}. \texttt{mipAssign} is used to define the standard \texttt{Prob} structure, which TOMLAB always uses to store all information about a problem. After \texttt{mipAssign}
has defined the structure `Prob` it is then easy to set or change fields in the structure. The solver `mipSolve` is using three different strategies to search the branch-and-bound tree. The default is the `Depth first` strategy, which is also the result if setting the field `Solver.Alg` as zero. Setting the field as one gives the `Breadth first` strategy and setting it as two gives the `Depth first, then breadth search` strategy. In the example our choice is the last strategy. The number of iterations might be many, thus the maximal number of iterations must be increased, the field `optParam.MaxIter`.

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

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

Optimal Objective function = -141278.000000002900000
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 0 -1 -1 0 0 0 0 -1 0 -1 0 -1 0 0 0 0
Branch and bound. Depth First, then Breadth. Priority Weighting.

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

Optimal Objective function = -141277.999999999830000
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 0 -1 -1 0 0 0 0 -1 0 -1 0 -1 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 -139508.000000000000000
Found new BEST Knapsack. Nodes left 1. Nodes deleted 0.
Best IP function value -140768.000000000000000
Found new BEST Knapsack. Nodes left 4. Nodes deleted 0.
Best IP function value -141278.000000000000000

--- Branch & Bound converged!!! ITER = 138

Optimal Objective function = -141278.000000002900000
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 0 -1 -1 0 0 0 -1 0 -1 0 -1 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 -139508.000000000000000
Found new BEST Knapsack. Nodes left 1. Nodes deleted 0.
Best IP function value -140768.000000000000000
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 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 \( \text{Result} = \text{tomMenu}; \) or just \( \text{tomMenu}; \) 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:

\[
\text{probNumber} = 7; \\
\text{Result} = \text{tomRun('mipSolve', 'mipnew_prob', probNumber);} \\
\]

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:

\[
\text{Prob} = \text{probInit('mipnew_prob',7);} \\
\text{PriLev} = 0; \\
\text{Prob.PriLevOpt} = 0; \\
\text{Result} = \text{tomRun('mipSolve', Prob, [], PriLev);} \\
\]

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

\[
\text{PrintResult(Result);} \\
\]

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.

\[
\text{probFile} = 'ownmip_prob'; \\
\text{Prob} = \text{probInit(probFile,1)}; \\
\]

43
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 Math Works 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,

\[
\begin{align*}
\min_x \quad & f(x) = \alpha \left(x_2 - x_1^2\right)^2 + (1-x_1)^2 \\
\text{s/t} \quad & -10 \leq x_1 \leq 2 \\
& -10 \leq x_2 \leq 2
\end{align*}
\]  

(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 Math Works 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 Math Works 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 Math Works Optimization TB v2.1 and Math Works 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

```matlab
% crbb_f - function value for Constrained Rosenbrocks Banana

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 $\alpha$ parameter in the example, the field $Prob.uP$ is recommended, and for matrices, vectors and structures subfields to the field $Prob.user$ and $Prob.userParam$ are recommended. See the Section 10.3 on how 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)

```matlab
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 $\alpha$ 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)

```matlab
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)

```matlab
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 48 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_J 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('=====================================================
');
fprintf('Rosenbrocks banana with explicit f(x), g(x) and H(x)
');
fprintf('=====================================================
');
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:

```matlab
... % 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 $\alpha$ 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 $\alpha$ value. The version below is slightly modified, doing the call to *conAssign* making the parameter definitions directly. The $\alpha$ 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.

```matlab
% *---------------------------------------------------------------------
% 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 _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

```matlab
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 $x_U$ 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_.

```matlab
```
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

```matlab
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*.

```matlab
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*,

\[
\begin{align*}
\min_x & \quad f(x) = \exp(x_1)(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1) \\
& \quad -10 \leq x_1 \leq 10 \\
& \quad -10 \leq x_2 \leq 10 \\
& \quad 0 \leq x_1 + x_2 \\
& \quad 1.5 \leq -x_1x_2 + x_1 + x_2 \\
& \quad -10 \leq x_1x_2 
\end{align*}
\]  \quad (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*.

```matlab
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
```

52
% 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 100 times the exponential problem with randomly generated initial point.

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 to 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:

```plaintext
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 8. 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:

```plaintext
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

```plaintext
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 $\sigma$ influences the accuracy of the line search each step. The default value is

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

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

```plaintext
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 really wishes these methods to use $\sigma = 0.9$, the value must be set slightly different to fool the test:

```plaintext
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.

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

54
7 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 corresponding 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.

7.1 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 very many 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. To know 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 occurring, 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

```matlab
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.

7.2 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 one element gets 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:

\[
\text{Prob.SOL.optPar}(40) = 0; \quad \% \text{Use Nonderivative instead of Derivative Linesearch}
\]

### 7.3 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.

\[
\text{Prob.SOL.PrintFile} = 'snoptp.out'; \quad \% \text{New name for Print File}
\]
\[
\text{Prob.SOL.SummFile} = 'snopts.out'; \quad \% \text{New name for Summary File}
\]

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.

\[
\text{Prob.SOL.optPar}(1) = 0; \quad \% \text{No print out}
\]
\[
\text{Prob.SOL.optPar}(1) = 0; \quad \% \text{Print File unit 0}
\]

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

56
avoid any opening of files by the following statements.

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

The important thing is that besides a zero print level, also the file unit number for \textit{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 \textit{SNOPT}, \textit{NPSOL} and \textit{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 \textit{SNOPT} on files with user defined names.

```matlab
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, \textit{NPSOL}, \textit{NLSSOL}, \textit{LSSOL}, \textit{QPOPT} and \textit{LPOPT}, only define the \textit{Print File} and \textit{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.

### 7.4 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 \textit{WarmDefSOL} after having solved the problem the first time, as the following principal example shows doing recursive calls to \textit{SNOPT}.

```matlab
... % 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);
    Result = tomSolve('snopt',Prob); % Call tomSolve to avoid checks of Prob
    ...
    % Postprocessing
  end
```

The \textit{WarmDefSOL} routine are setting the warm start flag \textit{Prob.WarmStart}, as true.

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

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

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

For \textit{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

```matlab
... % 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
    ... % Changes at time t in Prob structure
    Prob = WarmDefSOL('npsol', Prob, Result);
    Result = tomSolve('npsol',Prob); % Call tomSolve to avoid checks of 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.
8 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 \textit{glb} and global mixed-integer nonlinear programming \textit{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 \textit{examples}. Running the menu program \textit{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 \textit{glbDemo} and \textit{glcDemo}.

8.1 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 \textit{Shekel 5} test problem in a routine \textit{glb1.f}, the following statements are needed

function \textit{f} = \textit{glb1.f}(\textit{x}, \textit{Prob})
\begin{verbatim}
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
\end{verbatim}

To solve the \textit{Shekel 5} test problem define the following statements, available as \textit{glb1Demo} in \textit{glbDemo}.

function \textit{glb1Demo}
\begin{verbatim}
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);
\end{verbatim}

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 \textit{Shekel 5} problem, the optimal function value is known and could be set as target value with the following statements.

\begin{verbatim}
Prob.optParam.fGoal = -10.1532; %The optimal value set as target
Prob.optParam.eps_f = 0.01; %Convergence tolerance one percent
\end{verbatim}

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]';

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

8.2 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
9 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 Math Works 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.

9.1 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 [30].

Name='LSSOL test example';

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

% Linear constraints
A = [ones(1,8), 1:4, -2, 1 1 1 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 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 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 = [];
weightY = [];
weightType = [];
x_min = [];
x_max = [];

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.
9.2 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* [30].

% 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 1 1 1 1 1 2 0 0; 1 1 3 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 1 2 1 1 0 0 0 -1; ... 1 1 1 1 0 1 1 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
',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
',f);
9.3 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 lsI_r in the directory example with the statements

```matlab
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;

% Compute 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;

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.

```matlab
%---------------------------------------------------------------------
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; 3.5; 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',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.

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

9.4 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 [61]. 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.

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

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 [57]. 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.
### Table 14: Exponential fitting test problems.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exp.ArtP</code></td>
<td>Generate artificial exponential sum problems.</td>
</tr>
<tr>
<td><code>expInit</code></td>
<td>Find starting values for the exponential parameters $\lambda$.</td>
</tr>
<tr>
<td><code>exp.prob</code></td>
<td>Defines a exponential fitting type of problem, with data series $(t, y)$.</td>
</tr>
<tr>
<td><code>Helax.prob</code></td>
<td>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 <code>exp.prob</code>.</td>
</tr>
<tr>
<td><code>exp.r</code></td>
<td>Compute the residual vector $r_i(x), i = 1, ..., m. x \in \mathbb{R}^n$.</td>
</tr>
<tr>
<td><code>exp.J</code></td>
<td>Compute the Jacobian matrix $\partial r_i/dx_j, i = 1, ..., m, j = 1, ..., n$.</td>
</tr>
<tr>
<td><code>exp.d2r</code></td>
<td>Compute the 2nd part of the second derivative for the nonlinear least squares exponential fitting problem.</td>
</tr>
<tr>
<td><code>exp.c</code></td>
<td>Compute the constraints $\lambda_1 &lt; \lambda_2 &lt; ...$ on the exponential parameters $\lambda_i, i = 1, ..., p$.</td>
</tr>
<tr>
<td><code>exp.dc</code></td>
<td>Compute matrix of constraint normals for constrained exponential fitting problem.</td>
</tr>
<tr>
<td><code>exp.d2c</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>exp.q</code></td>
<td>Find starting values for exponential parameters $\lambda_i, i = 1, ..., p$.</td>
</tr>
<tr>
<td><code>exp.p</code></td>
<td>Find optimal number of exponential terms, $p$.</td>
</tr>
</tbody>
</table>
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 particularly important from a practical point of view. It often seems to be the case that either it is nearly impossible or the user has difficulties in coding the derivatives.

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.

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 \textit{fdng}, which is a direct implementation of the FD algorithm [29, page 343].

The \textit{fdng} routine is using the parameter field \texttt{DiffInt}, in the structure \texttt{optParam}, see Table 39, as the numerical step size. The user could either change this field or set the field \texttt{Prob.GradTolg}. The field \texttt{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 \texttt{Prob.GradTolg} is defined as a negative number, the \textit{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 \textit{fdng}, there are two routines \textit{FDJac} and \textit{FDHess}. \textit{FDJac} numerically estimates the Jacobian in nonlinear least squares problems or the constraint Jacobian in constrained nonlinear problems. \textit{FDJac} checks if the field \texttt{Prob.GradTolJ} is defined, with the same action as \textit{fdng}. \textit{FDHess} estimates the Hessian matrix in nonlinear problems and checks for the definition of the field \texttt{Prob.GradTolH}. Both routines use field \texttt{Prob.optParam.DiffInt} as the default tolerance if the other field is empty. Note that \textit{FDHess} has no automatic step size estimation. The implementation in \textit{fdng}, \textit{FDJac} and \textit{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.

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 \texttt{csapi}, \texttt{csaps} or \texttt{spaps} the user choose to use. The routines \textit{fdng2}, \textit{FDJac2} and \textit{FDHess2} implements the gradient estimation procedure for the different approximations needed. All routines use the tolerance in the field \texttt{Prob.optParam.CentralDiff} as the numerical step length. The basic principle is central differences, taking a small step in both positive and negative direction.

The fifth approximation method is a method by Squire and Trapp [67], 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 \textit{fdng3} implements the complex variable method for the estimation of the gradient and \textit{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 [51], 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.

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 \texttt{Prob} structure using the fields \texttt{Prob.JacPattern}, \texttt{Prob.ConsPattern} or \texttt{Prob.HessPattern}. If there are many zeros in the matrix, this leads to significant savings in each iteration of the optimization.

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 ...	omlab\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.

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

```matlab
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
% 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 are 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 \textit{tomSolve}. The syntax is similar to the syntax of \textit{tomRun}. Calling \textit{QPOPT} to solve a QP sub problem is made with the call

\begin{verbatim}
Result = tomRun('qpopt', Prob);
\end{verbatim}

The big advantage is that \textit{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. \textit{glcSolve} was used to find a good initial value for the main optimization and \textit{SNOPT} was used to find the exact solution. In each iteration several small optimization problems had to be solved. Here \textit{glcSolve} was just to find a good initial point close to the global optimum, and \textit{MINOS} then iterated until good accuracy was found.

The general TOM solvers \textit{clsSolve}, \textit{conSolve}, \textit{cutplane}, \textit{mipSolve}, \textit{nlpSolve}, \textit{qpSolve} and \textit{sTrustr} have all been designed so it shall be possible to change the subproblem solver. For example to solve the QP subproblems in \textit{conSolve} there are several alternatives, \textit{QPOPT}, \textit{qpSolve} or even \textit{SNOPT}. If using the BFGS update in \textit{conSolve}, which guarantees that the subproblems are convex, then furthermore \textit{QLD} or \textit{SQOPT} could be used. The QP, LP, FP (feasible point) and DLP (dual LP) subproblems have special treatment. A routine \textit{CreateProbQP} creates the \textit{Prob} structure for the subproblem. The routine checks on the fields \textit{Prob.QP.SOL} and \textit{Prob.QP.optParam} and move these to the usual places \textit{Prob.SOL} and \textit{Prob.optParam} for the subproblem. Knowing this, the user may send his own choices of these two important subfields as input to \textit{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 \textit{Prob.SolverQP} for QP subproblems and similar for the other subproblems. Note that the time consuming call to \textit{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 \textit{tomSolve} instead of \textit{tomRun}. But no checks are made on the structure \textit{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

\begin{verbatim}
Prob.LargeScale = 1;
\end{verbatim}

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:

```matlab
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 2.1.

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.

```matlab
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:

```matlab
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, \( x_{\text{opt}} \) was set as \((1, 1)\) in the problem definition files. If it is known that this point is a minimum point the definition of \( x_{\text{opt}} \) could be extended to

\[
x_{\text{opt}} = [1 1 \text{StatPntType}] ;
\]

where \text{StatPntType} equals 0, 1, or 2 depending on the type of the stationary point (minimum, saddle or maximum). In this case set \text{StatPntType} to 0 since \((1,1)\) is a minimum point. The extension becomes

\[
x_{\text{opt}} = [1 1 0] ; \quad \text{Known optimal point (optional)}.
\]

If there is more than one known stationary point, the points are defined as rows in a matrix with the values of \text{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 \( x_{\text{opt}} \) could then look like

\[
x_{\text{opt}} = [ -1 -1 1 \\
1 -2 0 \\
-3 5 2 ] ;
\]

Note that it is not necessary to define \( x_{\text{opt}} \). If \( x_{\text{opt}} \) is defined it is not necessary to define \text{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 \text{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 \text{Prob.PriLev}. The other printing level, defined in \text{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 \text{Prob.optParam.PriLev}, that determines how much output is written by the SOL solvers on files. See Section 7.3 for details on this.
The utility routine \textit{PrintResult} prints the results of an optimization given the \textit{Result} structure. The amount of printing is determined by a second input argument \textit{PriLev}. The driver routine \textit{tomRun} also makes a call to \textit{PrintResult} after the optimization, and if the input parameter \textit{PriLev} is greater than zero, the result will be the same as calling \textit{PrintResult} afterwards.

\textit{PrintResult} is using the global variables, \textit{MAX\_c}, \textit{MAX\_x} and \textit{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 \textit{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>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>Totally silent.</td>
</tr>
<tr>
<td>0</td>
<td>Error messages and warnings.</td>
</tr>
<tr>
<td>1</td>
<td>Final results including convergence test results and minor warnings.</td>
</tr>
<tr>
<td>2</td>
<td>Each iteration, short output.</td>
</tr>
<tr>
<td>3</td>
<td>Each iteration, more output.</td>
</tr>
<tr>
<td>4</td>
<td>Line search or QP information.</td>
</tr>
<tr>
<td>5</td>
<td>Hessian output, final output in solver.</td>
</tr>
</tbody>
</table>

There is a wait flag field in \textit{optParam}, \textit{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 \textit{RETURN} to continue execution. The fields in \textit{optParam} is described in Table 39.

The TOM solvers routines print large amounts of output if high values for the \textit{PriLev} parameter is set. To make the output look better and save space, several printing utilities have been developed, see Table 23 page 124.

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

The routine \textit{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 \textit{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 \textit{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.

A function $f$ is partially separable if $f(x) = \sum_{i=1}^{M} f_i(x)$, where, for each $i \in \{1, \ldots, M\}$ there exists a subspace $N_i \neq 0$ such that, for all $w \in N_i$ and for all $x \in X$, it holds that $f_i(x + w) = f_i(x)$. $X$ is the closed convex subset of $\mathbb{R}^n$ defined by the constraints.

Consider the problem DAS 2 in File: `tomlab/testprob/con_prob`:

\[
\begin{align*}
\min_{x} & \quad f(x) = \frac{1}{4} \sum_{i=1}^{6} r_i(x)^2 \\
\text{s.t} & \quad Ax \geq b \\
& \quad x \geq 0
\end{align*}
\]

where

\[
\begin{pmatrix}
\frac{\sqrt{T}}{6} x_1 - \frac{3}{\sqrt{11}} \\
\frac{x_2 - 3}{\sqrt{2}} \\
\frac{1}{\sqrt{0.0775}} x_3 + \frac{0.5}{\sqrt{0.0775}} \\
\frac{1}{\sqrt{2}} x_3 \\
\frac{1}{\sqrt{0.0775}} x_3 + \frac{3}{4} x_4 \\
\end{pmatrix},
\begin{pmatrix}
-1 & -2 & -1 & -1 \\
-3 & -1 & -2 & 1 \\
0 & 1 & 4 & 0 \\
-5 & -4 & 1.5
\end{pmatrix}
\]

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:

```matlab
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:

```matlab
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 Math Works 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 nargin. 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 nargin in Matlab 5.1 does lower on the arguments in the call to nargin, and the nargin
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>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtest</td>
<td>Runs all selected problems defined in a problem file for a given solver.</td>
<td>13.2.8</td>
<td>118</td>
</tr>
<tr>
<td>systest</td>
<td>Runs big test for each probType in TOMLAB.</td>
<td>13.2.10</td>
<td>119</td>
</tr>
</tbody>
</table>

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 \texttt{tomlabGUI.m}, i.e. by entering the command \texttt{tomlabGUI} at the Matlab prompt. There is a short command \texttt{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 9.

![Figure 9: The GUI after startup.](image)

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 than 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 9.

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 pushing 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 10.

A contour plot for a constrained problem and a plot of the data is given in Figure 11. 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 $(x_1^*, x_2^*) = (-9.5474, 1.0474)$.

In Figure 12 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 13, 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 9, 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.
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 14.

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 edit control named ‘Starting Values x3 - xn’. Starting values for \( x_1 \) and \( x_2 \) are given in the edit controls named...
Figure 11: A contour plot for a constrained problem

'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 heck 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.

**Code Generation**

Entering a name in the edit control Code Generation File and pushing 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 pushing 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.
Figure 13: 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.

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 [24]. In Figure 15 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 $\sigma$ (Sigma) between zero and one. Values 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 16. 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’.

When a problem is solved, the user can access the results from the Matlab Command Window, stored in the global structure \textit{Result}. If the user has not run the TOMLAB initialization command \textit{tomlabInit}, he must enter the command \textit{global Result} at the Matlab prompt to declare \textit{Result} as a global structure. To display the full structure, enter \textit{Result} at the prompt. To display a specific field in the structure, e.g. the solution found, enter \textit{Result.x_k}. All information stored in the structure are given in Table 45. When the check box ‘Hold Previous Run’ is selected, \textit{Result} becomes a structure array. As an example, to display the results from the third run, enter the command \textit{Result(3)}. To display the solution found in the third run, enter the command \textit{Result(3).x_k}.

The user could also access the plotting parameter structure \textit{plotData} in the same way as described for the Result structure in SectionB.
Figure 15: The Solver Parameters for the MINOS solver.
Figure 16: 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 17, is the selection of the type of optimization problem \( \text{probType} \).

![Figure 17: The choice of the Type of Optimization in tomMenu.](image)

The tomMenu sub-menu for unconstrained optimization is shown in Figure 18. 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 19.
Pushing the \textit{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. Pushing the \textit{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 \texttt{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 \texttt{tomMenu} by typing \texttt{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 \texttt{Result(3)}. To display the solution found in the third run, enter the command \texttt{Result(3).x_k}. The information stored in the structure are given in Table 45.

The menu program calls the driver routine \texttt{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 19: Setting optimization parameters for quadratic programming.
13 The TOMLAB Routines - Detailed Descriptions

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 of the solver.

13.1 Optimization Routines in TOMLAB - The TOM solvers

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

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

\[
\begin{align*}
\min_{x} & \quad f(x) = \frac{1}{2} r(x)^T r(x) \\
\text{s.t} & \quad x_L \leq x \leq x_U \\
& \quad b_L \leq Ax \leq b_U
\end{align*}
\]

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).

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.

LS Structure with special fields for nonlinear least squares, see Table 35.

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.

LineParam Structure with line search parameters, see Table 38.

varargin Other parameters directly sent to low level routines.
Description of Outputs

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: 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.
- **f_0**: Function value at start.
- **f_k**: Function value at optimum.
- **g_k**: Gradient value at optimum.
- **B_k**: Quasi-Newton approximation of the Hessian at optimum.
- **x_0**: Starting point.
- **x_k**: Optimal point.
- **v_k**: Lagrange multipliers.
- **r_k**: Residual at optimum.
- **J_k**: Jacobian matrix at optimum.
- **xState**: State of each variable, described in Table 46.
- **bState**: State of each linear constraint, described in Table 47.
- **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 [23] hybrid method, and the Hushens TSSM method [44]. 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 [24]. Bound constraints are partly treated as described in Gill, Murray and Wright [29]. 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

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

\[
\begin{align*}
\min_x f(x) \\
\text{s.t. } & x_L \leq x \leq x_U \\
& b_L \leq Ax \leq b_U \\
& c_L \leq c(x) \leq c_U
\end{align*}
\]
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:
  - 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.

- **optParam**  
  Structure with optimization parameters, see Table 39.
  Fields used are: \( b_{\text{Tol}}, c_{\text{Tol}}, \text{eps.absf}, \text{eps.g}, \text{eps.x}, \text{eps.Rank}, \text{IterPrint}, \text{MaxIter}, \text{QN_InitMatrix}, \text{size.f}, \text{size.x}, x_{\text{Tol}} \) and \( \text{wait} \).

- **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 \( \partial c(x)/\partial x \).

- **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:

- **Iter**: Number of iterations.
- **ExitFlag**: Flag giving exit status.
- **Inform**: Binary 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.

- **\( f_0 \)**: 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.
- **\( u_k \)**: Lagrange multipliers.
- **\( c_k \)**: Value of constraints at optimum.
- **\( cJac \)**: Constraint Jacobian at optimum.
- **\( xState \)**: State of each variable, described in Table 46.
- **\( bState \)**: State of each linear constraint, described in Table 47.
- **\( cState \)**: State of each general constraint.
- **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 [64]. 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.
<table>
<thead>
<tr>
<th>Solver.Alg</th>
<th>NumDiff</th>
<th>AutoDiff</th>
<th>isempty(USER.H)</th>
<th>Hessian computation</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Analytic Hessian</td>
<td>Schittkowski SQP</td>
</tr>
<tr>
<td>0</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>BFGS</td>
<td>Schittkowski SQP</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Analytic Hessian</td>
<td>Schittkowski SQP</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Numerical differences H</td>
<td>Schittkowski SQP</td>
</tr>
<tr>
<td>1</td>
<td>&gt; 0</td>
<td>0</td>
<td>any</td>
<td>Numerical differences g,H</td>
<td>Schittkowski SQP</td>
</tr>
<tr>
<td>1</td>
<td>&lt; 0</td>
<td>0</td>
<td>any</td>
<td>Numerical differences H</td>
<td>Schittkowski SQP</td>
</tr>
<tr>
<td>1</td>
<td>any</td>
<td>1</td>
<td>any</td>
<td>Automatic differentiation</td>
<td>Schittkowski SQP</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>any</td>
<td>BFGS</td>
<td>Schittkowski SQP</td>
</tr>
<tr>
<td>2</td>
<td>= 0</td>
<td>0</td>
<td>any</td>
<td>BFGS, numerical gradient g</td>
<td>Schittkowski SQP</td>
</tr>
<tr>
<td>2</td>
<td>any</td>
<td>1</td>
<td>any</td>
<td>BFGS, automatic diff gradient</td>
<td>Schittkowski SQP</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Analytic Hessian</td>
<td>Han-Powell SQP</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Numerical differences H</td>
<td>Han-Powell SQP</td>
</tr>
<tr>
<td>3</td>
<td>&gt; 0</td>
<td>0</td>
<td>any</td>
<td>Numerical differences g,H</td>
<td>Han-Powell SQP</td>
</tr>
<tr>
<td>3</td>
<td>&lt; 0</td>
<td>0</td>
<td>any</td>
<td>Numerical differences H</td>
<td>Han-Powell SQP</td>
</tr>
<tr>
<td>3</td>
<td>any</td>
<td>1</td>
<td>any</td>
<td>Automatic differentiation</td>
<td>Han-Powell SQP</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>any</td>
<td>BFGS</td>
<td>Han-Powell SQP</td>
</tr>
<tr>
<td>4</td>
<td>= 0</td>
<td>0</td>
<td>any</td>
<td>BFGS, numerical gradient g</td>
<td>Han-Powell SQP</td>
</tr>
<tr>
<td>4</td>
<td>any</td>
<td>1</td>
<td>any</td>
<td>BFGS, automatic diff gradient</td>
<td>Han-Powell SQP</td>
</tr>
</tbody>
</table>

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

\[
\min_{x} \quad f(x) = c^T x \\
\text{s.t.} \quad 0 \leq x \leq x_U \quad Ax = b, \quad x_j \in \mathbb{N} \forall j \in I
\]

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 \( 1, \ldots, n \) are restricted to be integers.

Calling Syntax

\[ [\text{Result}] = \text{cutplane}(\text{Prob}) \]
Description of Inputs

Prob                Problem description structure. The following fields are used:

optParam.method     Variable selection rule to be used:
0: Minimum reduced cost.
1: Bland’s rule (default).
2: Minimum reduced cost. Dantzig’s rule.

MIP.IntVars         Which of the n variables are integers.

QP.B                Active set B[i] 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: 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 46.
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.

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

\[
\begin{align*}
\min_x & \quad f(x) = c^T x \\
\text{s.t.} & \quad x_L \leq x \leq x_U \\
& \quad Ax = b_U
\end{align*}
\]

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

\[
\begin{align*}
\min_x & \quad f_P(x) = c^T x \\
\text{s.t.} & \quad \hat{A} x = b \\
& \quad x \geq 0
\end{align*}
\]

and the following dual problem is solved,

\[
\begin{align*}
\max_y & \quad f_D(y) = b^T y \\
\text{s.t.} & \quad \hat{A}^T y \leq c
\end{align*}
\]

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

**Calling Syntax**

\[
[\text{Result}] = \text{DualSolve}(\text{Prob})
\]

**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.

fk       Function value at optimum.
x0       Starting point.
xk       Optimal primal solution x.
vk       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 [36, 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  glbSolve

Purpose
Solve box-bounded global optimization problems.

glbSolve solves problems of the form

\[ \min_x f(x) \quad s.t. \quad x_L \leq x \leq x_U \]

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 [46]. 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 [60, 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*

13.1.6 *glbFast*

Purpose

Solve box-bounded global optimization problems.
$\text{glbFast}$ solves problems of the form

$$\min_x f(x) \quad \text{s.t.} \quad x_L \leq x \leq x_U$$

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

**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 $\text{glbFast}$ is an implementation of the DIRECT algorithm presented in [46]. The algorithm in $\text{glbFast}$ is a Fortran MEX implementation of the algorithm in $\text{glbSolve}$. 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 $\text{glbFast}$ 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 $\text{glbFast}$ with the final status of all parameters from the previous run, a so called warm start. Assume that a run has been made with $\text{glbFast}$ 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 $\text{glbFast}$ a flag **Prob.WarmStart** should be set to one. Then $\text{glbFast}$ is using output previously written to the file $\text{glbFastSave.mat}$ to make the restart. $\text{glbFast}$ also includes the subfunction $\text{conhull}$ which is an implementation of the algorithm GRAHAMHULL in [60, page 108] with the modifications proposed on page 109. $\text{conhull}$ 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

13.1.7 glcSolve

Purpose
Solve global mixed-integer nonlinear programming problems.

\[
\begin{align*}
\min_x & \quad f(x) \\
\text{s.t.} & \quad x_L \leq x \leq x_U \\
& \quad b_L \leq Ax \leq b_U \\
& \quad c_L \leq c(x) \leq c_U \\
& \quad x_i \text{ integer } \quad i \in I
\end{align*}
\]

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
\[
\text{Result} = \text{glcSolve}(\text{Prob}, \text{varargin})
\]

Description of Inputs

\textit{Prob}
Problem description structure. The following fields are used:

- \textit{optParam} Structure with special fields for optimization parameters, see Table 39.
  Fields used are: PriLev, cTol.
- \textit{x_L} Lower bounds for \( x \), must be given to restrict the search space.
- \textit{x_U} Upper bounds for \( x \), must be given to restrict the search space.
- \textit{A} Constraint matrix for linear constraints.
- \textit{b_L} Lower bounds on the linear constraints.
- \textit{b_U} Upper bounds on the linear constraints.
- \textit{c_L} Lower bounds on the general constraints.
- \textit{c_U} Upper bounds on the general constraints.
- \textit{USER.f} Name of m-file computing the objective function \( f(x) \).
- \textit{USER.g} Name of m-file computing the vector of constraint functions \( c(x) \).

\textit{GLOBAL} Special structure field containing:

- \textit{MaxEval} Number of function evaluations, default 200.
- \textit{Integers} Set of integer variables.
- \textit{epsilon} Global/local weight parameter, default \( 10^{-4} \).
- \textit{K} The Lipschitz constant. Not used.
- \textit{tolerance} Error tolerance parameter. Not used.

If restart is chosen in the menu system, the following fields in \textit{GLOBAL} are also used and contains information from the previous run:

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

\textit{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.
- **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 [47], 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*

13.1.8 glcFast

Purpose

Solve global mixed-integer nonlinear programming problems.

*glcFast* solves problems of the form

$$\min_{x} f(x) \quad s/t \quad x_L \leq x \leq x_U \quad b_L \leq Ax \leq b_U \quad c_L \leq c(x) \leq c_U \quad x_i \text{ integer } \quad i \in I$$

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

101
Calling Syntax
Result = glcFast(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.f    Name of m-file computing the vector of constraint functions \( c(x) \).
GLOBAL     Special structure field containing:
          MaxEval     Number of function evaluations, default 200.
          Integers    Set of integer variables.
          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.
          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.
          IL(i)      \( IL(i, j) \) is the lower bound for rectangle \( j \) in integer dimension \( I(i) \).
          IU(i)      \( IU(i, j) \) is the upper bound for rectangle \( j \) in integer dimension \( I(i) \).
          feasible   Flag indicating if a feasible point has been found.
          fmin       Best function value found at a feasible point.
          s0         \( s_0 \) 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 \( i \).
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.
- **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**: \( \text{Split}(i, j) \) 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, j)**: The lower bound for rectangle \( j \) in integer dimension \( I(i) \).
  - **I_U(i, j)**: 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 [47], 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`.

### 13.1.9 lpSolve

**Purpose**
Solve general linear programming problems.

lpSolve solves problems of the form

\[
\begin{align*}
\min_x \quad & f(x) = c^T x \\
\text{s.t.} \quad & x_L \leq x \leq x_U \\
& b_L \leq Ax \leq b_U
\end{align*}
\]

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**

\[ \text{Result} = \text{lpSolve}(\text{Prob}) \]
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: \( \text{MaxIter}, \text{PriLev}, \text{wait}, \text{eps}_f, \text{eps}_\text{Rank}, x\text{Tol} \) and \( b\text{Tol} \).

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 \( \text{ExitFlag} > 0 \), \( \text{Inform} = \text{ExitFlag} \).

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 46.

Solver
Solver used.

SolverAlgorithm
Solver algorithm used.

FuncEv
Number of function evaluations. Equal to \( \text{Iter} \).

ConstrEv
Number of constraint evaluations. Equal to \( \text{Iter} \).

Prob
Problem structure used.

Description
The routine \( \text{lpSolve} \) implements an active set strategy (Simplex method) for Linear Programming using an additional set of slack variables for the linear constraints. If the given starting point is not feasible then a Phase I objective is used until a feasible point is found.

M-files Used
\( \text{ResultDef.m} \)

See Also
\( \text{qpSolve} \)

13.1.10 \textbf{mipSolve}

Purpose
Solve mixed integer linear programming problems (MIP).
mipSolve solves problems of the form

$$\min \quad f(x) = c^T x$$
$$s/t \quad x_L \leq x \leq x_U$$
$$Ax = b, \quad x_j \in \mathbb{N} \forall j \in I$$

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.

**Calling Syntax**

[Result] = mipSolve(Prob)

**Description of Inputs**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prob</td>
<td>Problem description structure. The following fields are used:</td>
<td></td>
</tr>
<tr>
<td>optParam.method</td>
<td>Variable selection rule to be used:</td>
<td>0: Minimum reduced cost.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Bland’s rule (default).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2: Minimum reduced cost. Dantzig’s rule.</td>
</tr>
<tr>
<td>MIP.IntVars</td>
<td>Which of the $n$ variables are integers.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If IntVars is a scalar, then variables 1,...,IntVars are assumed to be integers.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If empty, all variables are assumed non-integer (LP problem)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If length(IntVars) == length(c) must hold,</td>
<td></td>
</tr>
<tr>
<td>MIP.VarWeight</td>
<td>Weight for each variable in the variable selection phase. A lower value gives higher priority.</td>
<td>Setting Prob.MIP.VarWeight = c; for knapsack problems improve convergence.</td>
</tr>
<tr>
<td>MIP.fIP</td>
<td>An upper bound on the IP value wanted. Makes it possible to cut branches and avoid non-optimal solutions.</td>
<td></td>
</tr>
<tr>
<td>MIP.xIP</td>
<td>The x-values giving the fIP value.</td>
<td></td>
</tr>
<tr>
<td>MIP.KNAPSACK</td>
<td>True if a knapsack problem is to be solved, then a knapsack heuristic is used.</td>
<td></td>
</tr>
<tr>
<td>QP.B</td>
<td>Active set $B_0$ at start:</td>
<td>$B(i) = 1$: Include variable $x(i)$ is in basic set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$B(i) = 0$: Variable $x(i)$ is set on its lower bound.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$B(i) = -1$: Variable $x(i)$ is set on its upper bound.</td>
</tr>
<tr>
<td>optParam</td>
<td>Structure with special fields for optimization parameters, see Table 39.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fields used are: MaxIter, PriLev, wait, eps_f, eps_Rank, xTol and bTol.</td>
<td></td>
</tr>
<tr>
<td>QP.c</td>
<td>Constant vector.</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>Constraint matrix for linear constraints.</td>
<td></td>
</tr>
<tr>
<td>b_L</td>
<td>Lower bounds on the linear constraints.</td>
<td></td>
</tr>
<tr>
<td>b_U</td>
<td>Upper bounds on the linear constraints.</td>
<td></td>
</tr>
<tr>
<td>x_L</td>
<td>Lower bounds on the variables.</td>
<td></td>
</tr>
<tr>
<td>x_U</td>
<td>Upper bounds on the variables.</td>
<td></td>
</tr>
<tr>
<td>x_0</td>
<td>Starting point.</td>
<td></td>
</tr>
</tbody>
</table>
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 $\text{ExitFlag} > 0$, $\text{Inform} = \text{ExitFlag}$.
  - **QP.B**: Optimal active set. See input variable $\text{QP.B}$.
  - **$f_0$$f_k$$g_k$$x_0$$x_k$$v_k$$xState$$Solver$$SolverAlgorithm$$FuncEv$$ConstrEv**
  - **Prob**: Problem structure used.

Description

The routine $\text{mipSolve}$ is an implementation of a branch and bound algorithm from Nemhauser and Wolsey [54, chap. 8.2]. $\text{mipSolve}$ normally uses the linear programming routines $\text{lpSolve}$ and $\text{DualSolve}$ to solve relaxed subproblems. $\text{mipSolve}$ calls the general interface routines $\text{SolveLP}$ and $\text{SolveDLP}$. By changing the setting of the structure fields $\text{Prob.Solver.SolverLP}$ and $\text{Prob.Solver.SolverDLP}$, different sub-solvers are possible to use, see the help for the interface routines.

Algorithm

See [54, chap. 8.2] and the code in $\text{mipSolve.m}$.

Examples

See $\text{exip39}$, $\text{exknap}$, $\text{expkorv}$.

M-files Used

$\text{lpSolve.m}$, $\text{DualSolve.m}$

See Also

$\text{cutplane}$, $\text{balas}$, $\text{SolveLP}$, $\text{SolveDLP}$

13.1.11 **nlpSolve**

Purpose

Solve general constrained nonlinear optimization problems.

$nlpSolve$ solves problems of the form

$$
\min_x \quad f(x) \\
\text{s.t.} \quad x_L \leq x \leq x_U \\
\quad b_L \leq Ax \leq b_U \\
\quad c_L \leq c(x) \leq c_U
$$

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_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 La

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.

  - 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 46.

- **bState**  State of each linear constraint, described in Table 47.

- **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 [25].

M-files Used

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

See Also

*conSolve*, *sTrustr*

13.1.12  **qpSolve**

Purpose

Solve general quadratic programming problems.
qpSolve solves problems of the form

$$\min_x f(x) = \frac{1}{2}x^TFx + c^Tx$$

subject to

$$s/t \ x_L \leq x \leq x_U$$
$$b_L \leq Ax \leq b_U$$

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: \(\lambda\) 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 46.
  - **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 \(conSolve\)
13.1.13  \texttt{sTrustr}

Purpose
Solve optimization problems constrained by a convex feasible region.

\texttt{sTrustr} solves problems of the form

\[
\begin{align*}
\min & \quad f(x) \\
\text{s.t} & \quad x_L \leq x \leq x_U \\
& \quad b_L \leq Ax \leq b_U \\
& \quad c_L \leq c(x) \leq c_U
\end{align*}
\]

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 = \texttt{sTrustr}(Prob, varargin)

Description of Inputs
\textit{Prob}  Problem description structure. The following fields are used:
  \textit{optParam}  Structure with special fields for optimization parameters, see Table 39.
    Fields used are: \texttt{eps.f}, \texttt{eps.g}, \texttt{eps.c}, \texttt{eps.Range}, \texttt{MaxIter}, \texttt{wait}, \texttt{size.x}, \texttt{size.f}, \texttt{xTol}, \texttt{LowIts}, \texttt{PriLev}, \texttt{method} and \texttt{QN_InitMatrix}.
  \textit{PartSep}  Structure with special fields for partially separable functions, see Table 40.
  \textit{A}  Constraint matrix for linear constraints.
  \textit{b.L}  Lower bounds on the linear constraints.
  \textit{b.U}  Upper bounds on the linear constraints.
  \textit{c.L}  Lower bounds on the general constraints.
  \textit{c.U}  Upper bounds on the general constraints.
  \textit{x.L}  Lower bounds on the variables.
  \textit{x.U}  Upper bounds on the variables.
  \textit{x.0}  Starting point.
  \textit{USER.f}  Name of m-file computing the objective function \( f(x) \).
  \textit{USER.g}  Name of m-file computing the gradient vector \( g(x) \).
  \textit{USER.H}  Name of m-file computing the Hessian matrix \( H(x) \).
  \textit{USER.c}  Name of m-file computing the vector of constraint functions \( c(x) \).
  \textit{USER.dc}  Name of m-file computing the matrix of constraint normals \( \partial c(x)/dx \).

\textit{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 46.
- **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 [17] and [62]. 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`

### 13.1.14 itrr

#### Purpose

Determine the initial trust region radius.

#### Calling Syntax

```matlab
[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 \( itr \) implements the initial trust region radius algorithm as described by Sartenaer in [62]. \( itr \) is called by \( sTrustr \).

See Also
\( sTrustr \)

13.1.15 \( ucSolve \)

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

\( ucSolve \) solves problems of the form
\[
\begin{align*}
\min_x f(x) \\
\text{s/t} \quad x_L \leq x \leq x_U
\end{align*}
\]

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 update of Hessians.
  6: Fletcher-Reeves CG.
  7: Polak-Ribiere CG.
  8: Fletcher conjugate descent CG-method.
  Solver.Method Method 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,f, eps,g, eps,x, eps_Rank, MaxIter, wait, size_x, size_f, LineSearch,
                 LineAlg, xTol, 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.
  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 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 \( \text{LowIts} \) iterations.
  101: Maximum number of iterations reached.
  102: Function value below given estimate.
  104: Convergence to a saddle point.
  f_0 Function value at start.
  f_k Function value at optimum.
  g_k Gradient value at optimum.
  H_k Hessian value at optimum.
  B_k Quasi-Newton approximation of the Hessian at optimum.
  x_0 Starting point.
  x_k Optimal point.
  v_k Lagrange multipliers.
  xState State of each variable, described in Table 46 .
  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 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 [49]. 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 [24]. Bound constraints are treated as described in Gill, Murray and Wright [29].

M-files Used
ResultDef.m, LineSearch.m, iniSolve.m, tomSolve.m, endSolve.m

See Also
clsSolve

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 ≥ 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 117.
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 45.
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 form 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

\[
\min_x \quad f(x)
\]
\[
s/t \quad x_L \leq x \leq x_U
\]
\[
b_L \leq Ax \leq b_U
\]

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

\[
[AA, bb, meq] = \text{cpTransf}(\text{Prob}, \text{TransfType}, \text{makeEQ}, \text{LowInf})
\]

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 \(-\text{Inf}\) or variables < `LowInf` are set to `LowInf` before transforming the problem. Default \(-10^{-4}\). \(|\text{LowInf}|\) 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 `TransfType = 1` the program is transformed into the form

\[
\min_x \quad f(x - x_L)
\]
\[
s/t \quad AA(x - x_L) \leq bb
\]
\[
x - x_L \geq 0
\]

where the first `meq` constraints are equalities. Translate back with (fixed variables do not change their values):

\[
x(\sim x_L=x_U) = (x-x_L) + x_L(\sim x_L=x_U)
\]

If `TransfType = 2` the program is transformed into the form

\[
\min_x \quad f(x)
\]
\[
s/t \quad AA(x) \leq bb
\]
\[
x_L \leq x \leq x_U
\]

where the first `meq` constraints are equalities.
If TransType = 3 the program is transformed into the form

\[
\min_x f(x) \quad \text{s/t} \quad AAx \leq bb \\
\quad x \geq x_L
\]

where the first meq constraints are equalities.

### 13.2.3 LineSearch

**Purpose**

LineSearch solves line search problems of the form

\[
\min_{0<\alpha_{\text{min}} \leq \alpha \leq \alpha_{\text{max}}} f(x^{(k)} + \alpha p)
\]

where \( x, p \in \mathbb{R}^n \).

**Calling Syntax**

\[
\text{Result} = \text{LineSearch}(f, g, x, p, f0, g0, \text{LineParam}, \alpha_{\text{max}}, pType, \text{PriLev}, \text{varargin})
\]

**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 \( p \).
- **f0**: Function value at \( \alpha = 0 \).
- **g0**: Gradient at \( \alpha = 0 \), the directed derivative at the present point.
- **LineParam**: Structure with line search parameters; 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. Other parameters directly sent to low level routines.
- **alphaMax**: Maximal value of step length \( \alpha \).
- **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:
  - \( \text{PriLev} > 0 \) Writes a lot of output in LineSearch.
  - \( \text{PriLev} > 3 \) 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 \( \alpha \).
- **f_alpha**: Optimal function value at line search step \( \alpha \).
- **g_alpha**: Optimal gradient value at line search step \( \alpha \).
- **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 [24]. 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 $f_{\text{Low}}$ on $f(\alpha)$. More precisely it is assumed that the user is prepared to accept any value of $f(\alpha)$ for which $f(\alpha) \leq f_{\text{Low}}$. For example in a nonlinear least squares problem $f_{\text{Low}} = 0$ 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 $f \leq f_{\text{Low}}$ is detected or a bracket on an interval of acceptable points is located. The sectioning phase generates a sequence of brackets $[a^{(k)}, b^{(k)}]$ whose lengths tend to zero. Each iteration pick a new point $\alpha^{(k)}$ in $[a^{(k)}, b^{(k)}]$ by minimizing a quadratic or a cubic polynomial which interpolates $f(a^{(k)}), f'(a^{(k)}), f(b^{(k)})$ and $f'(b^{(k)})$ 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**

\[
\text{alfa} = \text{intpol2}(x0, f0, g0, x1, f1, a, b, PriLev)
\]

**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 $x_1$.
- $g1$: 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 second degree polynomial in the interval $[a, b]$.

**Description**

In the line search routine *LineSearch* the problem of choosing $\alpha$ 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**

\[
\text{alfa} = \text{intpol3}(x0, f0, g0, x1, f1, g1, a, b, PriLev)
\]

**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

\textit{alfa} \quad The minimum of the interpolated third degree polynomial in the interval 
\([a, b]\).

Description

In the line search routine \textit{LineSearch} the problem of choosing \(\alpha\) in a given interval 
\([a, b]\) occurs both in the \textit{bracketing phase} and in the \textit{sectioning phase}. If cubic interpolation are to be used \textit{LineSearch} calls \textit{intpol3} which finds the minimum of a third degree polynomial approximation in the given interval.

See Also

\textit{LineSearch, intpol2}

13.2.6 \textbf{preSolve}

Purpose

Simplify the structure of the constraints and the variable bounds in a linear constrained program.

Calling Syntax

\texttt{Prob = preSolve(Prob)}

Description of Inputs

\textit{Prob} \quad Problem description structure. The following fields are used:

- \(A\) \quad Constraint matrix for linear constraints.
- \(b_L\) \quad Lower bounds on the linear constraints.
- \(b_U\) \quad Upper bounds on the linear constraints.
- \(x_L\) \quad Lower bounds on the variables.
- \(x_U\) \quad Upper bounds on the variables.

Description of Outputs

\textit{Prob} \quad Problem description structure. The following fields are changed:

- \(A\) \quad Constraint matrix for linear constraints.
- \(b_L\) \quad Lower bounds on the linear constraints, set to NaN for redundant constraints.
- \(b_U\) \quad Upper bounds on the linear constraints, set to NaN for redundant constraints.
- \(x_L\) \quad Lower bounds on the variables.
- \(x_U\) \quad Upper bounds on the variables.

Description

The routine \textit{preSolve} is an implementation of those presolve analysis techniques described by Gondzio in [37], which is applicable to general linear constrained problems. See [9] for a more detailed presentation.

\textit{preSolve} consists of the two routines \textit{clean} and \textit{mksp}. They are called in the sequence \textit{clean}, \textit{mksp}, \textit{clean}. The second call to \textit{clean} is skipped if the \textit{mksp} routine could not remove a single nonzero entry from \(A\).

\textit{clean} consists of two routines, \textit{r\_rw\_sng} that removes singleton rows and \textit{el\_cnsts} that improves variable bounds and uses them to eliminate redundant and forcing constraints. Both \textit{r\_rw\_sng} and \textit{el\_cnsts} check if empty rows appear and eliminate them if so. That is handled by the routine \textit{emptyrow}. In \textit{clean} the calls to \textit{r\_rw\_sng} and \textit{el\_cnsts} are repeated (in given order) until no further reduction is obtained.

Note that rows are actually not deleted or removed, instead \textit{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 \textbf{PrintResult}

Purpose

Prints the result of an optimization.

Calling Syntax

\texttt{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
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 solveType 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.

See Also
runtest
14 TOMLAB LDO

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 Math Works, 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

\[ \min_x f(x) = c^T x \]

\[ s/t \quad x_L \leq x \leq x_U, \]

\[ b_L \leq Ax \leq b_U \]  

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

The implementation of lpsimp2 is based on the standard revised simplex algorithm as formulated in Goldfarb and Todd [36, 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 [36, 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 [36, chap. 9]. The affine scaling variant of Karmakar’s method (akarmark) is an implementation of the algorithm in Bazaraa [36, 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpsolve2</td>
<td>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.</td>
<td>13.1.9</td>
<td>103</td>
</tr>
<tr>
<td>lpsimp1</td>
<td>The Phase I simplex algorithm. Finds a basic feasible solution (bfs) using artificial variables. Calls lpsimp2.</td>
<td>14.4.11</td>
<td>138</td>
</tr>
<tr>
<td>lpsimp2</td>
<td>The Phase II revised simplex algorithm with three selection rules.</td>
<td>14.4.12</td>
<td>139</td>
</tr>
<tr>
<td>karmark</td>
<td>Karmarkar’s algorithm. Canonical form.</td>
<td>14.4.6</td>
<td>135</td>
</tr>
<tr>
<td>lpkarma</td>
<td>Solves LP on equality form, by converting and calling karmark.</td>
<td>14.4.10</td>
<td>137</td>
</tr>
<tr>
<td>lpdual</td>
<td>The dual simplex algorithm.</td>
<td>14.4.9</td>
<td>137</td>
</tr>
<tr>
<td>akarmark</td>
<td>Affine scaling variant of Karmarkar’s algorithm.</td>
<td>14.4.1</td>
<td>131</td>
</tr>
</tbody>
</table>

14.1.2 Transportation Programming

Transportation problems are solved using an implementation of the transportation simplex method as described in Luenberger [50, chap 5.4] (TPsimplex). 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 [68, chap. 7.2]. The functions are described in Table 18.

Table 18: Routines for transportation programming.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPnw</td>
<td>Find initial bfs to TP using the northwest corner method.</td>
<td>14.5.6</td>
<td>147</td>
</tr>
<tr>
<td>TPmc</td>
<td>Find initial bfs to TP using the minimum cost method.</td>
<td>14.5.5</td>
<td>146</td>
</tr>
<tr>
<td>TPvogel</td>
<td>Find initial bfs to TP using Vogel’s approximation method.</td>
<td>14.5.7</td>
<td>147</td>
</tr>
<tr>
<td>TPsimplex</td>
<td>Implementation of the transportation simplex algorithm.</td>
<td>14.4.19</td>
<td>143</td>
</tr>
</tbody>
</table>

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 (\textit{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 (\textit{maxflow}). The implementation is based on the algorithm description in Luenberger [50, pages 144-145].

• Solve the minimum cost network flow problem (MCNFP) using a network simplex algorithm (\textit{NWsimp}). 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 (\textit{salesman}), an algorithm by Held and Karp [38].

The network programming routines are listed in Table 19.

Table 19: Routines for network programs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{gsearch}</td>
<td>Searching all reachable nodes in a network. Stack approach.</td>
<td>14.5.2</td>
<td>145</td>
</tr>
<tr>
<td>\textit{gsearchq}</td>
<td>Searching all reachable nodes in a network. Queue approach.</td>
<td>14.5.3</td>
<td>145</td>
</tr>
<tr>
<td>\textit{mintree}</td>
<td>Finds the minimum spanning tree of an undirected graph.</td>
<td>14.5.4</td>
<td>146</td>
</tr>
<tr>
<td>\textit{dijkstra}</td>
<td>Shortest path using Dijkstra’s algorithm.</td>
<td>14.4.3</td>
<td>133</td>
</tr>
<tr>
<td>\textit{labelcor}</td>
<td>Shortest path using a label correcting algorithm.</td>
<td>14.4.8</td>
<td>136</td>
</tr>
<tr>
<td>\textit{modlabel}</td>
<td>Shortest path using a modified label correcting algorithm.</td>
<td>14.4.14</td>
<td>140</td>
</tr>
<tr>
<td>\textit{maxflow}</td>
<td>Solving maximum flow problems using the Ford-Fulkerson augmenting path method.</td>
<td>14.4.13</td>
<td>139</td>
</tr>
<tr>
<td>\textit{salesman}</td>
<td>Symmetric traveling salesman problem (TSP) solver using Lagrangian relaxation and the subgradient method with the Polyak rule II.</td>
<td>14.4.18</td>
<td>142</td>
</tr>
<tr>
<td>\textit{NWsimp}</td>
<td>Solving minimum cost network flow problems (MCNFP) with a network simplex algorithm.</td>
<td>14.4.15</td>
<td>140</td>
</tr>
</tbody>
</table>

14.1.4 Mixed-Integer Programming

To solve mixed linear inequality integer programs two algorithms are implemented as part of TOMLAB, \textit{mipSolve} and \textit{cutplane}. Described in the Network Programming section 14.1.3 is the \textit{salesman} routine, which is an special type of integer programming problem. The directory tsplib contains test problems for the travelling salesman problem. The routine \textit{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 \textit{balas} [39].

14.1.5 Dynamic Programming

Two simple examples of dynamic programming are included. Both examples are from Winston [68, chap. 20]. Forward recursion is used to solve an inventory problem (\textit{dpinvent}) and a knapsack problem (\textit{dpknap}), see Table 20.

Table 20: Routines for dynamic programming.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{dpinvent}</td>
<td>Forward recursion DP algorithm for the inventory problem.</td>
<td>14.4.4</td>
<td>133</td>
</tr>
<tr>
<td>\textit{dpknap}</td>
<td>Forward recursion DP algorithm for the knapsack problem.</td>
<td>14.4.5</td>
<td>134</td>
</tr>
</tbody>
</table>
14.1.6 Quadratic Programming

Two simple routines for quadratic programming are included,

Table 21: Routines for quadratic programming.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>qpe</td>
<td>Solves a $qp$ problem, restricted to equality constraints, using a null</td>
<td>14.4.17</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>space method.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>qplm</td>
<td>Solves a $qp$ problem, restricted to equality constraints, using Lagrange's</td>
<td>14.4.16</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>method.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 [22], 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ksrelax$</td>
<td>Lagrangian relaxation with knapsack subproblems.</td>
<td>14.4.7</td>
<td>135</td>
</tr>
<tr>
<td>$urelax$</td>
<td>Lagrangian relaxation with knapsack subproblems, plot result.</td>
<td>14.4.20</td>
<td>144</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a2frstar$</td>
<td>Convert node-arc $A$ matrix to Forward-Reverse Star Representation.</td>
</tr>
<tr>
<td>$z2frstar$</td>
<td>Convert matrix of arcs (and costs) to Forward-Reverse Star.</td>
</tr>
<tr>
<td>$cpTransf$</td>
<td>Transform general convex programs to other forms.</td>
</tr>
<tr>
<td>$optParamDef$</td>
<td>Define optimization parameters in the TOMLAB format ($optParam$).</td>
</tr>
<tr>
<td>$lpDef$</td>
<td>Define optimization parameters in the Optimization Toolbox 1.x format ($optPar$).</td>
</tr>
<tr>
<td>$mPrint$</td>
<td>Print matrix, format: NAME(i,:) $a(i,1)a(i,2)...a(i,n)$.</td>
</tr>
<tr>
<td>$printmat$</td>
<td>Print matrix with row and column labels.</td>
</tr>
<tr>
<td>$vPrint$</td>
<td>Print vector in rows, format: NAME($i_1:i_n$) $v_{i_1}v_{i_2}...v_{i_n}$.</td>
</tr>
<tr>
<td>$xPrint$</td>
<td>Print vector $x$, row by row, with format.</td>
</tr>
<tr>
<td>$xPrinti$</td>
<td>Print integer vector $x$. Calls $xprint$.</td>
</tr>
<tr>
<td>$xPrinte$</td>
<td>Print integer vector $x$ in exponential format. Calls $xprint$.</td>
</tr>
</tbody>
</table>
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 \texttt{lpsimp1} and \texttt{lpsimp2}:

\[
A = \begin{bmatrix} 1 & 2 \\ 4 & 1 \end{bmatrix}; \\
b = \begin{bmatrix} 6 & 12 \end{bmatrix}; \\
c = \begin{bmatrix} -7 & -5 \end{bmatrix}; \\
meq = 0; \\
\text{optPar} = \text{lpDef}; \\
\text{optPar}(13) = \text{meq}; \\
[x_0, B_0, \text{optPar}, y] = \text{lpsimp1}(A, b, \text{optPar}); \\
[x, B, \text{optPar}, y] = \text{lpsimp2}(A, b, c, \text{optPar}, x_0, B_0);
\]

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{exinled.m}</td>
<td>First simple LP example from a course in Operations Research.</td>
</tr>
<tr>
<td>\textit{excycle}</td>
<td>Menu with cycling examples.</td>
</tr>
<tr>
<td>\textit{excycle1}</td>
<td>The Marshall-Surbarle cycling example. Run both the Bland’s cycle preventing rule and the default minimum reduced cost rule and compare results.</td>
</tr>
<tr>
<td>\textit{excycle2}</td>
<td>The Kuhn cycling example.</td>
</tr>
<tr>
<td>\textit{excycle3}</td>
<td>The Beale cycling example.</td>
</tr>
<tr>
<td>\textit{exKleeM}</td>
<td>The Klee-Minty example. Shows that the simplex algorithm with Dantzig’s rule visits all vertices.</td>
</tr>
<tr>
<td>\textit{exfl821}</td>
<td>Run exercise 8.21 from Fletcher, Practical methods of Optimization. Illustrates redundancy in constraints.</td>
</tr>
<tr>
<td>\textit{ex12b4s}</td>
<td>Wayne Winston example 4.12 B4, using \texttt{lpsimp1} and \texttt{lpsimp2}.</td>
</tr>
<tr>
<td>\textit{expertur}</td>
<td>Perturbed both right hand side and objective function for Luenberger 3.12-10,11.</td>
</tr>
<tr>
<td>\textit{ex6rev17}</td>
<td>Wayne Winston chapter 6 Review 17. Simple example of calling the dual simplex solver \texttt{lpdual}.</td>
</tr>
<tr>
<td>\textit{ex611a2}</td>
<td>Wayne Winston example 6.11 A2. A simple problem solved with the dual simplex solver \texttt{lpdual}.</td>
</tr>
</tbody>
</table>

Some test examples are collected in the file \textit{demoLP} and further described in Table 25.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{exwaw597}</td>
<td>Test of \texttt{karmark} and \texttt{lpsimp2} on Winston example page 597 and Winston 10.6 Problem A1.</td>
</tr>
<tr>
<td>\textit{exstrang}</td>
<td>Test of \texttt{karmark} and \texttt{lpsimp2} on Strangs’ \textit{nutshell} example.</td>
</tr>
<tr>
<td>\textit{exkarma}</td>
<td>Test of \texttt{akarmark}.</td>
</tr>
<tr>
<td>\textit{exKleeM2}</td>
<td>Klee-Minty example solved with \texttt{lpkarma} and \texttt{karmark}.</td>
</tr>
</tbody>
</table>
14.2.2 How to Solve Transportation Programming Problems

The following is a simple example of a transportation problem

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

where \( s \) is the supply vector, \( d \) is the demand vector and \( C \) is the cost matrix. See \textit{TPsimplex} Section 14.4.19. Solving (19) by use of the routine \textit{TPsimplex} is done by:

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

\[ [X, B, \text{optPar}, y, C] = \text{TPsimplex}(s, d, C) \]

When neither starting base nor starting point is given as input argument \textit{TPsimplex} calls \textit{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 (\textit{TPnw}) before the call to \textit{TPsimplex}:

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

\[ [X_0, B_0] = \text{TPnw}(s, d) \\
[X, B, \text{optPar}, y, C] = \text{TPsimplex}(s, d, C, X_0, B_0) \]

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{extp}</td>
<td>Test of the three routines that finds initial basic feasible solution to a TP problem, routines \textit{TPnw}, \textit{TPmc} and \textit{TPvogel}.</td>
</tr>
<tr>
<td>\textit{exlu119}</td>
<td>Luenberger TP page 119. Find initial basis with \textit{TPnw}, \textit{TPmc} and \textit{TPvogel} and run \textit{TPsimplex} for each.</td>
</tr>
<tr>
<td>\textit{exlu119U}</td>
<td>Test unbalanced TP on Luenberger TP page 119, slightly modified. Runs \textit{TPsimplex}.</td>
</tr>
<tr>
<td>\textit{extp}</td>
<td>Runs simple TP example. Find initial basic feasible solution and solve with \textit{TPsimplex}.</td>
</tr>
</tbody>
</table>

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

\[
A = \begin{pmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 1 & 0 & -1 & 0 \\
0 & 0 & -1 & 0 & 0 & 1 & 1 & -1 \\
0 & 0 & 0 & 0 & -1 & -1 & 0 & 1 \\
\end{pmatrix},
\]
\[
c = \begin{pmatrix}
2 \\
3 \\
1 \\
4 \\
2 \\
4 \\
1 \\
3 \\
\end{pmatrix}
\] (20)

See \textit{a2frstar} Section 14.5.1 for an explanation of the notation.

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

\[
A = \begin{pmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]
\[ C = [2 \ 3 \ 1 \ 4 \ 2 \ 4 \ 1 \ 3]; \]

\[ [P \ Z \ c \ T \ R \ x \ U] = a2frstar(A, C); \]

\[ [\text{pred dist}] = \text{modlabel}(1, P, Z, c); \]

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

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

### 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>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expkorv</td>
<td>Test of cutplane and mipSolve for example PKorv.</td>
</tr>
<tr>
<td>exIP39</td>
<td>Test example I39.</td>
</tr>
<tr>
<td>exbalas</td>
<td>Test of 0/1 IP (Balas algorithm) on simple example.</td>
</tr>
</tbody>
</table>

### 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 (22) is an example from Holmberg [39] and the inventory problem is an example from Winston [68, page 1013].

\[
\begin{align*}
\max_{u} \quad & f(u) = 7u_1 + u_2 + 4u_3 \\
\text{s.t.} \quad & 2u_1 + 3u_2 + 2u_3 \leq 4 \\
& 0 \leq u_1 \leq 1 \\
& 0 \leq u_2 \leq 1 \\
& 0 \leq u_3 \leq 2 \\
& u_j \in \mathbb{N}, j = 1, 2, 3
\end{align*}
\]

Problem (22) will be solved by the following definitions and call:
\[ A = \begin{bmatrix} 2 & 3 & 2 \end{bmatrix}; \]
\[ b = 4; \]
\[ c = \begin{bmatrix} 7 & 2 & 4 \end{bmatrix}; \]
\[ u_{\text{UPP}} = \begin{bmatrix} 1 & 1 & 2 \end{bmatrix}; \]
\[ \text{PriLev} = 0; \]
\[ [u, f_{\text{opt}}] = \text{dpknap}(A, b, c, u_{\text{UPP}}, \text{PriLev}); \]

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:

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exinvent</td>
<td>Test of dpinvent on two inventory examples.</td>
</tr>
<tr>
<td>exknap</td>
<td>Test of dpknap (calls mipSolve and cutplane) on five knapsack examples.</td>
</tr>
</tbody>
</table>

14.2.6 How to Solve Lagrangian Relaxation Problems

This section shows an example of using Lagrangian relaxation techniques implemented in the routine ksrelax to solve an integer programming problem. The problem to be solved, (23), is an example from Fischer [22].

\[
\begin{align*}
\max_{x} & \quad f(x) = 16x_1 + 10x_2 + 4x_4 \\
\text{s.t} & \quad 8x_1 + 2x_2 + x_3 + x_4 \leq 10 \\
& \quad x_1 + x_2 \leq 1 \\
& \quad x_3 + x_4 \leq 1 \\
& \quad x_j \in 0/1, j = 1, 2, 3, 4
\end{align*}
\]
\[ A = \begin{bmatrix} 8 & 2 & 1 & 4 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}; \]
\[ b = \begin{bmatrix} 10 & 1 \end{bmatrix}'; \]
\[ c = \begin{bmatrix} 16 & 10 & 0 & 4 \end{bmatrix}'; \]
\[ r = 1; \quad \text{% Do not relax the first constraint} \]
\[ x_{\text{UPP}} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}'; \]

\[
[x \ u \ f_{\text{opt}} \ \text{optPar}] = \text{ksrelax}(A, b, c, r, x_{\text{UPP}});
\]

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exrelax</td>
<td>Test of \text{ksrelax} on LP example from Fischer -85.</td>
</tr>
<tr>
<td>exrelax2</td>
<td>Simple example, runs \text{ksrelax}.</td>
</tr>
<tr>
<td>exIP39rx</td>
<td>Test example I39, relaxed. Calls \text{urelax} and plot.</td>
</tr>
</tbody>
</table>
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 \textit{PriLev}.

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

There is a wait flag in \texttt{optPar}, \texttt{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 \textit{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, \texttt{mPrint} and \texttt{printmat}. The routine \texttt{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, \texttt{printmat} will first display \texttt{A(:, 1:10)}, then \texttt{A(:, 11:20)} and so on.

The routine \texttt{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 \texttt{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 \texttt{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 LDO will be described.

14.4.1 \texttt{akarmark}

\textbf{Purpose}

Solve linear programming problems of the form

$$
\begin{align*}
\min_x & \quad f(x) = c^T x \\
\text{s/t} & \quad Ax = b \\
& \quad x \geq 0
\end{align*}
$$

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

131
Calling Syntax
\[ [x, \text{optPar}, y, x_0] = \text{akarmark}(A, b, c, \text{optPar}, x_0) \]

**Description of Inputs**
- \(A\) Constraint matrix.
- \(b\) Right hand side vector.
- \(c\) Cost vector.
- \(\text{optPar}\) Optimization parameter vector, see \textit{goptions.m}.
- \(x_0\) Starting point.

**Description of Outputs**
- \(x\) Optimal point.
- \(\text{optPar}\) Optimization parameter vector, see \textit{goptions.m}.
- \(y\) Dual parameters.
- \(x_0\) Starting point used.

**Description**
The routine \textit{akarmark} is an implementation of the affine scaling variant of Karmarkar’s method as described in Bazaraa [36, pages 411-413]. As the purification algorithm a modified version of the algorithm on page 385 in Bazaraa is used.

**Examples**
See \textit{exakarma}, \textit{exkarma}, \textit{exkleem2}.

**M-files Used**
\textit{lpDef.m}

**See Also**
\textit{lpkarma}, \textit{karmark}

14.4.2 \textbf{balas}

**Purpose**
Solve binary integer linear programming problems.

\textit{balas} solves problems of the form

\[
\begin{align*}
\min_x f(x) & = c^T x \\
\text{s/t} & \quad a_i^T x = b_i \quad i = 1, 2, \ldots, m_{eq} \\
& \quad a_i^T x \leq b_i \quad i = m_{eq} + 1, \ldots, m \\
& \quad x_j \in 0/1 \quad j = 1, 2, \ldots, n
\end{align*}
\]

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

**Calling Syntax**
\[ [x, \text{optPar}] = \text{balas}(A, b, c, \text{optPar}) \]

**Description of Inputs**
- \(A\) Constraint matrix, integer coefficients.
- \(b\) Right hand side vector, integer coefficients.
- \(c\) Cost vector, integer coefficients.
- \(\text{optPar}\) Optimization parameter vector, see \textit{goptions.m}.

**Description of Outputs**
- \(x\) Optimal point.
- \(\text{optPar}\) Optimization parameter vector, see \textit{goptions.m}.

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

**Examples**
See \textit{exbalas}.

**M-files Used**
\textit{lpDef.m}
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. Dijkstra’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

\[
\min_u f(u) = P_s(t) + P(t)^T u(t) + I(t)^T x(t)
\]

\[
s/t \quad u_L \leq u(t) \leq u_U
\]

\[
x_L \leq x(t) \leq x_U
\]

\[
0 \leq u(t) \leq x(t) + d(t)
\]

\[
u_j \in \mathbb{N} \quad j = 1, 2, ..., n
\]

\[
x_j \in \mathbb{N} \quad j = 1, 2, ..., n
\]

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

Calling Syntax
[u, Lopt, 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 [68, chap. 20].

Examples

See exinvent.

14.4.5 dpknap

Purpose

Solve knapsack problems of the form

\[
\max_{u} \quad f(u) = c^T u \\
\text{s.t} \quad Au \leq b \\
\quad u \leq u_U \\
\quad u_j \in \mathbb{N} \quad j = 1, 2, ..., n
\]

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 [68, 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

\[
\begin{align*}
\min_x \quad f(x) &= c^T x \\
\text{s.t} \quad Ax &= 0 \\
\sum_{j=1}^{n} x_j &= 1 \\
x &\geq 0
\end{align*}
\]

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
[\( x, \text{optPar} \)] = karmark(\( A, c, \text{optPar} \))

Description of Inputs
\( A \) Constraint matrix.
\( c \) Cost vector.
\( \text{optPar} \) Optimization parameter vector, see goptions.m.

Description of Outputs
\( x \) Optimal point.
\( \text{optPar} \) Optimization parameter vector, see goptions.m.

Description
The routine \texttt{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 [36, chap. 9]. As the purification algorithm a modified version of the algorithm on page 385 in Bazaraa is used. \texttt{karmark} is called by \texttt{lpkarma} which transforms linear maximization problems on inequality form into Karmakar’s form needed for \texttt{karmark}.

Examples
See exstrang, exww597.

M-files Used
\texttt{lpDef.m}

See Also
\texttt{lpkarma}, \texttt{akarmark}

14.4.7 ksrelax

Purpose
Solve integer linear problems of the form

\[
\begin{align*}
\max_x \quad f(x) &= c^T x \\
\text{s.t} \quad Ax &\leq b \\
x &\leq x_U \\
x_j &\in \mathbb{N} \quad j = 1, 2, ..., n
\end{align*}
\]

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

Calling Syntax
[\( x, P, u, LP, \text{optPar} \)] = ksrelax(\( A, b, c, r, x_U, \text{optPar} \))
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 \texttt{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 \cite{Fisher}, using subgradient iterations and a simple line search rule. LDO also contains a routine \texttt{urelax} which plots the result of each iteration.

Examples
See \texttt{exrelax}, \texttt{exrelax2}.

M-files Used
lpDef.m, dpknaps.m

See Also
\texttt{urelax}

14.4.8 labelcor

Purpose
Solve the shortest path problem.

Calling Syntax
\[ \pred, \dist = \text{labelcor}(s, P, Z, c) \]

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 \texttt{labelcor} is based on the algorithm \texttt{LABEL CORRECTING} in \cite[page 260]{Fisher} for solving shortest path problems. The algorithm belongs to the class of label correcting methods which are applicable to networks with arbitrary arc lengths. \texttt{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 \texttt{exgraph}.

See Also
\texttt{dijkstra}, \texttt{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
\[
\begin{align*}
\min_x f_P(x) & = c^T x \\
\text{s.t.} & \quad a_i^T x = b_i \quad i = 1, 2, ..., m_{eq} \\
& \quad a_i^T x \leq b_i \quad i = m_{eq} + 1, ..., m \\
& \quad x \geq 0
\end{align*}
\]

by rewriting it into standard form and solving the dual problem
\[
\begin{align*}
\max_y f_D(y) & = b^T y \\
\text{s.t.} & \quad A^T y \leq c \\
& \quad y \text{ urs}
\end{align*}
\]

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 [36, 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
\[
\begin{align*}
\max_x f(x) & = c^T x \\
\text{s.t.} & \quad Ax \leq b \\
& \quad x \geq 0
\end{align*}
\]
where \( x, c \in \mathbb{R}^n, A \in \mathbb{R}^{m \times n} \) and \( b \in \mathbb{R}^m \).

**Calling Syntax**

\[
[x, y, \text{optPar}] = lpkarma(A, b, c, \text{optPar})
\]

**Description of Inputs**
- \( A \): Constraint matrix.
- \( b \): Right hand side vector.
- \( c \): Cost vector.
- \( \text{optPar} \): Optimization parameter vector, see \textit{goptions.m}.

**Description of Outputs**
- \( x \): Optimal point.
- \( y \): Dual solution.
- \( \text{optPar} \): Optimization parameter vector, see \textit{goptions.m}.

**Description**

\textit{lpkarma} converts a linear maximization problem on inequality form into Karmakar's form and calls \textit{karmark} to solve the transformed problem.

**Examples**
See \textit{exstrang}, \textit{exw597}.

**M-files Used**
\textit{lpDef.m, karmark.m}

**See Also**
\textit{karmark, akarmark}

### 14.4.11 \texttt{lpsimp1}

**Purpose**

Find a basic feasible solution to linear programming problems.

\textit{lpsimp1} finds a basic feasible solution to problems of the form

\[
\min_x \quad f(x) = c^T x \\
\text{s.t.} \quad a_i^T x = b_i \quad i = 1, 2, \ldots, m_{eq} \\
\quad a_i^T x \leq b_i \quad i = m_{eq} + 1, \ldots, m \\
x \geq 0
\]

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**

\[
[x, B, \text{optPar}, y] = lpsimp1(A, b, \text{optPar})
\]

**Description of Inputs**
- \( A \): Constraint matrix.
- \( b \): Right hand side vector.
- \( \text{optPar} \): Optimization parameter vector, see \textit{goptions.m}.

**Description of Outputs**
- \( x \): Basic feasible solution.
- \( B \): Basic set at the solution \( x \).
- \( \text{optPar} \): Optimization parameter vector, see \textit{goptions.m}.
- \( y \): Lagrange multipliers.

**Description**

The routine \textit{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 \textit{lpsimp2} to solve the Phase I problem.

**Examples**
See \textit{exinled, excycle, excycle2, exKleeM, exfl821, ex412b4s}.
14.4.12 lpsimp2

Purpose
Solve linear programming problems.

\[ \begin{align*}
\min_{x} & \quad f(x) = c^T x \\
\text{s.t.} & \quad a_i^T x = b_i, \quad i = 1, 2, \ldots, m_{eq} \\
& \quad a_i^T x \leq b_i, \quad i = m_{eq} + 1, \ldots, m \\
& \quad x \geq 0
\end{align*} \]

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

Calling Syntax
\[ [x, B, optPar, y] = \text{lpsimp2}(A, b, c, optPar, x_0, B_0) \]

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 \text{lpsimp2} implements the Phase II standard revised Simplex algorithm as formulated in Goldfarb and Todd [36, 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
\text{lpDef.m}

See Also
\text{lpsimp1}, \text{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
\[ [\text{max}_f \text{low}, x] = \text{maxflow}(s, t, x_U, P, Z, T, R, \text{PriLev}) \]
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.
  (:, 1) Tail. (:, 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 [50, 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.
  (:, 1) Tail. (:, 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.

\[ \min_{x} \frac{1}{2}(x)^{T}Fx + c^{T}x \]

\[ \text{s/t} \quad Ax = b \]

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.

\[ \text{qpe solves problems of the form} \]
\[ \min_x f(x) = \frac{1}{2}(x)^T F x + c^T x \]
\[ \text{s/t} \quad A x = b \]

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
\([x, \lambda, QZ, RZ] = \text{qpe}(F, c, A, b)\)

Description of Inputs
- \( F \): Constant matrix, the Hessian.
- \( c \): 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
\( \text{qpBiggs, qpSolve, qplm} \)

14.4.18 salesman

Purpose
Solve the symmetric travelling salesman problem.

Calling Syntax
\([\text{Tour}, f_{\text{tour}}, \text{OneTree}, f_{\text{tree}}, w_{\text{max}}, my_{\text{max}}, \text{optPar}] = \text{salesman}(C, Zin, Zout, my, f_{\text{BestTour}}, \text{optPar})\)

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.
- \( f_{\text{BestTour}} \): Cost (total distance) of a known tour.
- \( \text{optPar} \): Optimization parameter vector, see goptions.m.

Description of Outputs
- \( \text{Tour} \): Arc list of the best tour found.
- \( f_{\text{tour}} \): Cost (total distance) of the best tour found.
- \( \text{OneTree} \): Arc list of the best 1-tree found.
- \( f_{\text{tree}} \): Cost of the best 1-tree found.
- \( w_{\text{max}} \): Best dual objective.
- \( my_{\text{max}} \): Lagrange multipliers at \( w_{\text{max}} \).
- \( \text{optPar} \): Optimization parameter vector, see goptions.m.

Description
The routine salesman is an implementation of an algorithm by Held and Karp [38] 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 \textit{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, \textit{salesman} could be used as a subroutine in a Branch and Bound algorithm, see \textit{travelng} which calls \textit{salesman}.

\textbf{Examples}
See \textit{ulyss16}.

\textbf{M-files Used}
lpDef.m, mintree.m

\textbf{See Also}
travelng

14.4.19 \textbf{TPsimp1x}

\textbf{Purpose}
Solve transportation programming problems.

\textit{TPsimp1x} solves problems of the form

$$
\min_x \ f(x) = \sum_{i}^{m} \sum_{j}^{n} c_{ij} x_{ij} \\
\text{s/t } \sum_{j}^{n} x_{ij} = s_i, \quad i = 1, 2, ..., m \\
\sum_{i}^{m} x_{ij} = d_j, \quad j = 1, 2, ..., n \\
x \geq 0
$$

where \( x, c \in \mathbb{R}^{m \times n} \), \( s \in \mathbb{R}^{m} \) and \( d \in \mathbb{R}^{n} \).

\textbf{Calling Syntax}

\texttt{[X, B, optPar, y, C] = TPsimp1x(s, d, C, X, B, optPar, Penalty)}

\textbf{Description of Inputs}

\begin{itemize}
\item \( s \) Supply vector.
\item \( d \) Demand vector.
\item \( C \) The cost matrix of linear objective function coefficients.
\item \( X \) Basic Feasible Solution matrix.
\item \( B \) Index \((i, j)\) of basis found.
\item \( optPar \) Optimization parameter vector, see \textit{goptions.m}.
\item \( Penalty \) 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 \( 10 \max(C_{ij}) \).
\end{itemize}

\textbf{Description of Outputs}

\begin{itemize}
\item \( X \) Solution matrix.
\item \( B \) Optimal set. Index \((i, j)\) of the optimal basis found.
\item \( optPar \) Optimization parameter vector, see \textit{goptions.m}.
\item \( y \) Lagrange multipliers.
\item \( C \) The cost matrix, changed if the problem is unbalanced.
\end{itemize}

\textbf{Description}
The routine \textit{TPsimp1x} is an implementation of the Transportation Simplex method described in Luenberger [50, chap 5.4]. In LDO, three routines to find a starting basic feasible solution for the transportation problem are included; the Northwest Corner method (\textit{TPnw}), the Minimum Cost method (\textit{TPmc}) and Vogel’s approximation method (\textit{TPvogel}). If calling \textit{TPsimp1x} without giving a starting point then Vogel’s method is used to find a starting basic feasible solution.

\textbf{Examples}
See \textit{extp.bfs}, \textit{exlua119}, \textit{exlua119U}, \textit{extp}.
14.4.20 urelax

Purpose
Solve integer linear problems of the form

\[
\begin{align*}
\max & \quad f(x) = c^T x \\
\text{s/t} & \quad Ax \leq b \\
& \quad x \leq x_U \\
& \quad x_j \in \mathbb{N} \quad j = 1, 2, \ldots, n
\end{align*}
\]

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

Calling Syntax

\[
[x_P, u, f_P] = urelax(u_{\text{max}}, A, b, c, r, x_{\text{U}}, \text{optPar})
\]

Description of Inputs

- \( u_{\text{max}} \): Upper bounds on \( u \).
- \( A \): Constraint matrix.
- \( b \): Right hand side vector.
- \( c \): Cost vector.
- \( r \): Constraint not to be relaxed.
- \( x_{\text{U}} \): Upper bounds on the variables.
- \( \text{optPar} \): Optimization parameter vector, see \( \text{goptions.m} \).

Description of Outputs

- \( x_P \): Primal solution.
- \( u \): Lagrangian multipliers.
- \( f_P \): Function value at \( x_P \).

Description

The routine \( \text{urelax} \) 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. \( \text{urelax} \) plots the result of each iteration. LDO also contains a more sophisticated routine, \( \text{ksrelax} \), for solving problems of this type.

Examples

See \( \text{exip39rrx} \).

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
\[ \{P, Z, c, T, R, u\} = \text{a2frstar}(A, C, U) \]

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.
- \( R \): Reverse pointer vector in \( T \) for each node.
- \( u \): Upper bounds on flow if \( U \) is given as input, else infinity.

Description
The routine \( \text{a2frstar} \) 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 \( \text{exflow} \), \( \text{exflow31} \), \( \text{exgraph} \), \( \text{pathflow} \).

14.5.2 \textbf{gsearch}

Purpose
Find all nodes in a network which is reachable from a given source node.

Calling Syntax
\[ \{\text{pred}, \text{mark}\} = \text{gsearch}(s, P, Z, c) \]

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
- \( \text{pred} \): \( \text{pred}(j) \) = Predecessor of node \( j \).
- \( \text{mark} \): If \( \text{mark}(j) = 1 \) the node is reachable from node \( s \).

Description
\( \text{gsearch} \) 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 \( \text{SEARCH} \) 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 \( \text{exgraph} \).

See Also
\( \text{gsearchq} \)

14.5.3 \textbf{gsearchq}

Purpose
Find all nodes in a network which is reachable from a given source node.
Calling Syntax

\[ [\text{pred}, \text{mark}] = \text{gsearchq}(s, P, Z, c) \]

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

\( \text{pred} \) \( \text{pred}(j) = \) Predecessor of node \( j \).
\( \text{mark} \) If \( \text{mark}(j) = 1 \) the node is reachable from node \( s \).

Description

\( \text{gsearchq} \) 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 \( \text{SEARCH} \) 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 \( \text{exgraph} \).

See Also

\( \text{gsearch} \)

14.5.4 \textbf{mintree}

Purpose

Find the minimum spanning tree of an undirected graph.

Calling Syntax

\[ [Z\text{\_tree}, \text{cost}] = \text{mintree}(C, \text{Zin, Zout}) \]

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 \( C_{ii} = NaN \).
\( \text{Zin} \) List of arcs which should be forced to be included in \( Z\text{\_tree} \).
\( \text{Zout} \) List of arcs which should not be allowed to be included in \( Z\text{\_tree} \) (could also be given as \( NaN \) in \( C \)).

Description of Outputs

\( Z\text{\_tree} \) List of arcs in the minimum spanning tree.
\( \text{cost} \) The total cost.

Description

\( \text{mintree} \) 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. \( \text{mintree} \) is called by \( \text{salesman} \).

14.5.5 \textbf{TPmc}

Purpose

Find a basic feasible solution to the Transportation Problem.

Calling Syntax

\[ [X,B] = \text{TPmc}(s, d, C) \]
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 [68, chap. 7.2].

Examples
See extp_bfs, exlu119, exlu119U, extp.

See Also
TPnw, TPvogel, TPsimplex

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 [68, chap. 7.2].

Examples
See extp_bfs, exlu119, exlu119U, extp.

See Also
TPmc, TPvogel, TPsimplex

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.
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 [68, 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.

\( R \)
Reverse 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 Init File in TOMLAB Init File format 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. The structure, named Prob, is shown in Table 31 and Table 32. 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 for user selection of which problem to be solved. The same type of problem structure, Prob, is also being defined in the TOMLAB Quick format.

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.

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 \textit{Prob}, part I. Fields defining sub-structures are defined in Table 33.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>Matrix with linear constraints, one constraint per row (dense or sparse).</td>
</tr>
<tr>
<td>\textit{AutoDiff}</td>
<td>If true, use automatic differentiation.</td>
</tr>
<tr>
<td>( b_L )</td>
<td>Lower bounds on the linear constraints.</td>
</tr>
<tr>
<td>( b_U )</td>
<td>Upper bounds on the linear constraints.</td>
</tr>
<tr>
<td>( c_L )</td>
<td>Lower bounds on the general constraints.</td>
</tr>
<tr>
<td>( c_U )</td>
<td>Upper bounds on the general constraints.</td>
</tr>
<tr>
<td>\textit{Name}</td>
<td>Name of each general constraint.</td>
</tr>
<tr>
<td>\textit{CHECK}</td>
<td>If true, no more check is done by \textit{ProbCheck}. Set to true (=1) after first call to \textit{ProbCheck}.</td>
</tr>
<tr>
<td>\textit{ConsDiff}</td>
<td>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 \textit{csapi}, \textit{csaps} or \textit{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.</td>
</tr>
<tr>
<td>\textit{ConsPattern}</td>
<td>Matrix with non-zero pattern in the constraint gradient matrix.</td>
</tr>
<tr>
<td>( f_{\text{Low}} )</td>
<td>Lower bound on optimal function value.</td>
</tr>
<tr>
<td>( f_{\text{opt}} )</td>
<td>Objective function value ( f(x^*) ) corresponding to the points given in ( x_{\text{opt}} ).</td>
</tr>
<tr>
<td>\textit{GradTol}</td>
<td>Size of step length to estimate first order derivatives in the gradient vector. If this field is empty, \textit{optParam.DiffInt} is used instead.</td>
</tr>
<tr>
<td>\textit{GradTolH}</td>
<td>Size of step length to estimate the Hessian matrix. If this field is empty, \textit{optParam.DiffInt} is used instead.</td>
</tr>
<tr>
<td>\textit{GradTolJ}</td>
<td>Size of step length to estimate the Jacobian matrix or the constraint gradient matrix. If this field is empty, \textit{optParam.DiffInt} is used instead.</td>
</tr>
<tr>
<td>\textit{HessPattern}</td>
<td>Matrix with non-zero pattern in the Hessian matrix.</td>
</tr>
<tr>
<td>\textit{JacPattern}</td>
<td>Matrix with non-zero pattern in the Jacobian matrix.</td>
</tr>
<tr>
<td>\textit{LargeScale}</td>
<td>Flag if the problem is large scale. If this flag is set no collection of search steps are made.</td>
</tr>
<tr>
<td>\textit{MENU}</td>
<td>Flag used to tell if a menu, the GUI or a driver is calling, to avoid unnecessary checks of the fields in \textit{Prob}(0).</td>
</tr>
<tr>
<td>\textit{Mode}</td>
<td>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.</td>
</tr>
<tr>
<td>\textit{nState}</td>
<td>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 + \textit{Inform} = ) Last call, see the \textit{Inform} parameter for the solver used.</td>
</tr>
<tr>
<td>( N )</td>
<td>Problem dimension (number of variables).</td>
</tr>
<tr>
<td>\textit{Name}</td>
<td>Problem name.</td>
</tr>
<tr>
<td>\textit{NumDiff}</td>
<td>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 \textit{csapi}, \textit{csaps} or \textit{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.</td>
</tr>
</tbody>
</table>
### Table 32: Information stored in the problem structure `Prob`, part II. Fields defining sub-structures are defined in Table 33.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>P</code></td>
<td>Problem number (1).</td>
</tr>
<tr>
<td><code>plotLine</code></td>
<td>Flag if to do a plot of the line search problem.</td>
</tr>
<tr>
<td><code>PriLev</code></td>
<td>Print level in the driver routines (0).</td>
</tr>
<tr>
<td><code>PriLevOpt</code></td>
<td>Print level in the TOM solver and the Matlab part of the SOL solver interface (0).</td>
</tr>
<tr>
<td><code>probFile</code></td>
<td>Name of m-file in which the problems are defined.</td>
</tr>
<tr>
<td><code>probType</code></td>
<td>TOMLAB problem type, see Table 1.</td>
</tr>
<tr>
<td><code>SolverDLP</code></td>
<td>Name of the solver that should solve dual LP sub-problems. Used by SolveDLP.</td>
</tr>
<tr>
<td><code>SolverLP</code></td>
<td>Name of the Solver that should solve LP sub-problems. Used by SolveLP.</td>
</tr>
<tr>
<td><code>SolverFP</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>SolverQP</code></td>
<td>Name of the solver that should solve QP sub problems. Used by SolveQP.</td>
</tr>
<tr>
<td><code>uP</code></td>
<td>User supplied parameters for the problem.</td>
</tr>
<tr>
<td><code>uPName</code></td>
<td>Problem name (<code>Prob.Name</code>) connected to the user supplied parameters in (<code>uP</code>).</td>
</tr>
<tr>
<td><code>WarmStart</code></td>
<td>Flag. If &gt; 0 the solver should do a warm start, if the solver supports a warm start.</td>
</tr>
<tr>
<td><code>x_0</code></td>
<td>Starting point.</td>
</tr>
<tr>
<td><code>x_L</code></td>
<td>Lower bounds on the variables <code>x</code>.</td>
</tr>
<tr>
<td><code>x_min</code></td>
<td>Lower bounds on plot region.</td>
</tr>
<tr>
<td><code>x_max</code></td>
<td>Upper bounds on plot region.</td>
</tr>
<tr>
<td><code>x_opt</code></td>
<td>Stationary points <code>x^*</code>, one per row (if known). If an extra column is defined, a zero indicates a minimum point, a one a saddle point, and a two a maximum.</td>
</tr>
<tr>
<td><code>x_U</code></td>
<td>Upper bounds on the variables <code>x</code>.</td>
</tr>
<tr>
<td><code>xName</code></td>
<td>Name of each decision variable in <code>x</code>.</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>QP</code></td>
<td>Structure with special fields for linear and quadratic problems, see Table 34.</td>
</tr>
<tr>
<td><code>LS</code></td>
<td>Structure with special fields for least squares problems, see Table 35.</td>
</tr>
<tr>
<td><code>MIP</code></td>
<td>Structure with special fields for mixed-integer programming, see Table 36.</td>
</tr>
<tr>
<td><code>ExpFit</code></td>
<td>Structure with special fields for exponential fitting problems, see Table 37.</td>
</tr>
<tr>
<td><code>LineParam</code></td>
<td>Structure with special fields for line search optimization parameters, see Table 38.</td>
</tr>
<tr>
<td><code>optParam</code></td>
<td>Structure with special fields for optimization parameters, see Table 39.</td>
</tr>
<tr>
<td><code>PartSep</code></td>
<td>Structure with special fields for partially separable functions, see Table 40.</td>
</tr>
<tr>
<td><code>SOL</code></td>
<td>Structure with special fields for SOL (Stanford Optimization Laboratory) solvers, see Table 41.</td>
</tr>
<tr>
<td><code>Solver</code></td>
<td>Structure with fields <code>Name</code>, <code>Alg</code> and <code>Method</code>. <code>Name</code> is the name of the solver. <code>Alg</code> is the solver algorithm to be used. <code>Method</code> is the solver sub-method technique. See the solver descriptions Section 13.1.</td>
</tr>
<tr>
<td><code>USER</code></td>
<td>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.</td>
</tr>
</tbody>
</table>
### 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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>Constant matrix $F$ in $\frac{1}{2}x'Fx + c'x$</td>
</tr>
<tr>
<td>$c$</td>
<td>Cost vector $c$ in $\frac{1}{2}x'Fx + c'x$</td>
</tr>
<tr>
<td>$B$</td>
<td>Logical vector of the same length as the number of variables. A zero corresponds to a variable in the basis.</td>
</tr>
<tr>
<td>$y$</td>
<td>Dual parameters $y$.</td>
</tr>
<tr>
<td>$Q$</td>
<td>Orthogonal matrix $Q$ in QR-decomposition.</td>
</tr>
<tr>
<td>$R$</td>
<td>Upper triangular matrix $R$ in QR-decomposition.</td>
</tr>
<tr>
<td>$E$</td>
<td>Pivoting matrix $E$ in QR-decomposition. Stored sparse.</td>
</tr>
<tr>
<td>$DualLimit$</td>
<td>Flag if to scale the $A$ matrix, the linear constraints.</td>
</tr>
<tr>
<td>$UseHot$</td>
<td>True if to use a crash basis (hot start).</td>
</tr>
<tr>
<td>$HotFile$</td>
<td>If nonempty and $UseCrash$ true, read basis from this file.</td>
</tr>
<tr>
<td>$HotFreq$</td>
<td>How often to save a crash basis.</td>
</tr>
<tr>
<td>$HotN$</td>
<td>The number of crash basis files.</td>
</tr>
<tr>
<td>$optParam$</td>
<td>Structure with special fields for optimization parameters, see Table 39.</td>
</tr>
<tr>
<td>$SOL$</td>
<td>Structure with special fields for SOL (Stanford Optimization Laboratory) solvers, see Table 41.</td>
</tr>
<tr>
<td>$Solver$</td>
<td>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 13.1.</td>
</tr>
</tbody>
</table>

### Table 35: Information stored in the structure Prob.LS

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>weightType</td>
<td>Weighting type:</td>
</tr>
<tr>
<td>0</td>
<td>No weighting.</td>
</tr>
<tr>
<td>1</td>
<td>Weight with data in $y$. If $y(t) = 0$, the weighting is 0, i.e. deleting this residual element.</td>
</tr>
<tr>
<td>2</td>
<td>Weight with weight vector or matrix in weightY. If weightY is a vector then weighting by weightY.*r (element wise multiplication). If weightY is a matrix then weighting by weightY * r (matrix multiplication).</td>
</tr>
<tr>
<td>3</td>
<td>nlp.r calls the routine weightY (must be a string with the routine name) to compute the residuals.</td>
</tr>
<tr>
<td>weightY</td>
<td>Either empty, a vector, a matrix or a string, see weightType.</td>
</tr>
<tr>
<td>$t$</td>
<td>Time vector $t$.</td>
</tr>
<tr>
<td>$y$</td>
<td>Vector or matrix with observations $y(t)$.</td>
</tr>
<tr>
<td>$E$</td>
<td>Fixed matrix, in linear least squares problem $E * x - y$.</td>
</tr>
<tr>
<td>$yUse$</td>
<td>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 $f(x,t)$.</td>
</tr>
<tr>
<td>SepAlg</td>
<td>If SepAlg = 1, use separable non linear least squares formulation (default 0).</td>
</tr>
</tbody>
</table>
Table 36: Information stored in the structure `Prob.MIP`

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntVars</td>
<td>Which variables are integer valued</td>
</tr>
<tr>
<td>VarWeight</td>
<td>Priority vector for each variable.</td>
</tr>
<tr>
<td>fIP</td>
<td>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</td>
</tr>
<tr>
<td>xIP</td>
<td>The point x giving the function value fIP.</td>
</tr>
<tr>
<td>PI</td>
<td>See the Tomlab /Xpress User’s Guide.</td>
</tr>
<tr>
<td>SC</td>
<td>See the Tomlab /Xpress User’s Guide.</td>
</tr>
<tr>
<td>SI</td>
<td>See the Tomlab /Xpress User’s Guide.</td>
</tr>
<tr>
<td>SOS1</td>
<td>See the Tomlab /Xpress User’s Guide.</td>
</tr>
<tr>
<td>SOS2</td>
<td>See the Tomlab /Xpress User’s Guide.</td>
</tr>
<tr>
<td>xpc control</td>
<td>See the Tomlab/ Xpress User’s Guide.</td>
</tr>
<tr>
<td>callback</td>
<td>See the Tomlab /Xpress User’s Guide.</td>
</tr>
<tr>
<td>KNAPSACK</td>
<td>See the Tomlab /Xpress User’s Guide.</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>Number of exponential terms (2).</td>
</tr>
<tr>
<td>wType</td>
<td>Weighting type (1).</td>
</tr>
<tr>
<td>eType</td>
<td>Type of exponential terms (1).</td>
</tr>
<tr>
<td>infCR</td>
<td>Information criteria for selection of best number of terms (0).</td>
</tr>
<tr>
<td>dType</td>
<td>Differentiation formula (0).</td>
</tr>
<tr>
<td>geoType</td>
<td>Type of equation (0).</td>
</tr>
<tr>
<td>qType</td>
<td>Length q of partial sums (0).</td>
</tr>
<tr>
<td>sigType</td>
<td>Sign to use in ((P \pm \sqrt{Q})/D) in exp_geo for (p = 3,4) (0).</td>
</tr>
<tr>
<td>lambda</td>
<td>Vector of dimension (p), intensities.</td>
</tr>
<tr>
<td>alpha</td>
<td>Vector of dimension (p), weights.</td>
</tr>
<tr>
<td>beta</td>
<td>Vector of dimension (p), weights in generalized exponential models.</td>
</tr>
<tr>
<td>x0Type</td>
<td>Type of starting value algorithm.</td>
</tr>
<tr>
<td>sumType</td>
<td>Type of exponential sum.</td>
</tr>
</tbody>
</table>

Table 38: Information stored in the structure `Prob.LineParam`

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LineAlg</td>
<td>Line search algorithm. 0 = quadratic interpolation, 1 = cubic interpolation, 2 = curvilinear quadratic interpolation (not robust), 3 = curvilinear cubic interpolation (not robust) (LineAlg = 1).</td>
</tr>
<tr>
<td>sigma</td>
<td>Line search accuracy; (0 &lt; \sigma &lt; 1). (\sigma = 0.9) inaccurate line search. (\sigma = 0.1) accurate line search (0.9).</td>
</tr>
<tr>
<td>InitStepLength</td>
<td>Initial length of step (1.0).</td>
</tr>
<tr>
<td>MaxIter</td>
<td>Maximum number of line search iterations.</td>
</tr>
<tr>
<td>fLowBnd</td>
<td>Lower bound on optimal function value. Used in the line search by Fletcher, m-file LineSearch (= −realmax).</td>
</tr>
<tr>
<td>rho</td>
<td>Determines the (\rho) line (0.01).</td>
</tr>
<tr>
<td>tau1</td>
<td>Determines how fast step grows in phase 1 (9).</td>
</tr>
<tr>
<td>tau2</td>
<td>How near end point of ([a,b]) (0.1).</td>
</tr>
<tr>
<td>tau3</td>
<td>Choice in ([a,b]) phase 2 (0.5).</td>
</tr>
<tr>
<td>eps1</td>
<td>Minimal length for interval ([a,b]) ((10^{-7})).</td>
</tr>
<tr>
<td>eps2</td>
<td>Minimal reduction ((100 \times \epsilon)).</td>
</tr>
</tbody>
</table>
Table 39: Information stored in the structure `Prob.optParam`. Default values in parenthesis. Items marked (SOLn) 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`.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PriLev</td>
<td>Solver major print level in file output (SOL-1).</td>
</tr>
<tr>
<td>PriFreq</td>
<td>Print frequency (SOL-5) in optimization solver.</td>
</tr>
<tr>
<td>SummFreq</td>
<td>Summary frequency (SOL-6) in optimization solver.</td>
</tr>
<tr>
<td>MinorPriLev</td>
<td>Minor print level in file output (SOL-2) in SOL sub-problem solver.</td>
</tr>
<tr>
<td>IterPrint</td>
<td>Flag for one-row-per-iteration printout during optimization (0). If SOL-1</td>
</tr>
<tr>
<td></td>
<td>not explicitly set, setting this flag will set SOL-1 to 1 for the SOL solvers.</td>
</tr>
<tr>
<td>wait</td>
<td>Flag, if true use pause statements after output in each iteration (0).</td>
</tr>
<tr>
<td>MaxFunc</td>
<td>Maximum number of function evaluations.</td>
</tr>
<tr>
<td>MaxIter</td>
<td>Maximum number of iterations (SOL-30).</td>
</tr>
<tr>
<td>MajorIter</td>
<td>Maximum number of iterations in major problem (SOL-35).</td>
</tr>
<tr>
<td>MinorIter</td>
<td>Maximum number of iterations in minor problem (SOL-36).</td>
</tr>
<tr>
<td>eps_f</td>
<td>Relative convergence tolerance in f ($10^{-8}$).</td>
</tr>
<tr>
<td>eps_absf</td>
<td>Absolute convergence tolerance for the function value ($\text{realmax}$).</td>
</tr>
<tr>
<td>eps_x</td>
<td>Relative convergence tolerance in parameter solution $x$ (SOL-10).</td>
</tr>
<tr>
<td>eps_d</td>
<td>Convergence tolerance for the directed derivative ($10^{-8}$).</td>
</tr>
<tr>
<td>eps_c</td>
<td>Feasibility tolerance for nonlinear constraints (SOL-9).</td>
</tr>
<tr>
<td>eps_g</td>
<td>Gradient (or reduced gradient) convergence tolerance ($10^{-7}$).</td>
</tr>
<tr>
<td>eps_Rank</td>
<td>Rank test tolerance (SOL-27).</td>
</tr>
<tr>
<td>EpsGlob</td>
<td>Global/local weight parameter in global optimization ($10^{-4}$).</td>
</tr>
<tr>
<td>fTol</td>
<td>Relative accuracy in the computation of the function value (SOL-41).</td>
</tr>
<tr>
<td>xTol</td>
<td>If $x \in [x_L, x_L + bTol]$ or $[x_U - bTol, x_U]$, fix $x$ on bound ($100 \times \epsilon = 2.2204 \times 10^{-13}$).</td>
</tr>
<tr>
<td>bTol</td>
<td>Feasibility tolerance for linear constraints (SOL-11).</td>
</tr>
<tr>
<td>cTol</td>
<td>Feasibility tolerance for nonlinear constraints (SOL-9).</td>
</tr>
<tr>
<td>MinorTolX</td>
<td>Relative convergence tolerance in parameters $x$ in sub-problem (SOL-12).</td>
</tr>
<tr>
<td>size_x</td>
<td>Size at optimum for the variables $x$, used in the convergence tests (1).</td>
</tr>
<tr>
<td></td>
<td>Only changed if scale very different, $x &gt; 1$.</td>
</tr>
<tr>
<td>size_f</td>
<td>Size at optimum for the function $f$, used in the convergence tests (1).</td>
</tr>
<tr>
<td></td>
<td>Only changed if scale very different, $f &gt; 1$.</td>
</tr>
<tr>
<td>size_c</td>
<td>Size at optimum for the constraints $c$, used in the convergence tests (1).</td>
</tr>
<tr>
<td></td>
<td>Only changed if scale very different, $c &gt; 1$.</td>
</tr>
<tr>
<td>Presolve</td>
<td>Flag if presolve analysis is to be applied on linear constraints (0).</td>
</tr>
<tr>
<td>DerLevel</td>
<td>Derivative Level, knowledge about nonlinear derivatives: 0 = Some compo-</td>
</tr>
<tr>
<td></td>
<td>nents of the objective gradient are unknown and some components of the</td>
</tr>
<tr>
<td></td>
<td>constraint gradient are unknown, 1 = The objective gradient is known but</td>
</tr>
<tr>
<td></td>
<td>some or all components of the constraint gradient are unknown, 2 = All</td>
</tr>
<tr>
<td></td>
<td>constraint gradients are known but some or all components of the objective</td>
</tr>
<tr>
<td></td>
<td>gradient are unknown, 3 = All objective and constraint gradients are</td>
</tr>
<tr>
<td></td>
<td>known (3,SOL-39).</td>
</tr>
<tr>
<td>GradCheck</td>
<td>0, 1, 2, 3 gives increasing level of user-supplied gradient checks (SOL-13).</td>
</tr>
<tr>
<td>DiffInt</td>
<td>Difference interval in derivative estimates (SOL-42).</td>
</tr>
<tr>
<td>CentralDiff</td>
<td>Central difference interval in derivative estimates (SOL-43).</td>
</tr>
<tr>
<td>QN_InitMatrix</td>
<td>Initial matrix for Quasi-Newton, may be set by the user. When QN_InitMatrix is empty, the identity matrix is used.</td>
</tr>
<tr>
<td>splineSmooth</td>
<td>Smoothness parameter sent to the SPLINE Toolbox routine csaps.m when</td>
</tr>
<tr>
<td></td>
<td>computing numerical approximations of the derivatives (0.4).</td>
</tr>
<tr>
<td>splineTol</td>
<td>Tolerance parameter sent to the SPLINE Toolbox routine spaps.m when</td>
</tr>
<tr>
<td></td>
<td>computing numerical approximations of the derivatives ($10^{-3}$).</td>
</tr>
<tr>
<td>BigStep</td>
<td>Unbounded step size. Used to detect unbounded nonlinear problems. (SOL-45).</td>
</tr>
<tr>
<td>BigObj</td>
<td>Unbounded objective value. Used to detect unbounded nonlinear problems.</td>
</tr>
<tr>
<td>(SOL-46).</td>
<td>CHECK</td>
</tr>
</tbody>
</table>


### Table 40: Information stored in the structure `Prob.PartSep`

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pSepFunc</code></td>
<td>Number of partially separable functions.</td>
</tr>
<tr>
<td><code>index</code></td>
<td>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. $f(x) = \sum_{i=1}^{M} f_i(x)$.</td>
</tr>
</tbody>
</table>

### Table 41: Information stored in the structure `Prob.SOL`

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SpecsFile</code></td>
<td>If nonempty gives the name of a file which is written in the SOL SPECS file format.</td>
</tr>
<tr>
<td><code>PrintFile</code></td>
<td>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 <code>Prob.SOL.optPar(1)</code>.</td>
</tr>
<tr>
<td><code>SummFile</code></td>
<td>If nonempty gives the name of a file which the SOL solver prints summary information on.</td>
</tr>
<tr>
<td><code>xs</code></td>
<td>Vector with solution $x$ and slack variables $s$. Used for warm starts.</td>
</tr>
<tr>
<td><code>hs</code></td>
<td>Vector with basis information in the SOL sparse solver format. Used for warm starts.</td>
</tr>
<tr>
<td><code>nS</code></td>
<td>Number of superbasics. Used for warm starts.</td>
</tr>
<tr>
<td><code>hElastic</code></td>
<td>Elastic variable information in <code>SQOPT</code>.</td>
</tr>
<tr>
<td><code>iState</code></td>
<td>Vector with basis information in the SOL dense solver format. Used for warm starts.</td>
</tr>
<tr>
<td><code>cLamda</code></td>
<td>Vector with Lagrange multiplier information in the SOL dense solver format. Used for warm starts.</td>
</tr>
<tr>
<td><code>H</code></td>
<td>Cholesky factor of Hessian Approximation. Either in natural order (<code>Hessian Yes</code>) or reordered (<code>Hessian No</code>). Used for warm starts using natural order with <code>NPSOL</code> and <code>NLSSOL</code>.</td>
</tr>
<tr>
<td><code>optPar</code></td>
<td>Vector with <code>optParN</code> 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 whose name always has the letters TL added, e.g. <code>minosTL</code>.</td>
</tr>
<tr>
<td><code>optParN</code></td>
<td>Number of elements in <code>optPar</code>, defined as 62.</td>
</tr>
</tbody>
</table>
Table 42: Information stored in the structure Prob.USER

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>Name of m-file computing the objective function ( f(x) ).</td>
</tr>
<tr>
<td>( g )</td>
<td>Name of m-file computing the gradient vector ( g(x) ). If Prob.USER.g is empty then numerical derivatives will be used.</td>
</tr>
<tr>
<td>( H )</td>
<td>Name of m-file computing the Hessian matrix ( H(x) ).</td>
</tr>
<tr>
<td>( c )</td>
<td>Name of m-file computing the vector of constraint functions ( c(x) ).</td>
</tr>
<tr>
<td>( dc )</td>
<td>Name of m-file computing the matrix of constraint normals ( \partial c(x)/dx ).</td>
</tr>
<tr>
<td>( d2c )</td>
<td>Name of m-file computing the 2nd part of 2nd derivative matrix of the Lagrangian function, ( \sum_{i=1}^{\lambda} \partial^2 c(x)/dx^2 ).</td>
</tr>
<tr>
<td>( r )</td>
<td>Name of m-file computing the residual vector ( r(x) ).</td>
</tr>
<tr>
<td>( J )</td>
<td>Name of m-file computing the Jacobian matrix ( J(x) ).</td>
</tr>
<tr>
<td>( d2r )</td>
<td>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} ).</td>
</tr>
</tbody>
</table>
The optimization output result structure

The results of the optimization attempts are stored in a structure array named Result. The currently defined fields in the structure are shown in Table 45. 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 43. The information stored in the mat-file glcSave.mat by the solver glcSolve is shown in Table 44.

Table 43: Information stored in the mat-file glbSave.mat by the solver glbSolve. Used for automatic restarts.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name of the problem, used as identification.</td>
</tr>
<tr>
<td>C</td>
<td>Matrix with all rectangle centerpoints in original coordinates.</td>
</tr>
<tr>
<td>F</td>
<td>Vector with function values.</td>
</tr>
<tr>
<td>D</td>
<td>Vector with distances from centerpoints to the vertices.</td>
</tr>
<tr>
<td>L</td>
<td>Matrix with all rectangle side lengths in each dimension.</td>
</tr>
<tr>
<td>d</td>
<td>Row vector of all different distances, sorted.</td>
</tr>
<tr>
<td>d_min</td>
<td>Row vector of minimum function value for each distance.</td>
</tr>
<tr>
<td>f_min</td>
<td>Best function value found at a feasible point.</td>
</tr>
<tr>
<td>E</td>
<td>Computed tolerance, dependent on f_min.</td>
</tr>
<tr>
<td>i_min</td>
<td>indices for all best points.</td>
</tr>
</tbody>
</table>

Table 44: Information stored in the mat-file glcSave.mat by the solver glcSolve. Used for automatic restarts.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name of the problem, used as identification.</td>
</tr>
<tr>
<td>C</td>
<td>Matrix with all rectangle centerpoints in original coordinates.</td>
</tr>
<tr>
<td>F</td>
<td>Vector with function values.</td>
</tr>
<tr>
<td>T</td>
<td>T(i) is the number of times rectangle i has been trisected.</td>
</tr>
<tr>
<td>D</td>
<td>Vector with distances from centerpoint to the vertices.</td>
</tr>
<tr>
<td>G</td>
<td>Matrix with constraint values for each point.</td>
</tr>
<tr>
<td>L_L</td>
<td>L_L(i,j) is the lower bound for rectangle j in integer dimension I(i).</td>
</tr>
<tr>
<td>L_U</td>
<td>L_U(i,j) is the upper bound for rectangle j in integer dimension I(i).</td>
</tr>
<tr>
<td>s_0</td>
<td>s_0 is used as s(0).</td>
</tr>
<tr>
<td>s</td>
<td>s(j) is the sum of observed rates of change for constraint j.</td>
</tr>
<tr>
<td>t</td>
<td>t(i) is the total number of splits along dimension i.</td>
</tr>
<tr>
<td>ignoreidx</td>
<td>Rectangles to be ignored in the rectangle selection procedure.</td>
</tr>
<tr>
<td>feasible</td>
<td>Flag indicating if a feasible point has been found.</td>
</tr>
<tr>
<td>Split</td>
<td>Split(i,j) is the number of splits along dimension i of rectangle j.</td>
</tr>
<tr>
<td>f_min</td>
<td>Best function value found at a feasible point.</td>
</tr>
</tbody>
</table>

The field xState describes the state of each of the variables. In Table 46 the different values are described. The different conditions for linear constraints are defined by the state variable in field bState. In Table 47 the different values are described.
Table 45: Information stored in the global Matlab structure Result.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Problem name.</td>
</tr>
<tr>
<td>P</td>
<td>Problem number.</td>
</tr>
<tr>
<td>probType</td>
<td>TOMLAB problem type.</td>
</tr>
<tr>
<td>Solver</td>
<td>Solver used.</td>
</tr>
<tr>
<td>SolverAlgorithm</td>
<td>Solver algorithm used.</td>
</tr>
<tr>
<td>solvType</td>
<td>TOMLAB solver type.</td>
</tr>
<tr>
<td>ExitFlag</td>
<td>0 if convergence to local min. Otherwise errors.</td>
</tr>
<tr>
<td>ExitText</td>
<td>Solver text about the result of the optimization.</td>
</tr>
<tr>
<td>Inform</td>
<td>Information parameter, type of convergence.</td>
</tr>
<tr>
<td>CPUtime</td>
<td>CPU time used in seconds.</td>
</tr>
<tr>
<td>REALtime</td>
<td>Real time elapsed in seconds.</td>
</tr>
<tr>
<td>iter</td>
<td>Number of major iterations.</td>
</tr>
<tr>
<td>MinorIter</td>
<td>Number of minor iterations (for some solvers).</td>
</tr>
<tr>
<td>FuncEv</td>
<td>Number of function evaluations needed.</td>
</tr>
<tr>
<td>GradEv</td>
<td>Number of gradient evaluations needed.</td>
</tr>
<tr>
<td>HessEv</td>
<td>Number of Hessian evaluations needed.</td>
</tr>
<tr>
<td>ConstrEv</td>
<td>Number of constraint evaluations needed.</td>
</tr>
<tr>
<td>ResEv</td>
<td>Number of residual evaluations needed (least squares).</td>
</tr>
<tr>
<td>JacEv</td>
<td>Number of Jacobian evaluations needed (least squares).</td>
</tr>
<tr>
<td>f_k</td>
<td>Function value at optimum.</td>
</tr>
<tr>
<td>g_k</td>
<td>Gradient value at optimum.</td>
</tr>
<tr>
<td>B_k</td>
<td>Quasi-Newton approximation of the Hessian at optimum.</td>
</tr>
<tr>
<td>H_k</td>
<td>Hessian value at optimum.</td>
</tr>
<tr>
<td>x_0</td>
<td>Starting point.</td>
</tr>
<tr>
<td>f_0</td>
<td>Function value at start i.e. f(x_0).</td>
</tr>
<tr>
<td>x_k</td>
<td>Optimal point.</td>
</tr>
<tr>
<td>y_k</td>
<td>Dual parameters.</td>
</tr>
<tr>
<td>v_k</td>
<td>Lagrange multipliers for constraints on variables, linear and nonlinear constraints.</td>
</tr>
<tr>
<td>r_k</td>
<td>Residual vector at optimum.</td>
</tr>
<tr>
<td>J_k</td>
<td>Jacobian matrix at optimum.</td>
</tr>
<tr>
<td>c_k</td>
<td>Value of constraints at optimum.</td>
</tr>
<tr>
<td>cJac</td>
<td>Constraint Jacobian at optimum.</td>
</tr>
<tr>
<td>xState</td>
<td>State of each variable, described in Table 46.</td>
</tr>
<tr>
<td>bState</td>
<td>State of each linear constraint, described in Table 47.</td>
</tr>
<tr>
<td>cState</td>
<td>State of each general constraint.</td>
</tr>
<tr>
<td>p_dx</td>
<td>Matrix where each column is a search direction.</td>
</tr>
<tr>
<td>alphaV</td>
<td>Matrix where row i stores the step lengths tried for the i:th iteration.</td>
</tr>
<tr>
<td>x_min</td>
<td>Lowest x-values in optimization. Used for plotting.</td>
</tr>
<tr>
<td>x_max</td>
<td>Highest x-values in optimization. Used for plotting.</td>
</tr>
<tr>
<td>F_X</td>
<td>F_X is a global matrix with rows: [iter_no f(x)].</td>
</tr>
<tr>
<td>SepLS</td>
<td>General result variable with fields z and Jz. Used when running separable nonlinear least squares problems</td>
</tr>
<tr>
<td>QP</td>
<td>Structure with special fields for QP problems. Used for warm starts, see Table 34.</td>
</tr>
<tr>
<td>SOL</td>
<td>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.</td>
</tr>
<tr>
<td>plotData</td>
<td>Structure with plotting parameters.</td>
</tr>
<tr>
<td>Prob</td>
<td>Problem structure, see Table 31 and Table 32.</td>
</tr>
</tbody>
</table>
Table 46: The state variable $xState$ for the variable.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A free variable.</td>
</tr>
<tr>
<td>1</td>
<td>Variable on lower bound.</td>
</tr>
<tr>
<td>2</td>
<td>Variable on upper bound.</td>
</tr>
<tr>
<td>3</td>
<td>Variable is fixed, lower bound is equal to upper bound.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Inactive constraint.</td>
</tr>
<tr>
<td>1</td>
<td>Linear constraint on lower bound.</td>
</tr>
<tr>
<td>2</td>
<td>Linear constraint on upper bound.</td>
</tr>
<tr>
<td>3</td>
<td>Linear equality constraint.</td>
</tr>
</tbody>
</table>
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 48 and 49.

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.

```matlab
... ...
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:

```matlab
... ...
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 48: The global variables used in TOMLAB

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXCOLS</td>
<td>Number of screen columns. Default 120.</td>
</tr>
<tr>
<td>MAXMENU</td>
<td>Number of menu items showed on one screen. Default 50.</td>
</tr>
<tr>
<td>MAX_c</td>
<td>Maximum number of constraints to be printed.</td>
</tr>
<tr>
<td>MAX_x</td>
<td>Maximum number of variables to be printed.</td>
</tr>
<tr>
<td>MAX_r</td>
<td>Maximum number of residuals to be printed.</td>
</tr>
<tr>
<td>CUTEPATH</td>
<td>The path ending with \cute.</td>
</tr>
<tr>
<td>CUTEDLL</td>
<td>Name of CUTE DLL file.</td>
</tr>
<tr>
<td>DLLPATH</td>
<td>Full path to the CUTE DLL file.</td>
</tr>
<tr>
<td>CUTE_g</td>
<td>Gradient.</td>
</tr>
<tr>
<td>CUTE_H</td>
<td>Hessian.</td>
</tr>
<tr>
<td>CUTE_Hx</td>
<td>Value of $x$ when computing $CUTE.H$.</td>
</tr>
<tr>
<td>CUTE_dc</td>
<td>Constraint normals.</td>
</tr>
<tr>
<td>CUTE_Equal</td>
<td>Binary vector, element $i$ equals 1 if constraint $i$ is an equality constraint.</td>
</tr>
<tr>
<td>CUTE_Linear</td>
<td>Binary vector, element $i$ equals 1 if constraint $i$ is a linear constraint.</td>
</tr>
<tr>
<td>$n_f$</td>
<td>Counter for the number of function evaluations.</td>
</tr>
<tr>
<td>$n_g$</td>
<td>Counter for the number of gradient evaluations.</td>
</tr>
<tr>
<td>$n_H$</td>
<td>Counter for the number of Hessian evaluations.</td>
</tr>
<tr>
<td>$n_c$</td>
<td>Counter for the number of constraint evaluations.</td>
</tr>
<tr>
<td>$n_dc$</td>
<td>Counter for the number of constraint normal evaluations.</td>
</tr>
<tr>
<td>$n_d2c$</td>
<td>Counter for the number of evaluations of the 2nd part of 2nd derivative matrix of the Lagrangian function.</td>
</tr>
<tr>
<td>$n_r$</td>
<td>Counter for the number of residual evaluations.</td>
</tr>
<tr>
<td>$n_J$</td>
<td>Counter for the number of Jacobian evaluations.</td>
</tr>
<tr>
<td>$n_d2r$</td>
<td>Counter for the number of evaluations of the 2nd part of the Hessian for a nonlinear least squares problem.</td>
</tr>
<tr>
<td>NLP_x</td>
<td>Value of $x$ when computing $NLP.f$.</td>
</tr>
<tr>
<td>NLP_f</td>
<td>Function value.</td>
</tr>
<tr>
<td>NLP_xg</td>
<td>Value of $x$ when computing $NLP.g$.</td>
</tr>
<tr>
<td>NLP_g</td>
<td>Gradient value.</td>
</tr>
<tr>
<td>NLP_xH</td>
<td>Value of $x$ when computing $NLP.H$.</td>
</tr>
<tr>
<td>NLP_H</td>
<td>Hessian value.</td>
</tr>
<tr>
<td>NLP_xc</td>
<td>Value of $x$ when computing $NLP.c$.</td>
</tr>
<tr>
<td>NLP_c</td>
<td>Constraints value.</td>
</tr>
<tr>
<td>NLP_pSepFunc</td>
<td>Number of partially separable functions.</td>
</tr>
<tr>
<td>NLP_pSepIndex</td>
<td>Index for the separated function computed.</td>
</tr>
</tbody>
</table>
Table 49: The global variables used in TOMLAB

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>US A</td>
<td>Problem dependent information sent between user routines. The user is recommended to always use this variable.</td>
</tr>
<tr>
<td>LS A</td>
<td>Problem dependent information sent from residual routine to Jacobian routine.</td>
</tr>
<tr>
<td>LS x</td>
<td>Value of $x$ when computing $LS_r$</td>
</tr>
<tr>
<td>LS r</td>
<td>Residual value.</td>
</tr>
<tr>
<td>LS xJ</td>
<td>Value of $x$ when computing $LS_J$</td>
</tr>
<tr>
<td>LS J</td>
<td>Jacobian value.</td>
</tr>
<tr>
<td>SEP z</td>
<td>Separated variables $z$.</td>
</tr>
<tr>
<td>SEP xJ</td>
<td>Jacobian for separated variables $z$.</td>
</tr>
<tr>
<td>wNLLS</td>
<td>Weighting of least squares residuals (internal variable in $nlp_r$ and $nlp_J$).</td>
</tr>
<tr>
<td>alphaV</td>
<td>Vector with all step lengths $\alpha$ for each iteration.</td>
</tr>
<tr>
<td>BUILDP</td>
<td>Flag.</td>
</tr>
<tr>
<td>F X</td>
<td>Matrix with function values.</td>
</tr>
<tr>
<td>pLen</td>
<td>Number of iterations so far.</td>
</tr>
<tr>
<td>p dx</td>
<td>Matrix with all search directions.</td>
</tr>
<tr>
<td>X max</td>
<td>The biggest $x$-values for all iterations.</td>
</tr>
<tr>
<td>X min</td>
<td>The smallest $x$-values for all iterations.</td>
</tr>
<tr>
<td>X NEW</td>
<td>Last $x$ point in line search. Possible new $x_k$.</td>
</tr>
<tr>
<td>X OLD</td>
<td>Last known base point $x_k$.</td>
</tr>
<tr>
<td>probType</td>
<td>Defines the type of optimization problem.</td>
</tr>
<tr>
<td>solvType</td>
<td>Defines the solver type.</td>
</tr>
<tr>
<td>answer</td>
<td>Used by the GUI for user control options.</td>
</tr>
<tr>
<td>instruction</td>
<td>Used by the GUI for user control options.</td>
</tr>
<tr>
<td>question</td>
<td>Used by the GUI for user control options.</td>
</tr>
<tr>
<td>plotData</td>
<td>Structure with plotting parameters.</td>
</tr>
<tr>
<td>Prob</td>
<td>Problem structure, see Table 31 and Table 32.</td>
</tr>
<tr>
<td>Result</td>
<td>Result structure, see Table 45.</td>
</tr>
<tr>
<td>runNumber</td>
<td>Vector index when $Result$ is an array of structures.</td>
</tr>
<tr>
<td>TIME0</td>
<td>Used to compute CPU time and real time elapsed.</td>
</tr>
<tr>
<td>TIME1</td>
<td>Used to compute CPU time and real time elapsed.</td>
</tr>
<tr>
<td>cJPI</td>
<td>Used to store sparsity pattern for the constraint Jacobian when automatic differentiation is used.</td>
</tr>
<tr>
<td>HPI</td>
<td>Used to store sparsity pattern for the Hessian when automatic differentiation is used.</td>
</tr>
<tr>
<td>JPI</td>
<td>Used to store sparsity pattern for the Jacobian when automatic differentiation is used.</td>
</tr>
<tr>
<td>glbSave</td>
<td>Used to save global variables in recursive calls to TOMLAB.</td>
</tr>
</tbody>
</table>
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 \texttt{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 \texttt{lptest1}.

1. Copy File: \texttt{tomlab/testprob/lp\_prob} to e.g. \texttt{lp\_new\_prob.m} in a working directory.

2. Edit \texttt{lp\_new\_prob.m}. Add the problem name, \texttt{lptest1}, to the menu choice:

```matlab
...,'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 \texttt{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 \(b_U\).

```matlab
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

```matlab
... 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.

```matlab
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 an Quadratic Programming Init File

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:

```matlab
...,'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 ] ;  
    c = [ 3 -4 ];  
    A = [ 1 1 ];  
    b_L = [-inf 0 ];  
    b_U = [ 5 0 ];  
    x_L = [ 0 0 ];  
    x_U = [ -1 -1 ];  
    x_0 = [ 0 1 ];  
    x_min=[-1 -1 ];  
    x_max=[ 6 6 ];  
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);  
...  
Result = tomRun('qpSolve', Prob);  

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 .  

169
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`:

```matlab
...
,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.

```matlab
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

```matlab...
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 uc_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 uc_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+2*x(2)^2+2*x(1)^2]*4, ...
        -2*x(1)*x(2)+x(1)^2+2*x(2)^2; 0, 1-x(1)^2+2*x(1)^2+2*x(2)^2+2*x(2)^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

```
min_x \; f(x) = \alpha \left( x_2 - x_1^2 \right)^2 + (1 - x_1)^2
\quad \text{s.t.} \quad -2 \leq x_1 \leq 2
\quad -2 \leq x_2 \leq 2
```

The standard value is \( \alpha = 100 \). To define (24) as a box-bounded global optimization problem follow the step wise instructions below (for all instructions is 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,

\[
\begin{align*}
\min_x & \quad f(x) = \alpha (x_2 - x_1^2)^2 + (1 - x_1)^2 \\
\text{s/t} & \quad -2 \leq x_1 \leq 2, \ x_1 \text{ integer} \\
& \quad -2 \leq x_2 \leq 2 \\
& \quad x_1 - x_2 \leq 1 \\
& \quad -x_1^2 - x_2 \leq 1
\end{align*}
\]

(25)
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 (25) 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.
% 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,

\[
\begin{align*}
\min_x \quad & f(x) = \alpha \left( x_2 - x_1^2 \right)^2 + (1 - x_1)^2 \\
\text{s.t.} \quad & -10 \leq x_1 \leq 2 \\
& -10 \leq x_2 \leq 2 \\
& x_1 - x_2 \leq 1 \\
& -x_1^2 - x_2 \leq 1
\end{align*}
\]

(26)

The standard value is \( \alpha = 100 \). The first two constraints are simple bounds on the variables. The third constraint is linear and 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 \textit{con.prob}, \textit{con.f}, \textit{con.g}, \textit{con.H}, \textit{con.c} and \textit{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 \textit{connew.prob}, \textit{connew.f}, \textit{connew.g}, \textit{connew.H}, \textit{connew.c} and \textit{connew.dc}.

2. Make the following modification in the beginning of \textit{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 \textit{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 \textit{New Constrained ...} is of course possible to change as the user likes. Now \textit{connew.prob.m} is one of the TOMLAB Init Files in the GUI data base and all problems in \textit{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 \textit{ls.prob.m}, \textit{ls.r.m} and \textit{ls.J.m} to the working directory.

2. Add the problem name to the menu choice in \textit{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;
   ```
3. Add the following in `ls_prob.m` after the last already existing problem (the optional parameters are not necessary to define):

```matlab
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`:

```matlab
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`:

```matlab
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

\[
\begin{bmatrix}
30 \\
50 \\
70 \\
90 \\
110 \\
130 \\
150 \\
170 \\
190 \\
210 \\
230 \\
250 \\
270 \\
290 \\
310 \\
330 \\
350 \\
370
\end{bmatrix}
\times
\begin{bmatrix}
18299 \\
15428 \\
13347 \\
11466 \\
10077 \\
8729 \\
7382 \\
6708 \\
5932 \\
5352 \\
4734 \\
4271 \\
3744 \\
3485 \\
3111 \\
2950 \\
2686 \\
2476
\end{bmatrix}
\times
\begin{bmatrix}
t = 10^{-3} \\
Y(t) = 10^{-4}
\end{bmatrix}
\]

(27)

here named SW.

To define (27) as an exponential sum fitting problem follow the step wise instructions below (for all instructions 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:

\[
\ldots
\begin{align*}
\ldots
\text{'CDFsim5'}... \\
\text{'CDFdata5\'}... \\
\text{'SW '...}
\end{align*}
\]

); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

if isempty(P)
    return;
end
\ldots

3. Add the following in exp_prob.m after the last already existing problem:

\[
\ldots
\begin{align*}
\text{elseif P==52}
\quad \text{Name='SW';} \\
\quad t=[30:20:370]'; \quad \text{Time in ms}
\quad y=[18299 15428 13947 11466 10077 8729 7382 6708 5932 5352 4734 4271 \ldots}
\end{align*}
\]

178
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.

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
    ...
    ...

179
3. Make the following modifications at the end of `lsnew_prob`

```matlab
...
%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

```matlab
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 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.

```matlab
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('
 Problem number %d:',P);
    fprintf(' %s',Prob.Name);
    Result = tomRun('clsSolve', Prob, ask, PriLev);
    fprintf('
 Number of iterations: %d',Result.Iter);
end
```

Another example is on how to solve the exponential sum fitting problem (27)
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:

```matlab
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

<table>
<thead>
<tr>
<th></th>
<th>f_k</th>
<th>0.000000000000000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>User given f(x_*)</td>
<td>0.000000000000000000</td>
<td></td>
</tr>
<tr>
<td>f(x_0)</td>
<td>24.199999999999996000</td>
<td></td>
</tr>
</tbody>
</table>

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_0: 1.000000 1.000000
Diff x-x0:
    2.200000e+00  -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:

```matlab
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
```

181
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.

```matlab
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('Problem number %d:', P);
    fprintf(' %s', Prob.Name);
    Result = tomRun(Solver, Prob, ask, PriLev);
    fprintf('Number of iterations: %d', Result.Iter);
end
```

182
E Interfaces

Some users may have been used to work with Math Works Optimization TB v2.1 or Math Works Optimization TB v1.5, or have code written for use with these toolboxes. For that reason TOMLAB has defined interfaces to simplify the transfer to TOMLAB. There are two ways in which the Math Works Optimization Toolbox may be used in TOMLAB. One way is to use the same type of the call to the main solvers as in Math Works 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 Optimization Toolbox solver. Interfaces have been made to both Math Works Optimization TB v2.1 and Math Works 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 Math Works Optimization TB v2.1 is not accessible.

E.1 Solver Call Compatible with Optimization Toolbox 2.1

TOMLAB is call compatible with Math Works 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 Math Works 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 50 gives a list of the solvers with compatible interfaces.

Table 50: Call compatible interfaces to Math Works Optimization TB v2.1.

<table>
<thead>
<tr>
<th>Function</th>
<th>Type of problem solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmincon</td>
<td>Constrained minimization.</td>
</tr>
<tr>
<td>fminsearch</td>
<td>Unconstrained minimization using Nelder-Mead type simplex</td>
</tr>
<tr>
<td></td>
<td>search method.</td>
</tr>
<tr>
<td>fminunc</td>
<td>Unconstrained minimization using gradient search.</td>
</tr>
<tr>
<td>linprog</td>
<td>Linear programming.</td>
</tr>
<tr>
<td>lsqcurvefit</td>
<td>Nonlinear least squares curvefitting.</td>
</tr>
<tr>
<td>lsqlin</td>
<td>Linear least squares.</td>
</tr>
<tr>
<td>lsqnonlin</td>
<td>Linear least squares with nonnegative variable constraints.</td>
</tr>
<tr>
<td>lsqnonneg</td>
<td>Nonlinear least squares.</td>
</tr>
<tr>
<td>quadprog</td>
<td>Quadratic programming.</td>
</tr>
</tbody>
</table>

In Table 51 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 51: Testroutines for the call compatible interfaces to Math Works Optimization TB v2.1 present in the it examples directory in the TOMLAB distribution.

<table>
<thead>
<tr>
<th>Function</th>
<th>Type of problem solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>testfmincon</td>
<td>Test of constrained minimization.</td>
</tr>
<tr>
<td>testfminsearch</td>
<td>Test of unconstrained minimization using a Nelder-Mead type</td>
</tr>
<tr>
<td></td>
<td>simplex search method.</td>
</tr>
<tr>
<td>testfminunc</td>
<td>Test of unconstrained minimization using gradient search.</td>
</tr>
<tr>
<td>testlinprog</td>
<td>Test of linear programming.</td>
</tr>
<tr>
<td>testlsqcurvefit</td>
<td>Test of nonlinear least squares curvefitting.</td>
</tr>
<tr>
<td>testlsqlin</td>
<td>Test of linear least squares.</td>
</tr>
<tr>
<td>testlsqnonlin</td>
<td>Test of linear least squares with nonnegative variable</td>
</tr>
<tr>
<td></td>
<td>constraints.</td>
</tr>
<tr>
<td>testlsqnonneg</td>
<td>Test of nonlinear least squares.</td>
</tr>
<tr>
<td>testquadprog</td>
<td>Test of quadratic programming.</td>
</tr>
</tbody>
</table>
E.1.1 Solving LP Similar to Optimization Toolbox 2.1

For linear programs the Math Works Optimization TB v2.1 solver is \textit{linprog}. The TOMLAB \textit{linprog} solver adds one extra input argument, the \textit{Prob} structure, and one extra output argument, the \textit{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 \textit{linprog} solver to solve test problem (11) illustrates the basic usage.

\textbf{File: tomlab/usersguide/lpTest4.m}

\begin{verbatim}
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 inequalities with different lower and upper bounds
   % the problem can be transformed using the TOMLAB routine \textit{cpTransf}.
   % See the example file tomlab/examples/testlinprog.m for an example.

   fprintf('
');
   fprintf('
');
   disp('Run TOMLAB linprog on LP Example');
   fprintf('
');
   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);
\end{verbatim}

The results from this test show the same results as previous runs in Section 5, because the same solver is called.

\textbf{File: tomlab/usersguide/lpTest4.out}

\textit{linprog} (\textit{linprog}): Optimization terminated successfully

\begin{verbatim}
Run TOMLAB linprog on LP Example

Constraints Ax-b_U  -8.881784e-016  0.000000e+000
Lambda.lower:  0.000000e+000  0.000000e+000
Lambda.upper:  0.000000e+000  0.000000e+000
Lambda.eqlin: 
Lambda.ineqlin:  -4.549052e+000  -4.453845e+000
x:  2.571429e+000  1.714286e+000
Output Structure
\end{verbatim}
E.1.2 Solving QP similar to Optimization Toolbox 2.1

For quadratic programs the Math Works 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

```matlab
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 inequalities 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(' %
');
fprintf(' %
');
disp('Run TOMLAB quadprog on QP Example');
fprintf(' %
');
fprintf(' %
');
xprintf(A*x-b_U, 'Constraints Ax-b_U ');
xprintf(Lambda.lower, 'Lambda.lower: ');
xprintf(Lambda.upper, 'Lambda.upper: ');
xprintf(Lambda.eqlin, 'Lambda.eqlin: ');
xprintf(Lambda.ineqlin, 'Lambda.ineqlin: ');
xprintf(x, 'x: ');
format compact
disp('Output Structure')
disp(Out)
fprintf('Function value %30.20f. ExitFlag %d\n',fVal,ExitFlag);

The restricted problem formulation in Math Works 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

186
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: 4
  algorithm: 'Active set strategy'
  cgiterations: []
  firstorderopt: []

Function value -0.02499999999999999400. ExitFlag 1

E.2 The Matlab Optimization Toolbox Interface

Included in TOMLAB is an interface to a number of the solvers in the Math Works Optimization TB v1.5 [14] and Math Works Optimization TB v2.1 [16]. The solvers that are directly possible to use, when a problem is generated in the TOMLAB format, are listed in Table 52. 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.

Math Works Optimization TB v1.5 is using a parameter vector OPTIONS of length 18, that the routine foptions is setting up the default values for. Math Works Optimization TB v2.1 is instead using a structure.

Table 52: Optimization toolbox routines with a TOMLAB interface.

<table>
<thead>
<tr>
<th>Function</th>
<th>Type of problem solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmincon</td>
<td>Constrained minimization.</td>
</tr>
<tr>
<td>fminsearch</td>
<td>Unconstrained minimization using Nelder-Mead type simplex search method.</td>
</tr>
<tr>
<td>fminunc</td>
<td>Unconstrained minimization using gradient search.</td>
</tr>
<tr>
<td>linprog</td>
<td>Linear programming.</td>
</tr>
<tr>
<td>lsqcurvefit</td>
<td>Nonlinear least squares curvefitting.</td>
</tr>
<tr>
<td>lsqlin</td>
<td>Linear least squares.</td>
</tr>
<tr>
<td>lsqnonlinlin</td>
<td>Linear least squares with nonnegative variable constraints.</td>
</tr>
<tr>
<td>lsqnonneg</td>
<td>Nonlinear least squares.</td>
</tr>
<tr>
<td>quadprog</td>
<td>Quadratic programming.</td>
</tr>
<tr>
<td>constr</td>
<td>Constrained minimization.</td>
</tr>
<tr>
<td>fmins</td>
<td>Unconstrained minimization using Nelder-Mead type simplex search method.</td>
</tr>
<tr>
<td>fminu</td>
<td>Unconstrained minimization using gradient search.</td>
</tr>
<tr>
<td>leastsq</td>
<td>Nonlinear least squares.</td>
</tr>
<tr>
<td>lp</td>
<td>Linear programming.</td>
</tr>
<tr>
<td>qp</td>
<td>Quadratic programming.</td>
</tr>
</tbody>
</table>

E.3 The CUTE Interface

The Constrained and Unconstrained Testing Environment (CUTE) [12, 13] 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 53 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 [12, 13].

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctools</td>
<td>Interface routine to constrained CUTE test problems.</td>
</tr>
<tr>
<td>utools</td>
<td>Interface routine to unconstrained CUTE test problems.</td>
</tr>
<tr>
<td>cto_prob</td>
<td>Initialization of constrained CUTE test problems.</td>
</tr>
<tr>
<td>ctl_prob</td>
<td>Initialization of large constrained CUTE test problems.</td>
</tr>
<tr>
<td>cto_f</td>
<td>Compute the objective function $f(x)$ for constrained CUTE test problems.</td>
</tr>
<tr>
<td>cto_g</td>
<td>Compute the gradient $g(x)$ for constrained CUTE test problems.</td>
</tr>
<tr>
<td>cto_H</td>
<td>Compute the Hessian $H(x)$ of $f(x)$ for constrained CUTE test problems.</td>
</tr>
<tr>
<td>cto_c</td>
<td>Compute the vector of constraint functions $c(x)$ for constrained CUTE test problems.</td>
</tr>
<tr>
<td>cto_dc</td>
<td>Compute the matrix of constraint normals for constrained CUTE test problems.</td>
</tr>
<tr>
<td>cto_d2c</td>
<td>Compute the second part of the second derivative of the Lagrangian function for constrained CUTE test problems.</td>
</tr>
<tr>
<td>uto_prob</td>
<td>Initialization of unconstrained CUTE test problems.</td>
</tr>
<tr>
<td>utl_prob</td>
<td>Initialization of large unconstrained CUTE test problems.</td>
</tr>
<tr>
<td>uto_f</td>
<td>Compute the objective function $f(x)$ for unconstrained CUTE test problems.</td>
</tr>
<tr>
<td>uto_g</td>
<td>Compute the gradient $g(x)$ for unconstrained CUTE test problems.</td>
</tr>
<tr>
<td>uto_H</td>
<td>Compute the Hessian $H(x)$ of $f(x)$ for unconstrained CUTE test problems.</td>
</tr>
</tbody>
</table>

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 [26] was built. The reason to choose AMPL was that it has a rudimentary Matlab interface written in C [28] 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 \textit{nl}, called \textit{nl}-files. This gives a file \textit{rosenbr.nl} for our example. Then AMPL invokes a solver with two arguments, the problem name, e.g. \textit{rosenbr}, and a string -\textit{AMPL}. The second argument is a flag telling AMPL is the caller. After solving the problem, the solver creates a file with extension \textit{sol}, e.g. \textit{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 \textit{nl}-format. TOMLAB assumes the \textit{nl}-files to be stored in directory /\textit{tomlab/ampl} or /\textit{tomlab/amplsp} (for sparse problems). When using the TOMLAB interface, the user either gets a menu of the \textit{nl}-files found or directly makes a choice of which problem to solve. The initialization routine in TOMLAB for AMPL problems, \textit{amp_prob}, either calls \textit{amplfunc} or \textit{spamfunc}, the two MEX-file interface routines written by Gay [28]. The low level routines \textit{amp_f}, \textit{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.
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 [7], AMPL [26], ASCEND [58], GAMS [8, 15] 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 [21].

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 [45, 59]. Nowadays, languages like Matlab has a rapid growth of users. Matlab was originally created [52] as a preprocessor to the standard Fortran subroutine libraries in numerical linear algebra, LINPACK [19] and EISPACK [66] [27], much the same idea as the modeling languages discussed above.

Matlab of today is an advanced and powerful tool, with graphics, animation and advanced menu design possibilities integrated with the mathematics. The Matlab language has made the development of toolboxes possible, which serves as a direct extension to the language itself. Using Matlab as an environment for solving optimization problems offers much more possibilities for analysis than just the pure solution of the problem. The increased quality of the Matlab MEX-file interfaces makes it possible to run Fortran and C-programs on both PC and Unix systems. 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 \texttt{lpSolve}, two algorithms implemented in the solver \texttt{linprog} from Optimization Toolbox 2.0 [16] and the Fortran solvers MINOS and QPOPT, available in TOMLAB v3.0, are compared. In all test cases the solvers converge to the same solution. The results are presented in five tables

Table 54, Table 55, Table 56, Table 57 and Table 58. The problem dimensions and all elements in (18) are chosen randomly. Since the simplex algorithm in \texttt{linprog} does not return the number of iterations as output, these figures could not be presented. \texttt{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 \texttt{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 54 show that problems with about 200 variables and 150 inequality constraints are solved by \texttt{lpSolve} fast and efficient. When comparing elapsed computational time for 20 problems, it is clear that \texttt{lpSolve} is much faster then the corresponding simplex algorithm implemented in the \texttt{linprog} solver. In fact \texttt{lpSolve}, with the minimum cost selection rule, is more than five times faster, a remarkable difference. \texttt{lpSolve} is also more than twice as fast as the other algorithm implemented in \texttt{linprog}, a primal-dual interior-point method aimed for large-scale problems [16]. There is a penalty about a factor of three to choose Bland’s rule to prevent cycling in \texttt{lpSolve}. The solvers written in Fortran, MINOS and QPOPT, of course run much faster, but the iteration count show that \texttt{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.

In Table 55 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 \texttt{lpSolve} and the Fortran solvers are not so striking for these lower dimensional problems. \texttt{lpSolve} is now more than nine times faster than the simplex algorithm in \texttt{linprog} and twice as fast as the primal-dual interior-point method in \texttt{linprog}.

A similar test on larger dense problems, running 20 problems with about 500 variables and 240 inequality constraints, shows no benefit in using the primal-dual interior-point method in \texttt{linprog}, see Table 58. In that test \texttt{lpSolve} is more than five times faster, and 15 times faster than the simplex algorithm in \texttt{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 59, 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 v3.0 is very important.
Table 54: 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.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$m$</th>
<th>$lpS$</th>
<th>$lpS$</th>
<th>$Minos$</th>
<th>qpopt</th>
<th>$linprog$</th>
<th>$lpS$</th>
<th>$lpS$</th>
<th>$Minos$</th>
<th>qpopt</th>
<th>$linprog$</th>
<th>$linprog$</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>32</td>
<td>37</td>
<td>12</td>
<td>10</td>
<td>11</td>
<td>16</td>
<td>1.05</td>
<td>0.61</td>
<td>0.33</td>
<td>0.31</td>
<td>9.06</td>
<td>1.14</td>
</tr>
<tr>
<td>129</td>
<td>60</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>17</td>
<td>0.63</td>
<td>0.59</td>
<td>0.24</td>
<td>0.21</td>
<td>9.20</td>
<td>2.07</td>
</tr>
<tr>
<td>125</td>
<td>45</td>
<td>8</td>
<td>9</td>
<td>16</td>
<td>7</td>
<td>14</td>
<td>0.57</td>
<td>0.59</td>
<td>0.35</td>
<td>0.32</td>
<td>8.20</td>
<td>1.34</td>
</tr>
<tr>
<td>81</td>
<td>65</td>
<td>27</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>12</td>
<td>1.30</td>
<td>0.54</td>
<td>0.23</td>
<td>0.21</td>
<td>3.51</td>
<td>1.38</td>
</tr>
<tr>
<td>102</td>
<td>40</td>
<td>25</td>
<td>9</td>
<td>12</td>
<td>8</td>
<td>12</td>
<td>1.00</td>
<td>0.60</td>
<td>0.39</td>
<td>0.33</td>
<td>5.26</td>
<td>1.01</td>
</tr>
<tr>
<td>96</td>
<td>33</td>
<td>13</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>0.65</td>
<td>0.41</td>
<td>0.34</td>
<td>0.32</td>
<td>4.72</td>
<td>0.84</td>
</tr>
<tr>
<td>110</td>
<td>61</td>
<td>29</td>
<td>10</td>
<td>9</td>
<td>9</td>
<td>15</td>
<td>1.38</td>
<td>0.66</td>
<td>0.25</td>
<td>0.33</td>
<td>6.34</td>
<td>1.73</td>
</tr>
<tr>
<td>113</td>
<td>27</td>
<td>25</td>
<td>8</td>
<td>161</td>
<td>8</td>
<td>10</td>
<td>0.87</td>
<td>0.50</td>
<td>0.41</td>
<td>0.34</td>
<td>6.72</td>
<td>0.77</td>
</tr>
<tr>
<td>127</td>
<td>58</td>
<td>16</td>
<td>9</td>
<td>13</td>
<td>8</td>
<td>14</td>
<td>0.91</td>
<td>0.58</td>
<td>0.26</td>
<td>0.34</td>
<td>8.58</td>
<td>1.82</td>
</tr>
<tr>
<td>85</td>
<td>58</td>
<td>10</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>14</td>
<td>0.68</td>
<td>0.59</td>
<td>0.25</td>
<td>0.31</td>
<td>3.70</td>
<td>1.45</td>
</tr>
<tr>
<td>103</td>
<td>31</td>
<td>15</td>
<td>7</td>
<td>9</td>
<td>6</td>
<td>12</td>
<td>0.69</td>
<td>0.52</td>
<td>0.35</td>
<td>0.33</td>
<td>5.39</td>
<td>0.87</td>
</tr>
<tr>
<td>101</td>
<td>41</td>
<td>22</td>
<td>9</td>
<td>11</td>
<td>9</td>
<td>11</td>
<td>0.87</td>
<td>0.56</td>
<td>0.36</td>
<td>0.22</td>
<td>5.20</td>
<td>0.98</td>
</tr>
<tr>
<td>83</td>
<td>41</td>
<td>9</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>12</td>
<td>0.54</td>
<td>0.36</td>
<td>0.38</td>
<td>0.33</td>
<td>3.55</td>
<td>0.98</td>
</tr>
<tr>
<td>118</td>
<td>39</td>
<td>28</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>13</td>
<td>0.89</td>
<td>0.57</td>
<td>0.36</td>
<td>0.34</td>
<td>7.23</td>
<td>1.14</td>
</tr>
<tr>
<td>92</td>
<td>33</td>
<td>13</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>12</td>
<td>0.63</td>
<td>0.53</td>
<td>0.23</td>
<td>0.33</td>
<td>4.33</td>
<td>0.90</td>
</tr>
<tr>
<td>110</td>
<td>46</td>
<td>21</td>
<td>7</td>
<td>15</td>
<td>6</td>
<td>13</td>
<td>0.81</td>
<td>0.46</td>
<td>0.25</td>
<td>0.34</td>
<td>6.37</td>
<td>1.26</td>
</tr>
<tr>
<td>82</td>
<td>65</td>
<td>25</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>15</td>
<td>1.21</td>
<td>0.51</td>
<td>0.38</td>
<td>0.22</td>
<td>3.41</td>
<td>1.63</td>
</tr>
<tr>
<td>104</td>
<td>29</td>
<td>6</td>
<td>10</td>
<td>4</td>
<td>11</td>
<td>0.47</td>
<td>0.36</td>
<td>0.23</td>
<td>0.34</td>
<td>5.52</td>
<td>0.85</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>48</td>
<td>28</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>13</td>
<td>1.13</td>
<td>0.50</td>
<td>0.24</td>
<td>0.35</td>
<td>3.53</td>
<td>1.15</td>
</tr>
<tr>
<td>90</td>
<td>50</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>11</td>
<td>0.44</td>
<td>0.35</td>
<td>0.24</td>
<td>0.23</td>
<td>4.13</td>
<td>1.18</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>45</td>
<td>19</td>
<td>8</td>
<td>17</td>
<td>7</td>
<td>13</td>
<td>0.84</td>
<td>0.52</td>
<td>0.30</td>
<td>0.30</td>
<td>5.70</td>
<td>1.23</td>
</tr>
</tbody>
</table>
Table 55: Computational results on randomly generated medium size LP problems for four different routines. \textit{Iter} is the number of iterations and \textit{Time} is the elapsed time in seconds on a Dell Latitude CPI 266XT running Matlab 5.3. The \textit{lpS} solver is the TOMLAB \textit{lpSolve}, and it is run with both Bland’s selection rule (iterations $I_t$, time $T_t$) and with the minimum cost selection rule (iterations $I_m$, time $T_m$). The \textit{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 $I_t$, time $T_t$). 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.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( m )</th>
<th>\textit{lpS}</th>
<th>\textit{lpS}</th>
<th>\textit{M inos}</th>
<th>\textit{qopt}</th>
<th>\textit{linprog}</th>
<th>\textit{lpS}</th>
<th>\textit{lpS}</th>
<th>\textit{M inos}</th>
<th>\textit{qopt}</th>
<th>\textit{linprog}</th>
<th>\textit{linprog}</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I_t )</td>
<td>( I_m )</td>
<td>\textit{Iter}</td>
<td>\textit{Iter}</td>
<td>\textit{Iter}</td>
<td>\textit{Iter}</td>
<td>\textit{Time}</td>
<td>\textit{Time}</td>
<td>\textit{Time}</td>
<td>\textit{Time}</td>
<td>\textit{Time}</td>
<td>\textit{Time}</td>
<td>\textit{Time}</td>
</tr>
<tr>
<td>228</td>
<td>132</td>
<td>32</td>
<td>10</td>
<td>17</td>
<td>12</td>
<td>22</td>
<td>3.41</td>
<td>1.56</td>
<td>0.49</td>
<td>0.39</td>
<td>38.66</td>
<td>11.51</td>
</tr>
<tr>
<td>191</td>
<td>164</td>
<td>20</td>
<td>9</td>
<td>10</td>
<td>18</td>
<td>3.12</td>
<td>1.85</td>
<td>0.49</td>
<td>0.26</td>
<td>24.91</td>
<td>12.50</td>
<td></td>
</tr>
<tr>
<td>212</td>
<td>155</td>
<td>63</td>
<td>16</td>
<td>30</td>
<td>16</td>
<td>7.90</td>
<td>2.76</td>
<td>0.54</td>
<td>0.41</td>
<td>33.36</td>
<td>12.57</td>
<td></td>
</tr>
<tr>
<td>185</td>
<td>158</td>
<td>53</td>
<td>25</td>
<td>16</td>
<td>16</td>
<td>6.86</td>
<td>4.00</td>
<td>0.38</td>
<td>0.43</td>
<td>23.88</td>
<td>11.29</td>
<td></td>
</tr>
<tr>
<td>222</td>
<td>168</td>
<td>35</td>
<td>12</td>
<td>0</td>
<td>12</td>
<td>5.38</td>
<td>2.56</td>
<td>0.64</td>
<td>0.42</td>
<td>40.13</td>
<td>17.78</td>
<td></td>
</tr>
<tr>
<td>207</td>
<td>162</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>21</td>
<td>1.91</td>
<td>1.69</td>
<td>0.51</td>
<td>0.27</td>
<td>33.74</td>
<td>15.66</td>
</tr>
<tr>
<td>229</td>
<td>130</td>
<td>42</td>
<td>12</td>
<td>21</td>
<td>19</td>
<td>4.31</td>
<td>1.81</td>
<td>0.42</td>
<td>0.44</td>
<td>44.53</td>
<td>11.69</td>
<td></td>
</tr>
<tr>
<td>213</td>
<td>136</td>
<td>56</td>
<td>6</td>
<td>21</td>
<td>6</td>
<td>19</td>
<td>6.02</td>
<td>1.19</td>
<td>0.51</td>
<td>0.39</td>
<td>36.54</td>
<td>11.07</td>
</tr>
<tr>
<td>227</td>
<td>146</td>
<td>95</td>
<td>19</td>
<td>33</td>
<td>20</td>
<td>23</td>
<td>10.91</td>
<td>2.94</td>
<td>0.45</td>
<td>0.45</td>
<td>44.84</td>
<td>15.82</td>
</tr>
<tr>
<td>192</td>
<td>150</td>
<td>25</td>
<td>6</td>
<td>13</td>
<td>5</td>
<td>16</td>
<td>3.22</td>
<td>1.26</td>
<td>0.53</td>
<td>0.27</td>
<td>27.07</td>
<td>10.79</td>
</tr>
<tr>
<td>195</td>
<td>155</td>
<td>12</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>22</td>
<td>2.19</td>
<td>1.76</td>
<td>0.52</td>
<td>0.39</td>
<td>27.40</td>
<td>14.76</td>
</tr>
<tr>
<td>221</td>
<td>160</td>
<td>30</td>
<td>12</td>
<td>10</td>
<td>11</td>
<td>22</td>
<td>4.66</td>
<td>2.41</td>
<td>0.59</td>
<td>0.43</td>
<td>36.95</td>
<td>18.00</td>
</tr>
<tr>
<td>183</td>
<td>144</td>
<td>61</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>20</td>
<td>7.08</td>
<td>1.62</td>
<td>0.37</td>
<td>0.39</td>
<td>22.34</td>
<td>11.22</td>
</tr>
<tr>
<td>200</td>
<td>165</td>
<td>19</td>
<td>10</td>
<td>0</td>
<td>14</td>
<td>19</td>
<td>3.27</td>
<td>2.22</td>
<td>0.61</td>
<td>0.42</td>
<td>27.94</td>
<td>14.43</td>
</tr>
<tr>
<td>199</td>
<td>137</td>
<td>16</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td>19</td>
<td>2.04</td>
<td>1.04</td>
<td>0.48</td>
<td>0.39</td>
<td>28.67</td>
<td>9.90</td>
</tr>
<tr>
<td>188</td>
<td>154</td>
<td>18</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>17</td>
<td>2.59</td>
<td>1.57</td>
<td>0.53</td>
<td>0.39</td>
<td>25.19</td>
<td>10.81</td>
</tr>
<tr>
<td>202</td>
<td>159</td>
<td>25</td>
<td>13</td>
<td>0</td>
<td>11</td>
<td>17</td>
<td>3.82</td>
<td>2.50</td>
<td>0.60</td>
<td>0.44</td>
<td>30.28</td>
<td>12.37</td>
</tr>
<tr>
<td>223</td>
<td>155</td>
<td>103</td>
<td>16</td>
<td>20</td>
<td>17</td>
<td>24</td>
<td>12.50</td>
<td>2.95</td>
<td>0.56</td>
<td>0.44</td>
<td>39.54</td>
<td>18.06</td>
</tr>
<tr>
<td>196</td>
<td>121</td>
<td>17</td>
<td>7</td>
<td>16</td>
<td>6</td>
<td>18</td>
<td>1.81</td>
<td>1.08</td>
<td>0.37</td>
<td>0.40</td>
<td>27.59</td>
<td>7.94</td>
</tr>
<tr>
<td>202</td>
<td>133</td>
<td>47</td>
<td>10</td>
<td>12</td>
<td>12</td>
<td>20</td>
<td>4.71</td>
<td>1.34</td>
<td>0.38</td>
<td>0.41</td>
<td>30.03</td>
<td>10.09</td>
</tr>
<tr>
<td>206</td>
<td>149</td>
<td>39</td>
<td>11</td>
<td>13</td>
<td>11</td>
<td>20</td>
<td>4.89</td>
<td>2.01</td>
<td>0.50</td>
<td>0.39</td>
<td>32.18</td>
<td>12.91</td>
</tr>
</tbody>
</table>
Table 56: Computational results on randomly generated medium size LP problems for four different routines. \textit{Iter} is the number of iterations and \textit{Time} is the elapsed time in seconds on a Dell Latitude CPl 266XT running Matlab 5.3. The \textit{lpS} solver is the TOMLAB \textit{lpSolve}, and it is run with both Bland’s selection rule (iterations $I_{b}$, time $T_{b}$) and with the minimum cost selection rule (iterations $I_{m}$, time $T_{m}$). The \textit{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 $I_{l}$, 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.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$m$</th>
<th>$lpS$</th>
<th>$lpS$</th>
<th>$Minos$</th>
<th>$qopt$</th>
<th>$linprog$</th>
<th>$lpS$</th>
<th>$lpS$</th>
<th>$Minos$</th>
<th>$qopt$</th>
<th>$linprog$</th>
<th>$linprog$</th>
</tr>
</thead>
<tbody>
<tr>
<td>328</td>
<td>192</td>
<td>174</td>
<td>26</td>
<td>33</td>
<td>34</td>
<td>24</td>
<td>34.73</td>
<td>6.59</td>
<td>0.70</td>
<td>0.76</td>
<td>121.57</td>
<td>50.52</td>
</tr>
<tr>
<td>326</td>
<td>212</td>
<td>65</td>
<td>10</td>
<td>24</td>
<td>12</td>
<td>20</td>
<td>14.67</td>
<td>3.28</td>
<td>0.82</td>
<td>0.57</td>
<td>116.00</td>
<td>49.87</td>
</tr>
<tr>
<td>325</td>
<td>185</td>
<td>15</td>
<td>15</td>
<td>33</td>
<td>15</td>
<td>33</td>
<td>4.19</td>
<td>4.31</td>
<td>0.78</td>
<td>0.55</td>
<td>112.43</td>
<td>63.63</td>
</tr>
<tr>
<td>327</td>
<td>186</td>
<td>21</td>
<td>11</td>
<td>14</td>
<td>13</td>
<td>26</td>
<td>4.49</td>
<td>2.86</td>
<td>0.75</td>
<td>0.55</td>
<td>112.95</td>
<td>49.85</td>
</tr>
<tr>
<td>327</td>
<td>192</td>
<td>22</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>19</td>
<td>5.01</td>
<td>1.92</td>
<td>0.73</td>
<td>0.48</td>
<td>113.05</td>
<td>40.58</td>
</tr>
<tr>
<td>285</td>
<td>181</td>
<td>9</td>
<td>7</td>
<td>11</td>
<td>7</td>
<td>21</td>
<td>2.33</td>
<td>1.98</td>
<td>0.64</td>
<td>0.44</td>
<td>80.13</td>
<td>30.33</td>
</tr>
<tr>
<td>323</td>
<td>219</td>
<td>24</td>
<td>10</td>
<td>15</td>
<td>11</td>
<td>22</td>
<td>6.44</td>
<td>3.39</td>
<td>0.88</td>
<td>0.56</td>
<td>110.42</td>
<td>59.27</td>
</tr>
<tr>
<td>284</td>
<td>201</td>
<td>45</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>24</td>
<td>9.46</td>
<td>3.21</td>
<td>0.71</td>
<td>0.35</td>
<td>81.13</td>
<td>44.80</td>
</tr>
<tr>
<td>285</td>
<td>199</td>
<td>22</td>
<td>9</td>
<td>14</td>
<td>8</td>
<td>21</td>
<td>4.85</td>
<td>2.62</td>
<td>0.71</td>
<td>0.33</td>
<td>78.64</td>
<td>39.07</td>
</tr>
<tr>
<td>296</td>
<td>228</td>
<td>33</td>
<td>11</td>
<td>10</td>
<td>13</td>
<td>23</td>
<td>9.00</td>
<td>3.78</td>
<td>0.77</td>
<td>0.39</td>
<td>89.67</td>
<td>59.23</td>
</tr>
<tr>
<td>310</td>
<td>185</td>
<td>28</td>
<td>14</td>
<td>19</td>
<td>16</td>
<td>25</td>
<td>5.62</td>
<td>3.30</td>
<td>0.73</td>
<td>0.54</td>
<td>96.93</td>
<td>43.75</td>
</tr>
<tr>
<td>311</td>
<td>219</td>
<td>23</td>
<td>12</td>
<td>12</td>
<td>17</td>
<td>22</td>
<td>6.53</td>
<td>4.13</td>
<td>0.78</td>
<td>0.60</td>
<td>97.05</td>
<td>53.90</td>
</tr>
<tr>
<td>280</td>
<td>206</td>
<td>58</td>
<td>23</td>
<td>28</td>
<td>17</td>
<td>20</td>
<td>12.20</td>
<td>5.80</td>
<td>0.76</td>
<td>0.40</td>
<td>75.66</td>
<td>38.22</td>
</tr>
<tr>
<td>319</td>
<td>204</td>
<td>17</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>23</td>
<td>4.41</td>
<td>3.45</td>
<td>0.64</td>
<td>0.54</td>
<td>106.16</td>
<td>52.84</td>
</tr>
<tr>
<td>287</td>
<td>202</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>17</td>
<td>2.43</td>
<td>1.79</td>
<td>0.75</td>
<td>0.34</td>
<td>78.26</td>
<td>32.93</td>
</tr>
<tr>
<td>328</td>
<td>202</td>
<td>44</td>
<td>9</td>
<td>11</td>
<td>10</td>
<td>18</td>
<td>9.32</td>
<td>2.72</td>
<td>0.76</td>
<td>0.53</td>
<td>117.09</td>
<td>41.86</td>
</tr>
<tr>
<td>307</td>
<td>213</td>
<td>85</td>
<td>12</td>
<td>34</td>
<td>12</td>
<td>30</td>
<td>19.35</td>
<td>3.97</td>
<td>0.86</td>
<td>0.51</td>
<td>98.97</td>
<td>70.47</td>
</tr>
<tr>
<td>285</td>
<td>199</td>
<td>29</td>
<td>11</td>
<td>11</td>
<td>9</td>
<td>24</td>
<td>6.43</td>
<td>3.27</td>
<td>0.71</td>
<td>0.47</td>
<td>78.32</td>
<td>44.30</td>
</tr>
<tr>
<td>315</td>
<td>194</td>
<td>22</td>
<td>10</td>
<td>8</td>
<td>9</td>
<td>20</td>
<td>5.14</td>
<td>3.00</td>
<td>0.73</td>
<td>0.52</td>
<td>102.28</td>
<td>41.73</td>
</tr>
<tr>
<td>310</td>
<td>181</td>
<td>38</td>
<td>6</td>
<td>7</td>
<td>5</td>
<td>22</td>
<td>6.95</td>
<td>1.80</td>
<td>0.71</td>
<td>0.46</td>
<td>96.99</td>
<td>36.93</td>
</tr>
<tr>
<td>308</td>
<td>200</td>
<td>39</td>
<td>11</td>
<td>16</td>
<td>12</td>
<td>23</td>
<td>8.68</td>
<td>3.36</td>
<td>0.75</td>
<td>0.50</td>
<td>98.18</td>
<td>47.20</td>
</tr>
</tbody>
</table>

195
Table 57: Computational results on randomly generated medium size LP problems for four different routines. \( \text{Iter} \) is the number of iterations and \( \text{Time} \) 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 \( I_{b} \), time \( T_{b} \)) and with the minimum cost selection rule (iterations \( I_{m} \), time \( T_{m} \)). The \( \text{linprog} \) 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 \( I_{l} \), 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 | \( I_{b} \) | \( I_{m} \) | \( \text{Iter} \) | \( \text{Iter} \) | \( I_{l} \) | \( T_{b} \) | \( T_{m} \) | \( \text{Time} \) | \( T_{m} \) | \( I_{l} \) |
| 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 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 $I_b$, time $T_b$) and with the minimum cost selection rule (iterations $I_m$, 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 $I_l$, 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.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$m$</th>
<th>$I_b$</th>
<th>$I_m$</th>
<th>Iter</th>
<th>Iter</th>
<th>$I_l$</th>
<th>$I_m$</th>
<th>Time</th>
<th>Time</th>
<th>$I_l$</th>
<th>$I_m$</th>
<th>$T_m$</th>
<th>$T_m$</th>
<th>$T_l$</th>
<th>$T_l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>528</td>
<td>232</td>
<td>35</td>
<td>7</td>
<td>6</td>
<td>28</td>
<td>12.33</td>
<td>3.50</td>
<td>1.28</td>
<td>0.86</td>
<td>453.03</td>
<td>124.19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>482</td>
<td>252</td>
<td>33</td>
<td>9</td>
<td>7</td>
<td>25</td>
<td>12.02</td>
<td>4.26</td>
<td>1.00</td>
<td>0.71</td>
<td>346.37</td>
<td>120.24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>503</td>
<td>251</td>
<td>72</td>
<td>15</td>
<td>38</td>
<td>17</td>
<td>25.45</td>
<td>6.79</td>
<td>1.49</td>
<td>1.01</td>
<td>387.91</td>
<td>170.35</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>507</td>
<td>259</td>
<td>142</td>
<td>18</td>
<td>46</td>
<td>27</td>
<td>50.68</td>
<td>8.55</td>
<td>1.43</td>
<td>1.33</td>
<td>397.67</td>
<td>147.41</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>487</td>
<td>240</td>
<td>48</td>
<td>17</td>
<td>33</td>
<td>19</td>
<td>16.69</td>
<td>7.02</td>
<td>1.29</td>
<td>1.03</td>
<td>346.64</td>
<td>114.96</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>506</td>
<td>251</td>
<td>46</td>
<td>8</td>
<td>11</td>
<td>8</td>
<td>16.92</td>
<td>4.19</td>
<td>1.13</td>
<td>0.78</td>
<td>394.38</td>
<td>119.71</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>504</td>
<td>256</td>
<td>35</td>
<td>9</td>
<td>16</td>
<td>8</td>
<td>14.73</td>
<td>4.97</td>
<td>1.26</td>
<td>0.81</td>
<td>395.37</td>
<td>183.20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>489</td>
<td>255</td>
<td>36</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>14.39</td>
<td>11.87</td>
<td>1.32</td>
<td>1.30</td>
<td>355.66</td>
<td>129.45</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>514</td>
<td>228</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3.24</td>
<td>1.80</td>
<td>1.05</td>
<td>0.51</td>
<td>399.44</td>
<td>133.82</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>524</td>
<td>245</td>
<td>64</td>
<td>11</td>
<td>27</td>
<td>14</td>
<td>21.99</td>
<td>5.34</td>
<td>1.26</td>
<td>1.00</td>
<td>439.31</td>
<td>135.32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>506</td>
<td>255</td>
<td>112</td>
<td>22</td>
<td>28</td>
<td>23</td>
<td>40.12</td>
<td>10.07</td>
<td>1.12</td>
<td>1.21</td>
<td>385.12</td>
<td>117.49</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>497</td>
<td>224</td>
<td>50</td>
<td>11</td>
<td>14</td>
<td>12</td>
<td>3.51</td>
<td>4.57</td>
<td>1.11</td>
<td>0.86</td>
<td>362.38</td>
<td>121.94</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>482</td>
<td>249</td>
<td>27</td>
<td>16</td>
<td>17</td>
<td>20</td>
<td>10.24</td>
<td>6.75</td>
<td>1.15</td>
<td>1.08</td>
<td>339.27</td>
<td>138.16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>485</td>
<td>249</td>
<td>18</td>
<td>6</td>
<td>21</td>
<td>5</td>
<td>6.36</td>
<td>2.87</td>
<td>1.35</td>
<td>0.55</td>
<td>340.35</td>
<td>95.15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>509</td>
<td>223</td>
<td>84</td>
<td>22</td>
<td>35</td>
<td>17</td>
<td>23.51</td>
<td>7.55</td>
<td>1.17</td>
<td>1.04</td>
<td>390.88</td>
<td>142.31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>506</td>
<td>224</td>
<td>38</td>
<td>12</td>
<td>11</td>
<td>14</td>
<td>11.89</td>
<td>4.65</td>
<td>1.09</td>
<td>0.94</td>
<td>383.13</td>
<td>132.21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>511</td>
<td>241</td>
<td>115</td>
<td>10</td>
<td>36</td>
<td>9</td>
<td>36.51</td>
<td>4.32</td>
<td>1.29</td>
<td>0.69</td>
<td>390.78</td>
<td>122.23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>497</td>
<td>230</td>
<td>78</td>
<td>23</td>
<td>43</td>
<td>12</td>
<td>23.60</td>
<td>8.27</td>
<td>1.29</td>
<td>0.75</td>
<td>362.08</td>
<td>109.30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>514</td>
<td>226</td>
<td>84</td>
<td>21</td>
<td>42</td>
<td>26</td>
<td>25.10</td>
<td>7.90</td>
<td>1.57</td>
<td>1.47</td>
<td>407.94</td>
<td>126.53</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>511</td>
<td>268</td>
<td>59</td>
<td>10</td>
<td>30</td>
<td>9</td>
<td>24.74</td>
<td>5.76</td>
<td>1.43</td>
<td>0.94</td>
<td>385.56</td>
<td>161.65</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>503</td>
<td>243</td>
<td>59</td>
<td>14</td>
<td>25</td>
<td>14</td>
<td>20.30</td>
<td>6.05</td>
<td>1.26</td>
<td>0.94</td>
<td>383.16</td>
<td>132.28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 $I_b$, time $T_b$) and with the minimum cost selection rule (iterations $I_m$, 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 $I_l$, 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.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$m$</th>
<th>$I_b$</th>
<th>$I_m$</th>
<th>Iter</th>
<th>Iter</th>
<th>$I_l$</th>
<th>$I_m$</th>
<th>Time</th>
<th>Time</th>
<th>$I_l$</th>
<th>$I_m$</th>
<th>$T_m$</th>
<th>$T_m$</th>
<th>$T_l$</th>
<th>$T_l$</th>
</tr>
</thead>
<tbody>
<tr>
<td>103</td>
<td>45</td>
<td>19</td>
<td>8</td>
<td>17</td>
<td>7</td>
<td>13</td>
<td>0.84</td>
<td>0.52</td>
<td>0.30</td>
<td>0.30</td>
<td>5.70</td>
<td>1.23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>49</td>
<td>39</td>
<td>11</td>
<td>13</td>
<td>11</td>
<td>20</td>
<td>12.99</td>
<td>2.01</td>
<td>0.50</td>
<td>0.39</td>
<td>32.18</td>
<td>12.91</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>308</td>
<td>200</td>
<td>39</td>
<td>11</td>
<td>16</td>
<td>12</td>
<td>23</td>
<td>8.68</td>
<td>3.36</td>
<td>0.75</td>
<td>0.50</td>
<td>98.18</td>
<td>47.20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>402</td>
<td>243</td>
<td>47</td>
<td>10</td>
<td>14</td>
<td>10</td>
<td>24</td>
<td>15.00</td>
<td>4.46</td>
<td>0.98</td>
<td>0.63</td>
<td>204.99</td>
<td>94.11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>503</td>
<td>243</td>
<td>59</td>
<td>14</td>
<td>25</td>
<td>14</td>
<td>29</td>
<td>20.30</td>
<td>6.05</td>
<td>1.26</td>
<td>0.94</td>
<td>383.16</td>
<td>132.28</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
References


