TOMLAB v1.0 User’s Guide

Kenneth Holmström, Mattias Björkman and Erik Dotzauer

Applied Optimization and Modeling Group (TOM) ²

Center for Mathematical Modeling
Department of Mathematics and Physics
Mälardalen University
P.O. Box 883, SE-721 23 Västerås, Sweden

Research Report in MATHEMATICS / APPLIED MATHEMATICS
Technical Report IMa-TOM-1999-01

April 26, 1999

KEYWORDS: MATLAB, Optimization, Mathematical Software, Algorithms, Nonlinear Least Squares

²The TOM home page is http://www ima mdh se/tom.
Contents

1 The TOMLAB Environment .................................................. 7
  1.1 Basic Questions About TOMLAB ................................... 7
  1.2 Background ................................................................. 7
  1.3 Installation of TOMLAB ............................................... 8
  1.4 Installation of NLPLIB TB ........................................... 8
    1.4.1 Installation on PC systems .................................... 8
    1.4.2 Installation on UNIX systems ................................. 8
  1.5 Installation of OPERA TB ............................................ 8
    1.5.1 Installation on PC systems .................................... 9
    1.5.2 Installation on UNIX systems ................................. 9
  1.6 Using Matlab 5.0 or 5.1 .............................................. 9

2 NLPLIB TB ........................................................................ 10
  2.1 The Design of NLPLIB ................................................. 10
    2.1.1 Global Variables .................................................. 29
  2.2 Solver Routines in NLPLIB TB ....................................... 25
  2.3 Utility Routines in NLPLIB TB ....................................... 28
    2.3.1 Low Level Routines and Test Problems .................... 28
    2.3.2 Test Routines for the System ............................... 32
  2.4 The Menu Systems ....................................................... 32
  2.5 The Graphical User Interface ........................................ 32
    2.5.1 The Advanced Mode ............................................. 34
  2.6 How to Define Optimization Problems in NLPLIB TB ....... 46
    2.6.1 Defining Unconstrained Problems ......................... 46
    2.6.2 Defining Box-bounded Global Optimization Problems ... 42
    2.6.3 Defining Nonlinear Least Squares Problems ............. 43
    2.6.4 Defining Constrained Problems ............................. 44
    2.6.5 Defining Global Mixed-Integer Nonlinear Programming Problems .... 46
    2.6.6 Defining Constrained Nonlinear Least Squares Problems 47
    2.6.7 Defining Quadratic Problems ................................. 48
    2.6.8 Defining Exponential Sum Fitting Problems ............. 49
    2.6.9 Defining Problems in Own Problem Definition Files .... 50
    2.6.10 Special Notes ....................................................... 52
  2.7 How to Solve Optimization Problems Using NLPLIB TB ...... 54
    2.7.1 Using the Driver Routines .................................... 54
    2.7.2 Direct Call to an Optimization Routine ................... 56
    2.7.3 A Direct Approach to a QP Solution ........................ 56
  2.8 Printing Utilities and Print Levels ................................. 57
  2.9 Notes about Special Features ....................................... 58
    2.9.1 Approximation of Derivatives ............................... 58
    2.9.2 Partially Separable Functions ............................... 58

3 Additional Information .................................................... 61
  3.1 TOMLAB Documentation .............................................. 61
  3.2 TOMLAB Team ............................................................ 61
  3.3 Future Development .................................................... 61
  3.4 Acknowledgments ........................................................ 61

4 Additional TOMLAB Information ........................................ 62
  4.1 Downloading TOMLAB .................................................. 62
  4.2 TOMLAB Contact Information ....................................... 62

5 Appendices ......................................................................... 63
  5.1 Specialized TOMLAB Commands ................................. 63
  5.2 Bibliography .................................................................. 63

6 Glossary ............................................................................ 64

7 Index .................................................................................. 65
2.9.3  Recursive solver calls ......................................................... 60

2.10  Driver Routines in NLPLIB TB ................................................. 61
  2.10.1  clsRun ................................................................. 61
  2.10.2  conRun ................................................................. 61
  2.10.3  glbRun ................................................................. 62
  2.10.4  gleRun ................................................................. 63
  2.10.5  lsRun ................................................................. 64
  2.10.6  qpRUn ................................................................. 65
  2.10.7  ucRUn ................................................................. 66

2.11  Optimization Routines in NLPLIB TB ....................................... 67
  2.11.1  clsSolve ............................................................. 67
  2.11.2  conSolve ............................................................. 67
  2.11.3  glbSolve ............................................................. 70
  2.11.4  gleSolve ............................................................. 71
  2.11.5  glbSolve ............................................................. 73
  2.11.6  gleSolve ............................................................. 75
  2.11.7  lsSolve ............................................................. 76
  2.11.8  nlpSolve ............................................................. 78
  2.11.9  qpe ................................................................. 79
  2.11.10  qpBiggs ........................................................... 80
  2.11.11  qplm .............................................................. 81
  2.11.12  qplSolve .......................................................... 81
  2.11.13  s1trustR .......................................................... 82
  2.11.14  ucSolve .......................................................... 84

2.12  Optimization Subfunction Utilities in NLPLIB TB ....................... 85
  2.12.1  intpol2 ............................................................ 85
  2.12.2  intpol3 ............................................................ 85
  2.12.3  itrr ............................................................... 86
  2.12.4  LineSearch ......................................................... 87
  2.12.5  preSolve ........................................................... 88

2.13  User Utility Functions in NLPLIB TB ...................................... 88
  2.13.1  PrintResult ......................................................... 88
  2.13.2  PrintSolvers ....................................................... 89
  2.13.3  runtest ............................................................ 89
  2.13.4  systest ............................................................ 90

3  OPERA TB ................................................................. 91

  3.1  Optimization Algorithms and Solvers in OPERA TB .................... 91
  3.1.1  Linear Programming ................................................. 91
  3.1.2  Transportation Programming ......................................... 93
  3.1.3  Network Programming ............................................... 93
  3.1.4  Integer Programming ................................................ 94
3.1.5 Dynamic Programming .................................................. 94
3.1.6 Lagrangian Relaxation .................................................. 94
3.1.7 Utility Routines ......................................................... 95
3.2 How to Solve Optimization Problems Using OPERA TB .................... 96
  3.2.1 How to Solve Linear Programming Problems .......................... 96
  3.2.2 How to Solve Transportation Programming Problems .............. 102
  3.2.3 How to Solve Network Programming Problems ...................... 102
  3.2.4 How to Solve Integer Programming Problems ....................... 103
  3.2.5 How to Solve Dynamic Programming Problems ..................... 104
  3.2.6 How to Solve Lagrangian Relaxation Problems ....................... 106
3.3 Printing Utilities and Print Levels ...................................... 107
3.4 Driver Routines in OPERA TB .......................................... 107
  3.4.1 lpRun ................................................................. 107
3.5 Optimization Routines in OPERA TB .................................... 108
  3.5.1 akarmark ............................................................ 108
  3.5.2 balas ............................................................... 109
  3.5.3 cutplane ........................................................... 111
  3.5.4 dijkstra ............................................................. 111
  3.5.5 dpinvent ........................................................... 111
  3.5.6 dpknapsack ......................................................... 112
  3.5.7 DualSolve ......................................................... 112
  3.5.8 karmark ............................................................. 114
  3.5.9 krelax .............................................................. 115
  3.5.10 labelcor ............................................................ 116
  3.5.11 lpduel ............................................................. 116
  3.5.12 lpkarma ............................................................ 117
  3.5.13 lpsimp1 ........................................................... 118
  3.5.14 lpsimp2 ........................................................... 118
  3.5.15 lpSolve ........................................................... 119
  3.5.16 maxflow ........................................................... 120
  3.5.17 minSolve .......................................................... 121
  3.5.18 modlabel .......................................................... 122
  3.5.19 NWSimplex ......................................................... 123
  3.5.20 Phase1Simplex ..................................................... 123
  3.5.21 Phase2Simplex ..................................................... 125
  3.5.22 salesman .......................................................... 126
  3.5.23 TPSimplex ........................................................ 126
  3.5.24 traveling .......................................................... 127
  3.5.25 urelax ............................................................. 128
3.6 Optimization Subfunction Utilities in OPERA TB ......................... 129
  3.6.1 a2frstar ........................................................... 129
  3.6.2 gsearch ............................................................ 129
3.6.3 gsearchq .................................................... 136
3.6.4 mintree ........................................................... 136
3.6.5 TPmce ............................................................ 131
3.6.6 TPnw .............................................................. 131
3.6.7 TPvogel ......................................................... 132
3.6.8 z2frstar ........................................................... 132
3.7 User Utility Functions in OPERA TB ................................ 133
3.7.1 cpTransf .......................................................... 133

4 Interfaces 134
4.1 The MEX-file Interface ........................................ 134
4.2 The Matlab Optimization Toolbox Interface ....................... 134
4.3 The CUTE Interface .............................................. 134
4.4 The AMPL Interface ............................................. 135

A Description of Algorithms in NLPLIB TB 136
A.1 clsSolve .......................................................... 136
   A.1.1 Convergence criterias ....................................... 138
   A.1.2 Stop criterias .................................................. 138
   A.1.3 Computation of Search Direction .................................. 138
   A.1.4 Update Procedure .............................................. 139
A.2 glibSolve .......................................................... 140
   A.2.1 conhull .......................................................... 141
   A.2.2 next ............................................................. 141
   A.2.3 pred ............................................................ 142
A.3 intpol2 ............................................................. 142
A.4 intpol3 ............................................................. 142
A.5 LineSearch ......................................................... 143
   A.5.1 Bracketing Phase ............................................. 143
   A.5.2 Sectioning Phase ............................................. 144
A.6 lsSolve ............................................................ 144
   A.6.1 Convergence criterias ......................................... 146
   A.6.2 Stop criterias .................................................... 146
   A.6.3 Computation of Search Direction .................................. 146
   A.6.4 Update Procedure .............................................. 146
A.7 ucSolve ............................................................. 148
   A.7.1 Convergence criterias ......................................... 150
   A.7.2 Stop criterias .................................................... 150
   A.7.3 Computation of Search Direction .................................. 150
   A.7.4 Update Procedure .............................................. 151

B Description of Algorithms in OPERA TB 153
B.1 akarmark ........................................................... 153
B.2 cutplane ................................................................. 153
B.3 dijkstra ................................................................. 154
B.4 dpinvent ................................................................. 155
B.5 dpknap ................................................................. 155
B.6 gsearch ................................................................. 156
B.7 gsearchq ................................................................. 156
B.8 karmark ................................................................. 156
B.9 ksrelax ................................................................. 157
B.10 labelcor ................................................................. 158
B.11 lp dual ................................................................. 158
B.12 lp karma ................................................................. 159
B.13 lpsimp1 ................................................................. 160
B.14 lpsimp2 ................................................................. 160
B.15 maxflow ................................................................. 161
B.16 modlabel ............................................................... 162
B.17 mintree ................................................................. 162
B.18 TP mc ................................................................. 162
B.19 TP nw ................................................................. 162
B.20 TPSimplex ........................................................... 163
B.21 TPvogel ............................................................... 164
B.22 urelax ................................................................. 165
1 The TOMLAB Environment

In this section the main features of TOMLAB are presented. This will include some frequently asked questions, stated and answered in Section 1.1, and its historical background, outlined in Section 1.2. The installation of its two major parts, the NLPLIB TB toolbox and the OPERA TB toolbox, are discussed in Section 1.4 and Section 1.5, respectively.

1.1 Basic Questions About TOMLAB

What is TOMLAB? TOMLAB is a general purpose, open and integrated development environment in Matlab for research and teaching in optimization. The main paper on TOMLAB is [33]. The main parts of TOMLAB is the toolboxes NLPLIB TB and OPERA TB.

What is NLPLIB TB? NLPLIB TB is a Matlab toolbox for nonlinear programming and parameter estimation, presented in [34].

What is OPERA TB? OPERA TB is a Matlab toolbox for linear and discrete optimization, presented in [35].

Why should I use TOMLAB? TOMLAB gives you easy access to a large set of standard test problems, optimization solvers and utilities. Furthermore, you can easily define your own problems and try to solve them using any solver. The basic design principle in TOMLAB is: Define your problem once, run all available solvers.

Can I reach other program packages using TOMLAB? Yes, by use of the TOMLAB MEX-file interfaces it is possible to call general-purpose solvers implemented in Fortran or C. It is also possible to call solvers in the Matlab Optimization Toolbox. Furthermore, using the MEX-file interfaces, problems in the CUTE test problem data base and problems defined in the AMPL modeling language can be solved.

How do I solve a problem using TOMLAB? You can solve a problem either by a direct call to a solver or a general multi-solver driver routine, or interactively, using a graphical user interface (GUI) [17] or a menu system.

1.2 Background

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 the motive for starting the development of TOMLAB: a general-purpose, open and integrated development environment in Matlab for research and teaching in optimization.

To solve optimization problems, traditionally the user has been forced to write a Fortran code that calls some standard solver written as a Fortran subroutine. For nonlinear problems, the user must also write subroutines computing the objective function value and the vector of constraint function values. The needed derivatives are either explicitly coded, computed by using numerical differences or derived using automatic differentiation techniques.

In recent years several modeling languages are developed, like AIMMS [8], AMPL [24], ASCEND [46], GAMS [9, 14] and LINGO [1]. The modeling system acts as a preprocessor. The user describes the details of his problem in a very verbal language; an opposite to the concise mathematical description of the problem. The problem description file is normally modified in a text editor, with help from example files solving the same type of problem. Much effort is directed to the development of more user friendly interfaces. The model system processes the input description file and calls any of the available solvers. For a solver to be accessible in the modeling system, special types of interfaces are developed.

The modeling language approach is suitable for many management and decision problems, but may not always be the best way for engineering problems, which often are nonlinear with a complicated problem description. Until recently, the support for nonlinear problems in the modeling languages has been crude. This is now rapidly changing [18].

For people with a mathematical background, modeling languages often seems to be a very tedious way to define an optimization problem. There has been several attempts to find languages more suitable than Fortran or C/C++ to describe mathematical problems, like the compact and powerful APL language [37, 47]. Nowadays, languages like Matlab has a rapid growth of users. Matlab was originally created [43] as a preprocessor to the standard Fortran subroutine libraries in numerical linear algebra, LINPACK [16] and EISPACK [51] [25], 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 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.

### 1.3 Installation of TOMLAB

The normal distribution of TOMLAB includes NLPLIB TB and OPERA TB and some extra sub directories described in the file contents.m in the main TOMLAB directory. This directory also includes a file tomlab.m, which describes the installation. There are two options. Either the Matlab search paths for TOMLAB should be made permanent or set temporarily for each run of TOMLAB. To make the Matlab search path permanent, either the file startup.m should be edited or the user may set the search paths according to the general instructions given by Math Works, Inc. To make temporarily search paths, the easiest way is to start Matlab, go to the TOMLAB main directory, and call findpath. If, for example, on a PC, TOMLAB is installed in \matlab\tomlab, execute

```bash
cd c:\matlab\tomlab
findpath
```

If you are using an old Matlab version, see the installation instructions for NLPLIB TB and OPERA TB below.

The normal distribution of TOMLAB does not include the DLL files for CUTE, AMPL and the MEX solvers that are needed on PC systems, neither the code to generate these files on Unix systems. Contact the authors if any of these options are needed.

### 1.4 Installation of NLPLIB TB

If NLPLIB TB is installed as a stand-alone toolbox, the routines Phase1Simplex, Phase2Simplex, lpDef, mPrint, printmat, xprint, xprinte and xprinti must be included from OPERA TB.

#### 1.4.1 Installation on PC systems

NLPLIB TB is normally installed as part of TOMLAB, with the subpath \tomlab\nlplib. On PC systems a normal choice of full path is \matlab\tomlab\nlplib or \matlab\toolbox\tomlab\nlplib. This path must be added to the Matlab search path. Before starting a session running NLPLIB TB, call nlplibInit, which sets the number of output characters per row used and declares nearly all the global variables. If the user has a screen with less than 120 columns, the variable MAXCOLS in nlplibInit should be changed to the correct number.

#### 1.4.2 Installation on UNIX systems

NLPLIB TB is normally installed as part of TOMLAB, with the subpath /tomlab/nlplib. A possible full path is /home/tomlab/nlplib or /home/matlab/toolbox/tomlab/nlplib. This path must be added to the Matlab search path. Before starting a session running NLPLIB TB, call nlplibInit, which sets the number of output characters per row used and declares nearly all the global variables. If the user has a screen with less than 120 columns, the variable MAXCOLS in nlplibInit should be changed to the correct number.

### 1.5 Installation of OPERA TB

If OPERA TB is installed as a stand-alone toolbox (not recommended), the routines inputR, inputSet, optParamDef, optParamSet, backsol and qoptions must be included from NLPLIB TB. The LP multi-driver routine lpRun, the LP menu program lpOpt, and the solvers lpSolve and DualSolve will not work without NLPLIB TB.
1.5.1 Installation on PC systems

OPERA TB is normally installed as part of TOMLAB, with the subpath \tomlab\opera. On PC systems a normal choice is \matlab\tomlab\opera or \matlab\toolbox\tomlab\opera. This path must be added to the Matlab search path. Before starting a session running OPERA TB, call `operaInit`, which sets the number of output characters per row used and declares nearly all the global variables. If the user has a screen with less than 120 columns, the variable `MAXCOLS` in `operaInit` should be changed to the correct number.

The example files are stored in a separate directory, \tomlab\operdemo. The full path should be added to the Matlab search path. As a possible alternative you can move to this directory when you want to run these files.

1.5.2 Installation on UNIX systems

OPERA TB is normally installed as part of TOMLAB, with the subpath /tomlab/opera. A possible full path is /home/tomlab/opera or /home/matlab/toolbox/tomlab/opera. This path must be added to the Matlab search path. Before starting a session running OPERA TB, call `operaInit`, which sets the number of output characters per row used and declares nearly all the global variables. If the user has a screen with less than 120 columns, the variable `MAXCOLS` in `operaInit` should be changed to the correct number.

The example files are stored in a separate directory, usually in a directory /home/tomlab/operdemo or /home/matlab/toolbox/tomlab/operdemo. The full path could be added to the Matlab search path. As a possible alternative you can move to this directory when you want to run these files.

1.6 Using Matlab 5.0 or 5.1

Are you running TOMLAB under Matlab 5.0 or 5.1?

If running on PC then the directory `matlab5.1` must be put before the directories `nlplib` and `opera` in the Matlab search path. This could be done by calling the routine `bug51`.

If running on Unix then the directory `unix5.1` must be put before the directories `nlplib` and `opera` in the Matlab search path. This could be done by calling the routine `unix51`.

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

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

The `unix5.1` directory contains one routine, `strempi`.
2 NLPLIB TB

NLPLIB TB is a Matlab toolbox for nonlinear programming and parameter estimation and gives you easy access to a large set of standard test problems, optimization solvers and utilities. Furthermore, you can easily define your own problems and try to solve them using any solver.

In the following subsections, NLPLIB TB is presented. In Section 2.1, its design and basic structure are discussed. Section 2.2 - Section 2.3.1 gives an overview of the implemented solver and utility routines. The menu system is presented in Section 2.4 and the Graphical User Interface in Section 2.5. How to define new problems is described in Section 2.6 and a description of how to solve a problem is given in Section 2.7. The possible amount of print output is discussed in Section 2.8. Finally, detailed descriptions of all implemented routines are given in Section 2.10 - 2.13.

2.1 The Design of NLPLIB

In this section we discuss the design of NLPLIB TB. As the scope of NLPLIB TB is large and broad, there is a clear need of a well-designed system. It is also necessary to use the power of the Matlab language, to make the system flexible and easy to use and maintain. We have used the concept of structure arrays and made heavy use of both the ability in Matlab to execute Matlab code defined as string expressions and to execute functions specified by a string.

Currently NLPLIB TB consists of about 48000 lines of m-file code in more than 265 files with algorithms, utilities and predefined problems. This motivates a well-defined naming convention and design.

NLPLIB TB solves a number of different types of optimization problems. Currently, we have defined the types listed in Table 1. The global variable probType is the current type to be solved. An optimization solver is defined to be of type solveType, where solveType is any of the probType entries in Table 1. It is clear that a solver of a certain solveType 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 and constrained nonlinear least squares problems.

<table>
<thead>
<tr>
<th>probType</th>
<th>Number</th>
<th>Description of the type of 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>exp</td>
<td>5</td>
<td>Exponential fitting problems.</td>
</tr>
<tr>
<td>cls</td>
<td>6</td>
<td>Constrained nonlinear least squares problems.</td>
</tr>
<tr>
<td>nts</td>
<td>7</td>
<td>Nonlinear time series.</td>
</tr>
<tr>
<td>lp</td>
<td>8</td>
<td>Linear programming.</td>
</tr>
<tr>
<td>gilb</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>
</tbody>
</table>

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. The probSet use is defined in order to make the inclusion of a few optimization problems of any type a simple and fast task. This method is to prefer when NLPLIB TB is used in optimization courses.

A flow-sheet of the process of optimization in NLPLIB TB is shown in Figure 1. Normally, a single optimization problem is solved running any of the menu systems (one for each solveType), 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. 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.
Figure 1: The process of optimization in TOMLAB.
Table 2: Defined test problem sets in TOMLAB.

<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>exp</td>
<td>5</td>
<td>Exponential fitting problems.</td>
</tr>
<tr>
<td>cls</td>
<td>6</td>
<td>Linear constrained nonlinear least squares problems.</td>
</tr>
<tr>
<td>nls</td>
<td>6</td>
<td>Nonlinear constrained nonlinear least squares 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>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>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>
<tr>
<td>uts</td>
<td>7</td>
<td>Nonlinear time series.</td>
</tr>
<tr>
<td>usr</td>
<td>1-9</td>
<td>User defined problems of probType 1-9.</td>
</tr>
</tbody>
</table>

i.e. for the menu systems, driver routines etc. Table 3 shows the naming convention. The names of the problem setup routine 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 4. An example, illustrating the constrained nonlinear programming case (solveType = con, probSet = con) is shown in Figure 2.

Table 3: Names of main m-file functions in NLPLIB TB.

<table>
<thead>
<tr>
<th>Generic variable</th>
<th>Purpose (solveType is ①, e.g. ①=con)</th>
</tr>
</thead>
<tbody>
<tr>
<td>⑨Opt</td>
<td>Menu program.</td>
</tr>
<tr>
<td>⑨Run</td>
<td>Multi-solver optimization driver routine.</td>
</tr>
<tr>
<td>⑨Def</td>
<td>Routine defining optimization parameters.</td>
</tr>
<tr>
<td>⑨Solve</td>
<td>(Prototype) solver.</td>
</tr>
</tbody>
</table>

The problem setup routine 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 5. 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.

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.

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 15. The use of structure arrays make advanced result presentation and statistics possible.

The field xState describes the state of each of the variables. In Table 16 the different values are described. The different conditions for linear constraints are defined by the state variable in field bState. In Table 17 the different values are described.

To conclude, the system design is flexible and easy to expand in many different ways.
Figure 2: Solution of constrained nonlinear problems in TOMLAB.
Table 4: Names on the low level m-files in NLPLIB TB.

<table>
<thead>
<tr>
<th>Generic name</th>
<th>Purpose ( ⬤ is any probSet, e.g. ⬤=amp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>⬤.prob</td>
<td>Define string matrix with problems and a structure prob for each 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, ∂c(x)/dx.</td>
</tr>
<tr>
<td>⬤.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 ).</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_i r_i(x) \partial^2 r_i(x)/dx^2 ).</td>
</tr>
</tbody>
</table>
Table 5: Information stored in the problem structure Prob.

<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, see Table 1.</td>
</tr>
<tr>
<td>probFile</td>
<td>Name of m-file in which problem are defined.</td>
</tr>
<tr>
<td>rName</td>
<td>Name of each decision variable</td>
</tr>
<tr>
<td>cName</td>
<td>Name of each general constraint.</td>
</tr>
<tr>
<td>optParam</td>
<td>Structure with special fields for optimization parameters, see Table 6</td>
</tr>
<tr>
<td>Solver</td>
<td>Structure with fields Name and Alg. Name is the name of the solver and Alg is the solver algorithm to be used. See the solver descriptions Section 2.11.</td>
</tr>
<tr>
<td>uP</td>
<td>User supplied parameters for the problem.</td>
</tr>
<tr>
<td>uPName</td>
<td>Problem name connected to the user supplied parameters.</td>
</tr>
<tr>
<td>ExpFit</td>
<td>Structure with special fields for exponential fitting problems, see Table 7</td>
</tr>
<tr>
<td>QP</td>
<td>Structure with special fields for quadratic problems, see Table 8.</td>
</tr>
<tr>
<td>NLLS</td>
<td>Structure with special fields for nonlinear least squares, see Table 9.</td>
</tr>
<tr>
<td>NTS</td>
<td>Structure with special fields for nonlinear time series, see Table 10</td>
</tr>
<tr>
<td>PartSep</td>
<td>Structure with special fields for partially separable functions, see Table 11.</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>Structure with special fields for global optimization, see Table 12.</td>
</tr>
<tr>
<td>A</td>
<td>Constraint matrix for linear constraints, one constraint per row.</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>r.L</td>
<td>Lower bounds on the variables.</td>
</tr>
<tr>
<td>r.U</td>
<td>Upper bounds on the variables.</td>
</tr>
<tr>
<td>r.0</td>
<td>Starting point.</td>
</tr>
<tr>
<td>N</td>
<td>Problem dimension (number of variables).</td>
</tr>
<tr>
<td>f.Low</td>
<td>Lower bound on function value. Used in line search by Fletcher, default, $-\text{realmax}=-1.7977E+308$.</td>
</tr>
<tr>
<td>r.opt</td>
<td>Optimal point $x^*$ (if known).</td>
</tr>
<tr>
<td>f.opt</td>
<td>Optimal objective function value $f(x^*)$.</td>
</tr>
<tr>
<td>AutoDiff</td>
<td>If true, use automatic differentiation.</td>
</tr>
<tr>
<td>NumDiff</td>
<td>Numerical approximation of derivatives. If set to 1, classical approach with forward or backward differences together with automatic step selection will be used. If set to 2, 3 or 4 the spline routines csapi, csaps or spaps in SPLINE Toolbox will be used. If set to 5, derivatives will be estimated by use of complex variables.</td>
</tr>
<tr>
<td>p.f</td>
<td>Name of gateway routine computing the objective function $f(x)$.</td>
</tr>
<tr>
<td>p.g</td>
<td>Name of gateway routine computing the gradient vector $g(x)$.</td>
</tr>
<tr>
<td>p.H</td>
<td>Name of gateway routine computing the Hessian matrix $H(x)$.</td>
</tr>
<tr>
<td>p.c</td>
<td>Name of gateway routine computing the vector of constraint functions $c(x)$.</td>
</tr>
<tr>
<td>p.dc</td>
<td>Name of gateway routine computing the matrix of constraint normals $\partial c(x)/\partial x$.</td>
</tr>
<tr>
<td>p.d2c</td>
<td>Name of gateway routine computing the 2nd part of 2nd derivative matrix of the Lagrangian function, $\sum_i \lambda_i \partial^2 c(x)/\partial x^2$.</td>
</tr>
<tr>
<td>p.r</td>
<td>Name of gateway routine computing the residual vector $r(x)$.</td>
</tr>
<tr>
<td>p.J</td>
<td>Name of gateway routine computing the Jacobian matrix $J(x)$.</td>
</tr>
<tr>
<td>p.d2r</td>
<td>Name of gateway routine computing the second part of the Hessian for a nonlinear least squares problem, i.e. $\sum_i r_i(x) \partial^2 r_i(x)/\partial x_i \partial x_k$.</td>
</tr>
<tr>
<td>USER</td>
<td>Structure with user defined names of the m-files computing the objective, gradient, Hessian etc. See Table 13. These routines are called from the corresponding gateway routine.</td>
</tr>
<tr>
<td>r.min</td>
<td>Lower plot region parameters.</td>
</tr>
<tr>
<td>r.max</td>
<td>Upper plot region parameters.</td>
</tr>
</tbody>
</table>
Table 6: Information stored in the structure **Prob.optParam**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alg</td>
<td>Optimization Algorithm. Dependent on type of problem. Default 0.</td>
</tr>
<tr>
<td>method</td>
<td>Solver sub-method technique. Default 0.</td>
</tr>
<tr>
<td>PriLever</td>
<td>Print level in optimization solver, default 1.</td>
</tr>
<tr>
<td>eps_(x)</td>
<td>Convergence tolerance in optimal solution (x), distance between successive (x,</td>
</tr>
<tr>
<td>eps_(f)</td>
<td>Convergence tolerance on (f). Also used when testing on the directed derivative, default (10^{-8}).</td>
</tr>
<tr>
<td>eps_dirg</td>
<td>Convergence tolerance on the directed derivative, default (10^{-8}).</td>
</tr>
<tr>
<td>eps_c</td>
<td>Constraint violation convergence tolerance, default (10^{-6}).</td>
</tr>
<tr>
<td>LineAlg</td>
<td>Line search algorithm. 0 = quadratic interpolation, 1 = cubic interpolation, 2 = curvilinear quadratic interpolation, 3 = curvilinear cubic interpolation. Default LineAlg = 0.</td>
</tr>
<tr>
<td>GradCheck</td>
<td>Set to 1 if you want to check user-supplied gradients, default 0.</td>
</tr>
<tr>
<td>MaxIter</td>
<td>Maximum number of iterations, default 500.</td>
</tr>
<tr>
<td>DiffGradMinChange</td>
<td>Minimum change in variables for finite difference gradients, default (10^{-8}).</td>
</tr>
<tr>
<td>DiffGradMaxChange</td>
<td>Maximum change in variables for finite difference gradients, default 0.1.</td>
</tr>
<tr>
<td>InitStepLength</td>
<td>Initial step length, default 1 or less.</td>
</tr>
<tr>
<td>eps_g</td>
<td>Gradient (or reduced gradient) convergence tolerance, default (10^{-6}).</td>
</tr>
<tr>
<td>eps_Rank</td>
<td>Rank test tolerance, default (10^{-10}).</td>
</tr>
<tr>
<td>wait</td>
<td>Flag if to use pause statements after output, default 0.</td>
</tr>
<tr>
<td>eps_absf</td>
<td>Convergence tolerance on absolute function value, default realmin.</td>
</tr>
<tr>
<td>PreSolve</td>
<td>Flag if presolve analysis is to be applied on linear constraints, default 0.</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>LineSearch</td>
<td>Structure with special fields for the line search, see Table 14.</td>
</tr>
<tr>
<td>Penalty</td>
<td>Penalty parameter for constrained problems.</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</td>
</tr>
<tr>
<td>bTol</td>
<td>Feasibility tolerance for linear constraints.</td>
</tr>
<tr>
<td>cTol</td>
<td>Feasibility tolerance for nonlinear constraints.</td>
</tr>
<tr>
<td>fTol</td>
<td>Accuracy in the computation of the function value, default (eps^{0.9}).</td>
</tr>
<tr>
<td>size_(x)</td>
<td>Size at optimum for the variables (x), used in the convergence tests.</td>
</tr>
<tr>
<td></td>
<td>Default 1.</td>
</tr>
<tr>
<td>size_(f)</td>
<td>Size at optimum for the function (f), used in the convergence tests.</td>
</tr>
<tr>
<td></td>
<td>Default 1.</td>
</tr>
<tr>
<td>size_(c)</td>
<td>Size at optimum for the constraints (c), used in the convergence tests.</td>
</tr>
<tr>
<td></td>
<td>Default 1.</td>
</tr>
<tr>
<td>IterNs</td>
<td>Number of iterations with low reduction before convergence.</td>
</tr>
<tr>
<td>NOT_release_all</td>
<td>Set to 1 if not to release more than one variable at the time.</td>
</tr>
<tr>
<td>subalg</td>
<td>Optimization sub algorithm. Dependent on type of problem. Default 0.</td>
</tr>
<tr>
<td>splineSmooth</td>
<td>Smoothness parameter sent to the SPLINE Toolbox routine <em>csaps.m</em> when computing numerical approximations of the gradient and the Jacobian. Default 0.4.</td>
</tr>
<tr>
<td>splineTol</td>
<td>Tolerance parameter sent to the SPLINE Toolbox routine <em>spaps.m</em> when computing numerical approximations of the gradient and the Jacobian. Default (10^{-3}).</td>
</tr>
</tbody>
</table>
Table 7: Information stored in the structure `Prob.ExpFit`

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

Table 8: Information stored in the structure `Prob.QP`

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>F</code></td>
<td>Constant matrix, the Hessian</td>
</tr>
<tr>
<td><code>c</code></td>
<td>Constant vector.</td>
</tr>
<tr>
<td><code>B</code></td>
<td>Logical vector of the same length as the number of variables. A one corresponds to a variable in the basis.</td>
</tr>
</tbody>
</table>

Table 9: Information stored in the structure `Prob.NLLS`

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>weightType</code></td>
<td>Weighting type:</td>
</tr>
<tr>
<td>0</td>
<td>No weighting.</td>
</tr>
<tr>
<td>1</td>
<td>Weight with data in (Yt). If (Yt = 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 <code>weightY</code>. If <code>weightY</code> is a vector then weighting by <code>weightY*r</code> (elementwise multiplication). If <code>weightY</code> is a matrix then weighting by <code>weightY*r</code> (matrix multiplication).</td>
</tr>
<tr>
<td>3</td>
<td><code>nlp.r</code> calls the routine <code>weightY</code> (must be a string with the routine name) to compute the residuals.</td>
</tr>
<tr>
<td><code>weightY</code></td>
<td>Either empty, a vector, a matrix or a string, see <code>weightType</code>.</td>
</tr>
<tr>
<td><code>t</code></td>
<td>Time vector t.</td>
</tr>
<tr>
<td><code>Yt</code></td>
<td>Matrix with observations <code>Y(t)</code>.</td>
</tr>
<tr>
<td><code>UseYt</code></td>
<td>If <code>UseYt = 0</code> compute residual as (f(x, t) - Y(t)) (default), otherwise Y(t) should be treated separately and the residual routines just return (f(x, t)).</td>
</tr>
<tr>
<td><code>SepAlg</code></td>
<td>If <code>SepAlg = 1</code>, use separable non linear least squares formulation, default 0.</td>
</tr>
</tbody>
</table>
Table 10: Information stored in the structure \textit{Prob.NTS}

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{SepAlg}</td>
<td>If \textit{SepAlg} = 1, use separable non linear least squares formulation, default 0.</td>
</tr>
<tr>
<td>\textit{ntsModel}</td>
<td>Nonlinear model number</td>
</tr>
<tr>
<td>\textit{p}</td>
<td>The number of terms (lags) in the model.</td>
</tr>
<tr>
<td>\textit{pL}</td>
<td>The number of nonlinear parameters.</td>
</tr>
<tr>
<td>\textit{pA}</td>
<td>The number of linear parameters.</td>
</tr>
<tr>
<td>\textit{ntsSeed}</td>
<td>Reset number for random generator or Time series number.</td>
</tr>
<tr>
<td>\textit{N}</td>
<td>Total number of data points.</td>
</tr>
<tr>
<td>\textit{t1}</td>
<td>The starting point for the estimation.</td>
</tr>
<tr>
<td>\textit{tN}</td>
<td>The end point for the estimation.</td>
</tr>
<tr>
<td>\textit{qamma}</td>
<td>Exponential weighting factor, default 0.99.</td>
</tr>
<tr>
<td>\textit{lambdaArt}</td>
<td>Nonlinear parameters used to create the artificial data.</td>
</tr>
<tr>
<td>\textit{alphaArt}</td>
<td>Linear parameters used to create the artificial data.</td>
</tr>
<tr>
<td>\textit{lambda}</td>
<td>Exponential parameters in autoregressive models.</td>
</tr>
<tr>
<td>\textit{alpha}</td>
<td>Weights in autoregressive models.</td>
</tr>
</tbody>
</table>

Table 11: Information stored in the structure \textit{Prob.PartSep}

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{pSepFunc}</td>
<td>Number of partially separable functions.</td>
</tr>
<tr>
<td>\textit{index}</td>
<td>Index for the partially separable function to compute, i.e. if \textit{i} = \textit{index}, compute \textit{f}<em>{i}(\textit{x}). If \textit{index} = 0, compute the sum of all, i.e. \textit{f}(\textit{x}) = \sum</em>{i=1}^{M} \textit{f}_{i}(\textit{x}).</td>
</tr>
</tbody>
</table>
Table 12: Information stored in the structure Prob.GLOBAL

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterations</td>
<td>Number of iterations, default 50.</td>
</tr>
<tr>
<td>MaxEval</td>
<td>Number of function evaluations, default 500.</td>
</tr>
<tr>
<td>Integers</td>
<td>Set of integer variables.</td>
</tr>
<tr>
<td>epsilon</td>
<td>Global/local weight parameter, default $10^{-4}$.</td>
</tr>
<tr>
<td>K</td>
<td>The Lipschitz constant. Not used.</td>
</tr>
<tr>
<td>tolerance</td>
<td>Error tolerance parameter. Not used.</td>
</tr>
<tr>
<td>C</td>
<td>Matrix with all rectangle centerpoints.</td>
</tr>
<tr>
<td>D</td>
<td>Vector with distances from centerpoint to the vertices.</td>
</tr>
<tr>
<td>L</td>
<td>Matrix with all rectangle side lengths in each dimension.</td>
</tr>
<tr>
<td>f'</td>
<td>Vector with function values.</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>Split</td>
<td>Split(i, j) is the number of splits along dimension i of rectangle j.</td>
</tr>
<tr>
<td>T</td>
<td>T(i) is the number of times rectangle i has been trisected.</td>
</tr>
<tr>
<td>G</td>
<td>Matrix with constraint values for each point.</td>
</tr>
<tr>
<td>ignoreidx</td>
<td>Rectangles to be ignored in the rectangle selection procedure.</td>
</tr>
<tr>
<td>ll</td>
<td>ll(i, j) is the lower bound for rectangle j in integer dimension I(i).</td>
</tr>
<tr>
<td>lu</td>
<td>lu(i, j) is the upper bound for rectangle j in integer dimension I(i).</td>
</tr>
<tr>
<td>feasible</td>
<td>Flag indicating if a feasible point has been found.</td>
</tr>
<tr>
<td>f.min</td>
<td>Best function value found at a feasible point.</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>
</tbody>
</table>

Table 13: 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>q</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, $\sum \lambda_i \partial^2 c(x)/dx^2$.</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>
Table 14: Information stored in the structure \textit{Prob.optParam.LineSearch}

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sigma</td>
<td>Line search accuracy; $0 &lt; \text{sigma} &lt; 1$. \text{sigma} = 0.9 inaccurate line search. \text{sigma} = 0.1 accurate line search, default 0.9.</td>
</tr>
<tr>
<td>rho</td>
<td>Determines the $\rho$ line, default 0.01.</td>
</tr>
<tr>
<td>tau1</td>
<td>Determines how fast step grows in phase 1, default 9.</td>
</tr>
<tr>
<td>tau2</td>
<td>How near end point of $[a, b]$, default 0.1.</td>
</tr>
<tr>
<td>tau3</td>
<td>Choice in $[a, b]$ phase 2, default 0.5.</td>
</tr>
<tr>
<td>eps1</td>
<td>Minimal length for interval $[a, b]$, default $10^{-7}$.</td>
</tr>
<tr>
<td>eps2</td>
<td>Minimal reduction, default $10^{-12}$.</td>
</tr>
<tr>
<td>MaxIter</td>
<td>Maximum number of line search iterations.</td>
</tr>
</tbody>
</table>
Table 15: Information stored in the global Matlab structure Result.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iter</td>
<td>Number of major iterations.</td>
</tr>
<tr>
<td>MinorIter</td>
<td>Number of minor iterations.</td>
</tr>
<tr>
<td>ExitFlag</td>
<td>0 if convergence to local min. Otherwise errors.</td>
</tr>
<tr>
<td>Inform</td>
<td>Information parameter, type of convergence.</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>H.k</td>
<td>Hessian value at optimum.</td>
</tr>
<tr>
<td>B.k</td>
<td>Quasi-Newton approximation of the Hessian 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>v.k</td>
<td>Lagrange multipliers.</td>
</tr>
<tr>
<td>r.k</td>
<td>Residual 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 16.</td>
</tr>
<tr>
<td>bState</td>
<td>State of each linear constraint, described in Table 17.</td>
</tr>
<tr>
<td>cState</td>
<td>State of each general constraint.</td>
</tr>
<tr>
<td>optParam</td>
<td>Structure with special fields for optimization parameters, see Table 6.</td>
</tr>
<tr>
<td>Name</td>
<td>Problem name.</td>
</tr>
<tr>
<td>P</td>
<td>Problem number.</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 steplengths 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>GLOBAL</td>
<td>Structure with special fields for global optimization, see Table 18.</td>
</tr>
<tr>
<td>SepNLLS</td>
<td>General result variable with fields z and Jz. Used when running separable nonlinear least squares problems</td>
</tr>
<tr>
<td>Solver</td>
<td>Solver used.</td>
</tr>
<tr>
<td>SolverAlgorithm</td>
<td>Solver algorithm used.</td>
</tr>
<tr>
<td>CPUtime</td>
<td>CPU time used.</td>
</tr>
<tr>
<td>REALtime</td>
<td>Real time elapsed.</td>
</tr>
<tr>
<td>Nfops</td>
<td>Number of floating point operations.</td>
</tr>
<tr>
<td>probType</td>
<td>TOMLAB problem type.</td>
</tr>
<tr>
<td>solvType</td>
<td>TOMLAB solver type.</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>ConstrEv</td>
<td>Number of constraint evaluations needed.</td>
</tr>
<tr>
<td>ResEv</td>
<td>Number of residual evaluations needed.</td>
</tr>
<tr>
<td>JacEv</td>
<td>Number of Jacobian evaluations needed.</td>
</tr>
<tr>
<td>Prob</td>
<td>Problem structure, see Table 5.</td>
</tr>
<tr>
<td>plotData</td>
<td>Structure with plotting parameters.</td>
</tr>
</tbody>
</table>
Table 16: 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 17: 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>

Table 18: Information stored in the structure Result.GLOBAL

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Matrix with all rectangle centerpoints in original coordinates.</td>
</tr>
<tr>
<td>D</td>
<td>Vector with distances from centerpoint to the vertices.</td>
</tr>
<tr>
<td>L</td>
<td>Matrix with all rectangle side lengths in each dimension.</td>
</tr>
<tr>
<td>F</td>
<td>Vector with function values.</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>Split</td>
<td>Split(i, j) is the number of splits along dimension i of rectangle j.</td>
</tr>
<tr>
<td>T</td>
<td>T(i) is the number of times rectangle i has been trisected.</td>
</tr>
<tr>
<td>G</td>
<td>Matrix with constraint values for each point.</td>
</tr>
<tr>
<td>ignoreidx</td>
<td>Rectangles to be ignored in the rectangle selection procedure.</td>
</tr>
<tr>
<td>LL</td>
<td>LL(i, j) is the lower bound for rectangle j in integer dimension I(i).</td>
</tr>
<tr>
<td>LU</td>
<td>LU(i, j) is the upper bound for rectangle j in integer dimension I(i).</td>
</tr>
<tr>
<td>feasible</td>
<td>Flag indicating if a feasible point has been found.</td>
</tr>
<tr>
<td>f.min</td>
<td>Best function value found at a feasible point.</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>
</tbody>
</table>
2.1.1 Global Variables

The use of globally defined variables in NLPLIB TB is well motivated. For example to avoid unnecessary evaluations, storage of sparse patterns, internal communication, computation of elapsed CPU time etc.

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 are used. The globalSave routine saves all global variables in a structure glbSave(depth) and then initialize all of them as empty. By using the depth variable, an arbitrarily number of recursions are possible. The other routine globalGet retrieves all global variables in the structure glbSave(depth).

The global variables used in NLPLIB TB are listed in Table 19 and 20.

<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>CUTE.DLL</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.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 20: The global variables used in NLPLIB TB

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>$SEP_Jz$</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 5.</td>
</tr>
<tr>
<td>$Result$</td>
<td>Result structure, see Table 15.</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>$SparseStructure$</td>
<td>Used by MINOS (sparse structure).</td>
</tr>
<tr>
<td>$NonZeros$</td>
<td>Number of nonzero matrix elements in $SparseStructure$.</td>
</tr>
<tr>
<td>$qblSave$</td>
<td>Used to save global variables in recursive calls to TOMLAB.</td>
</tr>
<tr>
<td>$PATHDEL$</td>
<td>PC or UNIX way of path delimiter i.e. &quot;\&quot; or &quot;/&quot;.</td>
</tr>
</tbody>
</table>
2.2 Solver Routines in NLPLIB TB

In Table 21 the optimization solvers in NLPLIB TB are listed. The solver for unconstrained optimization, \textit{ucSolve}, the nonlinear least squares solvers \textit{lsSolve} and \textit{clsSolve}, and the constrained solver \textit{conSolve}, are all written as prototype routines.

Table 21: Optimization solvers in NLPLIB TB

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{ucSolve}</td>
<td>A prototype routine for unconstrained optimization with simple bounds on the parameters. Implements Newton, quasi-Newton and conjugate-gradient methods.</td>
<td>2.11.14</td>
<td>84</td>
</tr>
<tr>
<td>\textit{gllSolve}</td>
<td>A routine for box-bounded global optimization.</td>
<td>2.11.5</td>
<td>73</td>
</tr>
<tr>
<td>\textit{gblSolve}</td>
<td>Stand-alone version of \textit{gllSolve}. Runs independently of NLPLIB TB.</td>
<td>2.11.3</td>
<td>70</td>
</tr>
<tr>
<td>\textit{gclSolve}</td>
<td>A routine for global mixed-integer nonlinear programming.</td>
<td>2.11.6</td>
<td>75</td>
</tr>
<tr>
<td>\textit{gclSolve}</td>
<td>Stand-alone version of \textit{gclSolve}. Runs independently of NLPLIB TB.</td>
<td>2.11.4</td>
<td>71</td>
</tr>
<tr>
<td>\textit{lsSolve}</td>
<td>A prototype algorithm for nonlinear least squares with simple bounds. Implements Gauss-Newton, and hybrid quasi-Newton and Gauss-Newton methods.</td>
<td>2.11.17</td>
<td>76</td>
</tr>
<tr>
<td>\textit{clsSolve}</td>
<td>A prototype algorithm for constrained nonlinear least squares. Currently 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>2.11.1</td>
<td>67</td>
</tr>
<tr>
<td>\textit{conSolve}</td>
<td>Constrained nonlinear minimization solver using two different sequential quadratic programming methods.</td>
<td>2.11.2</td>
<td>69</td>
</tr>
<tr>
<td>\textit{nlpSolve}</td>
<td>Constrained nonlinear minimization solver using filter SQP.</td>
<td>2.11.8</td>
<td>78</td>
</tr>
<tr>
<td>\textit{sTrustR}</td>
<td>Solver for constrained convex optimization of partially separable functions, using a structural trust region algorithm.</td>
<td>2.11.13</td>
<td>82</td>
</tr>
<tr>
<td>\textit{qplm}</td>
<td>Solves a general quadratic program.</td>
<td>2.11.10</td>
<td>80</td>
</tr>
<tr>
<td>\textit{qplm}</td>
<td>Solves a general quadratic program.</td>
<td>2.11.12</td>
<td>81</td>
</tr>
<tr>
<td>\textit{qpe}</td>
<td>Solves a \texttt{qp} problem, restricted to equality constraints, using a null space method.</td>
<td>2.11.9</td>
<td>79</td>
</tr>
<tr>
<td>\textit{npsol}</td>
<td>Solves a \texttt{qp} problem, restricted to equality constraints, using Lagrange’s method.</td>
<td>2.11.11</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 21 lists the NLPLIB TB internal solvers. To get a list of all available solvers, including Fortran, C and Matlab Optimization Toolbox solvers, for a certain \textit{solvType} the user just calls the routine \texttt{PrintSolvers} with \textit{solvType} as argument. \textit{solvType} should either be a string (‘uc’, ‘con’ etc.) or the corresponding \textit{solvType} number, see Table 1. As an example, assume you want a list of all available solvers of \textit{solvType} \texttt{con}. Then

\texttt{PrintSolvers(’con’)}

gives the printing output

<table>
<thead>
<tr>
<th>Solver</th>
<th>solvType Number</th>
<th>Multi-Solver Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{nlpSolve}</td>
<td>3</td>
<td>\textit{conRun}</td>
</tr>
<tr>
<td>\textit{conSolve}</td>
<td>3</td>
<td>\textit{conRun}</td>
</tr>
<tr>
<td>\textit{sTrustR}</td>
<td>3</td>
<td>\textit{conRun}</td>
</tr>
<tr>
<td>\textit{constr}</td>
<td>3</td>
<td>\textit{conRun}</td>
</tr>
<tr>
<td>\textit{minos}</td>
<td>3</td>
<td>\textit{conRun}</td>
</tr>
<tr>
<td>\textit{npsol}</td>
<td>3</td>
<td>\textit{conRun}</td>
</tr>
<tr>
<td>\textit{npopt}</td>
<td>3</td>
<td>\textit{conRun}</td>
</tr>
</tbody>
</table>
and if PrintSolvers is called with no given argument then all available solvers for all different solvType is printed.

The routine ucSolve implements a prototype algorithm for **unconstrained optimization** with simple bounds on the parameters (uc), i.e. solves the problem

\[
\min_x f(x) \tag{1}
\]
\[s/t \quad x_L \leq x \leq x_U.
\]

where \(x,x_L,x_U \in \mathbb{R}^n\) and \(f(x) \in \mathbb{R}\). ucSolve includes several of the most popular search step methods for unconstrained optimization. Bound constraints are treated as described in Gill et. al. [28]. The search step methods for unconstrained optimization included in ucSolve are: the Newton method, the quasi-Newton BFGS and inverse BFGS method, the quasi-Newton DFP and inverse DFP method, the Fletcher-Reeves and Polak-Ribiere conjugate-gradient method, and the Fletcher conjugate descent method. For the Newton and the quasi-Newton methods the code is using a subspace minimization technique to handle rank problems, see Lindström [41]. The quasi-Newton codes also use safe guarding techniques to avoid rank problem in the updated matrix.

The routine gIbSolve implements an algorithm for **box-bounded global optimization** (gIb), i.e. problems of the form (1) that have finite simple bounds on all the variables. gIbSolve implements the DIRECT algorithm [38], which is a modification of the standard Lipschitzian approach that eliminates the need to specify a Lipschitz constant. In gIbSolve no derivative information is used. For **global mixed-integer nonlinear programming** (gIc), gIcSolve implements an extended version of DIRECT, see [39], that handles problems with both nonlinear and integer constraints. There are also stand-alone versions of both gIbSolve and gIcSolve named gIbSolve and gIcSolve respectively. These stand-alone versions runs independently of NLPLIB TB.

For global optimization problems with expensive function evaluations the routine ego that implements the Efficient Global Optimization (EGO) algorithm [40]. 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 **constrained nonlinear optimization** problem (con) is defined as

\[
\min_x f(x) \tag{2}
\]
\[s/t \quad x_L \leq x \leq x_U, \quad b_L \leq Ax \leq b_U, \quad c_L \leq c(x) \leq c_U
\]

where \(x,x_L,x_U \in \mathbb{R}^n\), \(f(x) \in \mathbb{R}\), \(A \in \mathbb{R}^{m \times n}\), \(b_L,b_U \in \mathbb{R}^m\) and \(c_L,c(x),c_U \in \mathbb{R}^m\). For general constrained nonlinear optimization a sequential quadratic programming (SQP) method by Schittkowski [50] is implemented in the routine conSolve. Like ucSolve, IsSolve and cIsSolve, conSolve is a prototype routine and also includes an implementation of the HanPowell SQP method. There are also a routine nlpSolve which implements the Filter SQP by Roger Fletcher and Sven Leyffer presented in [23].

Another constrained solver in NLPLIB TB is the structural trust region algorithm sTrustR, combined with an initial trust region radius algorithm. The code is based on the algorithms in [15] and [49], and treats partially separable functions. Safeguarded BFGS or DFP are used for Quasi-Newton update, if not the analytical Hessian is used. Currently, sTrustR only solves problems where the feasible region defined by the constraints is convex.

A **quadratic program** (qp) is defined as

\[
\min_x f(x) = \frac{1}{2}x^TFx + c^Tx \tag{3}
\]
\[s/t \quad x_L \leq x \leq x_U, \quad b_L \leq Ax \leq b_U
\]

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 \(b_L,b_U \in \mathbb{R}^m\). Quadratic programs are solved with a standard active-set method [42], implemented in the routine qP Solve. qP Solve 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. NLPLIB TB also includes a similar routine \textit{qpBiggs}, which is using a more simple algorithm for negative definite quadratic problems, described by Bartholomew-Biggs in

NLPLIB TB includes two algorithms for solving quadratic programs restricted to equality constraints (EQP); a null space method (\textit{qpe}) and Lagrange’s method (\textit{qplm}).

The \textbf{nonlinear least squares problem} (\textit{ls}) is defined as

\[
\min_x f(x) = \frac{1}{2}r(x)^T r(x) \quad \text{s.t.} \quad x_L \leq x \leq x_U.
\]

where \(x, x_L, x_U \in \mathbb{R}^n\) and \(r(x) \in \mathbb{R}^N\).

In NLPLIB TB the prototype nonlinear least squares algorithm \textit{lsSolve} treats problems with bound constraints in a similar way as the routine \textit{ucSolve}.

The prototype routine \textit{lsSolve} includes four optimization methods for nonlinear least squares problems: the Gauss-Newton method, the Al-Baali-Fletcher [4] and the Fletcher-Xu [21] hybrid method, and the Huschens TSSM method [36]. If rank problems occur, the prototype algorithm is using subspace minimization. The line search algorithm used is the same as for unconstrained problems.

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

\[
\min_x f(x) = \frac{1}{2}r(x)^T r(x) \quad \text{s.t.} \quad x_L \leq x \leq x_U, \quad b_L \leq Ax \leq b_U, \quad c_L \leq c(x) \leq c_U
\]

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

The constrained nonlinear least squares solver \textit{clsSolve} is based on \textit{lsSolve} and its search steps methods. Currently \textit{clsSolve} treats linear equality and inequality constraints using an active-set strategy.
2.3 Utility Routines in NLPLIB TB

There are six menu programs defined in NLPLIB TB, see Table 22, one for each type of optimization problem (probType). NLPLIB TB also includes a graphical user interface (GUI), which has the same functionality as all the menu programs.

Table 22: Menu programs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>npllib</td>
<td>Graphical User Interface (GUI) for nonlinear optimization. Handles all types of nonlinear optimization problems.</td>
</tr>
<tr>
<td>ucOpt</td>
<td>Menu for unconstrained optimization.</td>
</tr>
<tr>
<td>gblOpt</td>
<td>Menu for box-bounded global optimization.</td>
</tr>
<tr>
<td>gmcOpt</td>
<td>Menu for global mixed-integer nonlinear programming.</td>
</tr>
<tr>
<td>gpOpt</td>
<td>Menu for quadratic programming.</td>
</tr>
<tr>
<td>conOpt</td>
<td>Menu for constrained optimization.</td>
</tr>
<tr>
<td>lsOpt</td>
<td>Menu for nonlinear least squares problems.</td>
</tr>
<tr>
<td>clsOpt</td>
<td>Menu for constrained nonlinear least squares problems.</td>
</tr>
</tbody>
</table>

Each menu program calls a corresponding driver routine, having the same probType, viz. either of ucRun, gblRun, gpRun, conRun, lsRun or clsRun.

NLPLIB TB is using the structure variable optParam, see Table 6, with optimization parameters. For each type of optimization problem, there is a corresponding definition routine which calls optParamDef and defines the default parameter values for optParam. Dependent on probType, it is any of ucDef, gpDef, conDef, lsDef or clsDef.

In Table 23, the utility functions needed by the solvers in Table 21 are displayed. The function ittr implements the initial trust region radius algorithm by Sartenaer [49].

The line search algorithm LineSearch, used by the solvers conSolve, lsSolve, clsSolve and ucSolve, is a modified version of an algorithm by Fletcher [22, chap. 2]. The use of quadratic (intpol2) and cubic interpolation (intpol3) is possible in the line search algorithm. For more details, see Section 2.12.4.

The routine presolve is running a presolve analysis on a system of linear equalities, linear inequalities and simple bounds. An algorithm by Gondzio [30], somewhat modified, is implemented in presolve. See [10] for a more detailed presentation.

Table 23: Utility routines for the optimization solvers.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ittr</td>
<td>Initial trust region radius algorithm.</td>
<td>2.12.3</td>
<td>86</td>
</tr>
<tr>
<td>LineSearch</td>
<td>Line search algorithm by Fletcher.</td>
<td>2.12.4</td>
<td>87</td>
</tr>
<tr>
<td>intpol2</td>
<td>Find the minimum of a quadratic interpolation. Used by LineSearch.</td>
<td>2.12.1</td>
<td>85</td>
</tr>
<tr>
<td>intpol3</td>
<td>Find the minimum of a cubic interpolation. Used by LineSearch.</td>
<td>2.12.2</td>
<td>85</td>
</tr>
<tr>
<td>presolve</td>
<td>Presolve analysis on linear constraints and simple bounds.</td>
<td>2.12.5</td>
<td>88</td>
</tr>
</tbody>
</table>

2.3.1 Low Level Routines and Test Problems

We 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 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 of the problem setup routines in NLPLIB TB (routines with 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');

ProbSet:
...

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 nonpresent or left out when calling mFiles.

NLPLIB TB is using gateway routines (nlp.f, nlp.g, nlp.H, nlp.c, nlp.dc, nlp.d2c, nlp.r, nlp.J, nlp.d2r). These names are put in Prob.p.f, Prob.p.g etc. by NLPLIB TB automatically. 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. In Figure 3, we illustrate 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. In Figure 4, we illustrate 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: 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.](image)

![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.](image)

Information about a problem is stored in the structure variable Prob, described in Table 5. 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. A more detailed description of how to define new problems is given in Section 2.6.

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 24 describes the low level test functions and the corresponding problem setup routines needed for the predefined constrained optimization (con) problems. For the predefined unstrained optimization (uc) problems, the global optimization (glb, glc) problems and the quadratic programming problems (qp) similar routines are needed.

The problem of fitting positive 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. Some empirical data series are predefined and artificial data series may also be generated. Algorithms to find starting values for different number of exponential terms are implemented. Table 25 shows the relevant routines.
Table 24: Generally constrained nonlinear (con) test problems.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>con.prob</td>
<td>Initialization of 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.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>

Table 25: Exponential fitting test problems.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp.ArtP</td>
<td>Generate artificial exponential sum problems.</td>
</tr>
<tr>
<td>exp.Init</td>
<td>Find starting values for the exponential parameters $\lambda$.</td>
</tr>
<tr>
<td>exp.prob</td>
<td>Defines a exponential fitting type of problem, with data series $(t,y)$. The file includes data from several different empirical test series.</td>
</tr>
<tr>
<td>Helax.prob</td>
<td>Defines 335 medical research problems supplied by Helax AB, Uppsala, where an exponential model is fitted to data. The actual data series $(t,y)$ are stored on one file each, i.e. 335 data files, 8MB large, and are not distributed. A sample of five similar files are part of exp.prob.</td>
</tr>
<tr>
<td>exp.r</td>
<td>Compute the residual vector $r_i(x), i = 1, ..., m$. $x \in \mathbb{R}^n$.</td>
</tr>
<tr>
<td>exp.J</td>
<td>Compute the Jacobian matrix $\partial r_i/\partial x_i, i = 1, ..., m, j = 1, ..., n$.</td>
</tr>
<tr>
<td>exp.d2r</td>
<td>Compute the 2nd part of the second derivative for the nonlinear least squares exponential fitting problem.</td>
</tr>
<tr>
<td>exp.c</td>
<td>Compute the constraints $\lambda_1 &lt; \lambda_2 &lt; ...$ on the exponential parameters $\lambda_i, i = 1, ..., p$.</td>
</tr>
<tr>
<td>exp.dc</td>
<td>Compute matrix of constraint normals for constrained exponential fitting problem.</td>
</tr>
<tr>
<td>exp.d2c</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>exp.q</td>
<td>Find starting values for exponential parameters $\lambda_i, i = 1, ..., p$.</td>
</tr>
<tr>
<td>exp.p</td>
<td>Find optimal number of exponential terms, $p$.</td>
</tr>
</tbody>
</table>

Table 26 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 27 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 [11, 12].

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.
### Table 26: 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, ..., 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) = \frac{\partial r_i}{\partial x_j}, i = 1, ..., m, j = 1, ..., 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>

### Table 27: Test problems from CUTE data base.

<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>
2.3.2 Test Routines for the System

NLPLIB TB is constantly being developed and improved. Therefore it is important to have some routines which run a whole bunch of test problems with all different solvers to check for bugs. The routines listed in Table 28 perform such tests.

<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>2.13.3</td>
<td>89</td>
</tr>
<tr>
<td>systest</td>
<td>Runs big test to check for bugs in NLPLIB TB.</td>
<td>2.13.4</td>
<td>90</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.

2.4 The Menu Systems

This section describes the menu routines ucOpt, qpOpt, conOpt, lsOpt, clsOpt and qlbOpt. The Graphical User Interface, which has the same functionality, is presented in Section 2.5. The ucOpt menu is shown in Figure 5. The other menus look the same, possibly with some extra 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 File and Problem button selects the problem setup file and the problem to be solved. Correspondingly, the Choice of optimization algorithm button selects the optimization algorithm 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. See Figure 6. The parameters are those stored in the optParam structure, see Table 6. 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.

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

When the problem is solved, it is possible to make different types of plots to illustrate the solution procedure. Pushing the Plot Menu button, a menu choosing type of plot will appear. A overview of the available plotting options are given in connection with the Graphical User Interface described in Section 2.5.

The menu routines are started by just typing the name of the routine (e.g. ucOpt) at the Matlab prompt. In Section 3.2.1 we illustrate how to use the menu system for linear programming problems (lpOpt). The menu routines in NLPLIB TB work in a similar way.

Calling any of the menu routines in NLPLIB TB (e.g. ucOpt) by typing Result = ucOpt will return a structure array containing the Result structures of all the runs made. As an example, to display the results from the third run, enter the command Result(3). To display the solution found in the third run, enter the command Result(3).x.k. The information stored in the structure are given in Table 15.

2.5 The Graphical User Interface

The Graphical User Interface is started by calling the Matlab m-file nlplib.m, i.e. by entering the command nlplib at the Matlab prompt. The GUI has two modes; Normal and Advanced. At start the GUI is in Normal mode, shown in Figure 7.

There are one axes area, four menus; Subject, Problem, Algorithm and Plot, and six push buttons; Defaults, Advanced, Plot, Info, Run and Close.

There are also eleven edit controls where it is possible to enter parameter values used by the solution algorithm. To the right of the axes area, starting values for two dimensional problems can be given. How to define starting values for problems with more than two decision variables is discussed in Section 2.5.1. The edit controls labeled
'Axes' set the axes in the contour plot and the mesh plot. The edit controls below the axes area are used to set the optimization parameters sent to the solver. These parameters are the maximum number of iterations (MaxIter), the line search accuracy $\sigma$ (Sigma), the termination tolerance on the change in the decision variables (EpsX), the termination tolerance on the function value (EpsF) and the termination tolerance on the gradient (EpsG). If a solver for constrained optimization is selected, a twelfth edit control (EpsC) is shown. This edit control sets the termination tolerance on the constraint violation.

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

The Subject menu is used to select subject, i.e. which type of problem to be solved. There are currently six main problem types; unconstrained optimization, quadratic programming, constrained optimization, nonlinear least squares, exponential sum fitting and constrained nonlinear least squares.

From the Problem menu, the user selects the problem to be solved. 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 Section 2.6.

The Algorithm menu is used to select solver. It can either be a NLPLIB TB internal solver, a solver in the Matlab Optimization Toolbox or a general-purpose solver implemented in Fortran or C.

Changing type of optimization problem in the Subject menu, will change the menu entries in the Problem menu and Algorithm 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 Defaults button, the default values for every parameter are displayed in the edit controls. 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.

The Advanced button and the Advanced mode is described in Section 2.5.1.

Pushing the Plot button gives a plot of the current problem. 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 8.

A contour plot for a constrained problem and a plot of the data and the obtained model for a nonlinear least squares problem are given in Figure 9. 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 \((x_1^*, x_2^*) = (-9.5474, 1.0474)\).

The Info button gives some information about the current problem, e.g. the number of variables.

When the user has chosen a solver and a problem, he then pushes the Run 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 nc algorithm is selected as in Figure 7, the Run button has the same function as the Plot button.

To close the GUI, push the Close button.

### 2.5.1 The Advanced Mode

When pushing the Advanced button, the GUI will change to Advanced mode. The axes area is replaced by more edit controls and menus, see Figure 10.

Furthermore, the Advanced button is renamed to Figure button. To change from Advanced mode to Normal mode, push the Figure button.

There are some new edit controls in the Advanced mode. FLow, the best guess on a lower bound for the optimal function value, is used by NLPLIB TB solver algorithms using the Fletcher line search algorithm [22]. The parameter EpsR is the rank test tolerance in the subspace minimization technique used when determining the search direction in some of the algorithms.

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 labeled 'Starting Values'. 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'.

There are six new menus in the Advanced mode. 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://simon.cs.cornell.edu/home/verma/AD, or by five different approaches for numerical differentiation. Three of them requires the Spline Toolbox to be installed. The second menu determines if a quadratic or a cubic interpolation shall be used in the line search algorithm.

Two menus are used to select the level of output from the optimization driver and the optimization solver. All output printed during the optimization are displayed in the Matlab Command Window. If the 'Pause Each Iteration' check box is selected, the NLPLIB TB solvers are using the pause statement to halt after each iteration. The menu 'Init File' selects the file defining the current set of problems. Changing the set of problems will automatically modify the Problem menu. The menu named 'Method' differs between problem types. 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.

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 about the parameters will appear when selecting the check box named 'User Control'. If the 'User Control' check box is not selected, default values will be used.

When selecting an exponential fitting problem, two new menus and a new edit control will appear. The number of exponential terms in the approximating model and which of four types of residual weighting to be used are determined by the user. Furthermore, there is a choice whether to solve the weighted least squares fitting problem using an ordinary or separable nonlinear least squares algorithm.

In the Advanced mode there are three new push buttons. If a contour plot is displayed in the axes area and the user pushes the button named 'x0', it is possible to select starting point for the current algorithm using the mouse. Pushing the 'ReOpt' button, the current problem is re-optimized with the starting point defined as the solution found in the previous solution attempt.

Entering a name in the edit control labeled 'Define' and pushing the Save button, two files will be generated; one Matlab mat-file and one Matlab.m-file. The name 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, and 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.

If entering a name in the 'Define' edit control and pushing the Defaults button, the default values for all parameters will be loaded from the current mat-file.

When a problem is solved, the user can access the results from the Matlab Command Window, stored in the global structure Result. If the user has not run the NLPLIB TB initialization command nlplibinit, he must enter the command global Result at the Matlab prompt to declare Result as a global structure. To display the full structure, enter Result at the prompt. To display a specific field in the structure, e.g. the solution found, enter Result.x.k. All information stored in the structure are given in Table 15. When the check box 'Hold Previous Run' is selected, Result becomes a structure array. As an example, to display the results from the third run, enter the command Result(3). To display the solution found in the third run, enter the command Result(3).x.k

The user could also access the plotting parameter structure plotData in the same way as described for the Result structure above.
Figure 6: Setting optimization parameters in ucOpt.
Figure 7: The GUI in Normal mode.
Figure 8: A contour plot with the search directions and marks for the line search trials for each iteration.

Figure 9: A contour plot for a constrained problem and a plot of data and model for a nonlinear least squares problem.
Figure 10: The GUI in Advanced mode.
2.6 How to Define Optimization Problems in NLPLIB TB

In NLPLIB TB there are principally three ways to define new problems. Which to choose is somewhat dependent if a quick or a more permanent solution is desired. When additions to NLPLIB TB are made, the best is to define a new directory and make additions to copies of already existing files. This new directory must be put before NLPLIB TB in the Matlab search path, or alternatively, the user must make his runs with this directory being the current directory. Making a special update directory makes it easy to update with new releases of NLPLIB TB without destroying any updates. In the following of this section, we assume that this new directory is called NLPNEW. All the problem definition files which we refer to in this section are found in the directory ...	omlab

In these example files, you can find all the modifications we describe.

The three alternative ways to define new problems in NLPLIB TB are:

1. Solving problems of a certain type, one can copy the basic files for this type of problem and edit these. For example, solving nonlinear least squares problems, copy the files ls.prob.m, ls.f.m, ls.g.m, etc. (note the underscore) to NLPNEW and either replace one of the existing problems, or add new ones. Section 2.6.1 - 2.6.6 describe how to modify these files for unconstrained, constrained, nonlinear least squares and constrained nonlinear least squares problems.

2. If many problems of a certain type are to be solved, we recommend you to make your own problem definition files for the function, gradient, constraints etc. Just copy the files that solve problems of the same or more general type. A general choice would be to copy the con.*.m files and change their names and edit these in the proper ways. Follow the instructions for alternative 1 and see Section 2.6.9 were we will make clear what extra modifications are needed.

3. To add one or more single problems, the easiest way is to copy the files usr.*.m to NLPNEW for modification. All different problem types are possible to define in these user problem definition files. At the end of each Section 2.6.1 - 2.6.6, we will describe how to modify these files.

Throughout this section (except for Section 2.6.7 and 2.6.8) we will show how to define the famous test problem Rosenbrock’s banana.

\[
\begin{align*}
\min_{s/t} \quad & f(x) = 100 (x_2 - x_1^2)^2 + (1 - x_1)^2 \\
n & -10 \leq x_j \leq 2, j = 1, 2,
\end{align*}
\]

(6)
as an unconstrained, constrained, nonlinear least squares and constrained nonlinear least squares problem. We have added simple bounds on the variables, and for the constrained problem types, we will also add constraints in illustrative purpose. We will call this problem RB BANANA in the following descriptions to avoid mixing it up with problems already defined in the problem definition files.

2.6.1 Defining Unconstrained Problems

To define (6) as an unconstrained problem follow the stepwise instructions below (for all instructions we assume that you edit the copied files in a text editor):

1. Copy the files uc.prob.m, uc.f.m, uc.g.m and uc.H.m to your directory NLPNEW.

2. Add the problem name to the menu choice in uc.prob.m

```matlab
... ...
', 'Fletcher U.2.6'...
', 'Fletcher U.3.3'...
', 'RB BANANA'...
); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES
if isempty(P)
    return;
```
3. Add the following in uc_prob.m after the last already existing problem (the optional parameters are not necessary to define):

```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.
    % CHANGE: elseif P == 18
    % CHANGE: Add an elseif entry and the other variable definitions needed
```

4. Make the following addition in uc_f.m:

```matlab
elseif P == 17  % Fletcher Q.3.3
    f = 0.5*(x(1)^2+x(2)^2)*exp(x(1)^2-x(2)^2);  
else P == 18  % RB BANANA
    f = 100*(x(2)-x(1)^2-x(2)^2 + (1-x(1))^2;  
end
```

5. Make the following addition in uc_g.m:

```matlab
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)];
else P == 18  % RB BANANA
    g = [-400*x(1)*(x(2)-x(1)^2-2*(1-x(1)); 200*(x(2)-x(1)^2) ];  
end
```

6. Make the following addition in uc_H.m

```matlab
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)];
```
\[ e = \exp(x(1)^2 - 2x(2)^2); \]
\[ H = \begin{bmatrix}
1 + 5x(1)^2 + 2x(1)x(2) + 2x(2)^2 + 2x(1)^4 \\
-2x(1)x(2) + (x(1)^2 + 2x(2)^2) \\
0 \\
1 - x(1)^2 + 2x(1)x(2) + 2x(2)^2 - 5x(2)^2 + 2x(2)^4 \\
\end{bmatrix}; \]
\[ H(2,1) = H(1,2); \]
\[ H = e^H; \]

else if \( P == 18 \) \( \% \) RB BANANA

\[ H = \begin{bmatrix}
1200x(1)^2 - 400x(2) + 2 & -400x(1) \\
-400x(1) & 200 \\
\end{bmatrix}; \]

end

...

7. Save all the files properly.

If you prefer alternative 3 you should instead copy the files \textit{usr.prob.m}, \textit{usr.f.m}, \textit{usr.q.m} and \textit{usr.H.m} in step 1.
In these files, replace the problem \textit{Own UC problem 1} with \textit{RB BANANA} in the same way as described above (do not forget the menu choice line in \textit{usr.prob.m}).

2.6.2 Defining Box-bounded Global Optimization Problems

Box-bounded global optimization problems are defined in the same way as unconstrained optimization problems. Since no derivative information is used, \textit{glb.prob} and \textit{glb.f} are the only problem definition files that need to be modified.

To define (6) as a box-bounded global optimization problem follow the stepwise instructions below (for all instructions we assume that you edit the copied files in a text editor). Note that we in this example change the lower variable bounds to \( x_L = (-2, -2)^T \). The reason for that is just to speed up the global search for the reader who wants to run this example.

1. Copy the files \textit{glb.prob.m} and \textit{glb.f.m} to your directory NLPNEW.

2. Add the problem name to the menu choice in \textit{glb.prob.m}:

\[ ...
...,'HGU 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 \textit{glb.prob.m} after the last already existing problem (the optional parameters are not necessary to define):

\[ ...
...
else if P == 29
    Name='RB BANANA';
    x_L = [-2; -2]; \% Lower bounds for x.
    x_U = [2; 2]; \% Upper bounds for x.
    x_opt = [1 1]; \% Known optimal point (optional).
    f_opt = 0; \% Known optimal function value (optional).\]
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.

% CHANGE: elseif P == 30
% CHANGE: Add an elseif entry and the other variable definitions needed
...
...

4. Make the following addition in glb.f.m:
...
...
eelseif P == 29 % RB BANANA
    f = 100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
end
...
...

5. Save all the files properly.

2.6.3 Defining Nonlinear Least Squares Problems

To define (6) as a nonlinear least squares problem follow the stepwise instructions below (for all instructions we assume that you edit the copied files in a text editor):

1. Copy the files ls_prob.m, ls_r.m and ls_J.m to your directory NLPNEW.

2. Add the problem name to the menu choice in ls_prob.m:
...
...
    ,'Plasmid Stability n=3 (subst.)'...
    ,'Plasmid Stability n=3 (probability)'
    , 'RB BANANA'...
); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

    if isempty(P)
        return;
    end
    ...
    ...

3. Add the following in ls_prob.m after the last already existing problem (the optional parameters are not necessary to define):
...
...
eelseif P==1:
    Name='RB BANANA':
    Yt=[0;0]; % r(x) = residual = model psi(t,x) - data Yt(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).
4. Make the following addition in \textit{ls}_r.m:

\begin{verbatim}

... ...

yMod=r;
elseif P==10
  \% RB BANANA
  r = [10*(x(2)-x(1)^2);1-x(1)];
end

if Prob.NLLS.UseYt && m==length(r), r=r-Yt; end ...

...

5. Make the following addition in \textit{ls}_J.m:

\begin{verbatim}

... ...

elseif P==10
  \% RB BANANA
  J = [-20*x(1) 10
        -1  0 ];
end ...

...

6. Save all the files properly.

If you prefer alternative 3 you should instead copy the files \textit{usr}_prob.m, \textit{usr}_r.m and \textit{usr}_J.m in step 1. In these files, replace the problem \textit{Own LS problem 1} with \textit{RB BANANA} in the same way as described above (do not forget the menu choice line in \textit{usr}_prob.m).

\subsection{2.6.4 Defining Constrained Problems}

To illustrate how to define a constrained problem, we add the constraints

\begin{equation}
    x_1 - x_2 \leq 1
\end{equation}

and

\begin{equation}
    -x_1^2 - x_2 \leq 1
\end{equation}

to (6). Constraint (7) is of linear type and will thereby be defined separated from the nonlinear constraint (8). The problem will be defined by following the stepwise instructions below (for all instructions we assume that you edit the copied files in a text editor):
1. Copy the files *con_prob.m*, *con_f.m*, *con_q.m*, *con_H.m*, *con_c.m*, *con_dc.m* and *con_d2c.m* to your directory NLPNEW.

2. Modify the files *con_prob.m*, *con_f.m*, *con_q.m* and *con_H.m* in the same way as described for the unconstrained case in Section 2.6.1.

3. Extend the problem definition in *con_prob.m* with the constraint parameters:

   ```
   ... 
   elseif P == 1b
       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(2,1);
       end
   
   end 
   ... 
   ```

6. Make the following addition in *con_d2c.m*:

   ```
   ... 
   ```
elseif p == 15 % RB BANANA
    if init==1
        d2c = [-2 0;0 0]*lam;
    else
        d2c = [1 0; 0 0]
    end
end
...  
...

7. Save all the files properly.

If you prefer alternative 3 you should instead copy the files *usr.prob.m*, *usr.f.m*, *usr.g.m*, *usr.H.m*, *usr.c.m*, *usr.dc.m* and *usr.d2c.m* in step 1. In these files, replace the problem *Own C problem 1* with *RB BANANA* in the same way as described above (do not forget the menu choice line in *usr.prob.m*).

### 2.6.5 Defining Global Mixed-Integer Nonlinear Programming Problems

To illustrate how to define a global mixed-integer nonlinear programming problem, we add the constraints (7), (8) and

\[ x_1 \text{ integer} \]  \hspace{1cm} (9)

to (6). Constraint (7) is of linear type and will thereby be defined separated from the nonlinear constraint (8).

To define (6) with the constraints (7), (8) and (9) as a global mixed-integer nonlinear programming problem follow the stepwise instructions below (for all instructions we assume that you edit the copied files in a text editor). Note that we in this example change the lower variable bounds to \( x_L = (-2, -2)^T \). The reason for that is just to speed up the global search for the reader who wants to run this example.

1. Copy the files *glc.prob.m*, *glc.f.m* and *glc.c.m* to your directory NLPNEW.

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 2.6.2.

3. Extend the problem definition in *glc.prob.m* with the constraint parameters:

```
...
...
elseif p == 24
    Name='RB BANANA';
x_L = [-2; -2];  % Lower bounds for x.
x_U = [2; 2];    % Upper bounds for x.
x_opt = [1 1];   % Known optimal point (optional).
f_opt = 0;       % Known optimal function value (optional).
A = [1 -1];      % Linear constraints matrix.
b_L = -inf;      % Lower bounds on linear constraints.
b_U = 1;        % Upper bounds on linear constraints.
c_L = -inf;     % Lower bounds on nonlinear constraints.
c_U = 1;        % Upper bounds on nonlinear constraints.
Integers = [1];  % Integer constraint.
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 `gcl.c.m`

```c

... ...
elseif P == 24 % RB BANANA
    cx = -x(1)^2 - x(2);
end ...
...
```

5. Save all the files properly.

### 2.6.6 Defining Constrained Nonlinear Least Squares Problems

To illustrate how to define a linear constrained nonlinear least squares problem we add the constraint (7) to (6). The problem will be defined by following the stepwise instructions below (for all instructions we assume that you edit the copied files in a text editor):

1. Copy the files `cls.prob.m`, `cls.r.m` and `cls.J.m` to your directory NLPNEW.

2. Modify the files `cls.prob.m`, `cls.r.m` and `cls.J.m` in the same way as described for for the unconstrained case in Section 2.6.3.

3. Extend the problem definition in `cls.prob.m` with the constraint parameters:

```matlab
...
...
elseif P==28
    Name='RB BANANA';
    Yt=[10;0];      % r(x) = residual = model psi(t,x) - data Yt(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.
    A = [1 -1];    % Linear constraints matrix.
    b_L = -inf;    % Lower bounds on linear constraints.
    b_U = 1;      % Upper bounds on linear constraints.
else
    disp('cls_prob: Illegal problem number')
    pause
    Name=[];
end ...
...
```

4. Save all the files properly.

If you prefer alternative 3 you should instead copy the files `usr.prob.m`, `usr.r.m` and `usr.J.m` in step 1. In these files, replace the problem `Own Constrained LS problem 1` with `RB BANANA` in the same way as described above (do not forget the menu choice line in `usr.prob.m`).
2.6.7 Defining Quadratic Problems

Quadratic programming problems are defined in only one problem definition file, \textit{qp.prob.m}. The problem

\[
\begin{align*}
\text{min} & \quad f(x) = 4x_1^2 + x_1x_2 + 4x_2^2 + 3x_1 - 4x_2 \\
\text{s.t} & \quad x_1 + x_2 \leq 5 \\
& \quad x_1 - x_2 = 0 \\
& \quad x_1 \geq 0 \\
& \quad x_2 > 0,
\end{align*}
\]

(10)

named \textit{QP EXAMPLE}, will be used to help us illustrate how to define a quadratic programming problem.

To define (10) as a quadratic programming problem follow the stepwise instructions below (for all instructions we assume that you edit the copied file in a text editor):

1. Copy the file \textit{qp.prob.m} to your directory NLPNEW.

2. Add the problem name to the menu choice in \textit{qp.prob.m}:

\[
\begin{align*}
\text{...}
& \quad,'Bazaara IQP 9.29b pg 405. F singular'... \\
& \quad,'Bunch and Kaufman Indefinite QP'...
& \quad,'QP EXAMPLE'... \\
& \quad); \% MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES
\end{align*}
\]

\[
\begin{align*}
\text{if isempty(P)}
& \quad return;
\end{align*}
\]

\[
\text{end}
\]

\[
\text{...}
\]

3. Add the following in \textit{qp.prob.m} after the last already existing problem:

\[
\begin{align*}
\text{...}
\text{...}
\text{elseif P==15}
& \quad Name='QP EXAMPLE';
& \quad F = [ 8 2 \quad \% \ Hessian \\
& \quad 2 8 ];
& \quad c = [ 3 -4 ]'; \\
& \quad A = [ 1 1 \quad \% \ \text{Constraint matrix} \\
& \quad 1 -1 ];
& \quad b_L = [-\infty 0 ]'; \% \ \text{Lower bounds on the constraints}
& \quad b_U = [ 5 0 ]'; \% \ \text{Upper bounds on the constraints}
& \quad x_L = [ 0 0 ]'; \% \ \text{Lower bounds on the variables}
& \quad x_U = [ \infty \infty ]'; \% \ \text{Upper bounds on the variables}
& \quad x_0 = [ 0 1 ]'; \% \ \text{Starting point}
& \quad x_{\text{min}}=[-1 -1 ]; \% \ \text{Plot region parameters}
& \quad x_{\text{max}}=[ 6 6 ]; \% \ \text{Plot region parameters}
\text{else}
& \quad disp('qp_prob: Illegal problem number')
& \quad pause
& \quad Name=[];
\text{end}
\text{...}
\text{...}
\end{align*}
\]

4. Save the file properly.
If you prefer alternative 3 you should instead copy the file *usr_prob.m* in step 1. In this file, replace the problem *Own QP problem 1* with *QP EXAMPLE* in the same way as described above (do not forget the menu choice line in *usr_prob.m*).

### 2.6.8 Defining Exponential Sum Fitting Problems

Exponential sum fitting problems are defined in only one problem definition file, *exp_prob.m*. Assume that we want to fit a sum of exponential terms to the data series

\[
\begin{pmatrix}
30 \\
50 \\
70 \\
90 \\
110 \\
130 \\
150 \\
170 \\
190 \\
210 \\
230 \\
250 \\
270 \\
290 \\
310 \\
330 \\
350 \\
370 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
18299 \\
15428 \\
13342 \\
11466 \\
10077 \\
8720 \\
7382 \\
6708 \\
5932 \\
5352 \\
4734 \\
4278 \\
3744 \\
3453 \\
3111 \\
2950 \\
2686 \\
2476 \\
\end{pmatrix},
\]

\[
t = 10^{-3}, \quad y(t) = 10^{-4}
\]

here named *SW*

To define (11) as an exponential sum fitting problem follow the stepwise instructions below (for all instructions we assume that you edit the copied file in a text editor):

1. Copy the file *exp_prob.m* to your directory NLPNEW.

2. Add the problem name to the menu choice in *exp_prob.m*:

```matlab
\ldots
\ldots
\ldots
῾Atcexp nr2 ᵋ\ldots
῾Atcexp nr2\~ ᵋ\ldots
῾SW ᵋ\ldots
\); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES
\ldots
\ldots
\ldots
\ldots
\ldots
\ldots
\ldots
\ldots
```

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

```matlab
\ldots
\ldots
elseif P==44
    Name='SW';
    t=[30:20:370]; % Time in ms
\ldots
```
Yt=18299 15428 13347 11466 10077 8729 7382 6708 5932 5352 4734 4271 ... 3744 3485 3111 2950 2866 24761);
t=t/1000;  % Scale to seconds. Gives lambda*1000, of order 1
Yt=Yt/10000;  % Scale function values. Avoid large alpha
else
    disp('exp_prob: Illegal problem number')
    ...
end

4. Save the file properly.

If you prefer alternative 3 you should instead copy the file \textit{usr.prob.m} in step 1. In this file, replace the problem \textit{Own EF problem 1} with \textit{SW} in the same way as described above (do not forget the menu choice line in \textit{usr.prob.m}).

There are four different types of exponential terms available in NLPLIB TB. The type of exponential terms is determined by the parameter \textit{Prob.ExpFil.eType} which is set by defining the parameter \textit{eType} in the problem definition file:

...  
elseif P==44  
Name='SW';
t=[30:20:370];  % Time in ms
Yt=[18299 15428 13347 11466 10077 8729 7382 6708 5932 5352 4734 4271 ... 3744 3485 3111 2950 2686 24761);
t=t/1000;  % Scale to seconds. Gives lambda*1000, of order 1
Yt=Yt/10000;  % Scale function values. Avoid large alpha
eType = 1;
else
    disp('exp_prob: Illegal problem number')
    ...
end

The above definition of \textit{eType} is not necessary and was made just in illustrative purpose since 1 is the default value of \textit{eType}.

The four different types of exponential terms available in NLPLIB TB are given in Table 29.

**Table 29:** The different types of exponential terms.

<table>
<thead>
<tr>
<th>Type</th>
<th>Exponential Term</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$f(t) = \sum_{i=1}^{p} a_i e^{-\beta_i t}$</td>
<td>$a_i \geq 0$, $0 \leq \beta_1 &lt; \beta_2 &lt; \ldots &lt; \beta_p$</td>
</tr>
<tr>
<td>2</td>
<td>$f(t) = \sum_{i=1}^{p} a_i (1 - e^{-\beta_i t})$,</td>
<td>$a_i \geq 0$, $0 \leq \beta_1 &lt; \beta_2 &lt; \ldots &lt; \beta_p$</td>
</tr>
<tr>
<td>3</td>
<td>$f(t) = \sum_{i=1}^{p} k_i e^{-\beta_i t}$,</td>
<td>$a_i \geq 0$, $0 \leq \beta_1 &lt; \beta_2 &lt; \ldots &lt; \beta_p$</td>
</tr>
<tr>
<td>4</td>
<td>$f(t) = \sum_{i=1}^{p} (d_i \gamma_i) e^{-\beta_i t}$,</td>
<td>$a_i, \gamma_i \geq 0$, $0 \leq \beta_1 &lt; \beta_2 &lt; \ldots &lt; \beta_p$</td>
</tr>
</tbody>
</table>

**2.6.9 Defining Problems in Own Problem Definition Files**

Assume you have a collection of e.g. nonlinear least squares problems which you want to define in your own problem definition files. Also assume you have defined your problems in \textit{ls.prob}, \textit{ls.r} and \textit{ls.J} as described in Section 2.6.3. Of course, you can remove the already existing problems and define your first problem as number one. The extra modifications needed are:
1. Rename the files to for example *own_prob*, *own_r* and *own_J*.

2. Make the following modification in the beginning of *own_prob*:

```matlab
... ... if ask=-1 & ~isempty(Prob)
    if isstruct(Prob)
        if ~isempty(Prob.P)
            if P==Prob.P & strcmp(Prob.probFile,'own_prob'), return; end
        end
    end
... ...
```

3. Make the following modifications at the end of *own_prob*:

```matlab
... ...
Prob=mFiles(Prob,'ls_f','ls_g','ls_H',[],[],[],'own_r','own_J','ls_d2r');
... ...
```

4. Modify the file *nameprob.m* as described in the file. It should now look like:

```matlab
... ...
elseif solvType==4
    % Nonlinear Least Squares
    F=str2mat('ls_prob','mgh_prob','exp_prob','usr_prob...
    ,'usr_prob...
    ,'nts_prob...
    ,'own_prob...
    );
    % USER: Duplicate the row above and insert your own file name
    %       inside the quotes
    % USER: Uncomment next row if your latest file should be the default one.
    % D=size(F,1);
    
    N=str2mat(...
    ,'ls Nonlinear Least Squares'...
    ,'mgh More, Garbow, Hillstrom'...
    ,'exp Exponential Fitting'...
    ,'usr Nonlinear Least Squares'...
    ,'usr Exponential Fitting'...
    ,'nts Nonlinear Time Series Fitting'...
    ,'own My own least squares'...
    );
    % USER: Duplicate the row above and insert your own file name
    %       and description inside the quotes. Add the probType number to
    %       the vector probTypV below.
    probTypV=[4 4 5 4 5 7 4];
... ...
```
5. Do not forget the uncomment procedure if your file should be the default one.

6. Save all the files properly.

2.6.10 Special Notes

User Supplied Problem Parameters

The best way to describe this will be by giving some examples. Assume you have a problem with variable dimension. If you want to interactively give the dimension of the problem during the problem setup, the routine askparam will help you. Let us take problem 27 in cls.prob as an example.

```plaintext
... elseif P==27
    Name='RELN'; % Test for releasing more than one bound with variable dimension
    uP=checkuP(Name,Prob);
    % n variable. 1 <= n . default n=10
    n = askparam(ask, 'Give problem dimension ', 1, [], 10, uP);
    uP(1)=n;
    Yt=zeros(n,1);
    x_0 = zeros(n,1);
    %x_0 = 1E-5*ones(n,1);
    x_opt = 3.5*ones(n,1);
... ...
```

The parameter $uP$ which is a field in the problem structure $Prob$ is aimed for this kind of problems and we can see above that $uP(1)$ is set to the dimension supplied by the user. Type help askparam for information about the parameters sent to askparam. When user supplied parameters are to be handled the routine checkuP should be called in the same way as above (directly after the definition of the name of the problem). checkuP checks that the user parameters set in $uP$ (or $Prob.uP$) are the ones that is set for the actual problem in the first place. If they are set outside the system checkuP will let them keep those values.

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.

```plaintext
... ...
elseif P==27
    % 'RELN:
    n = Prob.uP(1);
... ...
```

If you want questions to be asked during the problem setup you must set the ask flag true in the call to probInit. See the example below:

```plaintext
ask=1;
Prob = probInit('cls_prob',27,ask);
```

The system will now ask you to give the problem dimension, and let us assume that you choose the dimension to be 20:

Current value = 10

Give problem dimension 20

Now we call clsSolve to solve the problem,
Result=clsSolve(Prob);

which gives the printing output

==========================================================
Iteration no:  0  Func   80.00000000000000000000000000000000 Cond 1
==========================================================
Iteration no:  1  Func   1.25000000000000000000000000000000 Cond 1
*** Convergence 2, Projected gradient small ***

As a second example let us assume that the user will solve the problem above for all dimensions between 10 and 30. Then the following code snippet will help us

for dim=10:30
    Prob = [];
    Prob.uP(1) = dim;
    PriLev = 0;
    Result = clsRun([],Prob,[],PriLev,'cls_prob',27);
end

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 we have defined the problem RB BANANA
(6) in the previous sections we have defined $x_{opt}$ to (1,1) in the problem definition files. Since we now that this
point is a minimum point we could extend the definition of $x_{opt}$ to

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

where $StatPntType$ equals 0, 1, or 2 depending on the type of the stationary point (minimum, saddle or maximum).
In our case we will set $StatPntType$ to 0 since (1,1) is a minimum point and the extension becomes

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

If there is more than one known stationary point, the points are defined as rows in a matrix with the values of
$StatPntType$ as the last column. Assume that (-1,-1) is a saddle point, (1,-2) is a minimum point and (-3,5)
is a maximum point for a certain problem. The definition of $x_{opt}$ could then look like

$$x_{opt} = [\begin{array}{ccc}
-1 & -1 & 1 \\
1 & -2 & 0 \\
-3 & 5 & 2 \\
\end{array}] ;$$

Note that it is not necessary to define $x_{opt}$, and if $x_{opt}$ is defined it is not necessary to define $StatPntType$. 
2.7 How to Solve Optimization Problems Using NLPLIB TB

In general, solving a problem in NLPLIB TB demands that you have defined the problem in the problem definition files as described in Section 2.6. There are one exception, quadratic programming problems could be solved by first defining the problem parameters in the Matlab Command Window and then call the appropriate solver. When you have defined your problem in the problem definition files, there are several possible ways to solve it. You can use the Graphical User Interface routine nlplib, the menu systems ucOpt, conOpt etc. or the driver routines ucRun, conRun, etc. You could also solve your problem by a direct call to the optimization routine. Which approach to choose depends on your purpose.

The interactive environments in the menu systems and the Graphical User Interface (GUI) are the most straightforward approaches. These choices give you easy access to all available utilities. How to use the menu systems and the GUI are described in Section 2.4 and Section 2.5, respectively.

When several problems are to be solved, e.g. in an algorithmic development environment, it is inefficient to use an interactive system. In this case, we recommend you to solve your problems by directly call the driver routines. In the reminder of this section we will illustrate how these driver routines are called, how you directly call an general optimization routine and how you can solve a quadratic program by a direct call to the actual solver.

To run the examples in this section the reader could either define the particular problem as described in the previous section or he could use the problem definition files in the directory ..\tomlab\nlpnew. Note that the nlpnew directory must be put before the nlplib directory in the Matlab path or chosen as the current working directory.

2.7.1 Using the Driver Routines

As a first example, we will solve the problem RB BANANA (6) defined as an unconstrained problem. Default values will be used for all parameters not explicitly changed. The following calls will solve our problem:

```matlab
probFile = 'uc_prob'; % Problem definition file.
P = 18; % Problem number.
Prob = probInit(probFile, P); % Setup Prob structure.

Result = ucRun([J, Prob, [J, J, probFile, P]);
```

To display the result of your run you just call the print routine PrintResult with your Result structure,

PrintResult(Result);

which gives the following printing output:

```
== * * * =========================================================== * * *
Problem 18: RB BANANA
          f_k   0.000000000000000000001
          User given f(x*)  0.00000000000000000000
          f(x 0)   24.199999999999996000

Solver: ucSolve. EXIT=0. TNORM=2.
Safeguarded BFGS

FuncEv 48 GradEv 48
NLPLIB Global Variable Counters give:
FuncEv 48 GradEv 41 iter 36
Starting vector x:
  x_0:  -1.200000  1.000000
Optimal vector x:
  x_k:   1.000000  1.000000
Diff x-x0:
          2.200000e+000 -2.312176e-009
```
Gradient g.k:
\[
g_k: \begin{bmatrix} -4.162202e-009 & 9.227064e-010 \end{bmatrix}
\]
NLPLTR found no active constraints.

If you want to solve the problem by using the Matlab routine \textit{fminu} you just add the definition of \textit{Solver} and then call the driver routine \textit{ucRun}:

\begin{verbatim}
probFile = 'uc_prob';  % Problem definition file.
P = 18;                % Problem number.
Prob = probInit(probFile, P);  % Setup Prob structure.
Solver = 'fminu';       % Solver routine.

Result = ucRun(Solver, Prob, [], [], probFile, P);
\end{verbatim}

Our second example is of a more "testing and developing" characteristic. We want to illustrate how the driver routines could be used in an efficient way. By use of a simple \textbf{for} loop we will solve all the least squares problems defined in the files \textit{own_prob}, \textit{own.r} and \textit{own.j}, see Section 2.6.9. We have chosen to explicitly set the values of several parameters, just in illustrative purpose. This procedure is not necessary since you could use the default values. The function \textit{drv.test} below runs \textit{lsRun} for all problems defined in \textit{own_prob}, and then displays the number of iterations performed. Instead of just printing the number of iterations, you can store some of the results for later use in e.g. statistical analysis.

\begin{verbatim}
function drv_test();

probFile = 'own_prob';      % Solve problems defined in own_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.
usr    = 0;     % Do not solve problem defined in usr_prob.m.

Solver = '1sSolve';

optParam = lsDef;  % Set default values.

optParam.PriLev    = 0;    % No printing output.
optParam.\text{eps}_x = 1E-7; % Termination tolerance for \textit{X} (Default=1E-8).
optParam.\text{eps}_f = 1E-9; % Termination tolerance on \textit{F}.(Default=1E-10). Dir.derivative
optParam.\text{eps}_c = 1E-5; % Termination criterion on constraint violation (Default=1E-6);
optParam.method    = 1;    % Optimization solver sub-method technique.
optParam.MaxIter   = 200;  % Maximum number of iterations. (Default 100*no. of variables)
optParam.\text{eps}_g = 1E-5; % Termination tolerance on gradient.(Default=1E-6).
optParam.\text{eps}_Rank = 1E-11; % Rank test tolerance. Used in subspace minimization.
optParam.wait      = 0;     % If true, pause after iteration printout.
optParam.\text{eps}_absf = 1E-35; % Absolute convergence tolerance in function \textit{f}.

optParam.LineSearch.sigma = 0.5;  % Line search accuracy sigma. (Default=0.9)

for P = 1:size(probNames,1),
    probNumber = P;

    Prob        = probInit(probFile, P, ask, [], [], usr);
    Prob.optParam = optParam;
\end{verbatim}
fprintf('\n Problem number %d:',P);
fprintf(' %s',Prob.Name);

Result = lsRun(Solver, Prob, ask, PriLev, probFile, probNumber);
fprintf('\n Number of iterations: %d',Result.Iter);

and

As a third example, the exponential sum fitting problem (11) are solved by:

probFile = 'exp_prob'; % Problem definition file.
P = 44; % Problem number.
Prob = probInit(probFile, P); % Setup Prob structure.

Result = clsRun([], Prob, [], [], probFile, P);

2.7.2 Direct Call to an Optimization Routine

When you want to solve your 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 .prob routine (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. In this subsection, we will give examples of the two different approaches.

First, we will solve the problem RB BANANA (6) as an unconstrained problem. In this case, we will have to define the problem in the files uc.prob, uc.f, uc.g and uc.H as described in Section 2.6.1. Using the problem definition files in the directory NLPLNEW we solve the problem and print the result by the following calls.

probFile = 'uc_prob'; % Problem definition file.
P = 18; % Problem number.
Prob = probInit(probFile, P); % Setup Prob structure.

Result = ucSolve(Prob);

PrintResult(Result);

Now, we will solve the same problem as in the example above but we will define the parameters x_0, x_L and x_U by calling the routine ProbAssign. Note that in this case we will not use the file uc.prob, only the uc.f, uc.g and uc.H files will be needed. The call to the routine mFiles is to declare in which files our problem is defined.

optType = '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(optType, x_0, [], x_L, x_U); % Setup Prob structure.
Prob = mFiles(Prob,'uc_f','uc_g'); % Problem definition files.
Prob.P = 18; % Problem number.

Result = ucSolve(Prob);

PrintResult(Result);

2.7.3 A Direct Approach to a QP Solution

We end up this section with an example of how to solve the quadratic programming problem (10) by a direct call to the routine gpSolve. Using this approach will eliminate the need of defining the problem in the problem definition files. The following definitions and call will illustrate the procedure:
Prob = ProbDef:

Prob.QP.F = [ 8 2 ] % Hessian.
              2 8 ];
Prob.QP.c = [ 3 -4 ]; % Constant vector.
Prob.x.L = [ 0 0 ]; % Lower bounds on the variables
Prob.x.U = [inf inf]; % Upper bounds on the variables
Prob.x.0 = [ 0 1 ]; % Starting point

Prob.A = [ 1 1 ] % Constraint matrix
           1 -1 ];
Prob.b.L = [-inf 0 ]; % Lower bounds on the constraints
Prob.b.U = [ 5 0 ]; % Upper bounds on the constraints

Result = qpSolve(Prob);

2.8 Printing Utilities and Print Levels

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

The main driver or menu routine called may have a PriLev parameter among its input parameters. This parameter determines the level of printing output of the result of the optimization.

The optimization routines normally sets the PriLev parameter to Prob.optParam.PriLev. The structure optParam which itself is a field in the structure Prob is set to default values by a call to optParamdef. The user may then change any values before calling the main routine, see Table 30. The fields in optParam is described in Table 6.

<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 optParam, optParam.wait. If this flag is set true, the routines uses the pause statement to avoid the output just flushing by.

Three global variables, MAX_c, MAX_x and MAX_r, are used as upper bounds for the number of constraints, variables and residuals to be printed. Those variables, useful for large problems, are set to default values by calling nlplibInit.

The NLPLIB TB routines print large amounts of output if high values for the PriLev parameter is set. To make the output look better and save space, several printing utilities have been developed, see Table 41 page 95. There is also a routine PrintResult which prints the results of an optimization given the Result structure.

For matrices there are two routines, mPrint and printmat. The routine printmat prints a matrix with row and column labels. The default is to print the row and column number. The standard row label is eight characters long. The supplied matrix name is printed on the first row, the column label row, if the length of the name is at most eight characters. Otherwise the name is printed on a separate row.

The standard column label is seven characters long, which is the minimum space an element will occupy in the print out. On a 80 column screen, then it is possible to print a maximum of ten elements per row. Independence on the number of rows in the matrix, printmat will first display A(:, 1:10), then A(:, 11:20) and so on.
The routine printmat tries to be intelligent and avoid decimals when the matrix elements are integers. It determines the maximal positive and minimal negative number to find out if more than the default space is needed. If any element has an absolute value below \(10^{-9}\) (avoiding exact zeros) or if the maximal elements are too big, a switch is made to exponential format. The exponential format uses ten characters, displaying two decimals and therefore seven matrix elements are possible to display on each row.

For large matrices, especially integer matrices, the user might prefer the routine mPrint. With this routine a more dense output is possible. All elements in a matrix row is displayed (over several output rows) before next matrix row is printed. A row label with the name of the matrix and the row number is displayed to the left using the Matlab style of syntax.

The default in mPrint is to eight characters per element, with two decimals. However, it is easy to change the format and the number of elements displayed. For a binary matrix it is possible to display 36 matrix columns in one 80 column row.

2.9 Notes about Special Features

The aim of this section is to give short descriptions of some special features available in NLPLIB TB. The list (in form of subsections) does not claim to be complete so the reader should consult Section 2.1 to get a complete picture of the system.

2.9.1 Approximation of Derivatives

Both numerical differentiation and automatic differentiation are available. For numerical differentiation there are four different approaches.

First there is the classical approach with forward or backward differences together with an automatic step selection procedure. This is handled by the routines fdng which is a direct implementation of the FD algorithm [28, page 343].

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

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

Automatic differentiation is performed by use of the ADMAT TB, for information of how to get a copy of ADMAT TB see http://simon.cs.cornell.edu/home/verma/AD/. Below, we give a short instruction of how to install it.

1. Install the ADMAT TB at e.g. d:\Admat\. 

2. Change the path commands in ...	omlab\nlplib\admatInit.m and execute the file. (If you choose 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

ADMAI TB should be initialized by calling admatInit before running NLPLIB TB with automatic differentiation. Note that if NLPLIB TB should be fully compatible with the ADMAT TB then your functions must be defined
according to the ADMAT TB requirements. Some of the predefined test problems in NLPLIB TB do not fulfill those requirements.

In the Graphical User Interface, differentiation strategy selection is made from the Diff menu reachable in advanced mode. When running the menu routines you should 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 you just set Prob.AutoDiff equal to 1 for automatic differentiation or Prob.NumDiff to 1, 2, 3 or 4 for numerical differentiation, before calling drivers or solvers. Note that Prob.NumDiff = 1 will run the fdng routine and Prob.NumDiff = 2, 3, 4 will run the Spline Toolbox routines csapi, csaps and spaps correspondingly. The csaps demands that a smoothness parameter is set and the spaps routine demands that a tolerance parameter is set. Those parameters are asked for when the corresponding routine is chosen but could also be explicitly set by the user via the splineSmooth and splineTol fields in the optimization parameter structure optParam, see Table 6. The user should be aware of that there is no guarantee that the default values of splineSmooth and splineTol are appropriately chosen.

Here follows some examples of the use of approximative derivatives when running the driver routines ucRun and clsRun.

**Automatic Differentiation example**

```plaintext
probFile = 'uc_prob';
P = 1;
Prob = probInit(probFile, P);
Solver = 'ucSolve';
Prob.Solver.Alg = 1;
Prob.AutoDiff = 1; % Use Automatic Differentiation.
Result = ucRun(Solver, Prob, [], [], probFile, P);
```

**FD example**

```plaintext
probFile = 'uc_prob';
P = 1;
Prob = probInit(probFile, P);
Solver = 'ucSolve';
Prob.Solver.Alg = 1;
Prob.NumDiff = 1; % Use the fdng routine.
Result = ucRun(Solver, Prob, [], [], probFile, P);
```

**Spline example**

```plaintext
probFile = 'ls_prob';
P = 1;
Prob = probInit(probFile, P);
Solver = 'lsSolve';
Prob.Solver.Alg = 0;
Prob.NumDiff = 2; % Use the Spline Toolbox routine csapi.
Result = lsRun(Solver, Prob, [], [], probFile, P);
```

### 2.9.2 Partially Separable Functions

The routine sTrustR implements a structured trust region algorithm for partially separable functions (psf). We will here give the definition of a psf and illustrate how such a function is defined.

A function \( f(x) \) is partially separable if \( f(x) = \sum_{i=1}^{M} f_i(x) \), where, for each \( i \in \{1, ..., M\} \) 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 \( DAS 2 \):

\[
\begin{align*}
\min_x & \quad f(x) = \frac{1}{2} \sum_{i=1}^{8} r_i(x)^2 \\
\text{s.t.} & \quad Ax \geq b \\
& \quad x \geq 0
\end{align*}
\]

where

\[
\begin{bmatrix}
\sqrt{11} \cdot x_1 - \frac{3}{\sqrt{11}} \\
\frac{\sqrt{2}}{11} \cdot x_2 - \frac{3}{\sqrt{11}} \\
\sqrt{0.0775} \cdot x_3 + \frac{0.5}{\sqrt{0.0775}} \\
\frac{2}{\sqrt{2}} \cdot x_4 + \frac{3}{\sqrt{2}} \\
-\frac{5}{6} x_1 + 0.6 x_3 \\
0.75 x_3 + \frac{2}{3} x_4
\end{bmatrix}, \quad A = \begin{bmatrix}
-1 & -2 & -1 & -1 \\
-3 & -1 & -2 & 1 \\
0 & 1 & 4 & 0
\end{bmatrix}, \quad b = \begin{bmatrix}
-5 \\
-4 \\
1.5
\end{bmatrix}.
\]

The objective function in (12) 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 constrained problem 14 in \textit{con.prob}, \textit{con.f}, \textit{con.g} etc. to be an illustrative example of how to define a problem with a partially separable objective function. Note the definition of \textit{pSepFunc} in \textit{con.prob}.

Solving (12) with \textit{sTrustR} is done by the following definitions and call:

\begin{verbatim}
probFile = 'con_prob';
P     = 14;
Prob  = probInit(probFile,P);
Solver = 'sTrustR';
Result = conRun(Solver,Prob,[],[],probFile,P);
\end{verbatim}

### 2.9.3 Recursive solver calls

For solving some kinds of problems it could be suitable or even necessary to apply algorithms which is based on a recursive approach. Here, we by a recursive approach also include those cases where you in each iteration solves an optimization problem as a subproblem. For example, the EGO algorithm (implemented in the routine \textit{ego}) solves an unconstrained (uc) and a box-bounded global optimization problem (gb) in each iteration. As we mentioned in Section 2.1.1 NLPLIB TB uses a number of global variables. To avoid that those variables are not reinitialized or given new values by the underlying procedure NLPLIB TB 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.

The method described above to handle the problem of global variables in recursive algorithms are treated by the two routines \textit{globalSave} and \textit{globalGet}. The \textit{globalSave} routine saves all global variables in a structure \textit{glbSave(depth)} and then initialize all of them as empty. By using the depth variable, an arbitrarily number of recursions are possible. The other routine \textit{globalGet} retrieves all global variables in the structure \textit{glbSave(depth)}.

To illustrate the idea, we have pasted the parts of the \textit{ego} code where the routines \textit{globalSave} and \textit{globalGet} are called.

\begin{verbatim}
... globalSave(1);
EGOResult = glbSave(EGOProb);
globalGet(1);
...
... globalSave(1);
[DACEResult] = ucSolve(DACEProb);
\end{verbatim}
2.10 Driver Routines in NLPLIB TB

In the following subsections the driver routines in NLPLIB TB will be described.

2.10.1 clsRun

Purpose
Driver routine for constrained nonlinear least squares solvers.

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

Description of Inputs

- **Solver** 
The name of the solver that should be used to optimize the problem. Default clsSolve. If the solver may run several different optimization algorithms, then the values of Prob.optParam.alg and Prob.optParam.subalg determines which algorithm.

- **Prob** 
Problem description structure, see Table 5.

- **ask** 
Flag if questions should be asked during problem definition.
  - \( ask < 0 \) Use values in Prob.uP if defined or defaults.
  - \( ask = 0 \) Use defaults.
  - \( ask > 1 \) Ask questions in probFile.
  - \( ask = [-1] \) If Prob.uP = [-1], ask = -1, else ask = 0.

- **PriLev** 
Print level when displaying the result of the optimization in the routine PrintResult. See Section 2.13.1 page 88.
  - PriLev = 0 No output.
  - PriLev = 1 Final result, shorter version.
  - PriLev = 2 Final result.
  - PriLev = 3 Full results.

- **probFile** 
User problem init file, default cls_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 15.

Description
The driver routine clsRun is called by the menu routine clsOpt or the graphical user interface routine nlplib to solve constrained nonlinear least squares problems defined in your problem definition files. It is also possible for the user to call clsRun directly from the Matlab command prompt, see Section 2.7. Via clsRun you can run the TOMLAB internal solvers clsSolve and conSolve and the Matlab Optimization Toolbox solver constr. You can also, by use of a MEX-file interface run the commercial optimization solvers NLSSOL, MINOS, NPSOL and NPOPT.

M-files Used
xxxRun.m, xxxRun2.m, npopt.m, inibuild.m, clsDef.m, probInit.m, mknbound.m, clsSolve.m, conSolve.m, solrun.m, nlssol.m, minos.m, npsol.m, PrintResult.m, iniSolve.m, endSolve.m

2.10.2 conRun

Purpose
Driver routine for constrained optimization solvers.

Calling Syntax
Result = conRun(Solver, Prob, ask, PriLev, probFile, probNumber);
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.optParam alg* and *Prob.optParam subalg* determines which algorithm.

**Prob**
Problem description structure, see Table 5.

**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 2.13.1 page 88.
- *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.optParam.PriLev*.

**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 15.

Description

The driver routine *conRun* is called by the menu routine *conOpt* or the graphical user interface routine *nlplib* to solve constrained optimization problems defined in your problem definition files. It is also possible for the user to call *conRun* directly from the Matlab command prompt, see Section 2.7. Via *conRun* you can run the TOMLAB internal solvers *consolve*, *sTrustR* and *nlpsolve* and the Matlab Optimization Toolbox solver *constr*. You can also run by use of a MEX-file interface run the commercial optimization solvers MINOS, NPSOL and NPOPT.

**M-files Used**

`conRun.m`, `conRun2.m`, `PrintResult.m`, `inibuild.m`, `conDef.m`, `probInit.m`, `mkbound.m`, `consolve.m`, `nlpsolve.m`, `solrun.m`, `minos.m`, `npsol.m`, `npopt.m`

**2.10.3 gblRun**

**Purpose**
Driver routine for box-bounded global optimization.

**Calling Syntax**
Result = gblRun(Solver, Prob, ask, PriLev, probFile, probNumber);
Description of Inputs

**Solver**
The name of the solver that should be used to optimize the problem. Default `glsolve`. If the solver may run several different optimization algorithms, then the values of `Prob.optParam.alg` and `Prob.optParam.subalg` determines which algorithm.

**Prob**
Problem description structure, see Table 5.

**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 2.13.1 page 88.
- `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.optParam.PriLev`.

**probFile**
User problem init file, default `gls_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 15.

Description
The driver routine `glsRun` is called by the menu routine `glsOpt` or the graphical user interface routine `npltb` to solve global optimization problems defined in your problem definition files. It is also possible for the user to call `glsRun` directly from the Matlab command prompt, see Section 2.7. Via `glsRun` you can run the TOMLAB internal solver `glsolve`.

M-files Used
`xxxRun.m, xxxRun2.m, PrintResult.m, inibuild.m, ucDef.m, probInit.m, mkbound.m, glsSolve.m, iniSolve.m, endSolve.m`

2.10.4 **glsRun**

Purpose
Driver routine for global mixed-integer nonlinear programming.

Calling Syntax
Result = glsRun(Solver, Prob, ask, PriLev, probFile, probNumber)
Description of Inputs

Solver  The name of the solver that should be used to optimize the problem. Default
    glcSolve. If the solver may run several different optimization algorithms,
    then the values of Prob.optParam.alg and Prob.optParam.subalg determines
    which algorithm.

Prob  Problem description structure, see Table 5.
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 2.13.1 page 88.
    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.optParam.PriLev.

probFile  User problem init file, default glc 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 15.

Description

The driver routine glcRun is called by the menu routine glcOpt or the graphical user interface routine nlpLib to solve
constrained global optimization problems defined in your problem definition files. It is also possible for the user
to call glcRun directly from the Matlab command prompt, see Section 2.7. Via glcRun you can run the TOMLAB
internal solver glcSolve.

M-files Used
xxxRun.m, xxxRun2.m, PrintResult.m, inibuild.m, conDef.m, probInit.m, mkbound.m, glcSolve.m, iniSolve.m,
endSolve.m

2.10.5  IsRun

Purpose

Driver routine for nonlinear least squares solvers.

Calling Syntax

Result = IsRun(Solver, Prob, ask, PriLev, probFile, probNumber)
Description of Inputs

**Solver**
The name of the solver that should be used to optimize the problem. Default lsSolve. If the solver may run several different optimization algorithms, then the values of `Prob.optParam.alg` and `Prob.optParam.subalg` determines which algorithm.

**Prob**
Problem description structure, see Table 5.

**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`.

**PriLev**
Print level when displaying the result of the optimization in the routine `PrintResult`. See Section 2.13.1 page 88.
- `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.optParam.PriLev`.

**probFile**
User problem init file, default is `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 15.

Description

The driver routine `lsRun` is called by the menu routine `lsOpt` or the graphical user interface routine `nlplib` to solve nonlinear least squares problems defined in your problem definition files. It is also possible for the user to call `lsRun` directly from the Matlab command prompt, see Section 2.7. Via `lsRun` you can run the TOMLAB internal solvers `lsSolve` and `ucSolve` and the MatlabOptimization Toolbox solver `leastsq`. You can also, by use of a MEX-file interface run the commercial optimization solver NLSSOL.

M-files Used

`xxxRun.m`, `xxxRun2.m`, `PrintResult.m`, `inibuild.m`, `lsDef.m`, `probInit.m`, `mkbound.m`, `lsSolve.m`, `ucSolve.m`, `solrun.m`, `nlssl.m`, `iniSolve.m`, `endSolve.m`

2.10.6 **qpRun**

Purpose

Driver routine for quadratic programming solvers.

Calling Syntax

Result = qpRun(Solver, Prob, ask, PriLev, probFile, probNumber)
Description of Inputs

Solver The name of the solver that should be used to optimize the problem. Default \texttt{qpSolve}. If the solver may run several different optimization algorithms, then the values of \texttt{Prob.optParam.alg} and \texttt{Prob.optParam.subalg} determines which algorithm.

Prob Problem description structure, see Table 5.

ask Flag if questions should be asked during problem definition.
- \texttt{ask < 0} Use values in \texttt{uP} if defined or defaults.
- \texttt{ask = 0} Use defaults.
- \texttt{ask > 1} Ask questions in \texttt{probFile}.
- \texttt{ask = | |} If \texttt{uP = | |}, \texttt{ask} = \texttt{-1}, else \texttt{ask} = \texttt{0}.

PriLev Print level when displaying the result of the optimization in the routine \texttt{PrintResult}. See Section 2.13.1 page 88.
- \texttt{PriLev = 0} No output.
- \texttt{PriLev = 1} Final result, shorter version.
- \texttt{PriLev = 2} Final result.
- \texttt{PriLev = 3} Full results.

The printing level in the optimization solver is controlled by setting the parameter \texttt{Prob.optParam.PriLev}.

probFile User problem init file, default \texttt{qp_prob.m}.

probNumber Problem number in \texttt{probFile}. \texttt{probNumber = 0} gives a menu in \texttt{probFile}.

Description of Outputs

Result Structure with result from optimization, see Table 15.

Description

The driver routine \texttt{qpRun} is called by the menu routine \texttt{qpOpt} or the graphical user interface routine \texttt{nlplib} to solve quadratic programming problems defined in your problem definition files. It is also possible for the user to call \texttt{qpRun} directly from the Matlab command prompt, see Section 2.7. Via \texttt{qpRun} you can run the TOMLAB internal solvers \texttt{qpe}, \texttt{qplm}, \texttt{qpiOld} and \texttt{qpISolve} (not fully developed) and the Matlab Optimization Toolbox solver \texttt{qp}. Currently NLPLIB TB also includes a not fully developed routine \texttt{qpBiggs} for negative definite quadratic problems.

M-files Used
\texttt{xxeRun.m, xxeRun2.m, PrintResult.m, inibuild.m, conDef.m, probInit.m, mkbound.m, qpe.m, qplm.m, qpsolve.m, qpBiggs.m, iniSolve.m, endSolve.m}

2.10.7 \texttt{ucRun}

Purpose

Driver routine for unconstrained optimization solvers.

Calling Syntax

Result = \texttt{ucRun}(Solver, Prob, ask, PriLev, probFile, probNumber)
Description of Inputs

Solver  The name of the solver that should be used to optimize the problem. Default ucSolve. If the solver may run several different optimization algorithms, then the values of Prob.optParam.alg and Prob.optParam.subalg determines which algorithm.

Prob  Problem description structure, see Table 5.

ask  Flag if questions should be asked during problem definition.

\[ \text{ask} \begin{cases} < 0 & \text{Use values in uP if defined or defaults.} \\ = 0 & \text{Use defaults.} \\ > 1 & \text{Ask questions in probFile.} \end{cases} \]

ask = 1  If uP = [], ask = -1, else ask = 0.

PriLev  Print level when displaying the result of the optimization in the routine PrintResult. See Section 2.13.1 page 88.

\[ \text{PriLev} \begin{cases} = 0 & \text{No output.} \\ = 1 & \text{Final result, shorter version.} \\ = 2 & \text{Final result.} \\ = 3 & \text{Full results.} \end{cases} \]

The printing level in the optimization solver is controlled by setting the parameter Prob.optParam.PriLev.

probFile  User problem init file, default uc_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 15.

Description

The driver routine ucRun is called by the menu routine ucOpt or the graphical user interface routine nlpLib tc solve unconstrained optimization problems defined in your problem definition files. It is also possible for the user to call ucRun directly from the Matlab command prompt, see Section 2.7. Via ucRun you can run the TOMLAB internal solver ucSolve and the Matlab Optimization Toolbox solvers fmins and fminu. You can also, by use of a MEX-file interface run the commercial optimization solver MINOS.

M-files Used

\[ \text{xxxRun.m, xxxRun2.m, xxxRun3.m, initbuild.m, ucDef.m, probInit.m, mkbound.m, ucSolve.m, minos.m, iniSolve.m, endSolve.m} \]

2.11 Optimization Routines in NLPLIB TB

In the following subsections the optimization routines in NLPLIB TB will be described.

2.11.1 clsSolve

Purpose

Solve nonlinear least squares optimization problems with linear inequality and equality constraints and simple bounds on the variables.

clsSolve solves problems of the form

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

where \( x, x_L, x_U \in \mathbb{R}^n, r(x) \in \mathbb{R}^n, A \in \mathbb{R}^{m \times n} \) 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: Gauss-Newton (default).
1: Fletcher - Xu hybrid method; Gauss-Newton / BFGS.
2: Al-Baali - Fletcher hybrid method; Gauss-Newton/BFGS.
3: Huschens method.

optParam Structure with special fields for optimization parameters, see Table 6.
Fields used are: PreSolve, NOT.release.all, eps.f, eps.g, eps.c, eps.x, eps.Rank,
 eps.absf, MaxIter, want, size.x, size.f, f.Low, LineSearch, LineAlg, b.Tol, c.Tol, x.Tol,
LowItls, method, PriLev and QN.InitMatrix.

NLLS Structure with special fields for nonlinear least squares, see Table 9.

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.
p.H Name of m-file computing the Hessian matrix \( H(x) \).
p.r Name of m-file computing the residual vector \( r(x) \).
p.J Name of m-file computing the Jacobian matrix \( J(x) \).
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 Binary code giving exit status:

1: Iteration points are close.
2: Projected gradient small.
4: Function value close to 0.
8: Relative function value reduction low for LowItls 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.

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.
r.k Residual at optimum.
J.k Jacobian matrix at optimum.
xState State of each variable, described in Table 16.
bState State of each linear constraint, described in Table 17.
Solver Solver used.
SolverAlgorithm Solver algorithm used.
Prob Problem structure used.

Description

The prototype routine clsSolve includes four optimization methods for nonlinear least squares problems: the Gauss-Newton method, the Al-Baali-Fletcher [5] and the Fletcher-Xu [21] hybrid method, and the Huschens TSSM method [36]. If rank problem occur, the prototype algorithm is using subspace minimization. The line search is performed using the routine LineSearch which is a modified version of an algorithm by Fletcher [22]. Bound
Constraints are partly treated as described in Gill, Murray and Wright [28]. clsSolve treats linear equality and inequality constraints using an active set strategy and a null space method.

**Algorithm**
See Appendix A.1.

**M-files Used**
clsDef.m, ResultDef.m, preSolve.m, apSolve.m, qpoptSOL.m, LineSearch.m, secUpdat.m, iniSolve.m, endSolve.m

**See Also**
lsSolve, 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.

### 2.11.2 conSolve

**Purpose**
Solve general constrained nonlinear optimization problems.

conSolve solves problems of the form

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

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**
    Solver algorithm to be run:
    0: Schittkowski SQP.
    1: Han-Powell SQP.
  - **optParam**
    Structure with special fields for optimization parameters, see Table 6.
  - **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.
  - **p.f**
    Name of m-file computing the objective function \(f(x)\).
  - **p.g**
    Name of m-file computing the gradient vector \(g(x)\).
  - **p.H**
    Name of m-file computing the Hessian matrix \(H(x)\).
  - **p.c**
    Name of m-file computing the vector of constraint functions \(c(x)\).
  - **p.dc**
    Name of m-file computing the matrix of constraint normals \(\partial c(x)/\partial x\).
  - **f.Low**
    Lower bound on function value.

- **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: Small search direction.
    4: Merit function gradient small.
    8: Small p and constraints satisfied.
    101: Maximum number of iterations reached.
    102: Function value below given estimate.
    103: Close iterations, 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.

- **f0**
  - Function value at start.

- **fK**
  - Function value at optimum.

- **gK**
  - Gradient value at optimum.

- **Hk**
  - Hessian value at optimum.

- **x0**
  - Starting point.

- **xK**
  - Optimal point.

- **vK**
  - Lagrange multipliers.

- **cK**
  - Value of constraints at optimum.

- **cJac**
  - Constraint Jacobian at optimum.

- **xState**
  - State of each variable, described in Table 16.

- **bState**
  - State of each linear constraint, described in Table 17.

- **cState**
  - State of each general constraint.

- **Solver**
  - Solver used.

- **SolverAlgorithm**
  - Solver algorithm used.

- **Proh**
  - Problem structure used.

Description

The routine `con Solve` implements two SQP algorithms for general constrained minimization problems. One implementation, `optParam alg = 0`, is based on the SQP algorithm by Schittkowski with Augmented Lagrangian merit function described in [50]. The other, `optParam alg = 1`, is an implementation of the HanPowell SQP method.

M-files Used

- `conDef.m`
- `ResultDef.m`
- `qpsolve.m`
- `qpoptSOL.m`
- `LineSearch.m`
- `iniSolve.m`
- `endSolve.m`

See Also

- `nlpSolve`
- `sTrustR`

### 2.11.3 gblSolve

#### Purpose

Solve box-bounded global optimization problems. `gblSolve` is a stand-alone version of `glbSolve` and runs independently of NLPLIB TB.

**gblSolve** 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

```matlab
Result = gblSolve(fun, x_L, x_U, GLOBAL, PriLev);
```
Description of Inputs

- **fun** Name of m-file computing the function value, given as a string.
- **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.
- **GLOBAL** Structure field containing:
  - **iterations** Number of iterations, default 50.
  - **epsilon** Global/local weight parameter, default $10^{-4}$.

  If restart is wanted, the following fields in **GLOBAL** should be defined and equal the corresponding fields in the **Result.GLOBAL** structure from the previous run:
  - **U** 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.

- **PriLev** Printing level.

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:
    - **U** 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.

Description

The global optimization routine **gblSolve** is an implementation of the DIRECT algorithm presented in [38]. 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 **gblSolve** 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 **gblSolve** with the final status of all parameters from the previous run. Let’s say that you have run **gblSolve** on a certain problem for 50 iterations. Then you could run e.g. 40 iterations more and get the same result as if you had chosen to run 90 iterations in the first place. To restart **gblSolve** you must give the result of the first run as input to your next run. The m-file **gbSolve** also includes the subfunction **conhull** which is an implementation of the algorithm GRAHAMHULL in [48, 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.

Since **gblSolve** is a stand-alone version of **gblSolve** it runs independently of NLPLIB TB.

Algorithm

See Appendix A.2.

2.11.4 **gclSolve**

Purpose

Solve global mixed-integer nonlinear programming problems. **gclSolve** is a stand-alone version of **gclSolve** and runs independently of NLPLIB TB.
$gclSolve$ solves problems of the form

$$\min_x \ f(x)$$

subject to

$$s/t \quad x_L \leq x \leq x_U$$
$$b_L \leq Ax \leq b_U$$
$$c_L \leq c(x) \leq c_U$$

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**


**Description of Inputs**

- **p.f** Name of m-file computing the function value, given as a string
- **p.c** Name of m-file computing the function value, given as a string
- **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.
- **I** Set of integer variables (a vector).
- **GLOBAL** Structure field containing:
  - **MaxEval** Number of function evaluations, default 200.
  - **epsilon** Global/local weight parameter, default $10^{-4}$

If restart is wanted, the following fields in **GLOBAL** should be defined and equal the corresponding fields in the **Result.GLOBAL** structure 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.
- **LL** $LL(i, j)$ is the lower bound for rectangle $j$ in integer dimension $I(i)$.
- **LU** $LU(i, j)$ 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 $i$.

**PriLev** Printing level.
Description of Outputs

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

- **Iter** Number of iterations.
- **FuncFn** 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.
- **LL** \( LL(i, j) \) is the lower bound for rectangle \( j \) in integer dimension \( I(i) \).
- **LU** \( LU(i, j) \) is the upper bound for rectangle \( j \) in integer dimension \( I(i) \).
- **feasible** Flag indicating if a feasible point has been found.
- **f_min** Best function value found at a feasible point.
- **s_0** \( s_0 \) is used as \( s(0) \).
- **s** \( s(j) \) is the sum of observed rates of change for constraint \( j \).
- **t** \( t(i) \) is the total number of splits along dimension \( i \).

Description

The routine *gclSolve* implements an extended version of DIRECT, see [39], 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 *gclSolve* 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 *gclSolve* with the final status of all parameters from the previous run. Let’s say that you have run *gclSolve* on a certain problem for 50 function evaluations. Then you could run e.g. for 200 function evaluations more and let *gclSolve* search for a point that gives a lower function value. To restart *gclSolve* you must give the result of the first run as input to your next run.

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.

Since *gclSolve* is a stand-alone version of *gloSolve* it runs independently of NLPLIB TB.

### 2.11.5 *glbSolve*

**Purpose**

Solve box-bounded global optimization problems.

*glbSolve* solves problems of the form

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

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

**Calling Syntax**

Result = glbSolve(Prob, varargin)
**Description of Inputs**

*Prob*  
Problem description structure. The following fields are used:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>optParam</td>
<td>Structure with special fields for optimization parameters, see Table 6.</td>
</tr>
<tr>
<td>x.L</td>
<td>Lower bounds for x, must be given to restrict the search space.</td>
</tr>
<tr>
<td>x.U</td>
<td>Upper bounds for x, must be given to restrict the search space.</td>
</tr>
<tr>
<td>p.f</td>
<td>Name of m-file computing the objective function f(x).</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>Special structure field containing:</td>
</tr>
<tr>
<td>iterations</td>
<td>Number of iterations, default 50.</td>
</tr>
<tr>
<td>epsilon</td>
<td>Global/local weight parameter, default 10^{-4}.</td>
</tr>
<tr>
<td>K</td>
<td>The Lipschitz constant. Not used.</td>
</tr>
<tr>
<td>tolerance</td>
<td>Error tolerance parameter. Not used.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Matrix with all rectangle centerpoints.</td>
</tr>
<tr>
<td>D</td>
<td>Vector with distances from centerpoint to the vertices.</td>
</tr>
<tr>
<td>L</td>
<td>Matrix with all rectangle side lengths in each dimension.</td>
</tr>
<tr>
<td>F</td>
<td>Vector with function values.</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>
</tbody>
</table>

*varargin*  
Other parameters directly sent to low level routines.

**Description of Outputs**

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iter</td>
<td>Number of iterations</td>
</tr>
<tr>
<td>FuncEv</td>
<td>Number function evaluations.</td>
</tr>
<tr>
<td>x.k</td>
<td>Matrix with all points giving the function value f.k.</td>
</tr>
<tr>
<td>f.k</td>
<td>Function value at optimum.</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>Special structure field containing:</td>
</tr>
<tr>
<td>C</td>
<td>Matrix with all rectangle centerpoints.</td>
</tr>
<tr>
<td>D</td>
<td>Vector with distances from centerpoint to the vertices.</td>
</tr>
<tr>
<td>L</td>
<td>Matrix with all rectangle side lengths in each dimension.</td>
</tr>
<tr>
<td>f'</td>
<td>Vector with function values.</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>Solver</td>
<td>Solver used.</td>
</tr>
<tr>
<td>SolverAlgorithm</td>
<td>Solver algorithm used.</td>
</tr>
</tbody>
</table>

**Description**

The global optimization routine *gbiSolve* is an implementation of the DIRECT algorithm presented in [38]. 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 *gbiSolve* 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 *gbiSolve* with the final status of all parameters from the previous run. Let’s say that you have run *gbiSolve* on a certain problem for 50 iterations. Then you could run e.g. 40 iterations more and get the same result as if you had chosen to run 90 iterations in the first place. To restart *gbiSolve* you must give the result of the first run as input to your next run. The m-file *gbiSolve* also includes the subfunction *conhull* which is an implementation of the algorithm GRAHAMHULL in [48, 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.

**Algorithm**

See Appendix A.2.

**M-files Used**

*iniSolve.m, endSolve.m*
2.11.6 glcSolve

Purpose
Solve global mixed-integer nonlinear programming problems.

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

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

Calling Syntax
Result = glcSolve(Prob,varargin);

Description of Inputs
Prob

Problem description structure. The following fields are used:

- **optParam** Structure with special fields for optimization parameters, see Table 6.
  - Fields used are: PnlLev, 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.
  - \( p.f \) Name of m-file computing the objective function \( f(x) \).
  - \( p.c \) 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:

- \( U \) 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.
- **LL** \( \text{L}(i,j) \) is the lower bound for rectangle \( j \) in integer dimension \( I(i) \).
- **LU** \( \text{L}(i,j) \) 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 \( i \).

varargin

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

<table>
<thead>
<tr>
<th>Result</th>
<th>Structure with result from optimization. The following fields are changed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iter</td>
<td>Number of iterations.</td>
</tr>
<tr>
<td>FuncEv</td>
<td>Number function evaluations.</td>
</tr>
<tr>
<td>x.k</td>
<td>Matrix with all points giving the function value f_k.</td>
</tr>
<tr>
<td>f.k</td>
<td>Function value at optimum.</td>
</tr>
<tr>
<td>c.k</td>
<td>Nonlinear constraints values at x.k.</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>Special structure field containing:</td>
</tr>
<tr>
<td>C</td>
<td>Matrix with all rectangle centerpoints.</td>
</tr>
<tr>
<td>D</td>
<td>Vector with distances from centerpoint to the vertices.</td>
</tr>
<tr>
<td>F</td>
<td>Vector with function values.</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>T</td>
<td>T(i) is the number of times rectangle i has been trisected.</td>
</tr>
<tr>
<td>G</td>
<td>Matrix with constraint values for each point.</td>
</tr>
<tr>
<td>ignoredx</td>
<td>Rectangles to be ignored in the rectangle selection procedure.</td>
</tr>
<tr>
<td>LL</td>
<td>LL(i, j) is the lower bound for rectangle j in integer dimension I(i).</td>
</tr>
<tr>
<td>LU</td>
<td>LU(i, j) is the upper bound for rectangle j in integer dimension I(i).</td>
</tr>
<tr>
<td>feasible</td>
<td>Flag indicating if a feasible point has been found.</td>
</tr>
<tr>
<td>f.min</td>
<td>Best function value found at a feasible point.</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>Solver</td>
<td>Solver used.</td>
</tr>
<tr>
<td>SolverAlgorithm</td>
<td>Solver algorithm used.</td>
</tr>
</tbody>
</table>

Description

The routine glcSolve implements an extended version of DIRECT, see [39], 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. Let’s say that you have run glcSolve on a certain problem for 500 function evaluations. Then you could run e.g. for 200 function evaluations more and let glcSolve search for a point that gives a lower function value. To restart glcSolve you must give the result of the first run as input to your next run.

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

initSolve.m, endSolve.m

2.11.7 **IsSolve**

Purpose

Solve nonlinear least squares optimization problems with simple bounds on the variables.

IsSolve solves problems of the form

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

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

Calling Syntax

Result = IsSolve(Prob, varargin)
Description of Inputs
Prob Problem description structure. The following fields are used:
Solver.Alg Solver algorithm to be run:
0: Gauss-Newton (default).
1: Fletcher - Xu hybrid method; Gauss-Newton / BFGS.
2: Al-Baali - Fletcher hybrid method; Gauss-Newton/BFGS.
3: Huschens method.
optParam Structure with special fields for optimization parameters, see Table 6.
Fields used are: NOT.release_all, eps.f, eps.g, eps.c, eps.x, eps.Rank, eps.absf,
MaxIter, wait, size.x, size.f, tol, LineSearch, LineAlg, xTol, Lowlts, method,
PriLev and QN_InitMatrix.
NLLS Structure with special fields for nonlinear least squares, see Table 9.
x L Lower bounds on the variables.
x U Upper bounds on the variables.
x 0 Starting point.
p.H Name of m-file computing the Hessian matrix H(x).
p.r Name of m-file computing the residual vector r(x).
p.J Name of m-file computing the Jacobian matrix J(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: Function value close to 0.
8: Relative function value reduction low for Lowlts iterations.
32: Local minimum with all variables on bounds.
101: Maximum number of iterations reached.
102: Function value below given estimate.
J 0 Function value at start.
J k Function value at optimum.
q 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.
r k Residual at optimum.
J k Jacobian matrix at optimum.
zState State of each variable, described in Table 16.
Solver Solver used.
SolverAlgorithm Solver algorithm used.
Prob Problem structure used.

Description
The prototype routine IsSolve includes four optimization methods for nonlinear least squares problems: the Gauss-
[36]. If rank problem occur, the prototype algorithm is using subspace minimization. The line search is performed
using the routine LineSearch which is a modified version of an algorithm by Fletcher [22]. Bound constraints are
treated as described in Gill, Murray and Wright [28].

Algorithm
See Appendix A.6.
M-files Used
lsDef.m, ResultDef.m, LineSearch.m, secUpdat.m, iniSolve.m, endSolve.m

See Also
clsSolve, ucSolve

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

2.11.8 nlpSolve

Purpose
Solve general constrained nonlinear optimization problems.

nlpSolve 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 = 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 6

Fields used are: eps.g, eps.c, eps.x, MaxIter, wait, size.x, PriLev, method and
QN.InitMatrix.

A Constraint matrix for linear constraints.

b.L Lower bounds on the linear constraints.

b.U Upper bounds on the linear constraints.

c.L Lower bounds on the general constraints.

c.U Upper bounds on the general constraints.

x.L Lower bounds on the variables.

x.U Upper bounds on the variables.

x0 Starting point.

p.f Name of m-file computing the objective function \( f(x) \).

p.g Name of m-file computing the gradient vector \( g(x) \).

p.H Name of m-file computing the Hessian matrix \( H(x) \).

p.c Name of m-file computing the vector of constraint functions \( c(x) \).

p.dc Name of m-file computing the matrix of constraint normals \( \partial c(x)/\partial x \).

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.

- **f0**
  Function value at start.

- **fK**
  Function value at optimum.

- **gK**
  Gradient value at optimum.

- **H_k**
  Hessian value at optimum.

- **x0**
  Starting point.

- **xK**
  Optimal point.

- **v_k**
  Lagrange multipliers.

- **cK**
  Value of constraints at optimum.

- **cJac**
  Constraint Jacobian at optimum.

- **xState**
  State of each variable, described in Table 16.

- **bState**
  State of each linear constraint, described in Table 17.

- **cState**
  State of each general constraint.

- **Solver**
  Solver used.

- **SolverAlgorithm**
  Solver algorithm used.

- **Prob**
  Problem structure used.

Description

The routine *nlqSolve* implements the Filter SQP by Roger Fletcher and Sven Leyffer presented in the paper [23].

M-files Used

*conDef.m, lpDef.m, Phase1Simplex.m, iniSolve.m, endSolve.m*

See Also

*conSolve, sTrustR*

2.11.9 *qpe*

Purpose

Solve equality constrained quadratic programming problems.

The function *qpe* solves problems of the form

\[
\begin{align*}
\min_{x} & \quad f(x) = \frac{1}{2}(x)^T F x + c^T x \\
\text{subject to} & \quad s/t \quad A x = b
\end{align*}
\]

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

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

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
qplm, qpsolve, qbigg

2.11.10 qbigg

Purpose
Solve general quadratic programming problems.

qbigg solves problems of the form

\[
\min_{x} \quad f(x) = \frac{1}{2} (x^T F x + c^T x)
\]

subject to

\[
s/t \quad b_i = a_i^T x \quad i = 1, 2, ..., me
\]

\[
b_i \leq a_i^T x \quad i = me + 1, ..., m
\]

\[
x_L < x < x_U
\]

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 \( b \in \mathbb{R}^m \).

Calling Syntax

\[ [x, lambda, err, p.vec, alfa.vec] = qbigg(F, c, A, b, x.L, x.U, x0, me, PriLev, wait) \]

Description of Inputs

- \( F \): Constant matrix, the Hessian.
- \( c \): Constant vector.
- \( A \): Constraint matrix for the linear constraints.
- \( b \): Right hand side vector.
- \( x.L \): Lower bounds on the variables.
- \( x.U \): Upper bounds on the variables.
- \( x0 \): Starting point.
- \( me \): Number of equality constraints, stored first in \( A \) and \( b \).
- \( PriLev \): Print level: 0 None, 1 Final result, 2 Each iteration.
- \( wait \): Pause at each iteration if \( wait \) is true.

Description of Outputs

- \( x \): Optimal point.
- \( lambda \): Lagrange multipliers. Constraints, lower and upper variable bounds.
- \( err \): Error flag. 0 if OK; 1 – 4 different failures.
- \( p.vec \): All search directions \( p \).
- \( alfa.vec \): All step lengths \( \alpha \).

Description

The implementation of qbigg is similar to qpsolve, but for negative definite quadratic problems uses the algorithm described in M.C. Bartholomew-Biggs [6].

See Also
qpsolve, qpe, qplm

2.11.11 qplm

Purpose
Solve equality constrained quadratic programming problems.

qplm solves problems of the form

\[
\min_{x} \quad f(x) = \frac{1}{2} (x^T F x + c^T x)
\]

subject to

\[
s/t \quad Ax = b
\]

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 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.

Description

The routine `qpdlm` solves a quadratic programming problem, restricted to equality constraints, using the Lagrange method.

See Also

`qpBiggs`, `qpSolve`, `qpe`

2.11.12 `qpSolve`

Purpose

Solve general quadratic programming problems.

`qpSolve` solves problems of the form

$$
\begin{align*}
\text{minimize} & \quad f(x) = \frac{1}{2}x^T F x + c^T x \\
\text{subject to} & \quad s/t \quad x_L \leq x \leq x_U \\
& \quad b_L \leq Ax \leq b_U
\end{align*}
$$

where $x, x_L, x_U \in \mathbb{R}^n$, $F \in \mathbb{R}^{n \times n}$, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ 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 6
  - `QP.F`: Constant matrix, the Hessian.
  - `QP.c`: Constant vector.
  - `A`: Constraint matrix for linear constraints.
  - `b.L`: Lower 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: \( \lambda > 0 \). 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.
        q_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 16.
        Solver  Solver used.
        SolverAlgorithm  Solver algorithm used.
        Proh  Problem structure used.

Description

Implements an active set strategy for Quadratic Programming. For negative definite problems it computes
values and is using directions of negative curvature to proceed. To find an initial feasible point the Phase 1 LP
problem is solved calling Phase1Simplex. The routine is the standard QP solver used by nlpSolve, sTrustR and
conSolve.

M-files Used

qpDef.m, ResultDef.m, lpDef.m, Phase1Simplex.m, qpPhase1.m, iniSolve.m, endSolve.m

See Also

qpBiggs, qpe, qplm, nlpSolve, sTrustR and conSolve

2.11.13 sTrustR

Purpose

Solve optimization problems constrained by a convex feasible region.

sTrustR solves problems of the form

\[
\min_\mathbf{x} \quad f(\mathbf{x})
\]

s.t.

\[
x_L \leq x \leq x_U
\]

\[
b_L \leq Ax \leq b_U
\]

\[
c_L \leq c(x) \leq c_U
\]

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

Calling Syntax

Result = sTrustR(Prob, varargin);
Description of Inputs

Prob

Problem description structure. The following fields are used:

optParam Structure with special fields for optimization parameters, see Table 6.

Fields used are: eps.f, eps.g, eps.c, eps.x, eps_rank, MaxIter, wait, size.x, size.f, xTol, LowIts, PriLev, method and QN_InitMatrix.

PartSep Structure with special fields for partially separable functions, see Table 11.

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.

p.f Name of m-file computing the objective function f(x).

p.g Name of m-file computing the gradient vector g(x).

p.H Name of m-file computing the Hessian matrix H(x).

p.c Name of m-file computing the vector of constraint functions c(x).

p.dc Name of m-file computing the matrix of constraint normals dc(x)/dx.

varargin Other parameters directly sent to low level routines.

Description of Outputs

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

Iter Number of iterations.

ExitFlag Flag giving exit status.

Inform Binary code telling type of convergence:

1: Iteration points are close.

2: Projected gradient small.

4: Relative function value reduction low for LowIts iterations.

8: Too small trust region.

101: Maximum number of iterations reached.

102: Function value below given estimate.

103: Convergence to saddle point (eigenvalues computed).

f.0 Function value at start.

f.k Function value at optimum.

q.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.

c.Jac Constraint Jacobian at optimum.

zState State of each variable, described in Table 16.

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 [15] and [49]. BFGS or DFP is used for the Quasi-Newton update, if the analytical Hessian is not used. sTrustR calls itrR

M-files Used

itrR.m, conDef.m, gpoptSOL.m, gpoSolve.m, iniSolve.m, endSolve.m

See Also

conSolve, nlpSolve, clSolve
2.11.14  **ucSolve**

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

**ucSolve** solves problems of the form

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

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

**Calling Syntax**
Result = ucSolve(Prob, varargin)

**Description of Inputs**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Prob</strong></td>
<td>Problem description structure. The following fields are used:</td>
</tr>
<tr>
<td>Solver.Alg</td>
<td>Solver algorithm to be run:</td>
</tr>
<tr>
<td>0: Newton.</td>
<td></td>
</tr>
<tr>
<td>1: Safeguarded BFGS (default).</td>
<td></td>
</tr>
<tr>
<td>2: Safeguarded Inverse BFGS.</td>
<td></td>
</tr>
<tr>
<td>3: Safeguarded Inverse DFP.</td>
<td></td>
</tr>
<tr>
<td>4: Safeguarded DFP.</td>
<td></td>
</tr>
<tr>
<td>5: Fletcher-Reeves CG.</td>
<td></td>
</tr>
<tr>
<td>6: Polak-Ribiere CG.</td>
<td></td>
</tr>
<tr>
<td>7: Fletcher conjugate descent CG-method.</td>
<td></td>
</tr>
<tr>
<td>optParam</td>
<td>Structure with special fields for optimization parameters, see Table 6.</td>
</tr>
<tr>
<td>LineSearch, LineAlg, xTol, LowIts, method, PriLev and QN_InitMatrix.</td>
<td></td>
</tr>
<tr>
<td>x.L</td>
<td>Lower bounds on the variables.</td>
</tr>
<tr>
<td>x.U</td>
<td>Upper bounds on the variables.</td>
</tr>
<tr>
<td>x.0</td>
<td>Starting point.</td>
</tr>
<tr>
<td>p.f</td>
<td>Name of m-file computing the objective function ( f(x) ).</td>
</tr>
<tr>
<td>p.g</td>
<td>Name of m-file computing the gradient vector ( g(x) ).</td>
</tr>
<tr>
<td>p.H</td>
<td>Name of m-file computing the Hessian matrix ( H(x) ).</td>
</tr>
<tr>
<td>f.Low</td>
<td>Lower bound on function value.</td>
</tr>
<tr>
<td>varargin</td>
<td>Other parameters directly sent to low level routines.</td>
</tr>
</tbody>
</table>

**Description of Outputs**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Result</strong></td>
<td>Structure with result from optimization. The following fields are changed:</td>
</tr>
<tr>
<td>Iter</td>
<td>Number of iterations.</td>
</tr>
<tr>
<td>ExitFlag</td>
<td>0 if convergence to local min. Otherwise errors.</td>
</tr>
<tr>
<td>Inform</td>
<td>Binary code telling type of convergence:</td>
</tr>
<tr>
<td>1: Iteration points are close.</td>
<td></td>
</tr>
<tr>
<td>2: Projected gradient small.</td>
<td></td>
</tr>
<tr>
<td>4: Relative function value reduction low for LowIts iterations.</td>
<td></td>
</tr>
<tr>
<td>101: Maximum number of iterations reached.</td>
<td></td>
</tr>
<tr>
<td>102: Function value below given estimate.</td>
<td></td>
</tr>
<tr>
<td>104: Convergence to a saddle point</td>
<td></td>
</tr>
<tr>
<td>f.0</td>
<td>Function value at start.</td>
</tr>
<tr>
<td>f.k</td>
<td>Function value at optimum.</td>
</tr>
<tr>
<td>q.k</td>
<td>Gradient value at optimum.</td>
</tr>
<tr>
<td>H.k</td>
<td>Hessian value at optimum.</td>
</tr>
<tr>
<td>B.k</td>
<td>Quasi-Newton approximation of the Hessian at optimum.</td>
</tr>
<tr>
<td>x.0</td>
<td>Starting point.</td>
</tr>
<tr>
<td>x.k</td>
<td>Optimal point.</td>
</tr>
<tr>
<td>v.k</td>
<td>Lagrange multipliers.</td>
</tr>
<tr>
<td>xState</td>
<td>State of each variable, described in Table 16.</td>
</tr>
<tr>
<td>Solver</td>
<td>Solver used.</td>
</tr>
<tr>
<td>SolverAlgorithm</td>
<td>Solver algorithm used.</td>
</tr>
<tr>
<td>Prob</td>
<td>Problem structure used.</td>
</tr>
</tbody>
</table>
Description
The prototype routine 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 inverse BFGS method, the quasi-Newton DFP and inverse DFP method, the Fletcher-Reeves and Polak-Ribiére conjugate gradient method, and the Fletcher conjugate descent method. For the Newton and the quasi-Newton methods the code is using a subspace minimization technique to handle rank problem, see Lindström [41]. The quasi-Newton codes also use safe guarding techniques to avoid rank problem in the updated matrix. The line search is performed using the routine LineSearch which is a modified version of an algorithm by Fletcher [22]. Bound constraints are treated as described in Gill, Murray and Wright [28].

Algorithm
See Appendix A.7.

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

See Also
lsSolve

2.12 Optimization Subfunction Utilities in NLPLIB TB

In the following subsections the optimization subfunction utilities in NLPLIB TB will be described.

2.12.1 intpol2

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

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

Description of Inputs

<table>
<thead>
<tr>
<th>x0</th>
<th>Interpolation point x0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>f0</td>
<td>Function value at x0.</td>
</tr>
<tr>
<td>g0</td>
<td>Derivative value at x0.</td>
</tr>
<tr>
<td>x1</td>
<td>Interpolation point x1.</td>
</tr>
<tr>
<td>f1</td>
<td>Function value at x1.</td>
</tr>
<tr>
<td>a</td>
<td>Lower interval bound.</td>
</tr>
<tr>
<td>b</td>
<td>Upper interval bound.</td>
</tr>
<tr>
<td>PriLev</td>
<td>Printing level, PriLev ≥ 3 gives a lot of output.</td>
</tr>
</tbody>
</table>

Description of Outputs

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

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

Algorithm
See Appendix A.3.

See Also
LineSearch, intpol3

2.12.2 intpol3

Purpose
Find the minimum of a cubic approximation of a scalar function in a given interval.
Calling Syntax
alfa = intpol3(x0, f0, g0, x1, f1, g1 , a, b, PriLev)

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

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

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

Algorithm
See Appendix A.4.

See Also
LineSearch, intpol2

2.12.3  itrr

Purpose
Determine the initial trust region radius.

Calling Syntax
[D.0, f.0, x.0] = itrr(x.0, fS, gS, HS, jMax, iMax, Prob, varargin)

Description of Inputs
x.0    Starting point.
x.L    Lower bounds for x.
x.U    Upper bounds for x.
fS    String with function call sequence. \( x.k \) current point.
gS    String with gradient call sequence. \( x.k \) current point.
HS    String with Hessian call sequence. \( x.k \) current point.
jMax  Number of outer iterations, normally 1.
iMax  Number of inner iterations, normally 5.
Prob  Prob.PartSep.index is the index for the partial function to be analyzed.
varargin Extra user parameters, passed to \( f\), \( g\) and \( H\);

Description of Outputs
D.0    Initial trust region radius
f.0    Function value at the input starting point x.0.
x.0    Updated starting point, if \( jMax > 1\).

Description
The routine itrr implements the initial trust region radius algorithm as described by Sartenaer in [49]. itrr is called by sTrustR.

See Also
sTrustR
2.12.4 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 \( x, \alpha \in \mathbb{R}^n \).

**Calling Syntax**
Result = LineSearch(f, g, x, p, f.0, g.0, optParam, alphaMax, alpha.1, pType, PriLev, varargin)

**Description of Inputs**
- \( f \) Name of m-file computing the objective function \( f(x) \).
- \( g \) Name of m-file computing the gradient vector \( g(x) \).
- \( x \) Current iterate \( x \).
- \( p \) Search direction \( p \).
- \( f.0 \) Function value at \( \alpha = 0 \).
- \( g.0 \) Gradient at \( \alpha = 0 \), the directed derivative at the present point.
- \( \text{optParam} \) Structure with special fields for optimization parameters, the following fields are used:
  - \( \text{LineAlg} \) Type of line search algorithm, see Table 6.
  - \( \text{LineSearch} \) Structure with line search parameters, see Table 14.
- \( \text{alphaMax} \) Maximal value of step length \( \alpha \).
- \( \text{alpha.1} \) First step in \( \alpha \).
- \( \text{pType} \) Type of problem:
  - 0 Normal problem.
  - 1 Nonlinear least squares.
  - 2 Constrained nonlinear least squares.
  - 3 Merit function minimization.
  - 4 Penalty function minimization.
- \( \text{PriLev} \) Printing level:
  - \( \text{PriLev} > 0 \) Writes a lot of output in LineSearch.
  - \( \text{PriLev} > 3 \) Writes a lot of output in intpol2 and intpol3.
- \( \text{varargin} \) Other parameters directly sent to low level routines.

**Description of Outputs**
- \( \text{Result} \) Result structure with fields:
  - \( \text{alpha} \) Optimal line search step \( \alpha \).
  - \( f.\alpha \) Optimal function value at line search step \( \alpha \).
  - \( g.\alpha \) Optimal gradient value at line search step \( \alpha \).
  - \( \text{alphaVec} \) 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 [22]. 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
$a^{(k)}$ is an acceptable point.

**Algorithm**
See Appendix A.5.

**M-Files Used**
`intpol2.m, intpol3.m`

### 2.12.5 preSolve

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

**Calling Syntax**
`Prob = preSolve(Prob)`

**Description of Inputs**

<table>
<thead>
<tr>
<th>Prob</th>
<th>Problem description structure. The following fields are used:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Constraint matrix for linear constraints.</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>x L</td>
<td>Lower bounds on the variables.</td>
</tr>
<tr>
<td>x U</td>
<td>Upper bounds on the variables.</td>
</tr>
</tbody>
</table>

**Description of Outputs**

<table>
<thead>
<tr>
<th>Prob</th>
<th>Problem description structure. The following fields are changed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Constraint matrix for linear constraints.</td>
</tr>
<tr>
<td>b L</td>
<td>Lower bounds on the linear constraints, set to NaN for redundant constraints.</td>
</tr>
<tr>
<td>b U</td>
<td>Upper bounds on the linear constraints, set to NaN for redundant constraints.</td>
</tr>
<tr>
<td>x L</td>
<td>Lower bounds on the variables.</td>
</tr>
<tr>
<td>x U</td>
<td>Upper bounds on the variables.</td>
</tr>
</tbody>
</table>

**Description**
The routine `preSolve` is an implementation of those presolve analysis techniques described by Gondzio in [30], which is applicable to general linear constrained problems. See [10] for a more detailed presentation.

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

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

Note that rows are actually not deleted or removed, instead `preSolve` indicates that constraint $i$ is redundant by setting $b.L(i) = b.U(i) = NaN$ and leaves to the calling routine to decide what to do with those constraints.

### 2.13 User Utility Functions in NLPLIB TB

In the following subsections the user utility functions in NLPLIB TB will be described.

#### 2.13.1 PrintResult

**Purpose**
Prints the result of an optimization.

**Calling Syntax**
`PrintResult(Result, PriLev)`
Description of Inputs

Result
Result structure from optimization.

PriLev
Printing level:
0
Silent.
1
Problem number and name. Function value at the solution and at start. Known optimal function value (if given).
2
Optimal point $x$ and starting point $x_0$. Number of evaluations of the function, gradient etc. Lagrange multipliers, both returned and NLPLIB TB estimate. Distance from start to solution. The residual, gradient and projected gradient. ExitFlag and Inform.
3
Jacobian, Hessian or Quasi-Newton Hessian approximation.

2.13.2 PrintSolvers

Purpose
Prints the available solvers for a certain solveType.

Calling Syntax
PrintSolvers(solveType)

Description of Inputs

solveType
Either a string 'uc', 'con' etc. or the corresponding solveType number. See Table 1.

Description
The routine PrintSolvers prints all available solvers for a given solveType, including Fortran, C and Matlab Optimization Toolbox solvers. If solveType is not specified then PrintSolvers lists all available solvers for all different solveType. The input argument could either be a string such as 'uc', 'con' etc. or a number corresponding to the type of solver, see Table 1.

Examples
See Section 2.2

M-files Used
SolverList.m

2.13.3 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
2.13.4  systest

Purpose
Run big test to check for bugs in NLPLIB TB.

Calling Syntax
systest(solvTypes, PriLevOpt, PriLev, wait)

Description of Inputs

<table>
<thead>
<tr>
<th>solvTypes</th>
<th>A vector of numbers defining which solvType to test.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PriLevOpt</td>
<td>Printing level in the solver. Default 2, short information from each iteration.</td>
</tr>
<tr>
<td>wait</td>
<td>Set wait to 1 if pause after each problem. Default 1</td>
</tr>
<tr>
<td>PriLev</td>
<td>Printing level in PrintResult. Default 5, full information.</td>
</tr>
</tbody>
</table>

See Also
runtest
3 OPERA TB

OPERA TB is a Matlab toolbox for solving linear and discrete optimization problems in operations research and mathematical programming. Included are routines for linear programming, network programming, integer programming and dynamic programming.

3.1 Optimization Algorithms and Solvers in OPERA TB

In this section we describe OPERA TB by giving tables describing most Matlab functions with some comments. All function files are part of the directory OPERA.

There are two menu programs for linear programming. The *simplex* routine is a utility to interactively solve LP problems in canonical standard form. When the problem is defined, *simplex* calls the internal OPERA TB solvers *lpsimp1* and *lpsimp2*.

The menu program *lpOpt* is similar to the menu programs in NLPLIB TB. It calls the driver routine *lpRun*, which may call any of the predefined solvers written in Matlab, C or FORTRAN code. The user may run *lpOpt*, the driver routine *lpRun*, or directly call a solver routine.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>lpOpt</em></td>
<td>Menu program for LP problems.</td>
</tr>
<tr>
<td><em>lpRun</em></td>
<td>Driver routine that solves predefined LP problems.</td>
</tr>
<tr>
<td><em>simplex</em></td>
<td>Interactive input and solution of LP on canonical standard form.</td>
</tr>
</tbody>
</table>

Like the Matlab Optimization Toolbox, OPERA TB 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 *qoptions* is used instead of *foptions*.

The OPERA TB routines *lpOpt, lpRun, lsolve, Phase1Simplex, Phase2Simplex* and *DualSolve* are designed in the same way as the NLPLIB TB routines i.e. they use the same input and output format. They also use the optimization parameter structure *optParam* (Table 6) instead of *optPar*.

In OPERA TB the routine *lpDef* is used to define either the *optPar* vector or the *optParam* structure. *lpDef* is written to handle initial parameter setting both in the old part of OPERA TB as well as the new structure based NLPLIB TB parameter settings. If the user want *lpDef* to define the *optParam* structure the call to *lpDef* should look like

```matlab
optParam = lpDef(method, []);
```

or

```matlab
optParam = lpDef(method, optParam);
```

Otherwise, *lpDef* will return the *optPar* vector for the old format.

3.1.1 Linear Programming

There are several algorithms implemented for linear programming. Those implementations are divided into three groups:

1. Numerically stable solvers.
2. Solvers used in teaching courses.
3. Other solvers.
Table 32: Numerically stable solvers for linear programming.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>lpSolve</em></td>
<td>General solver for linear programming problems. Calls <em>Phase1Simplex</em> and <em>Phase2Simplex</em>.</td>
<td>3.5.15</td>
<td>119</td>
</tr>
<tr>
<td><em>Phase1Simplex</em></td>
<td>The Phase I simplex algorithm. Finds a basic feasible solution (bfs) using artificial variables. Calls <em>Phase2Simplex</em>.</td>
<td>3.5.20</td>
<td>123</td>
</tr>
<tr>
<td><em>Phase2Simplex</em></td>
<td>The Phase II revised simplex algorithm with three selection rules.</td>
<td>3.5.20</td>
<td>123</td>
</tr>
<tr>
<td><em>DualSolve</em></td>
<td>The dual simplex algorithm.</td>
<td>3.5.7</td>
<td>112</td>
</tr>
</tbody>
</table>

Table 32 lists the solvers from the first group, Table 33 lists all the solvers classified as solvers used in teaching courses and Table 34 lists the routines defined as other solvers.

The solvers classified as numerically stable (*lpSolve, Phase1Simplex, Phase2Simplex* and *DualSolve*), use the same input and output format as the NPLIB TB solvers described in Section 2.1. They use the optimization parameter structure *optParam* instead of the optimization parameter vector *optPar*. These routines are the routines for linear programming used by the NPLIB TB solvers and are also available from the Graphical User Interface.

*Phase1Simplex, Phase2Simplex* and *DualSolve* are refined versions of *lpsimp1, lpsimp2* and *lpdual* respectively. The last three are classified as solvers for linear programming to be used in teaching courses and are described below. *lpSolve* calls both the routines *Phase1Simplex* and *Phase2Simplex* to solve a general linear program (*lp*) defined as

\[
\min_x \ f(x) = c^T x \\
\text{s.t.} \ x_L \leq x \leq x_U, \ b_L < Ax < b_U
\]

(13)

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

Table 33: Solvers for linear programming used in teaching courses.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>lpsimp1</em></td>
<td>The Phase I simplex algorithm. Finds a basic feasible solution (bfs) using artificial variables. Calls <em>lpsimp2</em>.</td>
<td>3.5.13</td>
<td>118</td>
</tr>
<tr>
<td><em>lpsimp2</em></td>
<td>The Phase II revised simplex algorithm with three selection rules.</td>
<td>3.5.14</td>
<td>118</td>
</tr>
<tr>
<td><em>karmark</em></td>
<td>Karmarkar’s algorithm. Kanonical form.</td>
<td>3.5.8</td>
<td>114</td>
</tr>
<tr>
<td><em>lpkarma</em></td>
<td>Solves LP on equality form, by converting and calling <em>karmark</em>.</td>
<td>3.5.12</td>
<td>117</td>
</tr>
</tbody>
</table>

Table 34: Other solvers for linear programming.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>lpdual</em></td>
<td>The dual simplex algorithm.</td>
<td>3.5.11</td>
<td>116</td>
</tr>
<tr>
<td><em>akarmark</em></td>
<td>Affine scaling variant of Karmarkan’s algorithm.</td>
<td>3.5.1</td>
<td>108</td>
</tr>
</tbody>
</table>

The implementation of *lpsimp2* is based on the standard revised simplex algorithm as formulated in Goldfarb and Todd [29, 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 [29, 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. Karmarkar’s projective algorithm (*karmark*) is developed from the description in Bazarraa et. al. [7, page 386]. There is a choice of update, either according to Bazarara or the rule by Goldfarb and Todd [29, chap. 9]. The affine scaling variant of Karmarkar’s method...
(akarmark) is an implementation of the algorithm in Bazaraa [29, 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 lpdua 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). The same selection rules are used in Phase2Simplex and DualSolve.

### 3.1.2 Transportation Programming

Transportation problems are solved using an implementation of the transportation simplex method as described in Luenberger [42, 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 [52, chap. 7.2]. The functions are described in Table 35.

<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>3.6.6</td>
<td>131</td>
</tr>
<tr>
<td>TPmc</td>
<td>Find initial bfs to TP using the minimum cost method.</td>
<td>3.6.5</td>
<td>131</td>
</tr>
<tr>
<td>TPvogel</td>
<td>Find initial bfs to TP using Vogel’s approximation method.</td>
<td>3.6.7</td>
<td>132</td>
</tr>
<tr>
<td>TPsimplex</td>
<td>Implementation of the transportation simplex algorithm.</td>
<td>3.5.23</td>
<td>126</td>
</tr>
</tbody>
</table>

### 3.1.3 Network Programming

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

- Search for all reachable nodes in a network using a stack approach (gsearch). The implementation is a variation of the Algorithm SEARCH in [2, pages 231-233].
- Search for all reachable nodes in a network using a queue approach (gsearchq). The implementation is a variation of the Algorithm SEARCH in [2, pages 231-232].
- Find the minimal spanning tree of an undirected graph (mintree) with Kruskal’s algorithm described in Ahuja et. al. [3, page 520-521].
- Solve the shortest path problem using Dijkstra’s algorithm (dijkstra). A direct implementation of the Algorithm DIJKSTRA in [2, pages 250-251].
- Solve the shortest path problem using a label correcting method (labelcor). The implementation is based on Algorithm LABEL CORRECTING in [2, page 260].
- Solve the shortest path problem using a modified label correcting method (modlabel). The implementation is based on Algorithm MODIFIED LABEL CORRECTING in [2, page 262], including the heuristic rule discussed to improve running time in practice.
- Solve the maximum flow problem using the Ford-Fulkerson augmenting path method (maxflow). The implementation is based on the algorithm description in Luenberger [42, pages 144-145].
- Solve the minimum cost network flow problem (MCNFP) using a network simplex algorithm (NWSimplex). The implementation is based on Algorithm network simplex in Ahuja et. al. [3, page 415].
- Solve the symmetric traveling salesman problem using Lagrangian relaxation and the subgradient method with the Polyak rule II (salesman), an algorithm by Held and Karp [31].

The network programming routines are listed in Table 36.
Table 36: Routines for network programs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>gsearch</td>
<td>Searching all reachable nodes in a network. Stack approach.</td>
<td>3.6.2</td>
<td>129</td>
</tr>
<tr>
<td>gsearchq</td>
<td>Searching all reachable nodes in a network. Queue approach.</td>
<td>3.6.3</td>
<td>130</td>
</tr>
<tr>
<td>minnntree</td>
<td>Finds the minimum spanning tree of an undirected graph.</td>
<td>3.6.4</td>
<td>130</td>
</tr>
<tr>
<td>dijkstra</td>
<td>Shortest path using Dijkstra’s algorithm.</td>
<td>3.5.4</td>
<td>110</td>
</tr>
<tr>
<td>labelcor</td>
<td>Shortest path using a label correcting algorithm.</td>
<td>3.5.10</td>
<td>116</td>
</tr>
<tr>
<td>modlabel</td>
<td>Shortest path using a modified label correcting algorithm.</td>
<td>3.5.18</td>
<td>122</td>
</tr>
<tr>
<td>maxflow</td>
<td>Solving maximum flow problems using the Ford-Fulkerson augmenting path method.</td>
<td>3.5.16</td>
<td>120</td>
</tr>
<tr>
<td>salesman</td>
<td>Symmetric traveling salesman problem (TSP) solver using Lagrangian relaxation and the subgradient method with the Polyak rule II.</td>
<td>3.5.22</td>
<td>126</td>
</tr>
<tr>
<td>traveling</td>
<td>Solve TSP problems with branch and bound. Calls salesman.</td>
<td>3.5.24</td>
<td>127</td>
</tr>
<tr>
<td>NWsimplex</td>
<td>Solving minimum cost network flow problems (MCNFP) with a network simplex algorithm.</td>
<td>3.5.19</td>
<td>123</td>
</tr>
</tbody>
</table>

3.1.4 Integer Programming

To solve mixed linear inequality integer programs two algorithms are implemented. The first implementation (mipSolve) is a branch-and-bound algorithm from Nemhauser and Wolsey [45, chap. 8]. The second implementation (cutplane) is a cutting-plane algorithm using Gomory cuts. Both routines are using the linear programming routines in the toolbox OPERA TB 1.0 (Phase1Simplex, Phase2Simplex, DualSolve), to solve relaxed subproblems. Balas method for binary integer programs restricted to integer coefficients is implemented in the routine balas [32]. The routines for integer programming are described in Table 37.

Table 37: Routines for integer programming.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>cutplane</td>
<td>Cutting plane method using Gomory cuts for mixed-integer programs (MIP).</td>
<td>3.5.3</td>
<td>110</td>
</tr>
<tr>
<td>mipSolve</td>
<td>Branch and bound algorithm for mixed-integer programs (MIP).</td>
<td>3.5.17</td>
<td>121</td>
</tr>
<tr>
<td>balas</td>
<td>Branch and bound algorithm for binary IP using Balas method.</td>
<td>3.5.2</td>
<td>109</td>
</tr>
</tbody>
</table>

3.1.5 Dynamic Programming

Two simple examples of dynamic programming are included. Both examples are from Winston [52, chap. 20]. Forward recursion is used to solve an inventory problem (dpinvent) and a knapsack problem (dpknap), see Table 38.

Table 38: Routines for dynamic programming.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dpinvent</td>
<td>Forward recursion DP algorithm for the inventory problem.</td>
<td>3.5.5</td>
<td>111</td>
</tr>
<tr>
<td>dpknap</td>
<td>Forward recursion DP algorithm for the knapsack problem.</td>
<td>3.5.6</td>
<td>112</td>
</tr>
</tbody>
</table>

3.1.6 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 [20], using subgradient iterations and a simple line search rule. Lagrangian relaxation is also 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 39.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>krelax</td>
<td>Lagrangian relaxation with knapsack subproblems.</td>
<td>3.5.9</td>
<td>115</td>
</tr>
<tr>
<td>urelax</td>
<td>Lagrangian relaxation with knapsack subproblems, plot result.</td>
<td>3.5.25</td>
<td>128</td>
</tr>
</tbody>
</table>

### 3.1.7 Utility Routines

Table 40 describes the low level test functions and the corresponding set up routines needed for the predefined linear programming test problems. The driver routine lpRun may also call nonlinear solvers to solve the LP problem. therefore some extra low level routines are needed.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lp_prob</td>
<td>Initialization of lp test problems.</td>
</tr>
<tr>
<td>lp_f</td>
<td>Define the objective function for LP, $c^T x$ (for NLP solvers).</td>
</tr>
<tr>
<td>lp_g</td>
<td>Define the gradient function for LP, the vector $c$ (for NLP solvers).</td>
</tr>
<tr>
<td>lp_H</td>
<td>Define the Hessian matrix for LP, A zero matrix (for NLP solvers).</td>
</tr>
</tbody>
</table>

Table 41 lists the utility routines used in OPERA TB. Some of them are also used by NLPLIB TB.

<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>lpDef</td>
<td>Define optimization parameters. Handles both the Optimization Toolbox format (optPar) and the NLPLIB TB format (optParam).</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$(i_1 : i_n)$ $v_i$,$v_i$,...$v_i$.</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>
3.2 How to Solve Optimization Problems Using OPERA TB

In this section we will describe how to use OPERA TB to solve the different types of problems discussed in Section 3.1.

3.2.1 How to Solve Linear Programming Problems

To solve a linear programming problem in OPERA TB you can define your problem in an init file and then use the menu routine *lpOpt* or the driver routine *lpRun*. Another way of doing it is to call any of the solvers directly from the Matlab prompt. To illustrate the approach we will solve the problem

\[
\begin{align*}
\min_{x_1, x_2} & \quad f(x_1, x_2) = -7x_1 - 5x_2 \\
\text{s.t} & \quad x_1 + 2x_2 \leq 6 \\
& \quad 4x_1 + x_2 \leq 12 \\
& \quad x_1, x_2 > 0 
\end{align*}
\]

(14)

here named *lpTest1*, in some different ways.

If the problem is to be solved several times, perhaps with small changes in the coefficients or with different solvers, we recommend you to define the problem in an init file by following the stepwise description below (for all instructions we assume that you edit the copied files in a text editor).

1. Make a copy of *lp_prob.m* and place the copy in your working directory or in any other directory placed before the directory OPERA in the Matlab path.

2. Add the problem to the menu choice:

   ...
   ...
   , 'Winston Ex. 4.12 B4. Max || ||. Rewritten' ...
   , 'lpTest1' ...
   ); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

   if isempty(P)
     return;
   end
   ...
   ...

3. 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 you just define the lower bounds for the constraints *b.L* equal to *b.U*.

   ...
   ...
   elseif P == 13
     Name = 'lpTest1';
     c = [-7 -5]';
     A = [ 1 2
           4 1 ];
     b_U = [ 6 12 ]';
     x_L = [ 0 0 ]';
     x_min = [ 0 0 ]';
     x_max = [10 10 ]';
   else
     disp('lp_prob: Illegal problem number')
     pause
     Name=[];


end
...
...

4. Save the file properly.

You could also define the optional parameters $B$, $f_{\text{min}}$ and $x_{0}$ as described in the problem definition description in $lp_{.}\text{prob.}m$. If $B$ is not given, as in this case, a Phase I program is run.

The problem could now be solved by using the menu routine $lpOpt$, the driver routine $lpRun$ or by directly call the solver $lpSolve$. If your choice is the menu routine you just type $Result = lpOpt$ at the Matlab prompt and the main menu in Figure 11 will be displayed.

![menu](image)

Figure 11: The main menu in $lpOpt$

Pushing the *Choice of Problem File and Problem* button followed by the uppermost button will make the menu in Figure 12 to be displayed.

Push the $lp\text{test1}$ button to choose problem (14) and you will be back in the main menu. Now you can select optimization solver by pushing the *Choice of optimization algorithm* button and choose the routine you want to use to solve the problem. Back to the main menu you can change the default settings of the optimization parameters, the output printing level, convergence tolerances etc. Pushing the *Optimize* button will run the driver routine $lpRun$ 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 you can solve the problem by a direct call to $lp\text{Run}$ from the Matlab prompt or as a command in an m-file. This approach could be of great interest in an testing environment. The most
Figure 12: The problem choice menu in \textit{lpOpt}.

A straightforward way of doing it (when the problem is defined in \textit{lp\_prob.m}) is to give the following call from the Matlab prompt:

\begin{verbatim}
probNumber = 13;
Result = lpRun([], [], [], [], [], probNumber);
\end{verbatim}

The arguments not given to \textit{lpRun} is set to default values, see the \textit{lpRun} routine description Section 3.4.1 page 107. Let us also show how you can give a call by specifying some of the other arguments. Assume that you want to solve the problem with the following requirements:

- Start in the point \((1,1)\).
- No printing output neither in the driver routine nor in the solver.
- Use Matlab Optimization Toolbox solver \textit{lp}.

Then the call to \textit{lpRun} should be:

\begin{verbatim}
Solver = 'lp';
Prob = probInit('lp\_prob',13);
PriLev = 0;
\end{verbatim}
Prob.x_0 = [1;1];
Prob.optParam.PriLev = 0;

Result = lpRun(Solver, Prob, [], PriLev);

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

PrintResult(Result);

For a more advanced user it could be of interest to define the problems in an "own" problem definition file. This is of course possible in OPERA TB and we will now illustrate how to do (for all instructions we assume that you edit the copied files in a text editor).

1. Make a copy of *lp_prob.m* and place the copy in your working directory or in any other directory placed before the directory OPERA in the Matlab path.

2. Rename the file *lp_prob.m* to for example *ownlp_prob.m*.

3. Delete the already existing problems from the menu choice and add *lptest1* as the first problem:

   ...  
   ...  
   probList=str2mat(...  
   'lptest1'...  
   );  
   % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES  
   if isempty(P)
      return;
   end
   ...
   ...  

4. Make the following modification in MFILownlp_prob:

5. ...
   ...
   ...  
   if ask==-1 & ~isempty(Prob)
      if isstruct(Prob)
         if ~isempty(Prob.P)
            if P==Prob.P & strcmp(Prob.probFile,'ownlp_prob'), return; end
         end
      end
   end
   ...
   ...

6. 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 you just define the lower bounds for the constraints *b_L* equal to *b_U*.

   ...
   ...
   elseif P == 1
      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('ownlp_prob: Illegal problem number')
pause
Name=[];
end
...
...
7. Modify the file nameprob.m in the NLPLIB directory as described in the file. It should now look like:
...
...
elseif solvType==8
    % Linear programming
    F=str2mat('lp_prob...
        ,'ownlp_prob'...
        ,'usr_prob'...
    )

    % USER: Duplicate the row above and insert your own file name
    %       inside the quotes

    % USER: Uncomment next row if your latest file should be the default one.
    % D=size(F,1):

    N=str2mat(...
        ,'lp Linear Programming'...
        ,'ownlp My Own Linear Programming Problems'...
        ,'usr Linear Programming'...
    );

    % USER: Duplicate the row above and insert your own file name
    %       and description inside the quotes. Add the probTypV number to
    %       the vector probTypV below.
    probTypV=[8 8 8];
    ...
    ...

8. Save both the renamed file ownlp.prob and nameprob.m properly.

Now, when you push the Choice of Problem File and Problem button in the main menu of lpopt, Figure 11, the menu in Figure 13 should be displayed. Choose ownlp My Own Linear Programming Problems and proceed as described above.

We will now show how to give a direct call to lpRun in the case when the problem is defined in another init file than lp_prob.m. Assume the same requirements as itemized above.

Solver = 'lp';
probFile = 'ownlp_prob';
Prob = probInit(probFile,1);
PriLev = 0;
Prob.x_0 = [1;1];
Prob.optParam.PriLev = 0;

Result = lpRun(Solver, Prob, [], PriLev, probFile);
Finally, we will show how you can solve (14) by direct use of the optimization routines \texttt{lpsimp1} and \texttt{lpsimp2}.

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

For further illustrations of how to solve linear programming problems see the example files listed in Table 42 and Table 43.

\begin{table}[h]
\centering
\caption{Test examples for linear programming.}
\begin{tabular}{ll}
\hline
\textbf{Function} & \textbf{Description} \\
\hline
exinled.m & First simple LP example from a course in Operations Research. \\
excycle & Menu with cycling examples. \\
excycle1 & The Marshall-Surberalle cycling example. Run both the Bland's cycle preventing rule and the default minimum reduced cost rule and compare results. \\
excycle2 & The Kuhn cycling example. \\
excycle3 & The Beale cycling example. \\
exKleeM & The Klee-Minty example. Shows that the simplex algorithm with Dantzig's rule visits all vertices. \\
exf821 & Run exercise 8.21 from Fletcher, Practical methods of Optimization. Illustrates redundancy in constraints. \\
ex412b4s & Wayne Winston example 4.12 B4, using \texttt{lpsimp1} and \texttt{lpsimp2}. \\
expertur & Perturbed both right hand side and objective function for Luenberger 3.12-10,11. \\
ex6rev17 & Wayne Winston chapter 6 Review 17. Simple example of calling the dual simplex solver \texttt{lpdual}. \\
ex611a2 & Wayne Winston example 6.11 A2. A simple problem solved with the dual simplex solver \texttt{lpdual}. \\
\hline
\end{tabular}
\end{table}
Table 43: Test examples for linear programming running interior point methods.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exwn597</td>
<td>Test of karmark and lpsimp2 on Winston example page 597 and Winston 10.6 Problem A1.</td>
</tr>
<tr>
<td>exstranq</td>
<td>Test of karmark and lpsimp2 on Strangs’ nutshell example.</td>
</tr>
<tr>
<td>exkarma</td>
<td>Test of akarmark.</td>
</tr>
<tr>
<td>exKleeM2</td>
<td>Klee-Minty example solved with lpkarma and karmark.</td>
</tr>
</tbody>
</table>

3.2.2 How to Solve Transportation Programming Problems

We will as an example solve the transportation problem

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

(15)

where \( \mathbf{s} \) is the supply vector, \( \mathbf{d} \) is the demand vector and \( \mathbf{C} \) is the cost matrix. See TPsimplex Section 3.5.23. Solving (15) by use of the routine TPsimplex is done by:

```
s = [5 25 25]';
d = [10 10 20 15]';
C = [6 2 -1 0
     4 2 2 3
     3 1 2 1];
[X, B, optPar, y, C] = TPsimplex(s, d, C);
```

When neither starting base nor starting point is given as input argument TPsimplex calls TVogel (using Vogel’s approximation method) to find an initial basic feasible solution (bfs). If you want to use another method to find an initial bfs, e.g. the northwest corner method, you explicitly call the corresponding routine (TPnw) before the call to TPsimplex:

```
s = [5 25 25]';
d = [10 10 20 15]';
C = [6 2 -1 0
     4 2 2 3
     3 1 2 1];
[X_0, B_0] = TPnw(s, d)
[X, B, optPar, y, C] = 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 44.

3.2.3 How to Solve Network Programming Problems

In OPERA 1B there are several routines for network programming problems. We will here give an example of how to solve a shortest path problem. Given the network in Figure 14, where the numbers at each arc represent the distance of the arc, we want to 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:
Table 44: Test examples for transportation programming.

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

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

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

\[
P = \begin{pmatrix}
1 \\
3 \\
4 \\
6 \\
8 \\
9 \\
\end{pmatrix},
\quad Z = \begin{pmatrix}
1 & 2 \\
1 & 1 \\
1 & 3 \\
1 & 3 \\
1 & 4 \\
1 & 3 \\
\end{pmatrix},
\quad c = \begin{pmatrix}
2 \\
3 \\
4 \\
2 \\
4 \\
1 \\
\end{pmatrix},
\quad T = \begin{pmatrix}
1 \\
4 \\
2 \\
1 \\
7 \\
3 \\
\end{pmatrix},
\quad R = \begin{pmatrix}
1 \\
1 \\
3 \\
5 \\
7 \\
9 \\
\end{pmatrix}
\]

See a2frstar Section 3.6.1 for an explanation of the notation.

Our choice of solver for this example is modlabel, see Section 3.5.18, which uses a modified label correcting algorithm. First we define the incidence matrix \(A\) and the cost vector \(c\) and call the routine a2frstar to convert to a forward and reverse star representation (which is used by modlabel). Then the actual problem is solved

\[
A = \begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 1 & 1 & 0 & -1 & 0 \\
0 & 0 & -1 & 0 & 0 & 1 & 1 & -1 \\
0 & 0 & 0 & 0 & -1 & -1 & 0 & 1 \\
\end{bmatrix};
\]

\[
C = \begin{bmatrix}
2 & 3 & 1 & 4 & 2 & 4 & 1 & 3 \\
\end{bmatrix};
\]

\[
[P \ Z \ c \ T \ R \ x\_U] = \text{a2frstar}(A, \ C);
\]

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

For further illustrations of how to solve network programming problems see the example files listed in Table 45.

### 3.2.4 How to Solve Integer Programming Problems

The routines in OPERA TB for solving integer programming problems are cutplane, mipSolve and balas. To illustrate how to solve an integer programming problem we will solve the problem (14) with the addition of the
requirement of the variables to be positive integers. We have chosen to use the routine `cutplane`, see Section 3.5.3.

\[
A = \begin{bmatrix} 1 & 1 & 2 \\ 4 & 1 & \end{bmatrix}; \\
b = \begin{bmatrix} 6 & 12 \end{bmatrix}'; \\
c = \begin{bmatrix} -7 & -5 \end{bmatrix}'; \\
\text{meq} = 0; \\
optPar = ipDef; \\
optPar(13) = \text{meq}; \\
n_1 = 2;
\]

\[\{x, B, \text{optPar}, y\} = \text{cutplane}(A, b, c, \text{optPar}, L, L, n_1);\]

For further illustrations of how to solve integer programming problems see the example files listed in Table 46

3.2.5 How to Solve Dynamic Programming Problems

We will in this subsection illustrate the simple approach to solve both a knapsack problem and an inventory problem with help of the routines `dpknap` (see Section 3.5.6) and `dpinvent` (Section 3.5.5). The knapsack problem (18) is an example from Kaj H. [32] and the inventory problem is an example from Winston [52, page 1013].
Table 45: Test examples for network programming.

<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>
<tr>
<td>ulyss16</td>
<td>Traveling salesman (TSP) example Odyssey of Ulysses. Calls salesman.</td>
</tr>
<tr>
<td>exulyss16</td>
<td>TSP example Odyssey of Ulysses, 16 nodes. Calls traveling.</td>
</tr>
<tr>
<td>exulyss22</td>
<td>TSP example Odyssey of Ulysses, 22 nodes. Calls traveling.</td>
</tr>
<tr>
<td>expr96</td>
<td>TSP example Africa Subproblem, by Groetschel. 96 nodes. Calls traveling.</td>
</tr>
</tbody>
</table>

Table 46: Test examples for integer programming.

<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>ex139</td>
<td>Test example 139.</td>
</tr>
<tr>
<td>exbalas</td>
<td>Test of 0/1 IP (Balas algorithm) on simple example.</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{max} \quad & f(u) = 7u_1 + u_2 + 4u_3 \\
\text{s.t.} \quad & 2u_1 + 3u_2 + 2u_3 \leq 4 \\
& 0 \leq u_1 \leq 1 \\
& 0 < u_2 < 1 \\
& 0 < u_3 < 2 \\
& u_j \in \mathbb{N}, j = 1, 2, 3
\end{align*}
\] (18)

Problem (18) will be solved by the following definitions and call:

```plaintext
A = [1 2 3 2];
b = 4;
c = [7 2 4];
u_UPP = [1 1 2];
PriLev = 0;
[u, f_opt] = dpknap(A, b, c, u_UPP, PriLev);
```

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

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

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

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

For further illustrations of how to solve dynamic programming problems see the example files listed in Table 47.

<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>exknaps</td>
<td>Test of dpknaps (calls mipSolve and cutplane) on five knapsack examples</td>
</tr>
</tbody>
</table>

### 3.2.6 How to Solve Lagrangian Relaxation Problems

We end up this section with an example of how to solve an integer programming problem with the routine ksrelax, which uses a Lagrangian Relaxation technique. The problem to be solved, (19), is an example from Fischer [20].

\[
\begin{align*}
\max & \quad f(x) = 16x_1 + 10x_2 + 4x_4 \\
\text{s.t} & \quad 8x_1 + 2x_2 + x_3 + x_4 \leq 10 \\
& \quad x_1 + x_2 \leq 1 \\
& \quad x_2 + x_4 < 1 \\
& \quad x_i \in 0/1, j = 1, 2, 3, 4
\end{align*}
\]  \tag{19}

A = [1 8 2 1 4  
1 1 0 0  
0 0 1 1 ];

b = [10 1 1 ]';

c = [16 10 0 4 1'];

r = 1; % Do not relax the first constraint  
x_UPP = [1 1 1 1]';

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

For further illustrations of how to solve Lagrangian Relaxation problems see the example files listed in Table 48.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exrelax</td>
<td>Test of ksrelax on LP example from Fischer -85.</td>
</tr>
<tr>
<td>exrelax2</td>
<td>Simple example, runs ksrelax.</td>
</tr>
<tr>
<td>exIP39rx</td>
<td>Test example 139, relaxed. Calls urelax and plot.</td>
</tr>
</tbody>
</table>
3.3 Printing Utilities and Print Levels

This section is written for the part of OPERA TB which is not using the same input/output format and is not
designed in the same way as NLPLIB TB. Information about printing utilities and print levels for the other
routines could be found in Section 2.8.

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

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

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

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

The OPERA TB routines print large amounts of output if high values for the PriLev parameter is set. To make
the output look better and save space, several printing utilities have been developed, see Table 41.

For matrices there are two routines, mPrint and printmat. The routine printmat prints a matrix with row and
column labels. The default is to print the row and column number. The standard row label is eight characters
long. The supplied matrix name is printed on the first row, the column label row, if the length of the name is at
most eight characters. Otherwise the name is printed on a separate row.

The standard column label is seven characters long, which is the minimum space an element will occupy in the
print out. On a 80 column screen, then it is possible to print a maximum of ten elements per row. Independent
on the number of rows in the matrix, printmat will first display A(:, 1 : 10), then A(:, 11 : 20) and so on.

The routine printmat tries to be intelligent and avoid decimals when the matrix elements are integers. It determines
the maximal positive and minimal negative number to find out if more than the default space is needed. If any
element has an absolute value below 10^-9 (avoiding exact zeros) or if the maximal elements are too big, a switch
is made to exponential format. The exponential format uses ten characters, displaying two decimals and therefore
seven matrix elements are possible to display on each row.

For large matrices, especially integer matrices, the user might prefer the routine mPrint. With this routine a more
dense output is possible. All elements in a matrix row is displayed (over several output rows) before next matrix
row is printed. A row label with the name of the matrix and the row number is displayed to the left using the
Matlab style of syntax.

The default in mPrint is to eight characters per element, with two decimals. However, it is easy to change the
format and the number of elements displayed. For a binary matrix it is possible to display 36 matrix columns in
one 80 column row.

3.4 Driver Routines in OPERA TB

In the following subsections the driver routines in OPERA TB will be described.

3.4.1 lpRun

Purpose
Driver routine for linear programming solvers.

Calling Syntax
Result = lpRun(Solver, Prob, ask, PriLev, probFile, probNumber)
Description of Inputs

 Solver: The name of the solver that should be used to optimize the problem. Default is `lpSolve`. If the solver may run several different optimization algorithms, then the values of `Prob.optParam.alg` and `Prob.optParam.subalg` determines which algorithm.

 Prob: Problem description structure, see Table 5.

 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 2.13.1 page 88.

 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 controled by setting the parameter `Prob.optParam.PriLev`.

 probFile: User problem init file, default `lp_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 15.

 Description

 The driver routine `lpRun` is called by the menu routine `lpOpt` or the graphical user interface routine `ulplib` to solve linear programming problems defined in your problem definition files. It is also possible for the user to call `lpRun` directly from the Matlab command prompt, see Section 3.2. Via `lpRun` you can run the TOMLAB internal solvers `lpSolve`, `lpsimp2` and `akarmark` and the Matlab Optimization Toolbox solver `lp`. You can also, by use of a MEX-file interface run the commercial optimization solvers MINOS and QPOPT.

 M-files Used

 `xxxRun.m`, `xxxRun2.m`, `iniBuild.m`, `Phase1Simplex.m`, `lpDef.m`, `probInit.m`, `mkbound.m`, `lpSolve.m`, `lpsimp2.m`, `akarmark.m`, `lp.m`, `qpoptSOL.m`, `minos.m`, `PrintResult.m`, `iniSolve.m`, `endSolve.m`

3.5 Optimization Routines in OPERA TB

In the following subsections the optimization routines in OPERA TB will be described.

3.5.1 akarmark

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 \( b \in \mathbb{R}^m \).

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 `qoptions.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 [29, pages 411-413]. As the purification algorithm a modified version of the algorithm on page 385 in Bazaraa is used.

Algorithm
See Appendix B.1.

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

M-files Used
\textit{lpDef.m}

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

3.5.2 \textit{balas}

Purpose
Solve binary integer linear programming problems.

\textit{balas} solves problems of the form

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

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

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

Description of Inputs
\[ A \] Constraint matrix.
\[ b \] Right hand side vector.
\[ c \] Cost vector.
\texttt{optPar} Optimization parameter vector, see \textit{goptions.m}.

Description of Outputs
\[ x \] Optimal point.
\texttt{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.

Algorithm
See the code in \textit{balas.m}.

Examples
See \textit{exbalas}.

M-files Used
\textit{lpDef.m}

See Also
\textit{mipSolve}, \textit{cutplane}
3.5.3 cutplane

Purpose
Solve mixed integer linear programming problems (MIP).

cutplane solves problems of the form

\[
\begin{align*}
\min_x & \quad f(x) = c^T x \\
\text{s.t.} & \quad a_i^T x = b_i, \quad i = 1, 2, \ldots, m_q \\
& \quad a_i^T x \leq b_i, \quad i = m_q + 1, \ldots, m \\
& \quad x \geq 0 \\
& \quad x_j \in \mathbb{N}, \quad j = 1, 2, \ldots, n_I \\
& \quad x_j \in \mathbb{R}, \quad j = n_I + 1, \ldots, n
\end{align*}
\]

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

Calling Syntax
\[[x, B, optPar, y] = \text{cutplane}(A, b, \text{optPar}, x.0, B.0, n.I, \text{PriLev})\]

Description of Inputs
- \( A \) Constraint matrix.
- \( b \) Right hand side vector.
- \( c \) Cost vector.
- \( \text{optPar} \) Optimization parameter vector, see \text{goptions.m}.
- \( x.0 \) Starting point.
- \( B.0 \) Logical vector of length \( n \) for basic variables at start.
- \( n.I \) First \( n.I \) \( x \)-values are integer valued.
- \( \text{PriLev} \) Printing level:
  - \( \text{PriLev} = 0 \), no output.
  - \( \text{PriLev} = 1 \), output of convergence results.
  - \( \text{PriLev} > 1 \), output of each iteration.
  - \( \text{PriLev} > 2 \), output of each step in the simplex algorithm.

Description of Outputs
- \( x \) Optimal point.
- \( B \) Optimal basic set.
- \( \text{optPar} \) Optimization parameter vector, see \text{goptions.m}.
- \( y \) Lagrange multipliers at the solution.

Description
The routine \text{cutplane} is an implementation of a cutting plane algorithm with Gomorov cuts. \text{cutplane} uses the linear programming routines \text{Phase1Simplex}, \text{Phase2Simplex} and \text{DualSolve} to solve relaxed subproblems.

Algorithm
See Appendix B.2.

Examples
See \text{exp39}, \text{exknap}, \text{expkorv}.

M-files Used
\text{lpDef.m, Phase1Simplex.m, Phase2Simplex.m, DualSolve.m}

See Also
\text{mipSolve, balas, lpsimp1, lpsimp2, lpdual}

3.5.4 dijkstra

Purpose
Solve the shortest path problem.

Calling Syntax
\[[\text{pred}, \text{dist}] = \text{dijkstra}(s, P, Z, c)\]
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
\( \text{pred} \) \( \text{pred}(j) \) is the predecessor of node \( j \).
\( \text{dist} \) \( \text{dist}(j) \) is the shortest distance from node \( s \) to node \( j \).

Description
\textit{dijkstra} is a direct implementation of the algorithm \textit{DIJKSTRA} in \cite{2}, pages 250-251] for solving shortest path problems using Dijkstra's algorithm. Dijkstra's algorithm belongs to the class of \textit{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 \textit{labelcor} or \textit{modlabel} which belongs to the class of \textit{label correcting} methods.

Algorithm
See Appendix B.3.

Examples
See \textit{exgraph}, \textit{exflow31}.

See Also
\textit{labelcor}, \textit{modlabel}

Limitations
\textit{dijkstra} can only solve problems with nonnegative arc lengths.

3.5.5 \texttt{dpinvent}

Purpose
Solve production/inventory problems of the form

\[
\begin{align*}
\min_u \quad f(u) &= P_s(t) + P(t)^T u(t) + I(t)^T x(t) \\
\text{s.t.} \quad u_L &\leq u(t) & \leq u_U \\
\quad x_L &\leq x(t) & \leq x_U \\
\quad 0 &\leq u(t) & \leq x(t) + d(t) \\
\quad u_j &\in \mathbb{N} & j = 1,2,\ldots,n \\
\quad x_j &\in \mathbb{N} & j = 1,2,\ldots,n
\end{align*}
\]

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

Calling Syntax
\([u,\text{fopt},\text{exit}] = \text{dpinvent}(d, P_s, P, I, s, L, u, L, u, U, x, L, x, U, x, S, x, \text{LAST}, \text{PriLev})\)

Description of Inputs
\( d \) Demand vector.
\( P_s \) Production setup cost.
\( P \) Production cost vector.
\( L \) 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.
\( x, \text{LAST} \) Inventory state at finish.
\( \text{PriLev} \) Printing level:
\( \text{PriLev} = 0 \), no output.
\( \text{PriLev} = 1 \), output of convergence results.
\( \text{PriLev} > 1 \), output of each iteration.
Description of Outputs

- $u$: Optimal control.
- $f_{\text{opt}}$: Optimal function value.
- exit: Exit flag.

Description

dp invent solves production/inventory problems using a forward recursion dynamic programming technique as described in Winston [52, chap. 20].

Algorithm
See Appendix B.4.

Examples
See exinvent.

3.5.6 dpknap

Purpose
Solve knapsack problems of the form

$$\max \quad f(u) = c^T u$$

s.t. \quad Au \leq b

\quad u < u_i

\quad u_i \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

$[u, f_{\text{opt}}, \text{exit}] = \text{dpknap}(A, b, c, u.U, \text{PriLev})$

Description of Inputs

- $A$: Weight 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.
- $f_{\text{opt}}$: Optimal function value.
- exit: Exit flag.

Description
dpknap solves knapsack problems using a forward recursion dynamic programming technique as described in [52, chap. 20]. The Lagrangian relaxation routines ksrelax and urelax call dpknap to solve the knapsack subproblems.

Algorithm
See Appendix B.5.

Examples
See exknap.

3.5.7 DualSolve

Purpose
Solve linear programming problems when a dual feasible solution is available.
**DualSolve** solves problems of the form

\[
\begin{align*}
\min \quad f(x) &= c^T x \\
\frac{s}{t} \quad x_L &\leq x \leq x_U \\
\frac{b}{t} \quad A x &\leq b_U
\end{align*}
\]

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

by rewriting it into standard form as

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

and solving the dual problem

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

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

**Calling Syntax**

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

**Description of Inputs**

- **Prob** Problem description structure. The following fields are used:
  - **Solver.Alg** Variable selection rule to be used
    - 0: Minimum reduced cost (default).
    - 1: Bland’s anti-cycling rule.
    - 2: Minimum reduced cost. Dantzig’s rule.
  - **QP.B** Active set \(B.0\) at start:
    - \(B(i) = 1\): Include variable \(x(i)\) is in basic set.
    - \(B(i) = 0\): Variable \(x(i)\) is set on its lower bound.
    - \(B(i) = -1\): Variable \(x(i)\) is set on its upper bound.
  - **optParam** Structure with special fields for optimization parameters, see Table 6.
  - Fields used are: MaxIter, PriLev, wait, eps.f, eps.Rank and xTol.
  - **QP.c** Constant vector.
  - **A** Constraint matrix for linear constraints.
  - **b.L** Lower bounds on the linear constraints.
  - **b.U** Upper bounds on the linear constraints.
  - **x.L** Lower bounds on the variables.
  - **x.U** Upper bounds on the variables.
  - **x.0** Starting point, must be dual feasible.
  - **y.0** Dual parameters (Lagrangian multipliers) at \(x.0\).
Description of Outputs

Result

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

Iter
Number of iterations.

QP.B
Optimal active set.

ExitFlag
Exit flag:
0: OK.
1: Maximal number of iterations reached. No primal feasible solution found.
2: Infeasible Dual problem.
3: No dual feasible starting point found.
4: Illegal step length due to numerical difficulties. Should not occur.
5: Too many active variables in initial point.

f.k
Function value at optimum.

x.0
Starting point.

x.k
Optimal primal solution x.

v.k
Optimal dual parameters. Lagrange multipliers for linear constraints.

c
Constant vector in standard form formulation.

A
Constraint matrix for linear constraints in standard form.

b
Right hand side in standard form.

Description

When a dual feasible solution is available, the dual simplex method is possible to use. DualSolve implements this method using the algorithm in [29, 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

lpDef.m, cpTransf.m

See Also

lpSolve, Phase2Simplex

3.5.8 karmark

Purpose

Solve linear programming problems of Karmakar’s form

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

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

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

Calling Syntax

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

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 *karmark* is an implementation of Karmakar’s projective algorithm which is of polynomial complexity. The implementation uses the description in Bazaraa [7, page 386]. There is a choice of update, either according to Bazaraa or the rule by Goldfarb and Todd [29, chap. 9]. As the purification algorithm a modified version of the algorithm on page 385 in Bazaraa is used. *karmark* is called by *lpkarma* which transforms linear maximization problems on inequality form into Karmakar’s form needed for *karmark*.

Algorithm
See Appendix B.8.

Examples
See *exstrang, exww597*.

M-files Used
*lpDef.m*

See Also
*lpkarma, akarmark*

3.5.9 *ksrelax*

Purpose
Solve integer linear problems of the form

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

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

Calling Syntax
\[
[x, P, u, f.P, optPar] = ksrelax(A, b, c, r, x.U, optPar)
\]

Description of Inputs
- \( A \) : Constraint matrix.
- \( b \) : Right hand side vector.
- \( c \) : Cost vector.
- \( r \) : Constraint not to be relaxed.
- \( x.U \) : Upper bounds on the variables.
- \( optPar \) : Optimization parameter vector, see *goptions.m*.

Description of Outputs
- \( x \) : Primal solution.
- \( P \) : Lagrangian multipliers.
- \( f.P \) : Function value at \( x.P \).
- \( optPar \) : Optimization parameter vector, see *goptions.m*.

Description
The routine *ksrelax* uses Lagrangian Relaxation to solve integer linear programming problems with linear inequality constraints and simple bounds on the variables. The problem is solved by relaxing all but one constraint and then solve a simple knapsack problem as a subproblem in each iteration. The algorithm is based on the presentation in Fisher [20], using subgradient iterations and a simple line search rule. OPERA TB also contains a routine *urelax* which plots the result of each iteration.

Algorithm
See Appendix B.9.

Examples
See *exrelax, exrelax2*.

M-files Used
*lpDef.m, dpknap.m*
See Also

urdelx

3.5.10 labelcor

Purpose
Solve the shortest path problem.

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

Description of Inputs

\( s \)  
The starting node.

\( p \)  
Pointer vector to start of each node in the matrix \( Z \).

\( Z \)  
A matrix whose columns \( Z(:,1) \) and \( Z(:,2) \) are the tail and head of each arc, respectively.

\( c \)  
Costs related to the arcs in the matrix \( Z \).

Description of Outputs

\( \text{pred} \)  
\( \text{pred}(j) \) is the predecessor of node \( j \).

\( \text{dist} \)  
\( \text{dist}(j) \) is the shortest distance from node \( s \) to node \( j \).

Description
The implementation of \text{labelcor} is based on the algorithm \text{LABEL CORRECTING} in [2, page 260] for solving shortest path problems. The algorithm belongs to the class of \text{label correcting} methods which are applicable to networks with arbitrary arc lengths. \text{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.

Algorithm
See Appendix B.10.

Examples
See exgraph.

See Also
dijkstra, modlabel

Limitations
The network must not contain any negative directed cycle.

3.5.11 lpdual

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

\( \text{lpdual} \) solves problems of the form

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

by rewriting it into standard form and solving the dual problem

\[
\begin{align*}
\max_y \quad & f_D(y) = b^T y \\
\text{s.t.} \quad & A^T y \leq c \\
& y \geq 0
\end{align*}
\]

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

Calling Syntax
\([x, y, B, \text{optPar}] = \text{lpdual}(A, b, c, \text{optPar}, B.0, x.0, y.0)\)
### 3.5.12 lpkarma

**Purpose**

Solve linear programming problems of the form

\[
\begin{align*}
\max_x & \quad f(x) = c^T x \\
\text{s.t.} & \quad Ax \leq b \\
& \quad x \geq 0
\end{align*}
\]

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

**Calling Syntax**

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

**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 Karmarkar's form and calls \textit{karmark} to solve the transformed problem.

**Algorithm**

See Appendix B.12.
Examples
See exstrang, exww597.

M-files Used
lpDef.m, karmark.m

See Also
karmark, akarmark

3.5.13  lpsimp1

Purpose
Find a basic feasible solution to linear programming problems.

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

where \( x, c \in \mathbb{R}^n \), \( A \in \mathbb{R}^{m \times n} \) and \( b \in \mathbb{R}^m, b > 0 \).

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

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

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

Description
The routine \( \text{lpsimp1} \) 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 \( \text{lpsimp2} \) to solve the Phase I problem.

Algorithm
See Appendix B.13.

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

M-files Used
lpDef.m, lpsimp2.m

See Also
lpsimp2

3.5.14  lpsimp2

Purpose
Solve linear programming problems.

\[ \begin{align*}
\min \quad f(x) &= c^T x \\
\text{s.t.} \quad a_i^T x &= b_i \quad i = 1, 2, \ldots, m_e \\
\quad a_i^T x &\leq b_i \quad i = m_e + 1, \ldots, m \\
\quad x &\geq 0
\end{align*} \]
where \( x, c \in \mathbb{R}^n \), \( A \in \mathbb{R}^{m \times n} \) and \( b \in \mathbb{R}^m \).

**Calling Syntax**

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

**Description of Inputs**

- \( A \) Constraint matrix.
- \( b \) Right hand side vector.
- \( c \) Cost vector.
- \( \text{optPar} \) Optimization parameter vector, see \text{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.
- \( \text{optPar} \) Optimization parameter vector, see \text{goptions.m}.
- \( y \) Lagrange multipliers.

**Description**

The routine \text{lpsimp2} implements the Phase II standard revised Simplex algorithm as formulated in Goldfarb and Todd [29, 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).

**Algorithm**

See Appendix B.14.

**Examples**

See \text{exmled}, \text{excycle}, \text{excycle1}, \text{excycle2}, \text{excycle3}, \text{exKleeM}, \text{exfl821}, \text{ex4124b4s}, \text{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.

### 3.5.15 \text{lpsolve} 

**Purpose**

Solve general linear programming problems.

\text{lpsolve} solves problems of the form

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

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**

Result = \text{lpsolve}(Prob)
Description of Inputs

Prob
Problem description structure. The following fields are used:

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

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

optParam
Structure with special fields for optimization parameters, see Table 6.
Fields used are: MaxIter, PriLev, wait, eps.f, eps.Rank, xTol and bTol

QP.c
Constant vector.

A
Constraint matrix for linear constraints.

b.L
Lower bounds on the linear constraints.

b.U
Upper bounds on the linear constraints.

x.L
Lower bounds on the variables.

x.U
Upper bounds on the variables.

x.0
Starting point.

Description of Outputs

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

Iter
Number of iterations.

ExitFlag
0: OK
1: Maximal number of iterations reached.
2: Unbounded feasible region.
3: Rank problems. Can not find any solution point.
4: Illegal x.0 found in Phase2Simplex.
5: No feasible point x.0 found in Phase1Simplex.

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.

q.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 16.

Solver
Solver used.

SolverAlgorithm
Solver algorithm used.

FuncEv
Number of function evaluations. Equal to Iter.

ConstrEv
Number of constraint evaluations. Equal to Iter.

Prob
Problem structure used.

Description
The routine lpSolve implements an active set strategy (Simplex method) for Linear Programming. If the given starting point is not feasible then Phase1Simplex is called. The routine Phase2Simplex is called to solve the Phase 11 program.

M-files Used
lpDef.m, ResultDef.m, Phase1Simplex.m, Phase2Simplex.m

See Also
qpSolve

3.5.16 maxflow

Purpose
Solve the maximum flow problem.
Calling Syntax
\[ \text{[max.flow, x]} = \text{maxflow}(s, t, x.U, P, Z, T, R, \text{PriLev}) \]

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.
  - \( Z(:,1) \): Tail.
  - \( Z(:,2) \): Head.
- \( T \): Trace vector, points to \( Z \) with sorting order Head.
- \( R \): Pointer vector in \( T \) vector for each node.
- \( \text{PriLev} \): Printing Level: 0 Silent, 1 Print result (default).

Description of Outputs
- \( \text{max.flow} \): Maximal flow between node \( s \) and node \( t \).
- \( x \): The flow on each arc.

Description
\( \text{maxflow} \) 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 [42, page 144-145].

Algorithm
See Appendix B.15.

Examples
See \texttt{exflow}, \texttt{exflow31}, \texttt{pathflow}.

3.5.17 \texttt{mipSolve}

Purpose
Solve mixed integer linear programming problems (MIP).

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

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

Calling Syntax
\[ \text{[x, B, optPar, y]} = \text{mipSolve}(A, b, c, \text{optPar}, x.0, B.0, n.I, \text{PriLev}) \]

Description of Inputs
- \( A \): Constraint matrix.
- \( b \): Right hand side vector.
- \( c \): Cost vector.
- \( \text{optPar} \): Optimization parameter vector, see \texttt{options.m}.
- \( x.0 \): Starting point.
- \( B.0 \): Logical vector of length \( n \) for basic variables at start.
- \( n.I \): First \( n.I \) \( x \)-values are integer valued.
- \( \text{PriLev} \): Printing level:
  - \( \text{PriLev} = 0 \): no output.
  - \( \text{PriLev} = 1 \): output of convergence results.
  - \( \text{PriLev} > 1 \): output of each iteration.
  - \( \text{PriLev} > 2 \): output of each step in the simplex algorithm.
Description of Outputs

\[ x \]
Optimal point.

\[ B \]
Optimal basic set.

\[ optPar \]
Optimization parameter vector, see \texttt{options.m}.

\[ y \]
Lagrange multipliers at the solution.

Description

The routine \texttt{mipSolve} is an implementation of a branch and bound algorithm from Nemhauser and Wolsey [45, chap. 8.2]. \texttt{mipSolve} uses the linear programming routines \texttt{Phase1Simplex, Phase2Simplex} and \texttt{DualSolve} to solve relaxed subproblems.

Algorithm

See [45, chap. 8.2] and the code in \texttt{mipSolve.m}. \texttt{Phase1Simplex} to get the solution \( x \) and

Examples

See \texttt{exip39, exknap, expkorv}.

M-files Used

\texttt{bpDef.m, Phase1Simplex.m, Phase2Simplex.m, DualSolve.m}

See Also

\texttt{cutplane, balas, lpsimp1, lpsimp2, lpdual}

3.5.18 \texttt{modlabel}

Purpose

Solve the shortest path problem.

Calling Syntax

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

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

\[ \text{pred} \]
\( \text{pred}(j) \) is the predecessor of node \( j \).

\[ \text{dist} \]
\( \text{dist}(j) \) is the shortest distance from node \( s \) to node \( j \).

Description

The implementation of \texttt{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 \textit{node} to the beginning of \textit{LIST} if \textit{node} has been in \textit{LIST} before, otherwise add \textit{node} at the end of \textit{LIST}. The algorithm belongs to the class of \textit{label correcting} methods which are applicable to networks with arbitrary arc lengths. \texttt{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.

Algorithm

See Appendix B.16.

Examples

See \texttt{exgraph}.

See Also

\texttt{dijkstra, labelcor}

Limitations

The network must not contain any negative directed cycle.
3.5.19  **NWsimplx**

**Purpose**
Solve the minimum cost network flow problem.

**Calling Syntax**

\[ [Z, X, x_{\text{max}}, C, S, my, \text{optPar}] = \text{NWsimplx}(A, b, c, u, \text{optPar}) \]

**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 \( n \).
- \( \text{optPar} \) 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.
- \( x_{\text{max}} \) 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
- \( \text{optPar} \) 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].

**Algorithm**
See [3, page 415] and the code in `NWsimplx.m`.

**Examples**
See `exmcnrfp`.

**M-files Used**

`lpDef.m`, `a2frstar.m`

---

3.5.20  **Phase1Simplex**

**Purpose**
Find a basic feasible solution, i.e. a feasible point, to a constrained set for a general problem

\[
\begin{aligned}
\min_x \quad & f(x) \\
\text{s/t} \quad & x_L \leq x \leq x_U \\
& b_L \leq Ax \leq b_U
\end{aligned}
\]

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 obtain this feasible point, `Phase1Simplex` solves the following Phase 1 linear programming problem.

\[
\begin{aligned}
\min_{x, r_1, r_2, s_1, s_2} \quad & f(r_1, r_2) = e_1^T r_1 + e_2^T r_2 \\
\text{s/t} \quad & x_L \leq x \leq x_U \\
& Ax + \begin{pmatrix} r_1 \\ s_1 \end{pmatrix} = b_U \\
& -Ax + \begin{pmatrix} r_2 \\ s_2 \end{pmatrix} = -b_L \\
& r_1, r_2, s_1, s_2 > 0
\end{aligned}
\]
where $x, x_L, x_U \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b_L, b_U \in \mathbb{R}^m$. $r_1 \in \mathbb{R}^{m_1}$, $r_2 \in \mathbb{R}^{m_2}$, $s_1 \in \mathbb{R}^{m_3}$, $s_2 \in \mathbb{R}^{m_4}$, and $e_1 \in \mathbb{R}^{m_1}$, $e_2 \in \mathbb{R}^{m_2}$ are vectors of ones. It holds that $m_1 + m_3 = m$ and $m_2 + m_4 = m$.

**Calling Syntax**
Result = Phase1Simplex(Prob)

**Description of Inputs**

<table>
<thead>
<tr>
<th>Prob</th>
<th>Problem description structure. The following fields are used:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver.Alg</td>
<td>Variable selection rule used in Phase1Simplex:</td>
</tr>
<tr>
<td>0</td>
<td>Minimum reduced cost (default).</td>
</tr>
<tr>
<td>1</td>
<td>Bland’s anti-cycling rule.</td>
</tr>
<tr>
<td>2</td>
<td>Minimum reduced cost. Dantzig’s rule.</td>
</tr>
<tr>
<td>optParam</td>
<td>Structure with special fields for optimization parameters, see Table 6.</td>
</tr>
<tr>
<td>QP.c</td>
<td>Constant vector.</td>
</tr>
<tr>
<td>$A$</td>
<td>Constraint matrix for linear constraints.</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>$x_L$</td>
<td>Lower bounds on the variables.</td>
</tr>
<tr>
<td>$x_U$</td>
<td>Upper bounds on the variables.</td>
</tr>
</tbody>
</table>

**Description of Outputs**

<table>
<thead>
<tr>
<th>Result</th>
<th>Structure with result from optimization. The following fields are changed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>QP.B</td>
<td>The $n$ first elements in the optimal active set.</td>
</tr>
<tr>
<td>$B(i) = 1$</td>
<td>Include variable $x(i)$ is in basic set.</td>
</tr>
<tr>
<td>$B(i) = 0$</td>
<td>Variable $x(i)$ is set on its lower bound.</td>
</tr>
<tr>
<td>$B(i) = -1$</td>
<td>Variable $x(i)$ is set on its upper bound.</td>
</tr>
<tr>
<td>ExitFlag</td>
<td>Exit flag from Phase1Simplex:</td>
</tr>
<tr>
<td>0</td>
<td>OK.</td>
</tr>
<tr>
<td>1</td>
<td>Feasible region is empty. Some nonzero artificial variables left in the base.</td>
</tr>
<tr>
<td>Inform</td>
<td>Exit flag from Phase2Simplex:</td>
</tr>
<tr>
<td>0</td>
<td>OK.</td>
</tr>
<tr>
<td>1</td>
<td>Maximal number of iterations reached. No basic feasible solution found.</td>
</tr>
<tr>
<td>2</td>
<td>Unbounded feasible region.</td>
</tr>
<tr>
<td>3</td>
<td>Rank problems.</td>
</tr>
<tr>
<td>4</td>
<td>Illegal $x_0$.</td>
</tr>
<tr>
<td>Inform = Inform + 100 if any artificial variable still in base but on zero.</td>
<td></td>
</tr>
<tr>
<td>$x_0$</td>
<td>The full starting point.</td>
</tr>
<tr>
<td>$x.k$</td>
<td>The first $n$ variables in the solution $x$.</td>
</tr>
<tr>
<td>$v.k$</td>
<td>The full set of Lagrange multipliers for the linear constraints.</td>
</tr>
<tr>
<td>Prob.Prob</td>
<td>Problem structure for the Phase1 problem solved.</td>
</tr>
<tr>
<td>Prob.x.k</td>
<td>The full solution vector $x$ for the Phase 1 problem.</td>
</tr>
<tr>
<td>Prob.QP.B</td>
<td>The full optimal active set for the Phase 1 problem.</td>
</tr>
</tbody>
</table>

**Description**

The routine Phase1Simplex solves a Phase I linear programming problem to find a feasible point to a general set of simple bounds and linear constraints. It formulates an expanded LP problem on generalized standard form with slack variables, artificial variables and the original variables. Only the artificial variables have nonzero coefficients in the objective function. Slack variables are added to the inequality constraints with positive upper bound right hand sides and artificial variables are added to the rest of the inequality constraints and all equality constraints. The mathematical problem definition above is somewhat simplified. All linear equations with bounds on infinity are deleted, as well as the corresponding slack or artificial variable. Equalities are only included once. The actual problem to solve is hence reduced in size.

The simplex algorithm in the routine Phase2Simplex is used to solve the problem.

**M-files Used**

lpDef.m, Phase2Simplex.m
3.5.21 Phase2Simplex

Purpose
Solve a linear Phase II program (LP)

Phase2Simplex solves problems of the form

$$\min_{x} f(x) = c^T x$$
$$\text{s.t.} \quad x_L \leq x \leq x_U$$
$$b_L < Ax < b_U$$

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

Calling Syntax
Result = Phase2Simplex(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 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 6.
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.

Description of Outputs

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

Iter Number of iterations.

QP.B Optimal set. $B(i) = 1$, include variable $x(i)$ in basic set.

ExitFlag Exit flag from Phase2Simplex:
0: OK.
1: Maximal number of iterations reached. No basic feasible solution found.
2: Unbounded feasible region.
3: Rank problems.
4: Illegal x.0.

f.k Function value at optimum.

x.0 Starting point.

x.k Solution $x$

v.k Lagrange multipliers.

Description
The Phase2Simplex implements the Phase II standard revised Simplex algorithm. The implementation is based on the description in Goldfarb and Todd [29, page 91] generalized to bounded problems. Phase2Simplex uses QR factorization and numerical safeguarding.

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.

M-files Used

lpDef.m
3.5.22 salesman

Purpose
Solve the symmetric travelling salesman problem.

Calling Syntax
[Tour, f.tour, OneTree, f.tree, w_max, my_max, optPar] =
salesman(C, Zin, Zout, my, f.BestTour, optPar)

Description of Inputs

\( C \) Cost matrix of dimension \( n \times n \) where \( C_{ij} = C_{ji} \) is the cost of arc \((i, j)\). If there are no arc between node \( i \) and node \( j \) then set \( C_{ij} = C_{ji} = \infty \). It must hold that \( C_{ii} = NaN \).

\( Zin \) List of arcs forced in.

\( Zout \) List of arcs forced out.

\( my \) Lagrange multipliers.

\( f.BestTour \) Cost (total distance) of a known tour.

\( optPar \) Optimization parameter vector, see goptions.m.

Description of Outputs

\( Tour \) Arc list of the best tour found.

\( f.tour \) Cost (total distance) of the best tour found.

\( OneTree \) Arc list of the best 1-tree found.

\( f.tree \) Cost of the best 1-tree found.

\( w.max \) Best dual objective.

\( my.max \) Lagrange multipliers at \( w.max \).

\( optPar \) Optimization parameter vector, see goptions.m.

Description

The routine salesman is an implementation of an algorithm by Held and Karp [31] which solves the symmetric travelling salesman problem using Lagrangian Relaxation. The dual problem is solved using a subgradient method with the step length given by the Polyak rule II. The primal problem is to find a 1-tree. Here the routine mintree is called to get a minimum spanning tree. With this method there is no guarantee that an optimal tour is found, i.e. a zero duality gap can not be guaranteed. To ensure convergence, salesman could be used as a subroutine in a Branch and Bound algorithm, see traveling which calls salesman.

Algorithm
See [31] and the code in salesman.m.

Examples
See ulyss16.

M-files Used
lpDef.m, mintree.m

See Also
travelng

3.5.23 TPsimplex

Purpose
Solve transportation programming problems.

TPsimplex solves problems of the form

\[
\min_x \quad f(x) = \sum_{i}^{m} \sum_{j}^{n} c_{ij} x_{ij} \\
\frac{s}{t} \quad \sum_{j}^{n} x_{ij} = s_i \quad i = 1, 2, ..., m \\
\quad \sum_{i}^{m} x_{ij} = d_j \quad j = 1, 2, ..., n \\
\quad x \geq 0
\]
where $x, c \in \mathbb{R}^{m \times n}$, $s \in \mathbb{R}^m$ and $d \in \mathbb{R}^n$.

**Calling Syntax**

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

**Description of Inputs**

- $s$ Supply vector.
- $d$ Demand vector.
- $C$ The cost matrix of linear objective function coefficients.
- $X$ Basic Feasible Solution matrix.
- $B$ Index $(i, j)$ of basis found.
- $\text{optPar}$ Optimization parameter vector, see $qoptions.m$.
- $\text{Penalty}$ If the problem is unbalanced with $\sum_i s_i < \sum_j d_j$, a dummy supply point is added with cost vector $\text{Penalty}$. If the length of $\text{Penalty} < n$ then the value of the first element in $\text{Penalty}$ is used for the whole added cost vector. Default: Computed as $10 \max(C_{i,j})$.

**Description of Outputs**

- $X$ Solution matrix.
- $B$ Optimal set. Index $(i, j)$ of the optimal basis found.
- $\text{optPar}$ Optimization parameter vector, see $qoptions.m$.
- $y$ Lagrange multipliers.
- $C$ The cost matrix, changed if the problem is unbalanced.

**Description**

The routine $TPsimplex$ is an implementation of the Transportation Simplex method described in Luenberger [42, chap 5.4]. In OPERA TB, three routines to find a starting basic feasible solution for the transportation problem are included; the Northwestern Corner method ($TPnw$), the Minimum Cost method ($TPmc$) and Vogel’s approximation method ($TPvogel$). If calling $TPsimplex$ without giving a starting point then Vogel’s method is used to find a starting basic feasible solution.

**Algorithm**

See Appendix B.20.

**Examples**

See $expx.bfs$, $extx119$, $extx119U$, $extxp$.

**M-files Used**

$TPvogel.m$

**See Also**

$TPmc$, $TPnw$, $TPvogel$

**Warnings**

No check is done whether the given starting point is feasible or not.

### 3.5.24 travelng

**Purpose**

Solve the symmetric travelling salesman problem.

**Calling Syntax**

$[\text{BestTour}, f.\text{BestTour}] = travelng(Z, c, \text{optPar})$

**Description of Inputs**

- $Z$ Arcs outgoing from the nodes in increasing order.
  - $Z(:,1)$ Tail. $Z(:,2)$ Head.
- $c$ Costs related to the arcs in the matrix $Z$.
- $\text{optPar}$ Optimization parameter vector, see $qoptions.m$.

**Description of Outputs**

- $\text{BestTour}$ Arc list of the best tour found.
- $f.\text{BestTour}$ Cost (total distance) of the best tour found.
Description
The routine `travelng` is a main routine for the solution of the symmetric traveling salesman problem. This type of problem could be solved by `salesman` but it can’t guarantee that an optimal tour is found, i.e. a zero duality gap can not be guaranteed. To ensure convergence, `travelng` uses a Branch and Bound algorithm and calls `salesman` as a subroutine.

Algorithm
See the code in `travelng.m`.

Examples
See `exvr96`, `exyuls16`, `exyuls22`.

M-files Used
`salesman.m`

See Also
`salesman`

3.5.25  urelax

Purpose
Solve integer linear problems of the form

\[
\max_{s/t} \quad f(x) = c^T x \\
A x \leq b \\
x < x_U \\
x_j \in \mathbb{N} \quad j = 1, 2, ..., n
\]

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

Calling Syntax

\[
[x, P, u, fP] = \text{urelax}(u_{\text{max}}, A, b, c, r, x_U, \text{optPar})
\]

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.
- \( x_U \): Upper bounds on the variables.
- `optPar`: Optimization parameter vector, see `goptions.m`.

Description of Outputs
- \( x \): Primal solution.
- \( u \): Lagrangian multipliers.
- \( fP \): Function value at \( x.P \).

Description
The routine `urelax` is a simple example of the use of Lagrangian Relaxation to solve integer linear programming problems. The problem is solved by relaxing all but one constraint and then solve a simple knapsack problem as a subproblem in each iteration. `urelax` plots the result of each iteration. OPERA TB also contains a more sophisticated routine, `ksrelax`, for solving problems of this type.

Algorithm
See Appendix B.22.

Examples
See `exp39rx`.

M-files Used
`dpknap.m`

See Also
`ksrelax`
3.6 Optimization Subfunction Utilities in OPERA TB

In the following subsections the optimization subfunction utilities in OPERA TB will be described.

3.6.1 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
\begin{itemize}
  \item \( A \) The node-arc incidence matrix. \( A \) is \( m \times n \), where \( m \) is the number of arcs and \( n \) is the number of nodes.
  \item \( C \) Cost for each arc, \( n \)-vector.
  \item \( U \) Upper bounds on flow (optional).
\end{itemize}

Description of Outputs
\begin{itemize}
  \item \( P \) Pointer vector to start of each node in the matrix \( Z \).
  \item \( Z \) Arcs outgoing from the nodes in increasing order.
    \( Z(\cdot, 1) \) Tail. \( Z(\cdot, 2) \) Head.
  \item \( c \) Costs related to the arcs in the matrix \( Z \).
  \item \( T \) Trace vector, points to \( Z \) with sorting order Head.
  \item \( R \) Reverse pointer vector in \( T \) for each node.
  \item \( u \) Upper bounds on flow if \( U \) is given as input, else infinity.
\end{itemize}

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}.

3.6.2 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
\begin{itemize}
  \item \( s \) The starting node.
  \item \( P \) Pointer vector to start of each node in the matrix \( Z \).
  \item \( Z \) Arcs outgoing from the nodes in increasing order.
    \( Z(\cdot, 1) \) Tail. \( Z(\cdot, 2) \) Head.
  \item \( c \) Costs related to the arcs in the matrix \( Z \).
\end{itemize}

Description of Outputs
\begin{itemize}
  \item \( \text{pred} \) \( \text{pred}(j) = \) Predecessor of node \( j \).
  \item \( \text{mark} \) \( \text{mark}(j) = 1 \) the node is reachable from node \( s \).
\end{itemize}

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.

Algorithm
See Appendix B.6.
Examples
See exgraph.

See Also
gsearch

3.6.3 gsearch

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

Calling Syntax
[pre, mark] = gsearch(s, P, Z, c)

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

Description of Outputs
pre pred(j) = Predecessor of node j.
mark If mark(j) = 1 the node is reachable from node s.

Description
gsearch is searching for all nodes in a network which is reachable from the given source node s. The implementation is a variation of the Algorithm SEARCH in [2, pages 231-233]. The algorithm uses a 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.

Algorithm
See Appendix B.7.

Examples
See exgraph.

See Also
gsearch

3.6.4 mintree

Purpose
Find the minimum spanning tree of an undirected graph.

Calling Syntax
[Z.tree, cost] = mintree(C, Zin, Zout)

Description of Inputs
C Cost matrix of dimension n x n where \( C_{ij} = C_{ji} \) is the cost of arc \( (i, j) \). If there are no arc between node \( i \) and node \( j \) then set \( C_{ij} = C_{ji} = \infty \). It must hold that \( C_{ii} = NaN \).
Zin List of arcs which should be forced to be included in Z.tree.
Zout List of arcs which should not be allowed to be included in Z.tree (could also be given as NaN in C).

Description of Outputs
Z.tree List of arcs in the minimum spanning tree.
cost The total cost.

Description
mintree is an implementation of Kruskal's algorithm for finding a minimal spanning tree of an undirected graph. The implementation follows the algorithm description in [3, page 520-521]. It is possible to give as input, a list
of those arcs which should be forced to be included in the tree as well as a list of those arcs which should not be allowed to be included in the tree. *mintree* is called by *salesman*.

**Algorithm**
See Appendix B.17.

### 3.6.5 TPmc

**Purpose**
Find a basic feasible solution to the Transportation Problem.

**Calling Syntax**

```
[X,B] = TPmc(s, d, C)
```

**Description of Inputs**

- `s` Supply vector of length `m`.
- `d` Demand vector of length `n`.
- `C` The cost matrix of linear objective function coefficients.

**Description of Outputs**

- `X` Basic feasible solution matrix.
- `B` Index `(i, j)` of the basis found.

**Description**

*TPmc* is an implementation of the Minimum Cost method for finding a basic feasible solution to the transportation problem. The implementation of this algorithm follows the algorithm description in Winston [52, chap. 7.2].

**Algorithm**
See Appendix B.18.

**Examples**
See *expt.bfs, extu119, extu119U, extp*.

**See Also**

*TPnw, TPvogel, TPsimple*

### 3.6.6 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 [52, chap. 7.2].

**Algorithm**
See Appendix B.19.

**Examples**
See *expt.bfs, extu119, extu119U, extp*.

**See Also**

*TPmc, TPvogel, TPsimple*
3.6.7 TPvogel

Purpose
Find a basic feasible solution to the Transportation Problem.

Calling Syntax
[X, B] = TPvogel(s, d, C, PriLev)

Description of Inputs
s    Supply vector of length m.
d    Demand vector of length n.
C    The cost matrix of linear objective function coefficients.
PriLev If PriLev > 0, the matrix X is displayed in each iteration.
        If PriLev > 1, pause in each iteration.
        Default: PriLev = 0.

Description of Outputs
X    Basic feasible solution matrix.
B    Index (i, j) of the basis found.

Description
TPvogel is an implementation of Vogel's method for finding a basic feasible solution to the transportation problem. The implementation of this algorithm follows the algorithm description in Winston [52, chap. 7.2].

Algorithm
See Appendix B.21.

Examples
See extp.bfs, extu119, extu119U, extp.

See Also
TPmc, TPnw, TPsimpla

3.6.8 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 x 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].
3.7 User Utility Functions in OPERA TB

In the following subsections the user utility functions in OPERA TB will be described.

3.7.1 cpTransf

**Purpose**

Transform general convex programs on the form

\[
\min_x f(x) \\
\text{s.t.} \quad x_L \leq x \leq x_U \\
\quad b_L \leq Ax \leq b_U
\]

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**

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

**Description of Inputs**

- **Prob**
  Problem description structure. The following fields are used:
  
  \( QP.c \) Constant vector \( c^T x \).
  
  \( A \) Constraint matrix for linear constraints.
  
  \( b_L \) Lower bounds on the linear constraints.
  
  \( b.U \) Upper bounds on the linear constraints.
  
  \( x_L \) Lower bounds on the variables.
  
  \( x.U \) Upper bounds on the variables.

- **TransfType**
  Type of transformation, see the description below.

- **MakeEQ**
  Flag, if set true, make standard form (all equalities).

- **LowInf**
  Variables equal to \(-Inf\) or variables \(< LowInf\) are set to \(LowInf\) before transforming the problem. Default \(-10^{-4}\). \([LowInf]\) are limit if upper bound variables are to be used.

**Description of Outputs**

- **AA**
  The expanded linear constraint matrix.

- **bb**
  The expanded upper bounds for the linear constraints.

- **meq**
  The first \( meq \) equations are equalities.

**Description**

If \( TransfType = 1 \) the program is transformed into the form

\[
\min_x f(x - x_L) \\
\text{s.t.} \quad AA(x - x_L) \leq bb \\
\quad x - x_L \geq 0
\]

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

\( x(\neg x \equiv x \equiv U) = (x - x \equiv L) + x (\neg x \equiv x \equiv U) \)

If \( TransfType = 2 \) the program is transformed into the form

\[
\min_x f(x) \\
\text{s.t.} \quad AA(x) \leq bb \\
\quad x_L \leq x \leq x_U
\]

where the first \( meq \) constraints are equalities.

If \( TransfType = 3 \) the program is transformed into the form

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

where the first \( meq \) constraints are equalities.
4 Interfaces

4.1 The MEX-file Interface

TOOLAB is an open system with possibilities to interact with other program packages. An optimization solver implemented in Fortran or C is called from TOOLAB using a MEX-file interface. MEX-file interfaces for both Fortran and C are easy to develop for Unix machines. Interfaces to many solvers are available on Unix. On PC machines, there has been problems to make Fortran MEX-file interfaces that work properly. We have made general MEX-file interfaces in C and converted solvers written in Fortran to C using the Fortran to C converter f2c [19]. This solution is well-working and it should be easy to expand the list of available solvers to TOOLAB.

Presently, MEX-file interfaces has been developed for six general-purpose solvers available from the Systems Optimization Laboratory, Department of Operations Research, Stanford University, California; NPSOL 5.02 [27], NPOPT 1.0-10 (updated version of NPSOL), NLSSOL 5.0-2, QPOPT 1.0-10, LSSOL 1.05 and LPOPT 1.0-10. Furthermore, an interface to MINOS 5.5 [44] has been developed. MEX-file interfaces are available for both Unix and PC systems.

4.2 The Matlab Optimization Toolbox Interface

Included in TOOLAB is an interface the a number of the solvers in the Matlab Optimization Toolbox (OPTIM) [13]. The solvers that are possible to use are listed in Table 49, assuming the user has a valid license. The TOOLAB optimization driver routine checks if the routine is in the path and then calls the Matlab function feval to run it. Two low-level interface routines have been written. The constr solver needs both the objective function and the vector of constraint functions in the same call, which nlp_fe 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_quad.

OPTIM is using a parameter vector OPTIONS of length 18, that the routine foptions is setting up the default values for. TOOLAB is using a similar parameter vector of larger size, optPar, with the first 18 elements reserved to have the same interpretation as the OPTIONS vector. This makes the use of OPTIM routines trivial in TOOLAB.

<table>
<thead>
<tr>
<th>Function</th>
<th>Type of problem solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>constr</td>
<td>Constrained minimization.</td>
</tr>
<tr>
<td>leastsq</td>
<td>Nonlinear least squares.</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>lp</td>
<td>Linear programming.</td>
</tr>
<tr>
<td>qp</td>
<td>Quadratic programming.</td>
</tr>
<tr>
<td>nnls</td>
<td>Nonnegative linear least squares (no license needed).</td>
</tr>
<tr>
<td>consls</td>
<td>Constrained linear least squares.</td>
</tr>
</tbody>
</table>

Table 49: Matlab Optimization toolbox routines with a TOOLAB interface.

4.3 The CUTE Interface

The Constrained and Unconstrained Testing Environment (CUTE) [11, 12] 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; outdif.dat or outdif.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 outdif.dat or outdif.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 outsdf.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 `outsdf.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. Precompiled 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 NLPLIB TB, using any solver callable from the toolbox.

### 4.4 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 [24] was built. The reason to choose AMPL was that it has a rudimentary Matlab interface written in C [26] that could easily be used.

AMPL is using ASCII files to define a problem. The naming convention is to use the problem name and various extensions, e.g. `rosenbr.mod` and `rosenbr.dat` for the Rosenbrock banana function. These files are normally converted to binary files with the extension `nl`, called `nl`-files. This gives a file `rosenbr.nl` for our example. Then AMPL invokes a solver with two arguments, the problem name, e.g. `rosenbr`, and a string `-AMPL`. The second argument is a flag telling AMPL is the caller. After solving the problem, the solver creates a file with extension `sol`, e.g. `rosenbr.sol`, containing a termination message and the solution it has found.

The current TOMLAB AMPL interface is an interface to the problems defined in the AMPL `nl`-format. TOMLAB assumes the `nl`-files to be stored in directory `/tomlab/ampl` or `/tomlab/amplsp` (for sparse problems). When using the TOMLAB interface, the user either gets a menu of the `nl`-files found or directly makes a choice of which problem to solve. The initialization routine in TOMLAB for AMPL problems, `amp_prob`, either calls `amplfunc` or `spamfunc`, the two MEX-file interface routines written by Gay [26]. The low level routines `amp.f`, `amp.g`, etc. calls the same MEX-file interface routines, and dependent on the parameters in the call, the appropriate information is returned.

Note that the design of the AMPL solver interface makes it easy to run the NLPLIB TB solvers from AMPL using the Matlab Engine interface routines, a possible extension in the future. But indeed, any solver callable from NLPLIB TB may now solve problems formulated in the AMPL language.
A Description of Algorithms in NLPLIB TB

A.1 clsSolve

Transform the problem to the following form

\[
\min_{\bar{x}} \quad f(x) = \frac{1}{2} r^T(x)r(x) \\
\text{s.t.} \quad a_i^T \bar{x} = b_i, \quad i \in E \\
\quad a_i^T \bar{x} \geq b_i, \quad i \in I \\
\quad x_L \leq \bar{x} \leq x_U
\]

where \(E\) is the set of linear equalities and \(I\) the set of linear inequalities.

Set \(k = 0\), \(stop = 0\) and the number of consecutive zero steps, \(\alpha_0 = 0\).

Set \(G_{N\text{flag}} = 1\) and \(A_{H} = I\).

Set \(x^{(-1)} = \infty\), \(f^{(-1)} = \infty\), \(\alpha = 1\) and \(\alpha_{\text{max}} = 10^{20}\).

Set \(\bar{x}_i = \max(x_i^{(0)}, x_{L_i})\) and \(x_i^{(0)} = \min(\bar{x}_i, x_{U_i})\), \(i = 1, 2, \ldots, n\).

if \(p\text{Solve}\) then

Call the presolve analysis routine \(p\text{Solve}\).

if any constraint was deleted in the presolve analysis routine then

Update \(E\) and \(I\).

end if

end if

if \(x^{(0)}\) is not feasible (with respect to the constraints) then

Solve the QP:

\[
\min_{\tilde{x}} \quad f(\tilde{x}) = \frac{1}{2} \tilde{x}^T B \tilde{x} - x^{(0)T} B \tilde{x} \\
\text{s.t.} \quad a_i^T \tilde{x} = b_i, \quad i \in E \\
\quad a_i^T \tilde{x} \geq b_i, \quad i \in I \\
\quad x_L < \tilde{x} < x_U
\]

where \(B = \text{diag}\left(\frac{1}{\max(10^{-8},x_i^{(0)2})}\right)\) will minimize the relative deviation between \(\tilde{x}\) and \(x^{(0)}\) and \(B = I\) minimizes the absolute deviation.

Set \(x^{(0)} = \tilde{x}\).

end if

while not convergence do

if \(k = 0\) or \(I = \emptyset\) then

if \(x_i^{(k)}\) is beyond or very close to lower or upper bound, \(i = 1, 2, \ldots, n\) then

Move \(x_i^{(k)}\) to bound.

end if

Set up working sets for variables active on lower and upper bounds respectively.

\(V_L = \{i: x_i^{(k)} = x_{L_i}\}\), \(V_U = \{i: x_i^{(k)} = x_{U_i}\}\)

Set \(n_{r\text{act}}\) equal to the number of active variables.

end if

if any variable has been moved to bound or \(k = 0\) then

Compute \(r^{(k)}, J^{(k)}, f^{(k)}\) and \(g^{(k)}\).

end if

if \(k = 0\) then

Compute \(H^{(k)} = J^{(k)T} J^{(k)}\).

end if

if \(I = \emptyset\) then

Set the constraint working set \(W = E\).

Compute first order Lagrange multiplier estimate \(\lambda\).

\(\lambda_i = -g_i\) if \(i \in V_U\), \(\lambda_i = g_i\) if \(i \in V_L\) and \(\lambda_i = 0\) else.

if \(n_{r\text{act}} > 0\) and \(\alpha_0 < 3\) then

if \(n_{r\text{act}} < n\) then

Release all variables \(x_i\), not activated in the previous iteration, where \(x_L \neq x_U\) and \(\lambda_i < -b_{\text{tol}}\).
if $\alpha = 0$ then
  Release all variables $x_i$, inactive in the previous iteration, where $x_{L,i} \neq x_{U,i}$ and $\lambda_i < -b_{T,ol}$.
end if
else
  Release all variables $x_i$ where $x_{L,i} \neq x_{U,i}$ and $\lambda_i < -b_{T,ol}$.
end if
else
  if $k = 0$ then
    Set up the initial constraint working set $W = E \cup \{ i : i \in I \land a_i^T x = b_i \}$.
    The number of active variables and constraints must not exceed $n$.
  end if
if $k > 0$ and the release of more than one variable in the previous iteration resulted in a zero step then
  Activate the released variables.
  Do not allow more than one variable to be released in this iteration.
end if
if there are any active variable or constraint then
  Compute first order Lagrange multiplier estimate $\lambda$ by solving the overdetermined system
  \[
  \begin{pmatrix}
  A_{W(i)} \\
  e_i, \quad i \in V_L^{(k)} \\
  -e_i, \quad i \in V_U^{(k)}
  \end{pmatrix}^T \lambda = g^{(k)}
  \]
  where $e_i$ is the $i$th unit row vector.
  Set $\lambda_{\text{min}} = \min(\lambda)$
if $\lambda_{\text{min}} < -10^{-8}$ then
  if only one variable is allowed to be released then
    if $\lambda_{\text{min}}$ corresponds to a constraint then
      Release the constraint if it was not activated in the previous iteration.
    else
      Release the corresponding variable $x_i$, if it was not activated in the previous iteration and if $x_{L,i} \neq x_{U,i}$.
    end if
  else
    if $\lambda_{\text{min}}$ corresponds to a constraint then
      Release the constraint if it was not activated in the previous iteration.
    end if
    Release all variables $x_i$, not activated in the previous iteration, where $x_{L,i} \neq x_{U,i}$ and $\lambda_i < -10^{-8}$.
  end if
end if
if variables or constraints was released then
  Update the corresponding working sets and $nr_{act}$.
end if
if there has been any changes in the working sets or $k = 0$ then
  Compute $Z$, null space basis for $A_W = A_{ij} : i \in W \land j \notin V_L \cup V_U$
end if
if no variable or no constraint was released then
  if all Lagrange multipliers corresponding to the active inequality constraints are $\geq -10^{-8}$ and for all active variables $i$ there either holds that $\lambda_i \geq -10^{-8}$ or $x_{L,i} = x_{U,i}$ then
    Check convergence criterias, see A.1.1.
  end if
if any convergence criteria are fulfilled or $nr_{act} = n$ then
  Set $\text{stop} = 1$.
end if
end if
Check stop criterias, see A.1.2.
if any stop criteria are fullfilled then
    Set \( stop = 1 \).
end if
if \( stop \) then
END ALGORITHM
end if
Compute search direction \( p \) with chosen method, see A.1.3.
Set \( p_{full} = p_i \) if \( i \notin V_L \cup V_U \) else set \( p_{full} = 0 \).
Compute \( \alpha_1 \), step length estimate sent to line search routine.
if \( k = 0 \) or \( ||p|| = 0 \) then
    Set \( \tilde{\alpha}_1 = 1 \).
else
    Set \( \tilde{\alpha}_1 = \min \left( 1, \frac{2 \text{max}(f^{(k-1)} - f^{(k)}, 10\epsilon_2)}{\tilde{g}_i, p = 0} \right) \), where \( \tilde{g}_i \) = \( i \notin V_L \cup V_U \).
    if \( \tilde{\alpha}_1 < 0 \) then
        Change sign on \( \tilde{\alpha}_1 \), \( p_{full} \) and \( p \).
    end if
end if
Set \( \alpha_1 = \max(0.5, \tilde{\alpha}_1) \).
if \( ||p|| = 0 \) then
Set \( x^{(k+1)} = x^{(k)} \), \( f^{(k+1)} = f^{(k)} \), \( g^{(k+1)} = g^{(k)} \), \( r^{(k+1)} = r^{(k)} \) and \( J^{(k+1)} = J^{(k)} \).
else
    Compute \( \alpha_{max} \), the maximum step \( \alpha \) such that \( x + \alpha p_{full} \) is feasible with respect to the variable bounds and the nonactive inequality constraints.
    if \( \alpha_{max} < 10^{-14} \) then
        Set \( \alpha = 0 \)
    else
        Solve the line search problem \( \min_{0 < \alpha \leq \alpha_{max}} f(x + \alpha p_{full}) \).
    end if
    if \( \alpha = \alpha_{max} \) then
        if \( \alpha_{max} \) is restricted by a variable bound then
            Activate the corresponding variable.
        else
            Activate the corresponding constraint.
        end if
    end if
    if \( \alpha < 10^{-14} \) then
        Set \( \alpha_0 = \alpha_0 + 1 \).
    else
        Set \( \alpha_0 = 0 \).
    end if
Set \( x^{(k+1)} = x^{(k)} + \alpha p_{full} \), \( f^{(k+1)} \), \( g^{(k+1)} \), \( r^{(k+1)} \) and \( J^{(k+1)} \) was computed in the line search.
if \( \alpha > 10^{-14} \) then
    Depending on the chosen method, update the approximation of the Hessian, see A.1.4.
end if
end if
end while

A.1.1 Convergence criterias

- \( \max_i \frac{|x_i^{(k)} - x_i^{(k-1)}|}{\max(|x_i^{(k)}|, \text{size}_x)} \leq \epsilon_2 \) and \( \alpha_{max} > 10^{-14} \)
\[ \max \left( \left| Z^T \hat{g}^{(k)} \right| \max_{i \in V_L \cup V_U} \left| x_i^{(k)} \right|, \text{size}_x \right) \leq \epsilon_g \max \left( \text{abs}(f^{(k)}), \text{size}_f \right) \]

- \( f^{(k)} \leq \epsilon_{abs} \text{size}_f \)
- Relative function value reduction low for LowIts iterations.

A.1.2 Stop criteria

- \( k \geq \text{MaxIter} \)
- \( f^{(k)} \leq f_{\text{Low}} \)

A.1.3 Computation of Search Direction

**Gauss-Newton or hybrid method if** \( GN_{\text{flag}} = 1 \)

Solve the overdetermined system \( \hat{J} \tilde{Z} p = -r \) with rank estimation and a subspace minimization technique either using Singular Value Decomposition or using QR-Decomposition with or without pivoting.

\( \hat{J}_{ij} = J_{ij} : j \notin V_L \cup V_U \).

**Fletcher-Xu, Al-Baali-Fletcher and Huschens TSSM if** \( GN_{\text{flag}} = 0 \)

Solve \( Z^T \hat{H} \tilde{Z} p = -Z^T \hat{g} \) using Singular Value Decomposition with rank estimation and a subspace minimization technique.

\( \hat{H}_{ij} = H_{ij} : i, j \notin V_L \cup V_U \).

A.1.4 Update Procedure

**Fletcher-Xu**

Set \( z = \alpha p_{\text{full}} \).

if \( f^{(k)} \leq f^{(k+1)} \geq 0.2 f^{(k)} \) or \( \|z\| \leq \epsilon_z \) then

Set \( GN_{\text{flag}} = 1 \).

Set \( H^{(k+1)} = J^{(k+1)^T} J^{(k+1)} \).

else

Set \( GN_{\text{flag}} = 0 \).

Set \( y = J^{(k+1)^T} J^{(k+1)} z + \left(J^{(k+1)^T} - J^{(k)^T}\right) r^{(k+1)} \).

if \( z^T y < 0.01 z^T (g^{(k+1)} - g^{(k)}) \) then

Set \( w = g^{(k+1)} - g^{(k)} \).

else

Set \( w = y \).

end if

if \( z^T w < 10^{-13} \) or \( z^T H^{(k)} z < 10^{-13} \) then

Set \( GN_{\text{flag}} = 1 \).

Set \( H^{(k+1)} = J^{(k+1)^T} J^{(k+1)} \).

else

Set \( H^{(k+1)} = H^{(k)} + \frac{ww^T}{z^T w} - \frac{H^{(k)} z z^T H^{(k)} z}{z^T H^{(k)} z} \).

end if

end if

**Al-Baali-Fletcher**

Set \( z = \alpha p_{\text{full}} \).

if \( f^{(k)} \leq f^{(k+1)} \geq 0.2 f^{(k)} \) or \( \|z\| \leq \epsilon_z \) then

Set \( GN_{\text{flag}} = 1 \).

Set \( H^{(k+1)} = J^{(k+1)^T} J^{(k+1)} \).

else

Set \( GN_{\text{flag}} = 0 \).
Set \( y = J^{(k+1)^T} J^{(k+1)} z + (J^{(k+1)^T} - J^{(k)^T}) r^{(k+1)}. \)

if \( z^T y < 0.2 z^T H^{(k)} z \) then
  Set \( w = \frac{0.8 z^T H^{(k)} z}{z^T H^{(k)} z - z^T y} y + \left(1 - \frac{0.8 z^T H^{(k)} z}{z^T H^{(k)} z - z^T y}\right) H^{(k)} z. \)
else
  Set \( w = y. \)
end if

if \( z^T w < 10^{-10} \) or \( z^T H^{(k)} z < 10^{-10} \) then
  Set \( G N_{flag} = 1. \)
  Set \( H^{(k+1)} = J^{(k+1)^T} J^{(k+1)}, \)
else
  Set \( H^{(k+1)} = H^{(k)} + \frac{w y^T}{z^T w} - \frac{H^{(k)} z z^T H^{(k)} z}{z^T H^{(k)} z}. \)
end if

Huschens TSSM

Set:
\( G N_{flag} = 0, \)
\( z = p_{full}, \)
\( y^* = (J^{(k+1)} - J^{(k)})^T p_{(k+1)} \)
\( y = J^{(k+1)^T} J^{(k+1)} z + \|p_{(k+1)}\| y^* \)
\( B_s = J^{(k+1)^T} J^{(k+1)} + \|p_{(k+1)}\| A^{(k)} \)

if \( z^T B_s z > 0 \) and \( y^T z > 0 \) then
  Set \( v = y + \frac{y^T z}{z^T B_s z} B_s z. \)
else
  Set \( v = y. \)
end if

Set \( A^{(k+1)} = A^{(k)} + \frac{y^T - A^{(k)} z}{v^T B_s z} v + (y^T - A^{(k)} z)^T z v z^T}{v^T z} - \frac{(y^T - A^{(k)} z)^T z v z^T}{v^T z}. \)
Set \( H^{(k+1)} = H^{(k)} + \|p_{(k+1)}\| A^{(k+1)} \).

A.2 glbSolve

Set the global/local search weight parameter \( \epsilon \)
Set \( C_{i1} = \frac{1}{3} \) and \( L_{i1} = \frac{1}{3}, i = 1, 2, 3, ..., n. \)
Set \( F_1 = f(x), \) where \( x_i = x_{U_i} + C_{i1} (x_{U_i} - x_{L_i}), i = 1, 2, 3, ..., n. \)

Set \( D_1 = \left\{ \sum_{k=1}^{n} L_i^2 \right\}. \)
Set \( f_{min} = F_1 \) and \( i_{min} = 1. \)
for \( t = 1, 2, 3, ..., T \) do
  Set \( \hat{S} = \left\{ j : D_j \geq D_{i_{min}} \land F_j = \min_i \{ F_i : D_i = D_j \} \right\}. \)
  Define \( \alpha \) and \( \beta \) by letting the line \( y = \alpha x' + \beta \) pass through the points \( (D_{i_{min}}, F_{i_{min}}) \) and \( \left(\max(D_j), \min_j \{ F_i : D_i = \max(D_j) \}\right). \)
  Let \( \hat{S} \) be the set of all rectangles \( j \in \hat{S} \) fullfilling \( F_i \leq \alpha D_j + \beta + 10^{-12}. \)
  Let \( S \) be the set of all rectangles in \( \hat{S} \) which lies on the convex hull defined by the points \( (D_j, F_j), j \in \hat{S}. \)
while \( S \neq \emptyset \) do
  Select \( j \) as the first element in \( S. \)
  Set \( S = S \setminus \{ j \}. \)
  Let \( I \) be the set of dimensions with maximum rectangle side length, i.e. \( I = \left\{ i : D_{ij} = \max_k (D_{kj}) \right\}. \)
  Let \( \delta \) equal two-thirds of this maximum side length, i.e. \( \delta = \frac{2}{3} \max_k (D_{kj}). \)
  for all \( i \in I \) do
Set $c_k = C_{kj}, k = 1, 2, 3, ..., n$.
Set $\hat{c} = c + \delta e_i$ and $\hat{\hat{c}} = c - \delta e_i$, where $e_i$ is the $i$th unit vector.
Compute $f = f(\hat{c})$ and $\bar{f} = f(\hat{\hat{c}})$ where $\hat{c}_k = x_{L_k} + \hat{c}_k (x_{U_k} - x_{L_k})$ and $\hat{\hat{c}}_k = x_{L_k} + \hat{\hat{c}}_k (x_{U_k} - x_{L_k})$.

Set $w_i = \min(f, \bar{f})$.
Set $C = \begin{pmatrix} C & \hat{c} & \hat{\hat{c}} \end{pmatrix}$ and $F = \begin{pmatrix} F & f & \bar{f} \end{pmatrix}$.

end for

while $I \neq \emptyset$ do
Select the dimension $i \in I$ with the lowest value of $w_i$ and set $I = I \setminus \{i\}$.
Set $L_{ij} = \frac{1}{2} \delta$.
Let $j$ and $\hat{j}$ be the indices that corresponds to the points $\hat{c}$ and $\hat{\hat{c}}$ above.
Set $L_{kj} = L_{kj}$ and $L_{\hat{c}j} = L_{kj}$, $k = 1, 2, 3, ..., n$.

Set $D_j = \sqrt{\sum_{k=1}^{n} L_{kj}^2}$.
Set $D_j = D_{\hat{j}}$ and $D_{\hat{j}} = D_j$.
end while
end while
Set $f_{\text{min}} = \min(F_i)$.
Set $i_{\text{min}} = \arg\min \left( \frac{E_j - f_{\text{min}} + E}{D_j} \right)$, where $E = \max(\epsilon |f_{\text{min}}|, 10^{-8})$.
end for

A.2.1  conhull

The points $(x_i, y_i), i = 1, 2, 3, ..., m$ are given with $x_1 \leq x_2 \leq ... \leq x_m$.
Set $h = (1, 2, ..., m)$
if $m \geq 3$ then
Set $\text{START} = 1, v = \text{START}, w = m$ and $\text{flag} = 0$.
while $\text{next}(v) \neq \text{START}$ or $\text{flag} = 0$ do
if $\text{next}(v) = w$ then
Set $\text{flag} = 0$.
end if
Set $a = v, b = \text{next}(v)$ and $c = \text{next}(\text{next}(v))$.
Set $A = \begin{pmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{pmatrix}$.
if det$(A) \geq 0$ then
Set leftturn = 0.
else
Set leftturn = 1.
end if
if leftturn then
Set $v = \text{next}(v)$.
else
Set $j = \text{next}(v)$.
Set $x = (x_1, x_2, ..., x_{j-1}, x_{j+1}, ..., x_m), y = (y_1, y_2, ..., y_{j-1}, y_{j+1}, ..., y_m)$ and
$h = (h_1, h_2, ..., h_{j-1}, h_{j+1}, ..., h_m)$.
Set $m = m - 1, w = w - 1$ and $v = \text{pred}(v)$.
end if
end while
end if

A.2.2  next

if $v = m$ then
Set $i = 1$.
else

A.2.3 pred

if \( v = 1 \) then
  Set \( i = m \)
else
  Set \( i = v - 1 \).
end if

A.3 intpol2

Transform \( g_0 \) to \([0, 1] \) by setting \( \hat{g}_0 = g_0 \) \((x_1 - x_0)\).
Set \( c = f_1 - f_0 - \hat{g}_0 \).
if \( c = 0 \) then
  Set \( \alpha = \min(a, b) \).
else
  Transform \( a \) and \( b \) to \([0, 1] \) by setting \( \tilde{a} = \frac{a - x_0}{x_1 - x_0} \) and \( \tilde{b} = \frac{b - x_0}{x_1 - x_0} \).
  Define \( q(z) = f_0 + \tilde{g}_0 z + cz^2 \).
  Set \( z_{\text{min}} = \frac{\tilde{r}}{2c} \).
  if \( q(z_{\text{min}}) \leq q(\tilde{a}) \) and \( z_{\text{min}} \in [\tilde{a}, \tilde{b}] \) then
    if \( q(z_{\text{min}}) \leq q(\tilde{b}) \) then
      Set \( \alpha = x_0 + z_{\text{min}} \) \((x_1 - x_0)\).
    else
      Set \( \alpha = \tilde{b} \).
    end if
  else if \( q(\tilde{a}) < q(\tilde{b}) \) then
    Set \( \alpha = a \).
  else
    Set \( \alpha = b \).
  end if
end if

A.4 intpol3

Transform \( g_0 \) and \( g_1 \) to \([0, 1] \) by setting \( \hat{g}_0 = g_0 \) \((x_1 - x_0)\) and \( \hat{g}_1 = g_1 \) \((x_1 - x_0)\).
Set \( r = 3(f_1 - f_0) - 2\hat{g}_0 - \hat{g}_1 \).
Set \( s = \hat{g}_0 + \hat{g}_1 - 2(f_1 - f_0) \).
if \( |s| < 10^{-12} |r| \) then
  Call quadratic ineterpolation routine \textit{intpol2}.
else
  Transform \( a \) and \( b \) to \([0, 1] \) by setting \( \tilde{a} = \frac{a - x_0}{x_1 - x_0} \) and \( \tilde{b} = \frac{b - x_0}{x_1 - x_0} \).
  Define \( c(z) = f_0 + \tilde{g}_0 z + rz^2 + sz^3 \).
  Set \( z_1 = \frac{-r + \sqrt{r^2 - 3sz_0}}{3s} \) and \( z_2 = \frac{-r - \sqrt{r^2 - 3sz_0}}{3s} \).
  if \( z_1 \in [\tilde{a}, \tilde{b}] \) or \( z_2 \in [\tilde{a}, \tilde{b}] \) then
    if \( z_1 \in [\tilde{a}, \tilde{b}] \) then
      if \( z_2 \in [\tilde{a}, \tilde{b}] \) then
        if \( c(z_1) \leq c(z_2) \) then
          Set \( z_{\text{min}} = z_1 \).
        else
          Set \( z_{\text{min}} = z_2 \).
        end if
      end if
    else
      Set \( z_{\text{min}} = z_1 \).
    end if
  else
    Set \( z_{\text{min}} = z_2 \).
  end if
end if
Set \( z_{\min} = z_0 \).

end if

if \( c(z_{\min}) \leq c(\tilde{a}) \) then

if \( c(z_{\min}) \leq c(\tilde{b}) \) then

Set \( \alpha = x_0 + z_{\min}(x_1 - x_0) \)

else

Set \( \alpha = b \).

end if

else if \( c(\tilde{a}) < c(\tilde{b}) \) then

Set \( \alpha = a \).

else

Set \( \alpha = b \).

end if

else

if \( c(\tilde{a}) < c(\tilde{b}) \) then

Set \( \alpha = a \).

else

Set \( \alpha = t \)

end if

end if

end if

A.5 LineSearch

A.5.1 Bracketing Phase

Set \( \alpha^{(0)} = 0 \)

if \( f'(0) = 0 \) then

Set \( \mu = \alpha_{\text{max}} \).

else

Set \( \mu = \min \left( \alpha_{\text{max}}, \frac{f_{\text{low}} - f(0)}{\rho f'(0)} \right) \).

end if

if \( \mu < 0 \) then

Set \( \mu = \alpha_{\text{max}} \).

end if

Set \( \alpha^{(1)} = \min(1, \mu, \alpha_1) \).

if \( \alpha^{(1)} < 10^{-14} \) then

Set \( \alpha^{(1)} = 0 \).

Terminate Line Search.

end if

for \( k = 1, 2, 3, \ldots \) do

if \( f(\alpha^{(k)}) \leq f_{\text{low}} \) then

Terminate Line Search.

end if

if \( f(\alpha^{(k)}) \geq f(\alpha^{(k-1)}) \) or \( f(\alpha^{(k)}) > f(0) + \alpha^{(k)} \rho f'(0) \) then

if \( f(\alpha^{(k)}) = f(\alpha^{(k-1)}) \) and \( \|p\| \leq 10^{-8} \) then

Terminate Line Search.

end if

Set \( \alpha^{(k)} = \alpha^{(k-1)} \) and \( \tilde{b}^{(k)} = \alpha^{(k)} \).

Set \( \tilde{k} = k \).

Go to Sectioning Phase.

end if

if \( |f'(\alpha^{(k)})| \leq -\sigma f'(0) \) then

Set \( \alpha^{(k)} = 0 \).

Terminate Line Search.

end if

end if
if \( f'(a^{(k)}) \geq 0 \) then

Set \( a^{(k)} = \alpha^{(k)} \) and \( b^{(k)} = \alpha^{(k-1)} \).
Set \( \tilde{k} = k \).
Go to Sectioning Phase.
\)

end if

if \( \mu < 2\alpha^{(k)} - \alpha^{(k-1)} \) then

Set \( \alpha^{(k+1)} = \mu \).
else

Choose \( \alpha^{(k+1)} \in [2\alpha^{(k)} - \alpha^{(k-1)}, \min(\mu, \alpha^{(k)} + \tau_1 (\alpha^{(k)} - \alpha^{(k-1})))] \) using quadratic or cubic interpolation. See 2.12.1 and 2.12.2 respectively.
end if

if \( k > 30 \) then

Set \( \alpha^{(k)} = 0 \).
Terminate Line Search

end if

end for

A.5.2 Sectioning Phase

for \( k = \tilde{k}, \tilde{k} + 1, \ldots \) do

if \( |a^{(k)} - b^{(k)}| \leq \epsilon_1 \) then

if \( f(a^{(k)}) < f(b^{(k)}) \) or \( f(a^{(k)}) = f(b^{(k)}) \land a^{(k)} > b^{(k)} \) then

Set \( \alpha^{(k)} = a^{(k)} \).
else

Set \( \alpha^{(k)} = b^{(k)} \).
end if

Terminate Line Search

end if

Choose \( \alpha^{(k)} \in [a^{(k)} + \tau_2 (b^{(k)} - a^{(k)}), b^{(k)} - \tau_3 (b^{(k)} - a^{(k)})] \) using quadratic or cubic interpolation. See 2.12.1 and 2.12.2 respectively.

if \( (a^{(k)} - \alpha^{(k)}) f'(a^{(k)}) \leq \epsilon_2 \) then

Set \( \alpha^{(k)} = a^{(k)} \).
Terminate Line Search

end if

if \( f(a^{(k)}) > f(0) + \rho a^{(k)} f'(0) \) or \( f(a^{(k)}) \geq f(a^{(k)}) \) then

Set \( a^{(k+1)} = a^{(k)} \) and \( b^{(k+1)} = a^{(k)} \).
else

if \( |f'(a^{(k)})| \leq -\sigma f'(0) \) then

Terminate Line Search.

end if

Set \( a^{(k+1)} = a^{(k)} \).
if \( (b^{(k)} - a^{(k)}) f'(a^{(k)}) \geq 0 \) then

Set \( b^{(k+1)} = a^{(k)} \).
else

Set \( b^{(k+1)} = b^{(k)} \).
end if

end if

end for

A.6 IsSolve

Set \( k = 0 \), \( stop = 0 \) and the number of consecutive zero steps, \( a_0 = 0 \).
Set \( GN_{lag} = 1 \) and \( A_H = 1 \).
Set \( x^{(-1)} = \infty \), \( f^{(-1)} = \infty \) and \( \alpha = 1 \).
Set \( \tilde{x}_i = \max(x^{(0)}_i, x_{L_i}) \) and \( x^{(0)}_i = \min(\tilde{x}_i, x_{U_i}), i = 1, 2, \ldots, n. \)
while not convergence do
if $x^{(k)}_i$ is beyond or very close to lower or upper bound, $i = 1, 2, ..., n$ then
  Move $x^{(k)}_i$ to bound.
end if

Set up working sets for variables active on lower and upper bounds respectively.
$V_L = \{i : x^{(k)}_i = x_{L,i}\}, \ V_U = \{i : x^{(k)}_i = x_{U,i}\}$
Set $nr_{act}$ equal to the number of active variables.
if any variable has been moved to bound or $k = 0$ then
  Compute $r^{(k)}$, $J^{(k)}$, $f^{(k)}$ and $g^{(k)}$.
end if

if $k = 0$ then
  Compute $H^{(k)} = J^{(k)\top}J^{(k)}$.
end if

Compute first order Lagrange multiplier estimate $\lambda$.
$\lambda_i = -g_i$ if $i \in V_U$, $\lambda_i = g_i$ if $i \in V_L$ and $\lambda_i = 0$ else.
if $nr_{act} > 0$ and $\alpha_0 < 3$ then
  if $nr_{act} < n$ then
    Release all variables $x_i$, not activated in the previous iteration, where $x_{L,i} \neq x_{U,i}$ and $\lambda_i < -b_{Tol}$.
  if $\alpha = 0$ then
    Release all variables $x_i$, inactive in the previous iteration, where $x_{L,i} \neq x_{U,i}$ and $\lambda_i < -b_{Tol}$.
  end if
  else
    Release all variables $x_i$ where $x_{L,i} \neq x_{U,i}$ and $\lambda_i < -b_{Tol}$.
  end if
end if

if variables was released then
  Update $V_L$, $V_U$ and $nr_{act}$.
else
  if for all active variables $i$ there either holds that $\lambda_i \geq -10^{-8}$ or $x_{L,i} = x_{U,i}$ then
    Check convergence criteria, see A.6.1.
  end if

  if any convergence criteria are fulfilled or $nr_{act} = n$ then
    Set $stop = 1$.
  end if
end if

Check stop criteria, see A.6.2.
if any stop criteria are fulfilled then
  Set $stop = 1$.
end if
if $stop$ then
  END ALGORITHM
end if

if $\alpha = 0$ and variables was released in the current iteration based on first order Lagrange multiplier estimate then
  Search in the negative gradient direction for the released variables.
  Set $p_{full_i} = -g_i$ if variable $i$ was released, else set $p_{full_i} = 0$.
else
  Compute search direction $p$ with chosen method, see A.6.3.
  Set $p_{full_i} = p_i$ if $i \notin V_L \cup V_U$ else set $p_{full_i} = 0$.
end if

Compute $\alpha_1$, step length estimate sent to line search routine.
if $k = 0$ then
  Set $\tilde{\alpha}_1 = 1$.
else
  Set $\tilde{\alpha}_1 = \min \left(1, -2\frac{\max(f^{(k-1)} - f^{(k)}, 10^{-8})}{\tilde{g}^T \ p = 0}\right)$, where $\tilde{g}_i = g_i : i \notin V_L \cup V_U$.  

if $\alpha_1 < 0$ then
  Change sign on $\alpha_1$, $pf_{all}$ and $p$.
end if
end if
Set $\alpha_1 = \max (0.5, \alpha_1)$.
Compute $\alpha_{max}$, the maximum step $\alpha$ such that $x + \alpha pf_{all}$ is feasible with respect to the variable bounds.
if $\alpha_{max} < 10^{-14}$ then
  Set $\alpha = 0$.
else
  Solve the line search problem $\min_{0 < \alpha < \alpha_{max}} f(x + \alpha pf_{all})$.
end if
if $\alpha < 10^{-14}$ then
  Set $\alpha_n = \alpha_n + 1$.
else
  Set $\alpha_0 = 0$.
end if
Set $x^{(k+1)} = x^{(k)} + \alpha pf_{all}$.
$f^{(k+1)}$, $g^{(k+1)}$, $r^{(k+1)}$ and $J^{(k+1)}$ was computed in the line search.
Depending on the chosen method, update the approximation of the Hessian, see A.6.4.
Set $k = k + 1$.
end while

A.6.1 Convergence criterias

- $\max_i \frac{|x_i^{(k)} - x_i^{(k-1)}|}{\max_i \{|x_i^{(k)}|, size_x\}} \leq \epsilon_\text{d}$
- $\max_{i \notin V_L \cup V_U} \left( \left| g_i^{(k)} \right| \max_i \left( \left| x_i^{(k)} \right|, size_x \right) \right) \leq \epsilon_g \max_i \left( \left| f_i^{(k)} \right|, size_f \right)$
- $f^{(k)} \leq \epsilon_{absf} size_f$
- Relative function value reduction low for $\text{LowIt}$s iterations.

A.6.2 Stop criterias

- $k \geq \text{MaxIter}$
- $f^{(k)} < f_{\text{Low}}$

A.6.3 Computation of Search Direction

Gauss-Newton or hybrid method if $GN_{flag} = 1$

Solve the overdetermined system $\tilde{J} p = -r$ with rank estimation and a subspace minimization technique either using Singular Value Decomposition or using QR-Decomposition with or without pivoting.
$\tilde{J}_{ij} = J_{ij} : j \notin V_L \cup V_U$.

Fletcher-Xu, Al-Baali-Fletcher and Huschens TSSM if $GN_{flag} = 0$

Solve $H p = -\tilde{g}$ using Singular Value Decomposition with rank estimation and a subspace minimization technique.
$H_{ij} = H_{ij} : i, j \notin V_L \cup V_U$.

A.6.4 Update Procedure

Fletcher-Xu

Set $z = \alpha pf_{all}$.
if \( f^{(k)} - f^{(k+1)} \geq 0.2f^{(k)} \) or \( \|z\| \leq \epsilon_z \) then

Set \( GN_{flag} = 1 \).
Set \( H^{(k+1)} = J^{(k+1)^T} J^{(k+1)} \).
else
Set \( GN_{flag} = 0 \).
Set \( y = J^{(k+1)^T} J^{(k+1)} z + \left( J^{(k+1)^T} - J^{(k)^T} \right) r^{(k+1)} \).
if \( z^T y < 0.01z^T \left( g^{(k+1)} - g^{(k)} \right) \) then
Set \( w = g^{(k+1)} - g^{(k)} \).
else
Set \( w = y \).
end if
if \( z^T w < 10^{-13} \) or \( z^T H^{(k)} z < 10^{-13} \) then
Set \( GN_{flag} = 1 \).
Set \( H^{(k+1)} = J^{(k+1)^T} J^{(k+1)} \).
else
Set \( H^{(k+1)} = H^{(k)} + \frac{ww^T}{z^T w} - \frac{H^{(k)} + z^T H^{(k)^T}}{z^T H^{(k)} z} \).
end if
end if

Al-Baali-Fletcher

Set \( z = \alpha p_{full} \).
if \( f^{(k)} - f^{(k+1)} \geq 0.2f^{(k)} \) or \( \|z\| \leq \epsilon_z \) then
Set \( GN_{flag} = 1 \).
Set \( H^{(k+1)} = J^{(k+1)^T} J^{(k+1)} \).
else
Set \( GN_{flag} = 0 \).
Set \( y = J^{(k+1)^T} J^{(k+1)} z + \left( J^{(k+1)^T} - J^{(k)^T} \right) r^{(k+1)} \).
if \( z^T y < 0.2z^T H^{(k)} z \) then
Set \( w = \frac{0.8z^T H^{(k)} z}{z^T H^{(k)} z - z^T y} y + \left( 1 - \frac{0.8z^T H^{(k)} z}{z^T H^{(k)} z - z^T y} \right) H^{(k)} z \).
else
Set \( w = y \).
end if
if \( z^T w < 10^{-10} \) or \( z^T H^{(k)} z < 10^{-10} \) then
Set \( GN_{flag} = 1 \).
Set \( H^{(k+1)} = J^{(k+1)^T} J^{(k+1)} \).
else
Set \( H^{(k+1)} = H^{(k)} + \frac{ww^T}{z^T w} - \frac{H^{(k)} + z^T H^{(k)^T}}{z^T H^{(k)} z} \).
end if
end if

Huschens TSSM

Set:
\( GN_{flag} = 0 \),
\( z = p_{full} \),
\( y^p = \left( J^{(k+1)} - J^{(k)} \right)^T r^{(k+1)} \),
\( y = J^{(k+1)^T} J^{(k+1)} z + \|r^{(k+1)}\| y^p \) and
\( B_s = J^{(k+1)^T} J^{(k+1)} + \|r^{(k+1)}\| A^{(k)} \).
if \( z^T B_s z > 0 \) and \( y^T z > 0 \) then
Set \( v = y + \sqrt{\frac{y^T z}{z^T B_s z}} B_s z \).
else
Set \( v = y \).
end if
Set $A^{(k+1)}_H = A^{(k)}_H + \frac{\left(y^t - A^{(k)}_H z\right) v^t + v\left(y^t - A^{(k)}_H z\right)^t}{v^t z} - \frac{\left(y^t - A^{(k)}_H z\right)^t z v^t}{\left(v^t z\right)^2}$.
Set $H^{(k+1)} = H^{(k)} + \|r^{(k+1)}\|_2 A^{(k+1)}_H$.

A.7 ncSolve

Set $k = 0$, $stop = 0$ and the number of consecutive zero steps, $a_0 = 0$.
Set $\beta = 0$, $e_{\text{step}} = 1$ and $e_{\text{restart}} = 1$ if restart technique shall be used else set $e_{\text{restart}} = 0$.
Set $x^{(-1)} = x^{0}$, $f^{(-1)} = \infty$, $B^{(0)} = I$ and $\alpha = 1$.
Set $\bar{x}_i = \max(x_i^{(0)}, x_{L_i})$ and $x_i^{(0)} = \min(\bar{x}_i, x_{U_i})$, $i = 1, 2, ..., n$.

while not convergence do
  if $x_i^{(k)}$ is beyond or very close to lower or upper bound, $i = 1, 2, ..., n$ then
    Move $x_i^{(k)}$ to bound.
  end if
Set up working sets for variables active on lower and upper bounds respectively.
$V_L = \{i : x_i^{(k)} = x_{L_i}\}$, $V_U = \{i : x_i^{(k)} = x_{U_i}\}$
Set $n_{r_{\text{act}}}$ equal to the number of active variables.
if any variable has been moved to bound or $k = 0$ then
  Compute $f^{(k)}$ and $g^{(k)}$.
end if
Compute $H^{(k)}$.
if $n_{r_{\text{act}}} > 0$ and $a_0 < 3$ then
  Compute first order Lagrange multiplier estimate $\lambda$.
  $\lambda_i = -g_i$ if $i \in V_U$, $\lambda_i = g_i$ if $i \in V_L$ and $\lambda_i = 0$ else.
if $n_{r_{\text{act}}} < n$ then
  Release all variables $x_i$, not activated in the previous iteration, where $x_{L_i} \neq x_{U_i}$ and $\lambda_i < -b_{Tol}$.
  if $a = 0$ then
    Release all variables $x_i$, inactive in the previous iteration, where $x_{L_i} \neq x_{U_i}$ and $\lambda_i < -b_{Tol}$.
  end if
else
  Release all variables $x_i$ where $x_{L_i} \neq x_{U_i}$ and $\lambda_i < -b_{Tol}$.
end if
if variables was released then
  Update $V_L$, $V_U$ and $n_{r_{\text{act}}}$.
else
  Check convergence criteria, see A.7.1.
if any convergence criteria are fulfilled then
  Set $stop = 1$.
end if
if $stop$ and $n_{r_{\text{act}}} \in (0, n)$ and $a_0 < 3$ then
  Compute the search direction, $p$, for the free variables by solving $\tilde{H}p = -\tilde{g}$.
  where $\tilde{H}_{ij} = H_{ij}$ : $i, j \notin V_L \cup V_U$ and $\tilde{g}_i = g_i$ : $i \notin V_L \cup V_U$.
  Compute $\alpha_{\text{max}}$, the maximum step $\alpha$ such that $x + \alpha p_{\text{full}}$ is feasible with respect to the variable bounds, where $p_{\text{full}} = p_i$ if $i \notin V_L \cup V_U$ else $p_{\text{full}} = 0$.
  Compute second order Lagrange multiplier estimate, $\eta$, for the active variables.
  $\eta = \tilde{\lambda} + \tilde{H} \alpha_{\text{max}} p$.
  where $\tilde{\lambda} = \lambda_i$ : $i \in V_L \cup V_U$ and $\tilde{H} = H_{ij}$ : $i \in V_L \cup V_U, j \notin V_L \cup V_U$.
  if $\eta_i < -b_{Tol}$ and $x_{L_i} \neq x_{U_i}$ then
    Release the corresponding variable $x_i$, set $stop = 0$ and update $V_L$, $V_U$ and $n_{r_{\text{act}}}$.
  end if
end if
if $stop$ and $n_{r_{\text{act}}} < n$ then
  Check if $x$ is a saddle or a minimum point.
  Compute the eigenvalues of $\tilde{H}$ and let $\xi$ be the smallest eigenvalue.
if $\xi < -10^{-12}$ and no eigenvector search direction was used in previous iteration then
  Set $\text{stop} = 0$ and set $p$ equal to the eigenvector corresponding to $\xi$.
  Check if $p$ is a descent direction i.e. if $\hat{g}^T p < 0$.
  If $p$ is not a descent direction change sign on $p$.
end if
end if

Check stop criterias, see A.7.2.
if any stop criteria are fulfilled then
  Set $\text{stop} = 1$.
end if
if $\text{stop}$ then
  END ALGORITHM
end if
if $\alpha = 0$ and variables was released in the current iteration based on first order Lagrange multiplier estimate then
  Search in the negative gradient direction for the released variables.
  Set $p_{full i} = -g_i$ if variable $i$ was released, else set $p_{full i} = 0$.
else
  repeat
    Compute search direction $p$ with chosen method if not computed before in this iteration, see A.7.3.
    if $nr_{act} > 0$ then
      Compute second order Lagrange multiplier estimate $\eta$.
      if $\eta_i < -b_{tol}$ and $x_i \notin x_L$, $x_U$, then
        Release the corresponding variable $x_i$ and update $V_L$, $V_U$ and $nr_{act}$.
      end if
    end if
  until no variable is released.
  Set $p_{full i} = p_i$ if $i \notin V_L \cup V_U$, else set $p_{full i} = 0$.
end if

Compute $\alpha_1$, step length estimate sent to line search routine.
if $k > 0$ and $\hat{g}^T p \neq 0$ then
  Set $\tilde{\alpha}_1 = \min \left( 1, 2^{\frac{f(k-1) - f(k) - 10\epsilon}{\hat{g}^T p}} \right)$.
else
  Set $\tilde{\alpha}_1 = 1$.
end if

Set $\alpha_1 = \max(0.5, \tilde{\alpha}_1)$.
if $p$ is a descent direction then
  Compute $\alpha_{max}$.
  if $\alpha_{max} < 10^{-13}$ then
    Set $\alpha = 0$
  else
    Solve the line search problem $\min_{0 < \alpha \leq \alpha_{max}} f(x + \alpha p_{full})$.
  end if
else
  Compute the eigenvalues and their corresponding eigenvectors of $\hat{H}$.
  Let $P$ be the set of search directions containing all eigenvectors corresponding to negative eigenvalues, and
  the negative search direction $-p$.
  for all $p \in P$ and in order of most descent do
    Set $p_{full i} = p_i$ if $i \notin V_L \cup V_U$, else set $p_{full i} = 0$.
    Compute $\alpha_{max}$.
    if $\alpha_{max} > 10^{-7}$ then
      Set $\alpha_1 = 1$.
    end if
    Solve the line search problem $\min_{0 < \alpha \leq \alpha_{max}} f(x + \alpha p_{full})$.
  end for
Set \( \alpha = 0 \).

end if

if \( \alpha > 10^{-6} \) then

Accept the search direction \( pf_{all} \) and the step length \( \alpha \).

break for

end if

end for

end if

if \( \alpha < 10^{-14} \) then

Set \( \alpha_0 = \alpha_0 + 1 \).

else

Set \( \alpha_0 = 0 \).

end if

Set \( x^{(k+1)} = x^{(k)} + \alpha pf_{all} \).

\( f^{(k+1)} \) and \( q^{(k+1)} \) was computed in the line search.

Depending on the chosen method, update the approximation of the Hessian, the approximation of the inverse Hessian or \( \beta \), see A.7.4.

Set \( k = k + 1 \).

end while

A.7.1 Convergence criteria

- \( \max_i \frac{|x_i^{(k)} - x_i^{(k-1)}|}{\max(|x_i^{(k)}|,\text{size}_x)} \leq \epsilon_x \)

- \( \max_{i \in Y \cup V_t} \left( \left| g_i^{(k)} \right| \max \left( \left| x_i^{(k)} \right|, \text{size}_x \right) \right) \leq \epsilon_g \max \left( \left| f^{(k)} \right|, \text{size}_f \right) \)

- Relative function value reduction low for LowIts iterations.

A.7.2 Stop criteria

- \( k \geq \text{MaxIter} \)

- \( f^{(k)} \leq f_{Low} \)

A.7.3 Computation of Search Direction

Newton

Solve \( Hp = -\tilde{g} \) either using Singular Value Decomposition with rank estimation and a subspace minimization technique or using LU-Decomposition with or without pivoting.

Safeguarded quasi-Newton DFP or BFGS

Solve \( \hat{B}p = -\tilde{g} \) either using Singular Value Decomposition with rank estimation and a subspace minimization technique or using LU-Decomposition with or without pivoting.

Safeguarded quasi-Newton inverse DFP or BFGS

Set \( p = -\hat{B} \tilde{g} \).

Fletcher-Reeves, Polak-Ribiere and Fletcher conjugate descent CG

if \( cg_{restart} \) and \( cg_{step} = n + 1 \) then

Set \( cg_{step} = 1 \) and \( \beta = 0 \).

end if

if \( k=0 \) then

Set \( p = -\tilde{g} \).

else
Set $p = -\tilde{g} + \beta p.$

end if
Set $cg_{step} = cg_{step} + 1.$

A.7.4 Update Procedure

Safeguarded quasi-Newton BFGS
Set $z = \alpha p$.
if $\|z\| > \epsilon_x$ then
Set $y = \tilde{g}^{(k+1)} - \tilde{g}^{(k)}$.
if $z^T y < 0.2 \cdot z^T B z$ then
Set $w = \frac{0.8 z^T B z}{z^T B z - z^T y} y + \left(1 - \frac{0.8 z^T B z}{z^T B z - z^T y}\right) B z$.
else
Set $w = y$.
end if
if $z^T w = 0$ then
if $z^T B z \neq 0$ then
Set $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} - \frac{B z z^T B}{z^T B z}$.
end if
else if $z^T \tilde{B} z = 0$ then
Set $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} + \frac{w w^T}{z^T w}$.
else
Set $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} + \frac{w w^T}{z^T w} - \frac{B z z^T B}{z^T B z}$.
end if
end if
Safeguarded quasi-Newton inverse BFGS
Set $z = \alpha p$.
if $\|z\| > \epsilon_x$ then
Set $y = \tilde{g}^{(k+1)} - \tilde{g}^{(k)}$.
if $z^T y \neq 0$ then
Set $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} + \left(1 + \frac{y^T \tilde{B} y}{z^T y}\right) \frac{z z^T y}{z^T y} - \frac{z y^T B y + z z^T B}{z^T y}$.
end if
end if
Safeguarded quasi-Newton inverse DFP
Set $z = \alpha p$.
if $\|z\| > \epsilon_x$ then
Set $y = \tilde{g}^{(k+1)} - \tilde{g}^{(k)}$.
if $z^T y < 0.2 \cdot y^T \tilde{B} y$ then
Set $w = \frac{0.8 y^T \tilde{B} y}{y^T \tilde{B} y - z^T y} z + \left(1 - \frac{0.8 y^T \tilde{B} y}{y^T \tilde{B} y - z^T y}\right) \tilde{B} y$.
else
Set $w = z$.
end if
if $z^T w = 0$ then
if $y^T \tilde{B} y \neq 0$ then
Set $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} - \frac{\tilde{B} y y^T \tilde{B}}{y^T \tilde{B} y}$.
end if
else if $y^T \tilde{B} y = 0$ then
Set $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} + \frac{w w^T}{y^T w}$.
else
Set $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} + \frac{w w^T}{y^T w} - \frac{B y y^T B}{y^T B y}$.
end if
end if
Safeguarded quasi-Newton DFP

Set $z = \alpha p$.

if $\|z\| > \epsilon_z$ then
  Set $y = \hat{g}^{(k+1)} - \hat{g}^{(k)}$.
  if $z^T y \neq 0$ then
    Set $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} + \left(1 + \frac{z^T \tilde{B} z}{z^T y}\right) \frac{yy^T}{z^T y} - \frac{yz^T \tilde{B} z y^T}{z^T y}$.
  end if
end if

Fletcher-Reeves CG

Set $\beta = \frac{g^{(k+1)^T} g^{(k+1)}}{g^{(k)^T} g^{(k)}}$.

Polak-Ribiere CG

Set $\beta = \frac{(g^{(k+1)^T} - g^{(k)^T}) g^{(k+1)}}{g^{(k)^T} g^{(k)}}$.

Fletcher conjugate descent CG

Set $\beta = -\frac{g^{(k+1)^T} g^{(k+1)}}{g^{(k)^T} g^{(k)}}$. 
B Description of Algorithms in OPERA TB

B.1 akarmark

if $x_{0}$ is not given or $x_{0,j} = 0$ for any $j = 1, 2, ..., n$ then

Set $x^{(0)} = (\frac{1}{n}, ..., \frac{1}{n})$.

if $b - Ax^{(0)} \neq 0$ then

Set $A = (A - b - Ax^{(0)})$, $x^{(0)} = (x^{(0)T} 1)^T$ and $c = (c^T 2 \sum_{j=1}^{n} |c_j|)^T$.

Set $n = n + 1$

end if

else

Set $x^{(0)} = x_{0}$.

end if

Compute $L = |1 + \sum_{i} \sum_{j} \log_{2}(1 + |a_{ij}|)|$.

Set $tol = 2^{-12}$.

Compute $\alpha = \frac{n-1}{5n}$ and $r = \frac{1}{\sqrt{n(n-1)}}$.

Set $q = 0.97$ and $\mu = \min(10^{-12}, \frac{tol}{\alpha})$.

for $k = 0, 1, ..., k_{max} - 1$ do

Set $D = \text{diag}\{x_{1}^{(k)}, ..., x_{n}^{(k)}\}$, $\hat{c} = Dc$, and $\hat{A} = AD$.

Compute dual estimate $y^{(k)}$ by solving $\hat{A}y = \hat{c} - (\mu, ..., \mu)^T$.

Compute reduced cost vector $\tilde{R} = c - \frac{\mu}{x_{1}^{(k)}}(..., \frac{\mu}{x_{n}^{(k)}})^T - \hat{A}^Ty$, projected gradient vector $g_{p} = \hat{c} - (\mu, ..., \mu)^T - \hat{A}^Ty$ and search direction $d = -Dg_{p}$.

if $R \geq -10^{-10}$ and it either holds that $c^Tx^{(k)} - b^Ty^{(k)} < tol$ or that the function value reduction is less than $10^{-14}$ then

Let $W$ be a index set of the variables active on their lower bounds, i.e.

$W = \{ j : x_{j}^{(k)} \leq 10^{-12} \}$.

Set $r$ equal to the rank of $A$ plus the number of elements in $W$.

while $r < n$ do

Let $Z$ be a basis for the null space of the matrix $(A_{e_{i}, i \in W})$, where $e_{i}$ is the $i$:th unit row vector.

Let $d_{i} = Z_{i}$ if $i \notin W$ and $d_{i} = 0$ if $i \in W$.

if $c^Td \geq 0$ then

Set $d = -d$.

end if

Set $\alpha = \min_{i \in W, d_{i} < 0} \frac{-x_{i}^{(k)}}{d_{i}}$.

if $\{ i : i \notin W, d_{i} < 0 \} = \emptyset$ then

STOP, purification failed.

end if

Set $x^{(k+1)} = x^{(k)} + ad$.

Update $W$ and set $r$ equal to the rank of $B$ plus the number of elements in $W$.

end while

STOP, purification succeeded.

end if

Set $\lambda_{max}^{new} = \max_{j} \frac{d_{j}}{x_{j}^{(k)}}$.

Set $\alpha = \min\left(\frac{\mu}{\lambda_{max}^{new}}, \frac{(1-q)^2}{\mu}\right)$.

Set $x^{(k+1)} = x^{(k)} + ad$.

end for

B.2 cutplane

Set $m_{i} = m - m_{eq}$.

if $m_{i} > 0$ then
Add \( m_i \) slack variables to create a problem on standard form.

Update \( A, c \) and \( n \).

\textbf{end if}

Set \( \epsilon_1 = 10^{-12} \).

\textbf{if} \( B \neq 0 \) is not given \textbf{then}

Call Phase I simplex routine \textit{Phase1Simplex} to get the solution \( x \) and \( B \).

\textbf{if} no feasible Phase I solution were found \textbf{then}

STOP.

\textbf{end if}

\textbf{end if}

Set \( B_{idx} = \{ i : B_i = 1 \} \), the set of basic variables.

Set \( N_{idx} = \{ i : B_i = 0 \} \), the set of nonbasic variables.

\textbf{if} \( x_0 \) is not given \textbf{then}

Set \( x_j = 0, j \notin B_{idx} \).

Solve \( A_B x_B = b \), where \( A_B = A_{ij}, j \in B_{idx} \) and \( x_B = x_j, j \in B_{idx} \).

\textbf{else}

Set \( x = x_0 \).

\textbf{end if}

Call Phase II simplex routine \textit{Phase2Simplex} to get the solution \( x, B \) and \( y \).

\textbf{for} \( k = 0, 1, ..., k_{max} \) \textbf{do}

Update \( B_{idx} \) and \( N_{idx} \).

Set \( x_{idx} = x_j, j \leq n_I \wedge j \in B_{idx} \).

Set \( x_{ij} = \lfloor x_{idx} + \epsilon_1 \rfloor \).

Set \( x_{ri} = \max(0, x_{idx} - x_{ij}) \).

Determine the variable with its fractional part closest to 0.5 i.e. set \( i = \arg\min(|x_{ri} - 0.5|) \).

STOP, convergence.

\textbf{end if}

Set \( c_N = A_{-1} A_N \), where \( A_{-1} \) is the \( i \)-th row in \( A_{-1} \) and \( A_N = A_{ij}, j \in N_{idx} \).

Set \( a_j = 0 \) if \( j \notin N_{idx} \) else set \( a_j = \max(0, c_{N_i} - [c_{N_i} + \epsilon_1]) \).

Set \( A = \left( \begin{array}{c} A \ 0^{m \times 1} \\ a \ -1 \end{array} \right), b = (b^T \ x_{ri} )^T, c = (c^T \ 0)^T \) and \( x = (x^T \ -x_{ri})^T \).

Set \( B = (B \ 1) \), \( n = n + 1 \) and \( m = m + 1 \).

Call dual simplex routine \textit{lpdual} to get the solution \( x, B \) and \( y \).

\textbf{if} the dual simplex routine failed \textbf{then}

Use Phase I and Phase II simplex routines.

\textbf{end if}

\textbf{end for}

\subsection*{B.3 dijkstra}

Set \( n \) equal to the number of nodes.

Set \( B = \{ s \} \) and \( T = N \setminus \{ s \} \), where \( N \) is the set of all nodes.

Set \( \text{dist}(s) = 0 \) and \( \text{pred}(s) = 0 \).

\textbf{for} \( j = 1, 2, ..., n \) \textbf{do}

\textbf{if} \( (s, j) \in Z \) \textbf{then}

Set \( \text{dist}(j) = c_{sj} \), where \( c_{ij} \) means the cost of arc \((i, j)\).

Set \( \text{pred}(j) = s \).

\textbf{else}

Set \( \text{dist}(j) = \infty \).

\textbf{end if}

\textbf{end for}

\textbf{while} \( B \neq N \) \textbf{do}

Let \( i \in T \) be a node for which \( \text{dist}(i) = \min\{ \text{dist}(j) : j \in T \} \).

Set \( B = B \cup \{ i \} \) and \( T = T \setminus \{ i \} \).

\textbf{for} each \( j : (i, j) \in Z \) \textbf{do}

\textbf{end for}

\textbf{end while}

Let \( i \in T \) be a node for which \( \text{dist}(i) = \min\{ \text{dist}(j) : j \in T \} \).

Set \( B = B \cup \{ i \} \) and \( T = T \setminus \{ i \} \).

\textbf{for} each \( j : (i, j) \in Z \) \textbf{do}

\textbf{end for}

\textbf{end while}

Let \( i \in T \) be a node for which \( \text{dist}(i) = \min\{ \text{dist}(j) : j \in T \} \).

Set \( B = B \cup \{ i \} \) and \( T = T \setminus \{ i \} \).

\textbf{for} each \( j : (i, j) \in Z \) \textbf{do}

\textbf{end for}

\textbf{end while}

Let \( i \in T \) be a node for which \( \text{dist}(i) = \min\{ \text{dist}(j) : j \in T \} \).

Set \( B = B \cup \{ i \} \) and \( T = T \setminus \{ i \} \).

\textbf{for} each \( j : (i, j) \in Z \) \textbf{do}

\textbf{end for}

\textbf{end while}
if $dist(j) > dist(i) + c_{ij}$ then
   Set $dist(j) = dist(i) + c_{ij}$.
   Set $pred(j) = i$.
end if
end for
end while

**B.4 dpinvent**

Set $x_{\text{LOW}} = \min_i x_{l_i}$ and $x_{\text{UPP}} = \max_i x_{u_i}$.
Set $s = 1 + x_{\text{UPP}} - x_{\text{LOW}}$.
Set $U_{ij} = 0$, $i = 1,2,\ldots, s$, $j = 1,2,\ldots, n$.
Set $f_1 = \infty$ and $f_{P_i} = \infty$, $i = 1,2,\ldots, s$.
Set $f_{P_i + s - x_{\text{LOW}}} = 0$.
for $t = 1,2,\ldots, n$ do
   Set $u_{\text{LOW}} = u_{L_i}$ and $u_{\text{UPP}} = u_{U_i}$.
   if $t = n$ then
      Set $x_{\text{LOW}} = x_{\text{UPP}} = x_{\text{LAST}}$.
   else
      Set $x_{\text{LOW}} = x_{L_i}$ and $x_{\text{UPP}} = x_{U_i}$.
   end if
   for $i = x_{\text{LOW}}, x_{\text{LOW}} + 1,\ldots, x_{\text{UPP}}$ do
      Set $u_{min} = 0$ and $f_{min} = \infty$.
      for $j = u_{\text{LOW}}, u_{\text{LOW}} + 1,\ldots, u_{\text{UPP}}$ do
         Set $x = i - j + d_t$.
         if $x_{\text{LOW}} \leq x \leq x_{\text{UPP}}$ then
            Set $f_u = P_s(j > 0) + P_j I_s(i > 0) + I_t i + f_{P_i + s - x_{\text{LOW}}}$.
            if $f_u < f_{min}$ then
               Set $u_{min} = j$ and $f_{min} = f_u$.
            end if
         end if
      end for
      Set $f_{1+i-x_{\text{LOW}}} = f_{min}$ and $U_{1+i-x_{\text{LOW}},t} = u_{min}$.
   end for
   Set $f_P = f$.
end for
Set $x = x_{\text{LAST}}$.
for $t = n, n-1,\ldots, 1$ do
   Set $u_t = U_{1+x-x_{\text{LOW}},t}$.
   Set $x = x - u_t + d_t$.
end for
Set $f_{\text{opt}} = f_{1+x_{\text{LAST}}-x_{\text{LOW}}}$.

**B.5 dpknap**

if $u_U$ is not given then
   Set $u_{U_j} = \lfloor \frac{b}{A_i} \rfloor$, $j = 1,2,\ldots, n$.
else
   Set $u_U = u_U$.
end if
Set $U_{ij} = 0$, $i = 1,2,\ldots, b+1$, $j = 1,2,\ldots, n$.
Set $f_1 = 0$ and $f_{P_i} = 0$, $i = 1,2,\ldots, b+1$.
for $i = 1,2,\ldots, n$ do
   for $k = 1,2,\ldots, b+1$ do
      Set $u_{\text{max}} = 0$ and $f_{\text{max}} = f_{P_k}$.
      for $j = 1,2,\ldots, \min(u_{U_i}, \lfloor \frac{b}{A_i} \rfloor)$ do
         
      end for
   end for
end for
Set \( x = (k - 1) - A_{i,j} \).
if \( x < 0 \) then
    break for
else
    if \( c_{i,j} + f_{P,\lambda,s} > f_{\max} \) then
        Set \( u_{\max} = j \) and \( f_{\max} = c_{i,j} + f_{P,\lambda,s} \).
    end if
end if
end for
Set \( f_k = f_{\max} \) and \( U_{ki} = u_{\max} \).
end for
Set \( x = b + 1 \).
for \( k = n, n - 1, \ldots, 1 \) do
    Set \( u_k = U_{x,k} \).
    Set \( x = x - A_k U_{x,k} \).
end for
Set \( f_{\text{opt}} = f_{b+1} \).

B.6 \hspace{1em} \text{gsearch}

Set \( \text{pred}(i) = 0 \) and \( \text{mark}(i) = 0 \), \( i = 1, 2, \ldots, m \).
Set \( \text{pred}(s) = -1 \) and \( \text{mark}(s) = 1 \).
Set \( \text{LIST} = \{ s \} \).
while \( \text{LIST} \neq \emptyset \) do
    Set \( i \) equal to the first element in \( \text{LIST} \).
    if there is an arc from node \( i \) to node \( j \) and \( \text{mark}(j) = 0 \) then
        Set \( \text{mark}(j) = 1 \) and \( \text{pred}(j) = i \).
        Put \( j \) first in \( \text{LIST} \).
    else
        Delete the first element in \( \text{LIST} \).
    end if
end while

B.7 \hspace{1em} \text{gsearchq}

Set \( \text{pred}(i) = 0 \) and \( \text{mark}(i) = 0 \), \( i = 1, 2, \ldots, m \).
Set \( \text{pred}(s) = -1 \) and \( \text{mark}(s) = 1 \).
Set \( \text{LIST} = \{ s \} \).
while \( \text{LIST} \neq \emptyset \) do
    Set \( i \) equal to the first element in \( \text{LIST} \).
    Delete the first element in \( \text{LIST} \).
    for all arcs \( (i, j) \) outgoing from node \( i \) do
        if \( \text{mark}(j) = 0 \) then
            Set \( \text{mark}(j) = 1 \) and \( \text{pred}(j) = i \).
            Put \( j \) at the end of \( \text{LIST} \).
        end if
    end for
end while

B.8 \hspace{1em} \text{karmark}

Compute \( L = |1 + \log_2(1 + \max_j |c_{ij}|)\log_2(1 + m) + \sum_i \sum_j \log_2(1 + |a_{ij}|)| \).
Set \( tol = \max(2^{-L}, 10^{-0}) \).
Compute $\alpha = \frac{n-1}{3n}$ and $r = \frac{1}{\sqrt{n(n-1)}}$.

Set $x^{(0)} = (\frac{1}{n}, \ldots, \frac{1}{n})^T$.

Set $k = 0$.

while $c^T x^{(k)} > tol$ and $k < k_{\text{max}}$ do

Set $D = \text{diag}\{x_1^{(k)}, \ldots, x_n^{(k)}\}$, $\hat{x}^{(0)} = (\frac{1}{n}, \ldots, \frac{1}{n})$, $\hat{c} = Dc$ and $B = (A_D)$.

Compute $d = (B^T B B^T)^{-1} B - I) \hat{c}$.

if Goldfarb/Todd choice of update then
  
  Set $\alpha = 0.99$.

  Compute $\hat{x} = \hat{x}^{(0)} + \alpha \frac{d}{\|d\|}$.

else if Bazaar choice of update then
  
  Compute $\hat{x} = \hat{x}^{(0)} + \alpha r \frac{d}{\|d\|}$.

end if

Set $x^{(k+1)} = \frac{\hat{x}}{\|\hat{x}\|}$.

Set $k = k + 1$.

end while

Let $W$ be a index set of the variables active on their lower bounds, i.e. $W = \{ j : x_j^{(k)} \leq 10^{-12} \}$. Set $B = (i \bar{T})$ and set $r$ equal to the rank of $B$ plus the number of elements in $W$.

while $r < n$ do

Let $Z$ be a basis for the null space of the matrix $(e_i \bar{B})$, where $e_i$ is the $i$:th unit row vector.

Let $d_i = Z_i$ if $i \notin W$ and $d_i = 0$ if $i \in W$.

if $c^T d \geq 0$ then
  
  Set $d = -d$.

end if

Set $\alpha = \min_{i \in W, d_i < 0} \frac{x_i^{(k)}}{d_i}$.

Set $x^{(k+1)} = x^{(k)} + \alpha d$.

Update $W$ and set $r$ equal to the rank of $B$ plus the number of elements in $W$.

end while

**B.9 ksrelax**

Set $x_i = 0$, $j = 1, 2, \ldots, n$ and $u_i = 0$, $i = 1, 2, \ldots, m - 1$.

Set $\bar{A}_{ij} = A_{ij}$, $j = 1, 2, \ldots, n$ and $\bar{b} = b_r$.

Set $\bar{A}_{ij} = A_{ij}$ and $\bar{b}_i = b_i$, $i \notin \{1, 2, \ldots, m\} - \{r\}$, $j = 1, 2, \ldots, n$.

Set $\lambda = 2$, $\text{fail} = 0$, $f_P = 0$, $f_{D_{\text{uns}}} = \infty$ and $x_P = x$.

for $k = 1, 2, \ldots, k_{\text{max}}$ do

if $c^T x \leq -\infty$ then
  
  STOP, convergence.

end if

Set $\bar{c} = \bar{A}^T u$.

Call the knapsack problem solver $dpknap$ with the parameters $\bar{A}$, $\bar{b}$, $\bar{c}$ and $x_U$ to get the solution $x$ and $f_D$.

Set $f_D = f_D + u^T b$ and compute the subgradient $\bar{g} = b - \bar{A} x$.

if $\bar{g}_i > 0$, $i = 1, 2, \ldots, m - 1$ and $c^T x > f_P$ then

Set $f_P = c^T x$ and $x_P = x$.

Set $\text{fail} = 0$ and $\lambda = 2$.

end if

if $f_D < f_{D_{\text{uns}}}$ then
  
  Set $\text{fail} = 0$.

else

  Set $\text{fail} = \text{fail} + 1$.

end if

Set $f_{D_{\text{uns}}} = f_D$.

if $\text{fail} > 0$ then

  Set $\lambda = \frac{1}{2} \lambda$ and $\text{fail} = 0$.

end if
Set $s_{SO} = \hat{y}^T \hat{y} + (\hat{b} - \hat{A}x)^2$.
if $\alpha \leq 0$ or $s_{SO} \leq 10^{-14}$ then
STOP, convergence.
end if
Set $\alpha = \frac{\alpha}{s_{SO}}$ and $u_i = \max(0, u_i - \alpha \hat{y}_i)$, $i = 1, 2, ..., m - 1$.
end for

B.10 labelcor
Set $\text{dist}(j) = \infty$ for each $j \in N \setminus \{s\}$, where $N$ is the set of all nodes.
Set $\text{dist}(s) = 0$ and $\text{pred}(s) = 0$.
repeat
for all arcs $(i, j) \in Z$ do
if $\text{dist}(j) > \text{dist}(i) + c_{ij}$ then
Set $\text{dist}(j) = \text{dist}(i) + c_{ij}$.
Set $\text{pred}(j) = i$
end if
end for
until no changes in $\text{dist}$ are made

B.11 lp dual
Set $m_i = m - m_{eq}$.
if $m_i > 0$ then
Add $m_i$ slack variables to create a problem on standard form.
Update $A$, $c$ and $n$.
end if
if $B.0$ is given then
Set $B^{(0)} = \{i : B.0_i = 1\}$, the set of basic variables.
Set $N^{(0)} = \{i : B.0_i = 0\}$, the set of nonbasic variables.
else
Set $B^{(0)} = \{n - m + 1, n - m + 2, ..., n\}$.
Set $N^{(0)} = \{1, 2, ..., n - m\}$.
end if
if $x.0$ is given then
Set $x = x.0$.
else
Set $x_j = 0 : j \in N$.
Solve $A_Bx_B = b$, where $A_B = A_{ij} : j \in B$ and $x_B = x_j : j \in B$.
end if
if $y.0$ is given then
Set $y = y.0$.
else
Compute initial shadow prices $y$ by solving $A_By = c_B$, where $c_B = c_j : j \in B$.
end if
Compute initial reduced costs $\hat{c}_N = c_N - A_N^Ty$, where $A_N = A_{ij} : j \in N$ and $c_N = c_j : j \in N$.
if $\hat{c}_{N_i} < 0$ for any $i = 1, 2, ..., n - m$ then
if $\hat{c}_{N_i} < -10^{-13}$ for any $i = 1, 2, ..., n - m$ then
STOP, initial shadow prices $y$ is not dual feasible.
else
Set $\hat{c}_{N_i} = 0$ for all $i$ such that $-10^{-13} < \hat{c}_{N_i} < 0$.
end if
end if
for $k = 1, 2, ..., k_{\text{max}}$ do
Compute the objective function value $\hat{f} = b^Ty$,
if $x_i > -10^{-19}$ for all $i = 1, 2, ..., n$ then
if $x_i < 0$ for any $i = 1, 2, ..., n$ then
  Set $x_i = 0$ for all $i$ such that $x_i < 0$.
end if
STOP, convergence.
end if
Choose the variable $x_p$ to exclude from the basis either using Bland's rule or Minimal Reduced Cost rule.
Solve $A_p^T u = e_p$, where $B_p = p$.
Compute $v = A_N^T u$.
if $v_j \geq 0$ for all $j = 1, 2, ..., n - m$ then
  STOP, infeasible dual problem.
end if
Determine nonbasic variable $x_q$ to enter the base. Choose
\[
-\hat{c}_{N_i} \over v_q = \min \left\{ -\hat{c}_{N_i} \over v_j : v_j < -10^{-10}, j = 1, 2, ..., n - m \right\} \overset{\text{def}}{=} \gamma,
\]
where $q = N_q$.
if $\gamma = \emptyset$ then
  Choose
  \[
  -\hat{c}_{N_i} \over v_q = \min \left\{ -\hat{c}_{N_i} \over v_j : v_j < 0, j = 1, 2, ..., n - m \right\} \overset{\text{def}}{=} \gamma.
  \]
end if
if $\gamma = \emptyset$ or $\gamma > 10^5$ then
  STOP, numerical difficulties.
end if
Set $\hat{c}_N = \hat{c}_N + \gamma v$, $\hat{c}_p = \gamma$, $\hat{c}_q = 0$ and $y = y - \gamma u$.
Compute primal search direction $d$ by solving $A_B d = -a_q$, where $a_q$ is the $q$:th column in $A$.
Set $x_q = \alpha = \hat{c}_q \over v_q$, $x_B = x_B + \alpha d$ and $x_p = 0$.
Set $B = B \cup \{q\} \backslash \{p\}$ and $N = N \cup \{p\} \backslash \{q\}$.
if $\alpha > 10^5$ then
  Numerical difficulties, recompute $x$ by setting $x_N = 0$ and solving $A_B x_B = b$.
end if
end for

B.12 Ipkarma

Set $k = m + n + 2$ and $l = 2(m + n) + 2$.
Set $M = 3 \left| \sum_{j=1}^{n} c_j + \sum_{i=1}^{m} \left| b_i \right| + 2 \max(c_j, \left| b_i \right|) \right|$
Set $\hat{B} = \begin{pmatrix}
c^T & -b^T & 0^{1 \times k} \\
A & 0^{m \times m} & I^{m \times m} & 0^{m \times n+1} & -b \\
0^{n \times n} & A^T & 0^{n \times m} & -I^{n \times n} & 0^{n \times 1} & -c \\
1^{1 \times l-1} & -M & 
\end{pmatrix}$
Set $v_i = \sum_{j=1}^{l} B_{ij}$, $i = 1, 2, ..., k$.
Set $B = ( \hat{B} \ - v \ )$ and $d = \begin{pmatrix} 0^{1 \times l} & 1 \end{pmatrix}^T$.
Call propark with constraint matrix $B$ and cost vector $d$ to get the solution $\hat{x}$.
Set $x = (M + 1)\hat{x}_i$, $j = 1, 2, ..., n$
Set $y = (M + 1)\hat{x}_j$, $j = n + 1, ..., n + m$. 
B.13  **lpsimp1**

Solve the LP problem

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

with \( \tilde{A} = (A \ I_{m \times m}^T) \), \( \tilde{c} = (0^n \ I_{m_{eq}} \ 0^{m-m_{eq}})^T \), \( x.0 = (0^n \ b^T)^T \) and

\[
B.0 = (0^n \ 1^m)^T.
\]

if \( f(\tilde{x}) < 10^{-10} \) then

if there are no artificial variable left in the base then

Set \( x \) and \( B \) equal to those entries in \( \tilde{x} \) and \( \tilde{B} \) corresponding to the non artificial variables.

end if

else

No feasible solution exist.

Set \( x \) and \( B \) equal to those entries in \( \tilde{x} \) and \( \tilde{B} \) corresponding to the non artificial variables.

end if

B.14  **lpsimp2**

Set \( m_i = m - m_{eq} \).

if \( m_i > 0 \) then

Add \( m_i \) slack variables to create a problem on standard form.

Update \( A, c \) and \( n \).

if \( x.0 \) is given then

Extend \( x.0 \) with zeros for the added slack variables.

end if

end if

if neither \( B.0 \) nor \( x.0 \) is given then

Set \( B^{(0)} = \{n - m + 1, n - m + 2, \ldots, n\} \).

Set \( N^{(0)} = \{1, 2, \ldots, n - m\} \).

else if \( B.0 \) is given then

Set \( B^{(0)} = \{i : B.0 = 1\} \), the set of basic variables.

Set \( N^{(0)} = \{i : B.0 = 0\} \), the set of nonbasic variables.

end if

if \( x.0 \) is given then

Set \( x = x.0 \).

Set \( B^{(0)} = \{i : x_i > 0\} \).

Set \( N^{(0)} = \{i : x_i \leq 0\} \).

if the number of elements in \( B^{(0)} \) is less than \( m \) then

Add to \( B^{(0)} \), index elements \( i \) corresponding to \( x_i = 0 \) to have \( B^{(0)} \) contain \( m \) elements.

Delete the same elements \( i \) from \( N^{(0)} \).

end if

else

Set \( x_j = 0 : j \in N \).

Solve \( A_B x_B = b \), where \( A_B = A_{ij} : j \in B \) and \( x_B = x_i : j \in B \).

end if

for \( k = 1, 2, \ldots, k_{max} \) do

Compute the objective function value \( \bar{f} = c_B^T x_B \), where \( c_B = c_j : j \in B \).

Compute shadow prices \( y \) by solving \( A_B^T y = c_B \).

Compute reduced costs \( \bar{c}_N = c_N - A_N^T y \), where \( A_N = A_{ij} : j \in N \) and \( c_N = c_j : j \in N \).

if \( \bar{c}_N \geq -\epsilon_f \) then

STOP, \( x \) is optimal.

end if

Choose the variable \( x_a \) to include in the new basis either using Bland’s anti-cycling rule or the Minimal Reduced Cost rule.
Compute the search direction $d$ by solving $A_B d = -a_q$, where $a_q$ is the $q$:th column in $A$.  
Set $P = \{ i : d_i < 0 \}$.
\begin{figure*}[h!]
\begin{algorithm}
\caption{Algorithm for the search direction $d$.}
\begin{algorithmic}
\State Set $P = \emptyset$.
\If{$P = \emptyset$}
\State The problem is unbounded, STOP.
\Else
\State Set the step length $\alpha = -\frac{x_{d_P}}{d_P} = \min_{i \in P} \left( -\frac{x_{d_i}}{d_i} \right)$
\State Variable $x_p$ is to be excluded from the basis.
\EndIf
\State Set $x_R = x_R + \alpha d$.
\State Set $x_p = 0$ and $x_q = \alpha$.
\State Set $B^{(k+1)} = B^{(k)} \cup \{q\} \setminus \{p\}$ and $N^{(k+1)} = N^{(k)} \cup \{p\} \setminus \{q\}$.
\end{algorithmic}
\end{algorithm}
\end{figure*}
\end{figure*}

end for

**B.15 maxflow**

Set $m$ equal to the number of nodes.
Set $x_{ij} = 0 \ \forall (i,j) \in Z$.
Set $max_{-flow} = 0$.
\begin{figure*}[h!]
\begin{algorithm}
\caption{Algorithm for maxflow.}
\begin{algorithmic}
\While{not convergence}
\State Set $pred(i) = 0$ and $flow(i) = 0$, $i = 1, 2, ... , m$.
\State Set $pred(s) = -1$ and $flow(s) = \infty$.
\State Set $LIST = s$.
\While{$LIST \neq \emptyset$ and $pred(t) = 0$}
\For{all arcs $(i,j)$ outgoing from node $i$}
\If{$pred(j) = 0$ and $x_{ij} < u_{ij}$}
\State Set $pred(j) = i$ and $flow(j) = \min (flow(i), x_{ij} - x_{ij})$.
\State Put $j$ at the end of $LIST$.
\EndIf
\EndFor
\For{all arcs $(j,i)$ coming in to node $i$}
\If{$pred(j) = 0$ and $x_{ji} < u_{ji}$}
\State Set $pred(j) = i$ and $flow(j) = \min (flow(i), x_{ji})$.
\State Put $j$ at the end of $LIST$.
\EndIf
\EndFor
\State Delete the first element in $LIST$.
\EndWhile
\If{$pred(t) > 0$}
\State Set $j = t$ and $i = pred(t)$.
\State Set $x_{ij} = x_{ij} + flow(t)$.
\While{$i \neq s$}
\State Set $j = i$ and $i = pred(i)$.
\If{$(i,j) \in Z$}
\State Set $x_{ij} = x_{ij} + flow(t)$.
\Else
\State Set $x_{ij} = x_{ij} - flow(t)$.
\EndIf
\EndWhile
\State Set $max_{-flow} = max_{-flow} + flow(t)$.
\Else
\State STOP, the maximum flow is $max_{-flow}$.
\EndIf
\EndWhile
\end{algorithmic}
\end{algorithm}
\end{figure*}

end while
B.16 modlabel

Set dist(j) = $\infty$ for each $j \in N \setminus \{s\}$, where $N$ is the set of all nodes.
Set dist(s) = 0 and pred(s) = 0.
Set LIST = {s}.

while LIST \neq \emptyset do
    Set $i$ equal to the first element in LIST.
    Delete the first element in LIST.
    for all arcs (i, j) outgoing from node $i$ do
        if dist(j) > dist(i) + c$_{ij}$ then
            Set dist(j) = dist(i) + c$_{ij}$.
            Set pred(j) = $i$
        end if
        if $j \notin$ LIST then
            if $j$ has been in in LIST before then
                Put $j$ first in LIST.
            else
                Put $j$ at the end of LIST.
            end if
        end if
    end for
end while

B.17 mintree

Set Z$_{tree} =$ Zin.

while the number of arcs in Z$_{tree}$ is less than $n - 1$ do
    Choose the arc (i, j) \notin Z$_{tree} \cup$ Zin for which $C_{ij} = \min\{C_{ij} : (i, j) \notin Z_{tree} \cup$ Zin\}.
    if the arc (i, j) does not create a cycle with the arcs in Z$_{tree}$ then
        Add the arc (i, j) to Z$_{tree}$.
    end if
    Set $C_{ij} = C_{ji} =$ Inf.
end while

B.18 TPmc

Initially set $x$ to a zero matrix of dimension $m \times n$.
Set $M = \max(c) + 1$.
for $k = 1, 2, ..., m + n - 1$ do
    Choose (i, j) for which $i + j = \min\{i + j : c_{ij} = \min(c)\}$
    if $s_i > d_j$ then
        Set $x_{ij} = d_j$.
        Set $s_i = s_i - d_j$.
        Set all elements in the $j$:th column of $c$ equal to $M$.
    else
        Set $x_{ij} = s_i$.
        Set $d_j = d_j - s_i$.
        Set all elements in the $i$:th row of $c$ equal to $M$.
    end if
end for

B.19 TPnw

Initially set $x$ to a zero matrix of dimension $m \times n$.
Set $i = 1$ and $j = 1$. 
for $k = 1, 2, ..., m + n - 1$ do
  if $s_i > d_j$ then
    Set $x_{ij}$ = $d_j$.
    Set $B_k = (i, j)$.
    Set $s_i = s_i - d_j$.
    Set $j = j + 1$.
  else
    Set $x_{ij} = s_i$.
    Set $B_k = (i, j)$.
    Set $d_j = d_j - s_i$.
    Set $i = i + 1$.
  end if
end for

B.20 TPsimplex

if $\sum_{i} s_i > \sum_{j} d_j$ then
  Add a dummy demand point with zero cost.
else if $\sum_{i} s_i < \sum_{j} d_j$ then
  Add a dummy supply point with high cost.
end if
if $x$ and $B$ is not given then
  Call TP vogel to get a starting basic feasible solution.
else if only $x$ is given then
  Set $B$ to represent the nonzero entries in $x$.
else if only $B$ is given then
  Compute $x$ for the given basis $B$.
end if
for $k = 1, 2, ..., k_{max}$ do
  Compute the simplex multipliers $y = (u)$ by setting $v_n = 0$ and solving the $m + n - 1$ equations $u_i + v_j = c_{ij}$ for $(i, j) \in B$.
  Compute the reduced costs $\acute{c}_{ij} = c_{ij} - u_i - v_j$.
  Set $\acute{c}_{min} = \min(\acute{c})$.
  if $\acute{c}_{min} \geq 0$ then
    STOP, $x$ is optimal.
  else
    Set $q = (q_1, q_2)$ where $\acute{c}_{q_1q_2} = \acute{c}_{min}$.
  end if
  Determine the cycle of change vector $\mu$ by solving $A\mu = b$, where $A \in \mathbb{R}^{m+n \times n+m}$ and $b \in \mathbb{R}^{n+m}$. $A_{B_i, i} = 1$, $A_{m+B_{i,2},i} = 1$ for $i = 1, 2, ..., m+n$. $A_{m+n, m+n} = 1$ and the rest of the entries in $A$ is zero. $b_{q_1} = -1$, $b_{q_2} = -1$ and the rest of the entries in $b$ is zero.
  Set $\theta = \min \{ x_{B_{1,1}, B_{1,2}} : \mu_i < 0 \}$.
  if $\theta = 0$ then
    STOP, the problem has an unbounded feasible region.
  else
    Set $p = (p_1, p_2)$ where $x_{p, p_i} = \theta$.
  end if
  Set $x_{B_{1,1}, B_{1,2}} = x_{B_{1,1}, B_{1,2}} + \theta \mu_i$.
  Set $x_{q_1q_2} = \theta$.
  Set $B = B \cup \{ q \} \setminus \{ p \}$.
end for
B.21 TPvogel

Initially set $x$ to a zero matrix of dimension $m \times n$.
Set $k = 1$.

while $k \leq m + n - 1$ do

Compute for each column $j$ a penalty $p_{cj}$ equal to the difference between the two smallest costs in the column, using entries that do not lie in a crossed-out row or column.
if there is a column where only one entry is not crossed-out then

for $j = 1, 2, \ldots, n$ do

if column $j$ is a column with only one crossed-out entry then

Choose $i$ so that $(i, j)$ corresponds to that entry.
if $s_i > d_j$ then

Set $x_{ij} = d_j$.
Set $s_i = s_i - d_j$.
else

Set $x_{ij} = s_i$.
Set $d_j = d_j - s_i$.
end if
Set $B_k = (i, j)$.
Set $k = k + 1$.
end if
end for

else

Compute for each row $i$ a penalty $p_{ri}$, equal to the difference between the two smallest costs in the row, using entries that do not lie in a crossed-out row or column.
if there is a row where only one entry is not crossed-out then

for $i = 1, 2, \ldots, m$ do

if row $i$ is a row with only one crossed-out entry then

Choose $j$ so that $(i, j)$ corresponds to that entry.
if $s_i > d_j$ then

Set $x_{ij} = d_j$.
Set $s_i = s_i - d_j$.
else

Set $x_{ij} = s_i$.
Set $d_j = d_j - s_i$.
end if
Set $B_k = (i, j)$.
Set $k = k + 1$.
end if
end for

else

if $\max(p_{ri}) > \max(p_{cj})$ then

Set $i = \text{argmax}(p_{ri})$.
Choose $j$ so that $(i, j)$ corresponds to the smallest cost in row $i$ of the non crossed-out entries
else

Set $j = \text{argmax}(p_{cj})$.
Choose $i$ so that $(i, j)$ corresponds to the smallest cost in column $j$ of the non crossed-out entries.
end if
if $s_i > d_j$ then

Set $x_{ij} = d_j$.
Set $s_i = s_i - d_j$.
Cross out column $j$.
else

Set $x_{ij} = s_i$.
Set $d_j = d_j - s_i$.
Cross out row $i$.
end if
Set $B_k = (i, j)$.
Set $k = k + 1$.
end if
end if
end while

B.22 urelax

Set $x_j = 0$, $j = 1, 2, ..., n$ and $u_i = -1$, $i = 1, 2, ..., m - 1$.
Set $A_{rj} = A_{rj}$, $j = 1, 2, ..., n$ and $b = b_r$.
Set $A_{ij} = A_{ij}$ and $\tilde{b}_i = b_i$, $i \in \{1, 2, ..., m\} - \{r\}$, $j = 1, 2, ..., n$.
Set $f_P = 0$ and $x_P = x$.
for $k = 0, 1, ..., u_{\text{max}}$ do
  Set $u_i = u_i + 1$, $i = 1, 2, ..., m - 1$.
  Set $\tilde{c} = \tilde{A}^T u$.
  Call the knapsack problem solver $dpknaps$ with the parameters $\tilde{A}$, $\tilde{b}$, $\tilde{c}$ and $x_U$ to get the solution $x$ and $f_D$.
  Set $f_D = f_D + u^T b$ and compute the subgradient $\tilde{g} = b - Ax$.
if $\tilde{g}_i \geq 0$, $i = 1, 2, ..., m - 1$ and $c^T x > f_P$ then
  Set $f_P = c^T x$ and $x_P = x$.
end if
end for
References


