TOMLAB v1.0 User's Guide

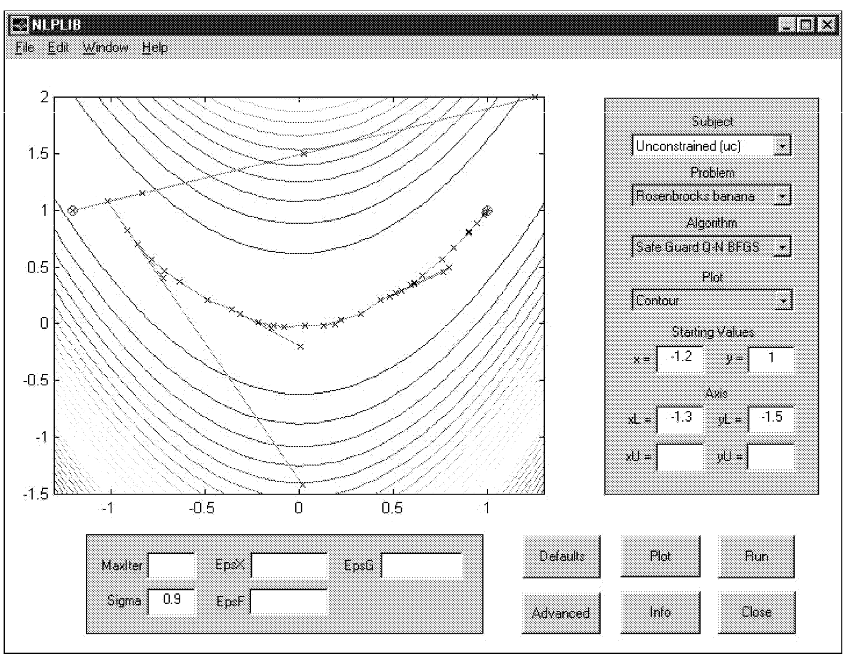
Kenneth Holmström, Mattias Björkman and Erik Dotzauer

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.

[2]The **TOM** home page is **http://www.ima.mdh.se/tom**.

Contents

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

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

A bug in Matlab 5.1 on PC for the *nargin* command makes it necessary to call *nargin* with only non-capitalized letters. The routine *xnargin* in Matlab 5.1 does lower on the arguments in the call to *nargin*, and the *xnargin* routine in the *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, *strcmpi*.

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 *solvType*, where *solvType* is any of the *probType* entries in Table 1. It is clear that a solver of a certain *solvType* is able to solve a problem defined to be of another type. For example, a constrained nonlinear programming solver should be able to solve unconstrained problems and constrained nonlinear least squares problems.

Table 1: The different types of optimization problems treated in NLPLIB TB.

probType	Number	Description of the type of problem
uc	1	Unconstrained optimization (incl. bound constraints).
qp	2	Quadratic programming.
con	3	Constrained nonlinear optimization.
ls	4	Nonlinear least squares problems (incl. bound constraints).
exp	5	Exponential fitting problems.
cls	6	Constrained nonlinear least squares problems.
nts	7	Nonlinear time series.
lp	8	Linear programming.
glb	9	Box-bounded global optimization.
glc	10	Global mixed-integer nonlinear programming.

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* **usr** 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 *solvType*), 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,

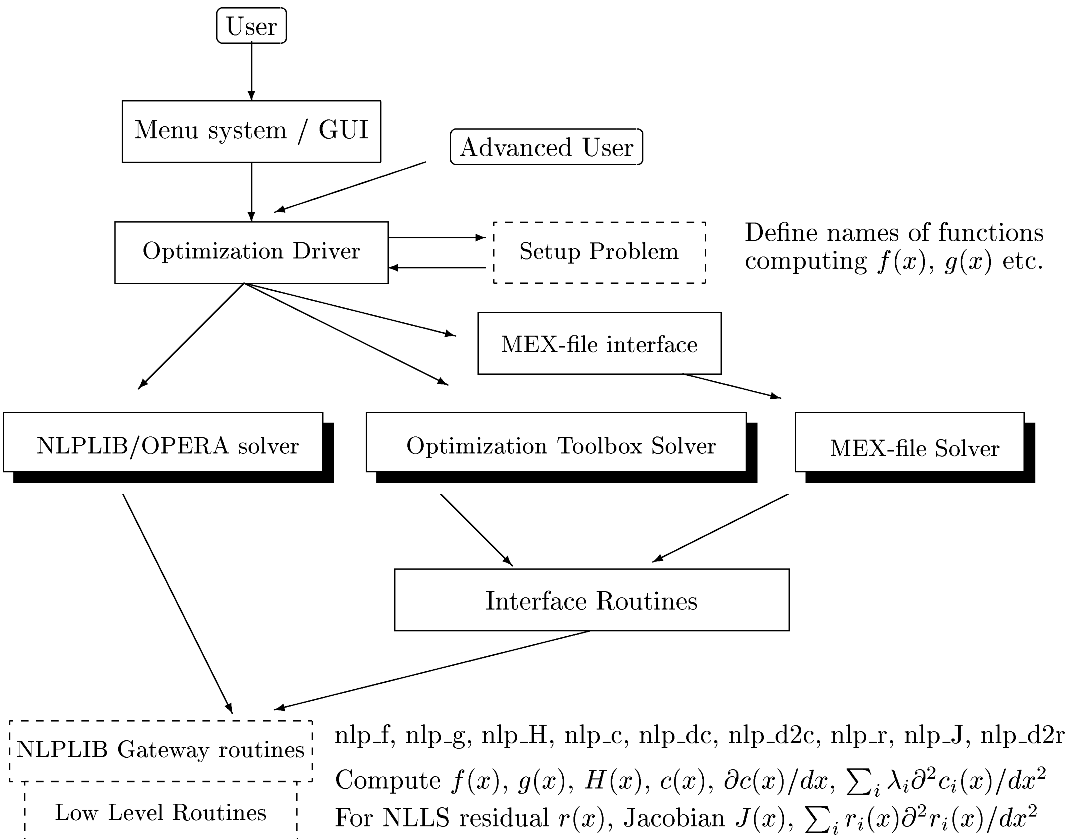


Figure 1: The process of optimization in TOMLAB.

Table 2: Defined test problem sets in TOMLAB.

probSet	probType	Description of test problem set
uc	1	Unconstrained test problems.
qp	2	Quadratic programming test problems.
con	3	Constrained test problems.
ls	4	Nonlinear least squares test problems.
exp	5	Exponential fitting problems.
cls	6	Linear constrained nonlinear least squares problems.
nls	6	Nonlinear constrained nonlinear least squares problems.
glb	9	Box-bounded global optimization test problems.
glc	10	Global MINLP test problems.
mgh	4	More, Garbow, Hillstrom nonlinear least squares problems.
amp	3	AMPL test problems as *nl*-files.
cto	3	CUTE constrained test problems as *dll*-files.
ctl	3	CUTE large constrained test problems as *dll*-files.
uto	1	CUTE unconstrained test problems as *dll*-files.
utl	1	CUTE large unconstrained test problems as *dll*-files.
nts	7	Nonlinear time series.
usr	1-9	User defined problems of *probType* 1-9.

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 (*solvType* = **con**, *probSet* = **con**) is shown in Figure 2.

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

Generic variable	Purpose (*solvType* is ⋄, e.g. ⋄=**con**)
⋄Opt	Menu program.
⋄Run	Multi-solver optimization driver routine.
⋄Def	Routine defining optimization parameters.
⋄Solve	(Prototype) solver.

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.

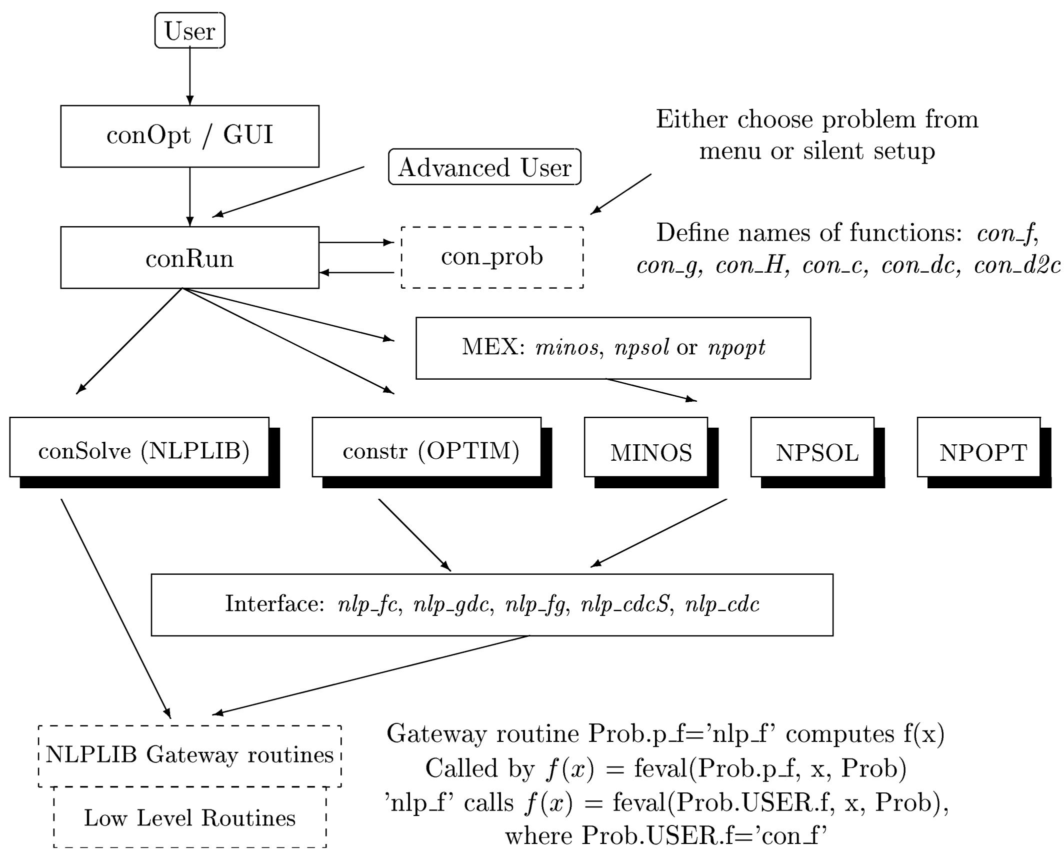


Figure 2: Solution of constrained nonlinear problems in TOMLAB.

Table 4: Names on the low level m-files in NLPLIB TB.

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

Table 5: Information stored in the problem structure *Prob*.

Field	Description
Name	Problem name.
P	Problem number.
probType	TOMLAB problem type, see Table 1.
probFile	Name of m-file in which problem are defined.
xName	Name of each decision variable.
cName	Name of each general constraint.
optParam	Structure with special fields for optimization parameters, see Table 6.
Solver	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.
uP	User supplied parameters for the problem.
uPName	Problem name connected to the user supplied parameters.
ExpFit	Structure with special fields for exponential fitting problems, see Table 7.
QP	Structure with special fields for quadratic problems, see Table 8.
NLLS	Structure with special fields for nonlinear least squares, see Table 9.
NTS	Structure with special fields for nonlinear time series, see Table 10.
PartSep	Structure with special fields for partially separable functions, see Table 11.
GLOBAL	Structure with special fields for global optimization, see Table 12.
A	Constraint matrix for linear constraints, one constraint per row.
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.
N	Problem dimension (number of variables).
f_Low	Lower bound on function value. Used in line search by Fletcher, default, $-realmax = -1.7977E308$.
x_opt	Optimal point x^* (if known).
f_opt	Optimal objective function value $f(x^*)$.
AutoDiff	If true, use automatic differentiation.
NumDiff	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.
p_f	Name of gateway routine computing the objective function $f(x)$.
p_g	Name of gateway routine computing the gradient vector $g(x)$.
p_H	Name of gateway routine computing the Hessian matrix $H(x)$.
p_c	Name of gateway routine computing the vector of constraint functions $c(x)$.
p_dc	Name of gateway routine computing the matrix of constraint normals $\partial c(x)/dx$.
p_d2c	Name of gateway routine computing the 2nd part of 2nd derivative matrix of the Lagrangian function, $\sum_i \lambda_i \partial^2 c(x)/dx^2$.
p_r	Name of gateway routine computing the residual vector $r(x)$.
p_J	Name of gateway routine computing the Jacobian matrix $J(x)$.
p_d2r	Name of gateway routine computing the second part of the Hessian for a nonlinear least squares problem, i.e. $\sum_{i=1}^{m} r_i(x) \frac{\partial^2 r_i(x)}{\partial x_j \partial x_k}$.
USER	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
x_min	Lower plot region parameters.
x_max	Upper plot region parameters.

Table 6: Information stored in the structure *Prob.optParam*

Field	**Description**
alg	Optimization Algorithm. Dependent on type of problem. Default 0.
method	Solver sub-method technique. Default 0.
PriLev	Print level in optimization solver, default 1.
eps_x	Convergence tolerance in optimal solution x, distance between successive x, $\|\|x_{k+1} - x_k\|\|$, default 10^{-8}.
eps_f	Convergence tolerance on f. Also used when testing on the directed derivative, default 10^{-8}.
eps_dirg	Convergence tolerance on the directed derivative, default 10^{-8}.
eps_c	Constraint violation convergence tolerance, default 10^{-6}.
LineAlg	Line search algorithm. 0 = quadratic interpolation, 1 = cubic interpolation, 2 = curvilinear quadratic interpolation, 3 = curvilinear cubic interpolation. Default $LineAlg = 0$.
GradCheck	Set to 1 if you want to check user-supplied gradients, default 0.
MaxIter	Maximum number of iterations, default 500.
DiffGradMinChange	Minimum change in variables for finite difference gradients, default 10^{-8}.
DiffGradMaxChange	Maximum change in variables for finite difference gradients, default 0.1.
InitStepLength	Initial step length, default 1 or less.
eps_g	Gradient (or reduced gradient) convergence tolerance, default 10^{-6}.
eps_Rank	Rank test tolerance, default 10^{-10}.
wait	Flag if to use pause statements after output, default 0.
eps_absf	Convergence tolerance on absolute function value, default *realmin*.
PreSolve	Flag if presolve analysis is to be applied on linear constraints, default 0.
QN_InitMatrix	Initial matrix for Quasi-Newton, may be set by the user. When *QN_InitMatrix* is empty, the identity matrix is used.
LineSearch	Structure with special fields for the line search, see Table 14.
Penalty	Penalty parameter for constrained problems.
xTol	If $x \in [x\_L, x\_L + bTol]$ or $[x\_U - bTol, x\_U]$, fix x on bound.
bTol	Feasibility tolerance for linear constraints.
cTol	Feasibility tolerance for nonlinear constraints.
fTol	Accuracy in the computation of the function value, default $eps^{0.9}$.
size_x	Size at optimum for the variables x, used in the convergence tests. Default 1.
size_f	Size at optimum for the function f, used in the convergence tests. Default 1.
size_c	Size at optimum for the constraints c, used in the convergence tests. Default 1.
LowIts	Number of iterations with low reduction before convergence.
NOT_release_all	Set to 1 if not to release more than one variable at the time.
subalg	Optimization sub algorithm. Dependent on type of problem. Default 0.
splineSmooth	Smoothness parameter sent to the SPLINE Toolbox routine *csaps.m* when computing numerical approximations of the gradient and the Jacobian. Default 0.4.
splineTol	Tolerance parameter sent to the SPLINE Toolbox routine *spaps.m* when computing numerical approximations of the gradient and the Jacobian. Default 10^{-3}.

Table 7: Information stored in the structure *Prob.ExpFit*

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

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

Field	Description
F	Constant matrix, the Hessian
c	Constant vector.
B	Logical vector of the same length as the number of variables. A one corresponds to a variable in the basis.

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

Field	Description
weightType	Weighting type: *0* No weighting. *1* Weight with data in *Yt*. If $Yt = 0$, the weighting is 0, i.e. deleting this residual element. *2* Weight with weight vector or matrix in *weightY*. If *weightY* is a vector then weighting by $weigthY. * r$ (elementwise multiplication). If *weightY* is a matrix then weighting by $weigthY * r$ (matrix multiplication). *3* *nlp_r* calls the routine *weigthtY* (must be a string with the routine name) to compute the residuals.
weightY	Either empty, a vector, a matrix or a string, see *weightType*.
t	Time vector t.
Yt	Matrix with observations $Y(t)$.
UseYt	If $UseYt\ = 0$ compute residual as $f(x,t) - Y(t)$ (default), otherwise $Y(t)$ should be treated separately and the residual routines just return $f(x,t)$.
SepAlg	If $SepAlg = 1$, use separable non linear least squares formulation, default 0.

Table 10: Information stored in the structure *Prob.NTS*

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

Table 11: Information stored in the structure *Prob.PartSep*

Field	Description
pSepFunc	Number of partially separable functions.
index	Index for the partially separable function to compute, i.e. if $i = index$, compute $f_i(x)$. If $index = 0$, compute the sum of all, i.e. $f(x) = \sum_{i=1}^{M} f_i(x)$.

Table 12: Information stored in the structure *Prob.GLOBAL*

Field	Description
iterations	Number of iterations, default 50.
MaxEval	Number of function evaluations, default 500.
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.
C	Matrix with all rectangle centerpoints.
D	Vector with distances from centerpoint to the vertices.
L	Matrix with all rectangle side lengths in each dimension.
F	Vector with function values.
d	Row vector of all different distances, sorted.
d_min	Row vector of minimum function value for each distance.
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 proceedure.
I_L	$I\_L(i,j)$ is the lower bound for rectangle j in integer dimension $I(i)$.
I_U	$I\_U(i,j)$ is the upper bound for rectangle j in integer dimension $I(i)$.
feasible	Flag indicating if a feasible point has been found.
f_min	Best function value found at a feasible point.
s_0	s_0 is used as $s(0)$.
s	$s(j)$ is the sum of observed rates of change for constraint j.
t	$t(i)$ is the total number of splits along dimension i.

Table 13: Information stored in the structure *Prob.USER*

Field	Description
f	Name of m-file computing the objective function $f(x)$.
g	Name of m-file computing the gradient vector $g(x)$. If *Prob.USER.g* is empty then numerical derivatives will be used.
H	Name of m-file computing the Hessian matrix $H(x)$.
c	Name of m-file computing the vector of constraint functions $c(x)$.
dc	Name of m-file computing the matrix of constraint normals $\partial c(x)/dx$.
d2c	Name of m-file computing the 2nd part of 2nd derivative matrix of the Lagrangian function, $\sum_i \lambda_i \partial^2 c(x)/dx^2$.
r	Name of m-file computing the residual vector $r(x)$.
J	Name of m-file computing the Jacobian matrix $J(x)$.
d2r	Name of m-file computing the 2nd part of the Hessian for nonlinear least squares problem, i.e. $\sum_{i=1}^{m} r_i(x) \frac{\partial^2 r_i(x)}{\partial x_j \partial x_k}$.

Table 14: Information stored in the structure *Prob.optParam.LineSearch*

Field	Description
sigma	Line search accuracy; $0 < sigma < 1$. $sigma = 0.9$ inaccurate line search. $sigma = 0.1$ accurate line search, default 0.9.
rho	Determines the ρ line, default 0.01.
tau1	Determines how fast step grows in phase 1, default 9.
tau2	How near end point of $[a, b]$, default 0.1.
tau3	Choice in $[a, b]$ phase 2, default 0.5.
eps1	Minimal length for interval $[a, b]$, default 10^{-7}.
eps2	Minimal reduction, default 10^{-12}.
MaxIter	Maximum number of line search iterations.

Table 15: Information stored in the global Matlab structure *Result*.

Field	Description
Iter	Number of major iterations.
MinorIter	Number of minor iterations.
ExitFlag	0 if convergence to local min. Otherwise errors.
Inform	Information parameter, type of convergence.
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.
f_0	Function value at start i.e. $f(x\_0)$.
x_k	Optimal point.
v_k	Lagrange multipliers.
r_k	Residual at optimum.
J_k	Jacobian matrix at optimum.
c_k	Value of constraints at optimum.
cJac	Constraint Jacobian at optimum.
xState	State of each variable, described in Table 16 .
bState	State of each linear constraint, described in Table 17.
cState	State of each general constraint.
optParam	Structure with special fields for optimization parameters, see Table 6.
Name	Problem name.
P	Problem number.
p_dx	Matrix where each column is a search direction.
alphaV	Matrix where row i stores the steplengths tried for the i:th iteration.
x_min	Lowest x-values in optimization. Used for plotting.
x_max	Highest x-values in optimization. Used for plotting.
F_X	F_X is a global matrix with rows: [iter_no f(x)].
GLOBAL	Structure with special fields for global optimization, see Table 18.
SepNLLS	General result variable with fields z and Jz. Used when running separable nonlinear least squares problems
Solver	Solver used.
SolverAlgorithm	Solver algorithm used.
CPUtime	CPU time used.
REALtime	Real time elapsed.
Nflops	Number of floating point operations.
probType	TOMLAB problem type.
solvType	TOMLAB solver type.
FuncEv	Number of function evaluations needed.
GradEv	Number of gradient evaluations needed.
ConstrEv	Number of constraint evaluations needed.
ResEv	Number of residual evaluations needed.
JacEv	Number of Jacobian evaluations needed.
Prob	Problem structure, see Table 5.
plotData	Structure with plotting parameters.

Table 16: The state variable *xState* for the variable.

Value	Description
0	A free variable.
1	Variable on lower bound.
2	Variable on upper bound.
3	Variable is fixed, lower bound is equal to upper bound.

Table 17: The state variable *bState* for each linear constraint.

Value	Description
0	Inactive constraint.
1	Linear constraint on lower bound.
2	Linear constraint on upper bound.
3	Linear equality constraint.

Table 18: Information stored in the structure *Result.GLOBAL*

Field	Description
C	Matrix with all rectangle centerpoints in original coordinates.
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.
Split	$Split(i, j)$ is the number of splits along dimension i of rectangle j.
T	$T(i)$ is the number of times rectangle i has been trisected.
G	Matrix with constraint values for each point.
ignoreidx	Rectangles to be ignored in the rectangle selection procedure.
I_L	$I\_L(i, j)$ is the lower bound for rectangle j in integer dimension $I(i)$.
I_U	$I\_U(i, j)$ is the upper bound for rectangle j in integer dimension $I(i)$.
feasible	Flag indicating if a feasible point has been found.
f_min	Best function value found at a feasible point.
s_0	s_0 is used as $s(0)$.
s	$s(j)$ is the sum of observed rates of change for constraint j.
t	$t(i)$ is the total number of splits along dimension i.

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 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 19: The global variables used in NLPLIB TB

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

Table 20: The global variables used in NLPLIB TB

Name	**Description**
LS_A	Problem dependent information sent from residual routine to Jacobian routine.
LS_x	Value of x when computing *LS_r*
LS_r	Residual value.
LS_xJ	Value of x when computing *LS_J*
LS_J	Jacobian value.
SEP_z	Separated variables z.
SEP_Jz	Jacobian for separated variables z.
wNLLS	Weighting of least squares residuals (internal variable in *nlp_r* and *nlp_J*).
alphaV	Vector with all step lengths α for each iteration.
BUILDP	Flag.
F_X	Matrix with function values.
pLen	Number of iterations so far.
p_dx	Matrix with all search directions.
X_max	The biggest x-values for all iterations.
X_min	The smallest x-values for all iterations.
X_NEW	Last x point in line search. Possible new x_k.
X_OLD	Last known base point x_k
probType	Defines the type of optimization problem.
solvType	Defines the solver type.
answer	Used by the GUI for user control options.
instruction	Used by the GUI for user control options.
question	Used by the GUI for user control options.
plotData	Structure with plotting parameters.
Prob	Problem structure, see Table 5.
Result	Result structure, see Table 15.
runNumber	Vector index when *Result* is an array of structures.
TIME0	Used to compute CPU time and real time elapsed.
TIME1	Used to compute CPU time and real time elapsed
cJPI	Used to store sparsity pattern for the constraint Jacobian when automatic differentiation is used.
HPI	Used to store sparsity pattern for the Hessian when automatic differentiation is used.
JPI	Used to store sparsity pattern for the Jacobian when automatic differentiation is used.
SparseStructure	Used by MINOS (sparse structure).
NonZeros	Number of nonzero matrix elements in *SparseStructure*.
glbSave	Used to save global variables in recursive calls to TOMLAB.
PATHDEL	PC or UNIX way of path delimiter i.e. "\" or "/".

2.2 Solver Routines in NLPLIB TB

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

Table 21: Optimization solvers in NLPLIB TB.

Function	Description	Section	Page
ucSolve	A prototype routine for unconstrained optimization with simple bounds on the parameters. Implements Newton, quasi-Newton and conjugate-gradient methods.	2.11.14	84
glbSolve	A routine for box-bounded global optimization.	2.11.5	73
gblSolve	Stand-alone version of *glbSolve*. Runs independently of NLPLIB TB.	2.11.3	70
glcSolve	A routine for global mixed-integer nonlinear programming.	2.11.6	75
gclSolve	Stand-alone version of *glcSolve*. Runs independently of NLPLIB TB.	2.11.4	71
lsSolve	A prototype algorithm for nonlinear least squares with simple bounds. Implements Gauss-Newton, and hybrid quasi-Newton and Gauss-Newton methods.	2.11.7	76
clsSolve	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.	2.11.1	67
conSolve	Constrained nonlinear minimization solver using two different sequential quadratic programming methods.	2.11.2	69
nlpSolve	Constrained nonlinear minimization solver using filter SQP.	2.11.8	78
sTrustR	Solver for constrained convex optimization of partially separable functions, using a structural trust region algorithm.	2.11.13	82
qpBiggs	Solves a general quadratic program.	2.11.10	80
qpSolve	Solves a general quadratic program.	2.11.12	81
qpe	Solves a **qp** problem, restricted to equality constraints, using a null space method.	2.11.9	79
qplm	Solves a **qp** problem, restricted to equality constraints, using Lagrange's method.	2.11.11	80

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 *solvType* the user just calls the routine *PrintSolvers* with *solvType* as argument. *solvType* should either be a string ('uc', 'con' etc.) or the corresponding *solvType* number, see Table 1. As an example, assume you want a list of all available solvers of *solvType* **con**. Then

```
PrintSolvers('con')
```

gives the printing output

```
----------------------------------------------------
   Solver    solvType Number  Multi-Solver Driver
----------------------------------------------------
   nlpSolve         3                conRun
   conSolve         3                conRun
   sTrustR          3                conRun
   constr           3                conRun
   minos            3                conRun
   npsol            3                conRun
   npopt            3                conRun
```

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

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

where $x, x_L, x_U \in \mathbb{R}^n$ and $f(x) \in \mathbb{R}$. *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 *glbSolve* implements an algorithm for **box-bounded global optimization** (**glb**), i.e. problems of the form (1) that have finite simple bounds on all the variables. *glbSolve* implements the DIRECT algorithm [38], which is a modification of the standard Lipschitzian approach that eliminates the need to specify a Lipschitz constant. In *glbSolve* no derivative information is used. For **global mixed-integer nonlinear programming** (**glc**), *glcSolve* 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 *glbSolve* and *glcSolve* named *gblSolve* and *gclSolve* 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

$$\begin{array}{ll} \min\limits_{x} & f(x) \\ & \\ s/t & \begin{array}{ccccc} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U \\ c_L & \leq & c(x) & \leq & c_U \end{array} \end{array} \tag{2}$$

where $x, x_L, x_U \in \mathbb{R}^n$, $f(x) \in \mathbb{R}$, $A \in \mathbb{R}^{m_1 \times n}$, $b_L, b_U \in \mathbb{R}^{m_1}$ and $c_L, c(x), c_U \in \mathbb{R}^{m_2}$. For general constrained nonlinear optimization a sequential quadratic programming (SQP) method by Schittkowski [50] is implemented in the routine *conSolve*. Like *ucSolve*, *lsSolve* and *clsSolve*, *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

$$\begin{array}{ll} \min\limits_{x} & f(x) = \frac{1}{2}x^T F x + c^T x \\ & \\ s/t & \begin{array}{ccccc} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U \end{array} \end{array} \tag{3}$$

where $c, x, x_L, x_U \in \mathbb{R}^n$, $F \in \mathbb{R}^{n \times n}$, $A \in \mathbb{R}^{m_1 \times n}$, and $b_L, b_U \in \mathbb{R}^{m_1}$. Quadratic programs are solved with a standard active-set method [42], implemented in the routine *qpSolve*. *qpSolve* explicitly treats both inequality and equality constraints, as well as lower and upper bounds on the variables (simple bounds). It converges to a local minimum for indefinite quadratic programs. NLPLIB TB also includes a similar routine *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 (*qpe*) and Lagrange's method (*qplm*).

The **nonlinear least squares problem** (**ls**) is defined as

$$\begin{array}{ll} \min\limits_{x} & f(x) = \frac{1}{2} r(x)^T r(x) \\ & \\ s/t & x_L \leq x \leq x_U, \end{array} \tag{4}$$

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 *lsSolve* treats problems with bound constraints in a similar way as the routine *ucSolve*.

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

$$\begin{array}{cccccc} \min\limits_{x} & f(x) = \frac{1}{2} r(x)^T r(x) & & & & \\ & & & & & \\ & x_L & \leq & x & \leq & x_U, \\ s/t & b_L & \leq & Ax & \leq & b_U \\ & c_L & \leq & c(x) & \leq & c_U \end{array} \tag{5}$$

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 *clsSolve* is based on *lsSolve* and its search steps methods. Currently *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.

Function	Description
nlplib	Graphical User Interface (GUI) for nonlinear optimization. Handles all types of nonlinear optimization problems.
ucOpt	Menu for unconstrained optimization.
glbOpt	Menu for box-bounded global optimization.
glcOpt	Menu for global mixed-integer nonlinear programming.
qpOpt	Menu for quadratic programming.
conOpt	Menu for constrained optimization.
lsOpt	Menu for nonlinear least squares problems.
clsOpt	Menu for constrained nonlinear least squares problems.

Each menu program calls a corresponding driver routine, having the same *probType*, viz. either of *ucRun*, *glbRun*, *qpRun*, *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*, *qpDef*, *conDef*, *lsDef* or *clsDef*.

In Table 23, the utility functions needed by the solvers in Table 21 are displayed. The function *itrr* 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.

Function	Description	Section	Page
itrr	Initial trust region radius algorithm.	2.12.3	86
LineSearch	Line search algorithm by Fletcher.	2.12.4	87
intpol2	Find the minimum of a quadratic interpolation. Used by *LineSearch*.	2.12.1	85
intpol3	Find the minimum of a cubic interpolation. Used by *LineSearch*.	2.12.2	85
preSolve	Presolve analysis on linear constraints and simple bounds.	2.12.5	88

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*.

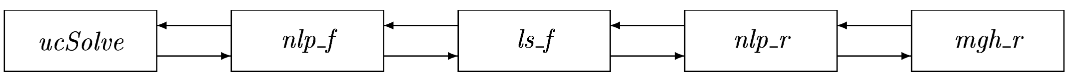


Figure 3: The chain of calls when computing the objective function value in *ucSolve* for a nonlinear least squares problem defined in *mgh_prob*, *mgh_r* and *mgh_J*.

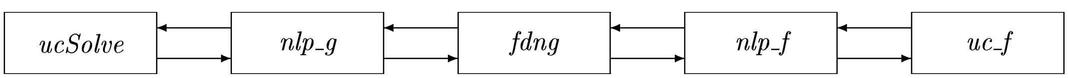


Figure 4: The chain of calls when computing the numerical approximation of the gradient in *ucSolve* for an unconstrained problem defined in *uc_prob* and *uc_f*.

Information about a problem is stored in the structure variable *Prob*, described in 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 unconstrained 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.

Function	**Description**
con_prob	Initialization of **con** test problems.
con_f	Compute the objective function $f(x)$ for **con** test problems.
con_g	Compute the gradient $g(x)$ for **con** test problems.
con_H	Compute the Hessian matrix $H(x)$ of $f(x)$ for **con** test problems.
con_c	Compute the constraint residuals $c(x)$ for **con** test problems.
con_dc	Compute the derivative of the constraint residuals for **con** test problems.
con_fm	Compute merit function $\theta(x_k)$.
con_gm	Compute gradient of merit function $\theta(x_k)$.

Table 25: Exponential fitting test problems.

Function	**Description**
exp_ArtP	Generate artificial exponential sum problems.
expInit	Find starting values for the exponential parameters λ.
exp_prob	Defines a exponential fitting type of problem, with data series (t, y). The file includes data from several different empirical test series.
Helax_prob	Defines 335 medical research problems supplied by Helax AB, Uppsala, where an exponential model is fitted to data. The actual data series (t, y) are stored on one file each, i.e. 335 data files, 8MB large, and are not distributed. A sample of five similar files are part of *exp_prob*.
exp_r	Compute the residual vector $r_i(x), i = 1, ..., m$. $x \in \mathbb{R}^n$
exp_J	Compute the Jacobian matrix $\partial r_i/dx_j, i = 1, ..., m, j = 1, ..., n$.
exp_d2r	Compute the 2nd part of the second derivative for the nonlinear least squares exponential fitting problem.
exp_c	Compute the constraints $\lambda_1 < \lambda_2 < ...$ on the exponential parameters $\lambda_i, i = 1, ..., p$.
exp_dc	Compute matrix of constraint normals for constrained exponential fitting problem.
exp_d2c	Compute second part of second derivative matrix of the Lagrangian function for constrained exponential fitting problem. This is a zero matrix, because the constraints are linear.
exp_q	Find starting values for exponential parameters $\lambda_i, i = 1, ..., p$.
exp_p	Find optimal number of exponential terms, p.

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.

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

Table 27: Test problems from CUTE data base.

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

2.3.2 Test Routines for the System

NLPLIB TB is constantly being developed and improved. Therefore it is important to have some routines who 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 28: System test routines.

Function	Description	Section	Page
runtest	Runs all selected problems defined in a problem file for a given solver.	2.13.3	89
systest	Runs big test to check for bugs in NLPLIB TB.	2.13.4	90

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 *glbOpt*. 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

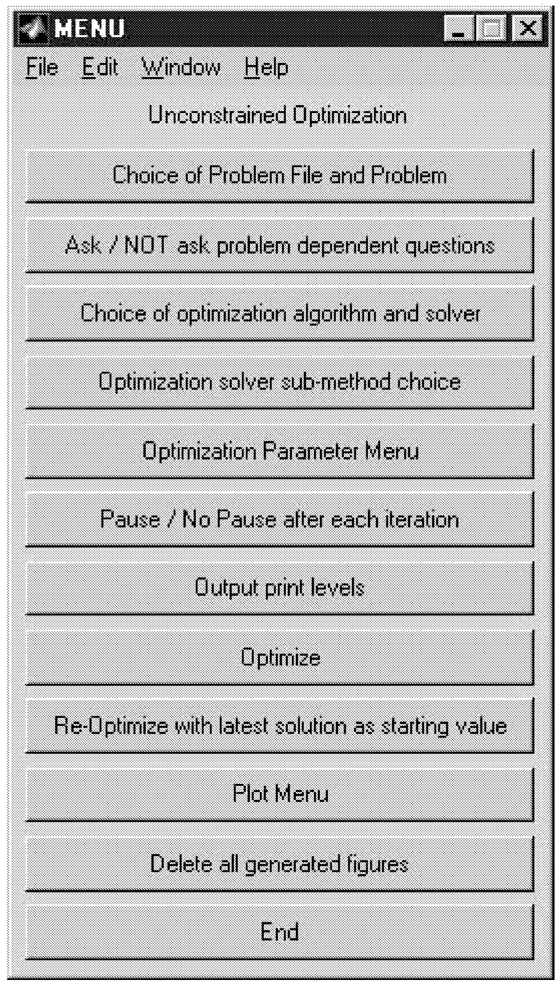


Figure 5: The main menu in *ucOpt*.

'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 σ (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 no 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 Matlabm-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.

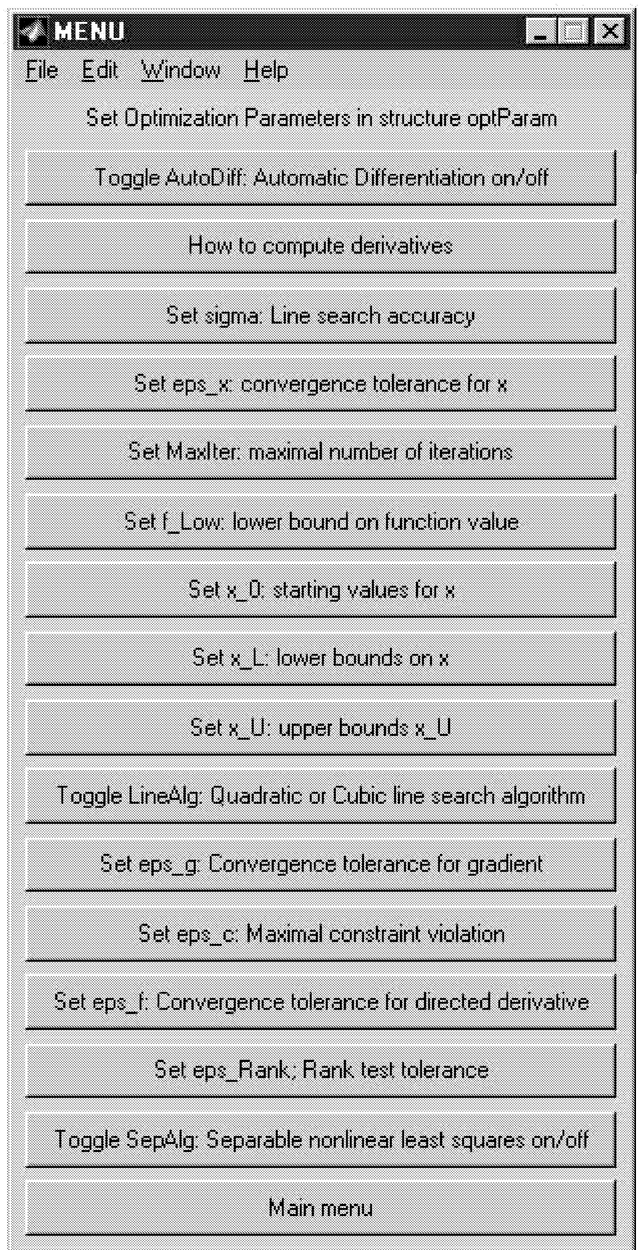


Figure 6: Setting optimization parameters in *ucOpt*.

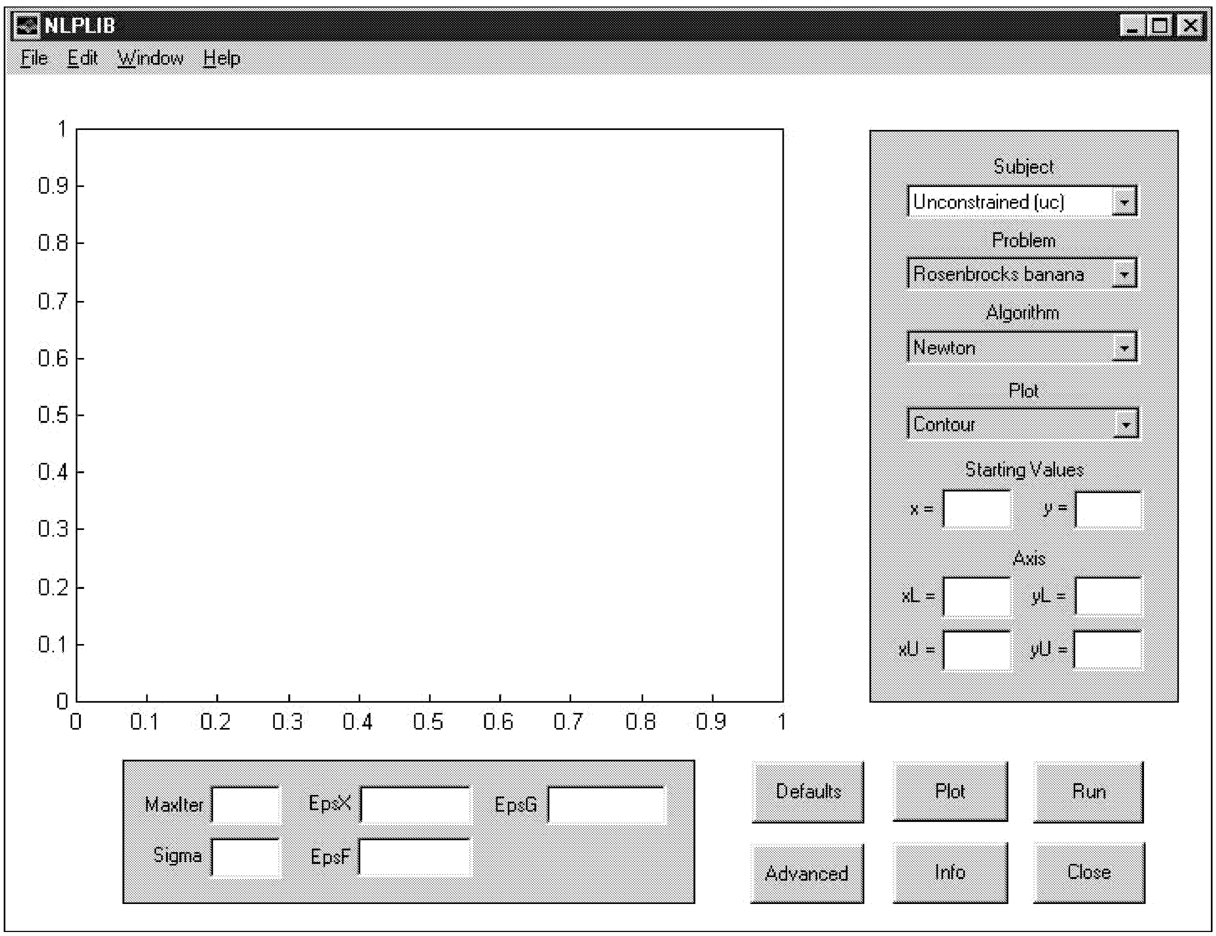


Figure 7: The GUI in Normal mode.

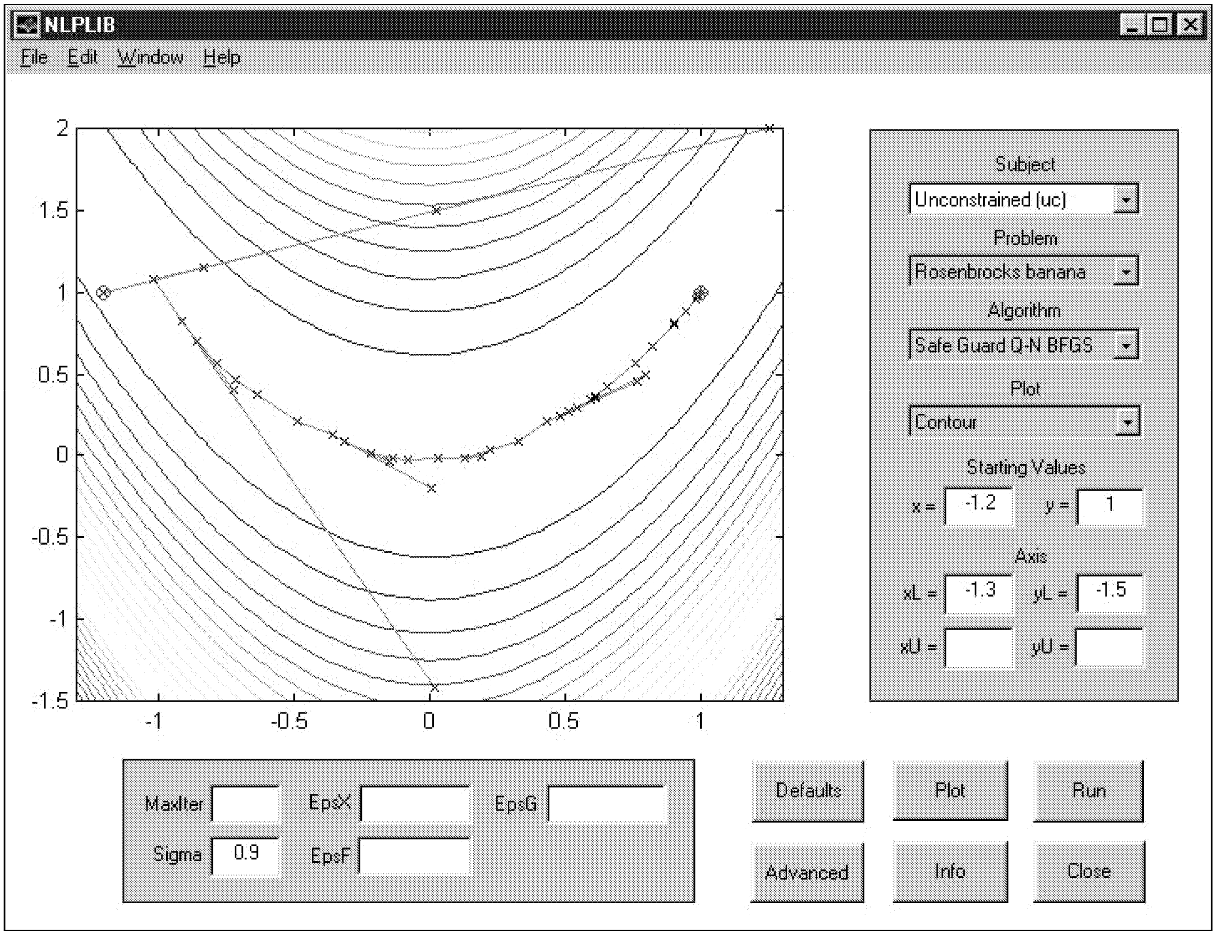


Figure 8: A contour plot with the search directions and marks for the line search trials for each iteration.

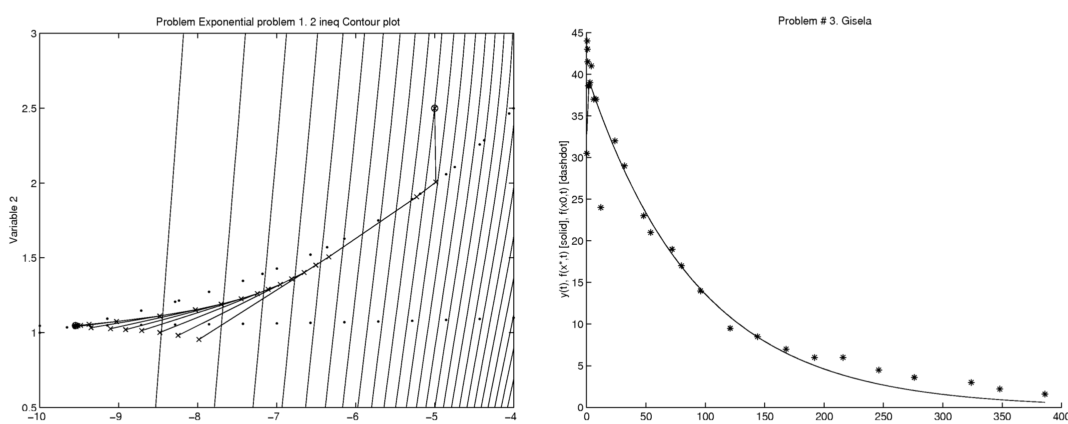


Figure 9: A contour plot for a constrained problem and a plot of data and model for a nonlinear least squares problem.

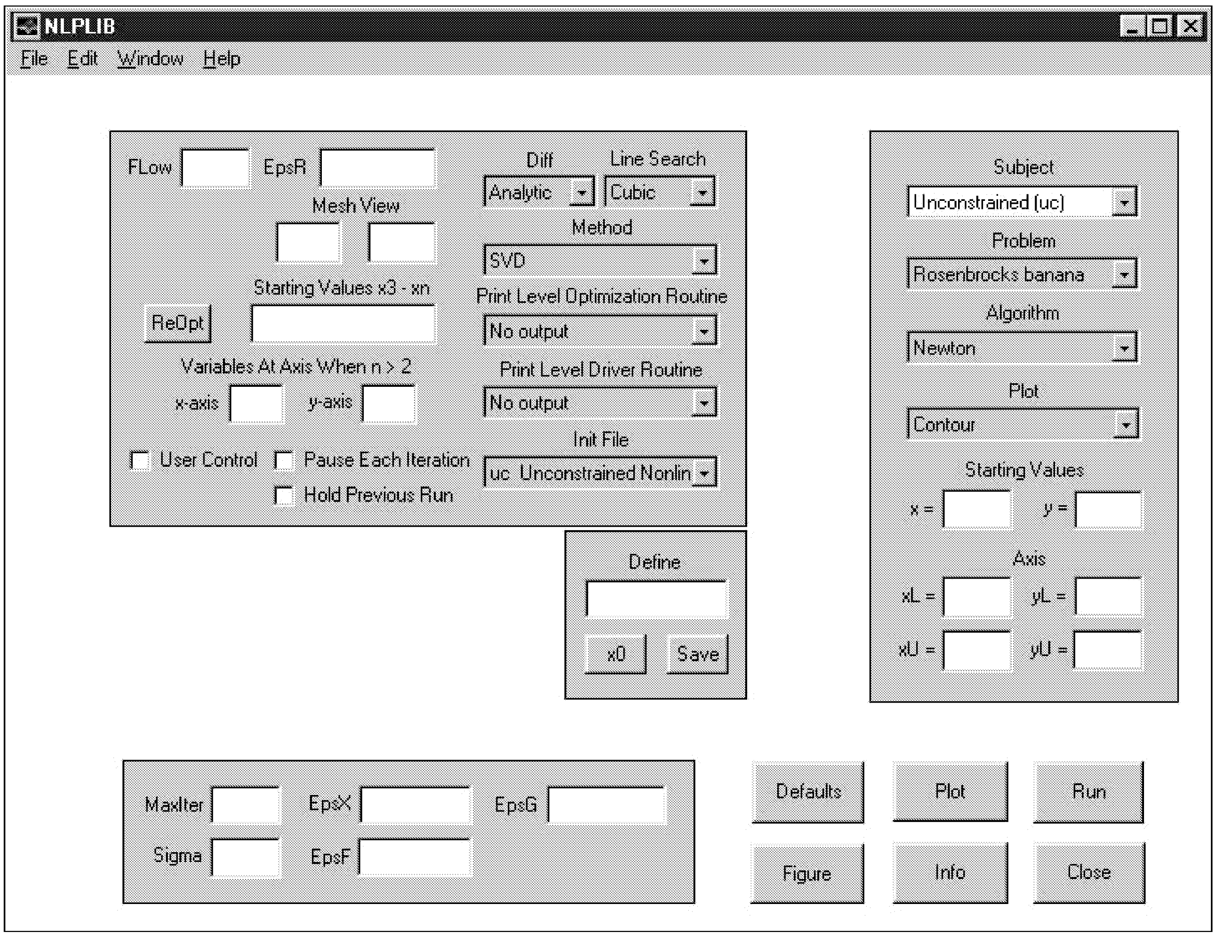


Figure 10: The GUI in Advanced mode.

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 ...\tomlab\nlpnew. 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*,

$$\begin{array}{ll} \min\limits_{x} & f(x) = 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2 \\ s/t & -10 \leq x_j \leq 2, j = 1, 2, \end{array} \tag{6}$$

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*:

```
...
...
      ,'Fletcher Q.2.6'...
      ,'Fletcher Q.3.3'...
      ,'RB BANANA'...
           ); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

if isempty(P)
   return;
```

```
end
...
...
```

3. Add the following in *uc_prob.m* after the last already existing problem (the optional parameters are not necessary to define):

```
...
...
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*:

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

5. Make the following addition in *uc_g.m*:

```
...
...
elseif P == 17  % Fletcher Q.3.3
   %f = 0.5*(x(1)^2+x(2)^2)*exp(x(1)^2-x(2)^2);
   e = exp(x(1)^2-x(2)^2);
   g = e*[x(1)*(1+x(1)^2+x(2)^2); x(2)*(1-x(1)^2-x(2)^2)];
elseif P == 18  % RB BANANA
   g = [-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*:

```
...
...
elseif P == 17  % Fletcher Q.3.3
   %f = 0.5*(x(1)^2+x(2)^2)*exp(x(1)^2-x(2)^2);
   %g = e*[x(1)*(1+x(1)^2+x(2)^2); x(2)*(1-x(1)^2-x(2)^2)];
```

```
      e = exp(x(1)^2-x(2)^2);
      H = [1+5*x(1)^2+2*x(1)^2*x(2)^2+x(2)^2+2*x(1)^4, ...
           -2*x(1)*x(2)*(x(1)^2+x(2)^2);
           0 , 1-x(1)^2+2*x(1)^2*x(2)^2-5*x(2)^2+2*x(2)^4 ];
      H(2,1)=H(1,2);
      H = e*H;
   elseif P == 18  % RB BANANA
      H = [ 1200*x(1)^2-400*x(2)+2 , -400*x(1);
                  -400*x(1)        ,    200  ];
   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* and *usr_H.m* in step 1. In these files, replace the problem *Own UC problem 1* with *RB BANANA* in the same way as described above (do not forget the menu choice line in *usr_prob.m*).

2.6.2 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, *glb_prob* and *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 *glb_prob.m* and *glb_f.m* to your directory NLPNEW.
2. Add the problem name to the menu choice in *glb_prob.m*:

```
...
...
      ,'HGO 468:2'...
      ,'Spiral'...
      ,'RB BANANA'...
          ); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

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

3. Add the following in *glb_prob.m* after the last already existing problem (the optional parameters are not necessary to define):

```
...
...
elseif P == 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*:

```
...
...
elseif 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):

```
...
...
elseif P==10
   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).
```

```
   f_min=0;             % Lower bound on function (optional).
   x_max=[ 1.3  1.3];   % Plot region parameters.
   x_min=[-1.1  -0.2];  % Plot region parameters.
else
   disp('ls_prob: Illegal problem number')
   pause
   Name=[];
end
...
...
```

4. Make the following addition in *ls_r.m*:

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

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

5. Make the following addition in *ls_J.m*:

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

6. Save all the files properly.

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 LS problem 1* with *RB BANANA* in the same way as described above (do not forget the menu choice line in *usr_prob.m*).

2.6.4 Defining Constrained Problems

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

$$x_1 - x2 \leq 1 \tag{7}$$

and

$$-x_1^2 - x_2 \leq 1 \tag{8}$$

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_g.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_g.m* and *con_H.m* in the same way as described for for the unconstrained case in Section 2.6.1.

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

```
...
...
elseif P == 15
   Name='RB BANANA';
   x_0 = [-1.2 1]';       % Starting values for the optimization.
   x_L = [-10;-10];       % Lower bounds for x.
   x_U = [2;2];           % Upper bounds for x.
   x_opt = [1 1];         % Known optimal point (optional).
   f_opt = 0;             % Known optimal function value (optional).
   f_min = 0;             % Lower bound on function (optional).
   x_max = [ 1.3  1.3];   % Plot region parameters.
   x_min = [-1.1  -0.2];  % Plot region parameters.

   A   = [1 -1];          % Linear constraints matrix.
   b_L = -inf;            % Lower bounds on linear constraints.
   b_U = 1;               % Upper bounds on linear constraints.
   c_L = -inf;            % Lower bounds on nonlinear constraints.
   c_U = 1;               % Upper bounds on nonlinear constraints.
end
...
...
```

4. Make the following addition in *con_c.m*:

```
...
...
elseif P == 15  % RB BANANA
   cx = -x(1)^2 - x(2);
end
...
...
```

5. Make the following addition in *con_dc.m*:

```
...
...
elseif P == 15  % RB BANANA
   if init==0
      dc = [-2*x(1);-1];
   else
      dc = ones(2,1);
   end
end
...
...
```

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

```
...
...
```

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

7. Save all the files properly.

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

$$x_1 \quad \text{integer} \tag{9}$$

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 *glc_c.m*:

```
...
...
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:

```
...
...
elseif P==28
   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).
   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, *qp_prob.m*. The problem

$$\begin{array}{ll}\min\limits_{x} & f(x) = 4x_1^2 + x_1x_2 + 4x_2^2 + 3x_1 - 4x_2 \\ s/t & x_1 + x_2 \le 5 \\ & x_1 - x_2 = 0 \\ & x_1 \ge 0 \\ & x_2 \ge 0,\end{array} \tag{10}$$

named *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 *qp_prob.m* to your directory NLPNEW.
2. Add the problem name to the menu choice in *qp_prob.m*:

```
...
...
     ,'Bazaara IQP 9.29b pg 405. F singular'...
     ,'Bunch and Kaufman Indefinite QP'...
     ,'QP EXAMPLE'...
       ); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

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

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

```
...
...
elseif P==15
   Name='QP EXAMPLE';
   F = [ 8   2          % Hessian
         2   8 ];
   c = [ 3  -4 ]';
   A = [ 1   1          % Constraint matrix
         1  -1 ];
   b_L = [-inf  0   ]';  % Lower bounds on the constraints
   b_U = [  5   0   ]';  % Upper bounds on the constraints
   x_L = [  0   0   ]';  % Lower bounds on the variables
   x_U = [ inf inf ]';   % Upper bounds on the variables
   x_0 = [  0   1   ]';  % Starting point
   x_min=[-1 -1 ];       % Plot region parameters
   x_max=[ 6  6 ];       % Plot region parameters
else
   disp('qp_prob: Illegal problem number')
   pause
   Name=[];
end
...
...
```

4. 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

$$t = 10^{-3}\begin{pmatrix} 30\\ 50\\ 70\\ 90\\ 110\\ 130\\ 150\\ 170\\ 190\\ 210\\ 230\\ 250\\ 270\\ 290\\ 310\\ 330\\ 350\\ 370 \end{pmatrix}, Y(t) = 10^{-4}\begin{pmatrix} 18299\\ 15428\\ 13347\\ 11466\\ 10077\\ 8729\\ 7382\\ 6708\\ 5932\\ 5352\\ 4734\\ 4271\\ 3744\\ 3485\\ 3111\\ 2950\\ 2686\\ 2476 \end{pmatrix}, \tag{11}$$

here named SW.

To define (11) as a 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*:

```
...
...
           ,'Atcexp nr2 '...
           ,'Atcexp nr2\~ '...
           ,'SW '...
           ); % MAKE COPIES OF THE PREVIOUS ROW AND CHANGE TO NEW NAMES

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

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

```
...
...
elseif P==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 2476]';
      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')
   ...
   ...
```

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 EF problem 1* with *SW* in the same way as described above (do not forget the menu choice line in *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 *Prob.ExpFit.eType* which is set by defining the parameter *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 2476]';
   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')
...
...
```

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

$f(t) = \sum_i^p \alpha_i e^{-\beta_i t}$,	$\alpha_i \geq 0$,	$0 \leq \beta_1 < \beta_2 < ... < \beta_p$,	$eType = 1$.
$f(t) = \sum_i^p \alpha_i (1 - e^{-\beta_i t})$,	$\alpha_i \geq 0$,	$0 \leq \beta_1 < \beta_2 < ... < \beta_p$,	$eType = 2$.
$f(t) = \sum_i^p t\alpha_i e^{-\beta_i t}$,	$\alpha_i \geq 0$,	$0 \leq \beta_1 < \beta_2 < ... < \beta_p$,	$eType = 3$.
$f(t) = \sum_i^p (t\alpha_i - \gamma_i) e^{-\beta_i t}$,	$\alpha_i, \gamma_i \geq 0$,	$0 \leq \beta_1 < \beta_2 < ... < \beta_p$,	$eType = 4$.

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 *ls_prob*, *ls_r* and *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*:

```
...
...
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
end
...
...
```

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

```
...
...
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:

```
...
...
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.

```
...
...
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.

```
...
...
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:

```
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       80.00000000000000000000 Cond 1
==================================================
Iteration no:    1  Func        1.25000000000000000000 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];      % 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 1 0];      % Known optimal point (optional).
```

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

```
   x_opt = [ -1 -1  1
              1 -2  0
             -3  5  2 ];
```

Note that it is not necessary to define *x_opt*, 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:

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

Result = ucRun([], Prob, [], [], 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       0.000000000000000001
                                 User given f(x_*)      0.000000000000000000
                                            f(x_0)     24.199999999999996000

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

FuncEv   48 GradEv   40
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: -4.162202e-009  9.227064e-010
NLPLIB found no active constraints.

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

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

```
   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);
```

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 **for** loop we will solve all the least squares problems defined in the files *own_prob*, *own_r* and *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 *drv_test* below runs *lsRun* for all problems defined in *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.

```
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 = 'lsSolve';

optParam  = lsDef;  % Set default values.

optParam.PriLev    = 0;     % No printing output.
optParam.eps_x     = 1E-7;  % Termination tolerance for X (Default=1E-8).
optParam.eps_f     = 1E-9;  % Termination tolerance on F.(Default=1E-10). Dir.derivative
optParam.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.eps_g     = 1E-5;  % Termination tolerance on gradient.(Default=1E-6).
optParam.eps_Rank  = 1E-11; % Rank test tolerance. Used in subspace minimization.
optParam.wait      = 0;     % If true, pause after iteration printout.
optParam.eps_absf  = 1E-35; % Absolute convergence tolerance in function 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;
```

```
   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);

end
```

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 NLPNEW 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_L* 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 *qpSolve*. 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 30: *PriLev* in the optimization routines

Value	Description
< 0	Totally silent.
0	Error messages and warnings.
1	Final results including convergence test results and minor warnings.
2	Each iteration, short output.
3	Each iteration, more output.
4	Line search or QP information.
5	Hessian output, final output in solver.

There is a wait flag field in *optParam*, *optParam.wait*. If this flag is set true, 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. Independent on the number of rows in the matrix, *printmat* will first display $A(:, 1:10)$, then $A(:, 11:20)$ and so on.

The routine *printmat* tries to be intelligent and avoid decimals when the matrix elements are integers. It determines the maximal positive and minimal negative number to find out if more than the default space is needed. If any element has an absolute value below 10^{-5} (avoiding exact zeros) or if the maximal elements are too big, a switch is made to exponential format. The exponential format uses ten characters, displaying two decimals and therefore seven matrix elements are possible to display on each row.

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

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

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 ...\tomlab\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.

ADMAT 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

```
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

```
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

```
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.

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

Consider the problem *DAS 2*:

$$\begin{array}{ll} \min\limits_{x} & f(x) = \frac{1}{2}\sum\limits_{1}^{6} r_i(x)^2 \\ s/t & Ax \geq b \\ & x \geq 0 \end{array} \qquad (12)$$

where

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

The objective function in (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 *con_prob*, *con_f*, *con_g* etc. to be an illustrative example of how to define a problem with a partially separable objective function. Note the definition of *pSepFunc* in *con_prob*.

Solving (12) with *sTrustR* is done by the following definitions and call:

```
probFile = 'con_prob';
P        = 14;
Prob     = probInit(probFile,P);
Solver   = 'sTrustR';

Result   = conRun(Solver,Prob,[],[],probFile,P);
```

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 *ego*) solves an unconstrained (**uc**) and a box-bounded global optimization problem (**glb**) 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 *globalSave* and *globalGet*. The *globalSave* routine saves all global variables in a structure *glbSave(depth)* and then initialize all of of them as empty. By using the depth variable, an arbitrarily number of recursions are possible. The other routine *globalGet* retrieves all global variables in the structure *glbSave(depth)*.

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

```
...
...
   globalSave(1);
   EGOResult = glbSolve(EGOProb);
   globalGet(1);
...
...
   globalSave(1);
   [DACEResult] = ucSolve(DACEProb);
```

```
   globalGet(1);
...
...
```

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. $ask \geq 1$ Ask questions in *probFile*. $ask = [\,]$ If $Prob.uP = [\,]$, $ask = -1$, 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 *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*, *mkbound.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. $ask \geq 1$ Ask questions in *probFile*. $ask = [\,]$ If $uP = [\,]$, $ask = -1$, else $ask = 0$.
PriLev	Print level when displaying the result of the optimization in the routine *PrintResult*. See Section 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, by use of a MEX-file interface run the commercial optimization solvers MINOS, NPSOL and NPOPT.

M-files Used

xxxRun.m, *xxxRun2.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 glbRun

Purpose

Driver routine for box-bounded global optimization.

Calling Syntax

Result = glbRun(Solver, Prob, ask, PriLev, probFile, probNumber)

Description of Inputs

Solver	The name of the solver that should be used to optimize the problem. Default *glbSolve*. 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. $ask \geq 1$ Ask questions in *probFile*. $ask = [\,]$ If $uP = [\,]$, $ask = -1$, else $ask = 0$.
PriLev	Print level when displaying the result of the optimization in the routine *PrintResult*. See Section 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 *glb_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 *glbRun* is called by the menu routine *glbOpt* or the graphical user interface routine *nlplib* to solve global optimization problems defined in your problem definition files. It is also possible for the user to call *glbRun* directly from the Matlab command prompt, see Section 2.7. Via *glbRun* you can run the TOMLAB internal solver *glbSolve*.

M-files Used

xxxRun.m, *xxxRun2.m*, *PrintResult.m*, *inibuild.m*, *ucDef.m*, *probInit.m*, *mkbound.m*, *glbSolve.m*, *iniSolve.m*, *endSolve.m*

2.10.4 glcRun

Purpose

Driver routine for global mixed-integer nonlinear programming.

Calling Syntax

Result = glcRun(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. $ask \geq 1$ Ask questions in *probFile*. $ask = [\,]$ If $uP = [\,]$, $ask = -1$, else $ask = 0$.
PriLev	Print level when displaying the result of the optimization in the routine *PrintResult*. See Section 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 lsRun

Purpose

Driver routine for nonlinear least squares solvers.

Calling Syntax

Result = lsRun(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. $ask \geq 1$ Ask questions in *probFile*. $ask = [\,]$ If $uP = [\,]$, $ask = -1$, else $ask = 0$.
PriLev	Print level when displaying the result of the optimization in the routine *PrintResult*. See Section 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 *ls_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*, *nlssol.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 *qpSolve*. 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. $ask \geq 1$ Ask questions in *probFile*. $ask = [\,]$ If $uP = [\,]$, $ask = -1$, else $ask = 0$.
PriLev	Print level when displaying the result of the optimization in the routine *PrintResult*. See Section 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 *qp_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 *qpRun* is called by the menu routine *qpOpt* or the graphical user interface routine *nlplib* to solve quadratic programming problems defined in your problem definition files. It is also possible for the user to call *qpRun* directly from the Matlab command prompt, see Section 2.7. Via *qpRun* you can run the TOMLAB internal solvers *qpe*, *qplm*, *qpiOld* and *qpiSolve* (not fully developed) and the Matlab Optimization Toolbox solver *qp*. Currently NLPLIB TB also includes a not fully developed routine *qpBiggs* for negative definite quadratic problems.

M-files Used

xxxRun.m, *xxxRun2.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 ucRun

Purpose

Driver routine for unconstrained optimization solvers.

Calling Syntax

Result = 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. $ask < 0$ Use values in *uP* if defined or defaults. $ask = 0$ Use defaults. $ask \geq 1$ Ask questions in *probFile*. $ask = [\,]$ If $uP = [\,]$, $ask = -1$, else $ask = 0$.
PriLev	Print level when displaying the result of the optimization in the routine *PrintResult*. See Section 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 *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* to 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

xxxRun.m, *xxxRun2.m*, *xxxRun3.m*, *inibuild.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

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

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

Calling Syntax

Result = clsSolve(Prob, varargin)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	Solver.Alg	Solver algorithm to be run: 0: 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*, *wait*, *size_x*, *size_f*, *f_Low*, *LineSearch*, *LineAlg*, *bTol*, *cTol*, *xTol*, *LowIts*, *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	Flag giving exit status.
	Inform	Binary code telling type of convergence: 1: Iteration points are close. 2: Projected gradient small. 4: Function value close to 0. 8: Relative function value reduction low for *LowIts* 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 Hushens 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*, *qpSolve.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

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

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

Calling Syntax
Result = conSolve(Prob, varargin)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	Solver.Alg	Solver algorithm to be run: 0: Schittkowski SQP. 1: Han-Powell SQP.
	optParam	Structure with special fields for optimization parameters, see Table 6. Fields used are: *eps_f*, *eps_g*, *eps_c*, *eps_x*, *eps_Rank*, *eps_absf*, *MaxIter*, *wait*, *size_x*, *size_f*, *size_c*, *f_Low*, *LineSearch*, *LineAlg*, *xTol*, *LowIts*, *PriLev*, *method* and *QN_InitMatrix*.
	A	Constraint matrix for linear constraints.
	b_L	Lower bounds on the linear constraints.
	b_U	Upper bounds on the linear constraints.
	c_L	Lower bounds on the general constraints.
	c_U	Upper bounds on the general constraints.
	x_L	Lower bounds on the variables.
	x_U	Upper bounds on the variables.
	x_0	Starting point.
	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 $\partial c(x)/dx$.
	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	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.
f_0	Function value at start.
f_k	Function value at optimum.
g_k	Gradient value at optimum.
H_k	Hessian value at optimum.
x_0	Starting point.
x_k	Optimal point.
v_k	Lagrange multipliers.
c_k	Value of constraints at optimum.
cJac	Constraint Jacobian at optimum.
xState	State of each variable, described in Table 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 *conSolve* 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

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

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

Calling Syntax

Result = 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: *C* Matrix with all rectangle centerpoints. *D* Vector with distances from centerpoint to the vertices. *L* Matrix with all rectangle side lengths in each dimension. *F* Vector with function values. *d* Row vector of all different distances, sorted. *d_min* Row vector of minimum function value for each distance.
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: *C* Matrix with all rectangle centerpoints. *D* Vector with distances from centerpoint to the vertices. *L* Matrix with all rectangle side lengths in each dimension. *F* Vector with function values. *d* Row vector of all different distances, sorted. *d_min* Row vector of minimum function value for each distance.

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 *gblsolve* 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 *glbSolve* 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 *glcSolve* and runs independently of NLPLIB TB.

gclSolve solves problems of the form

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

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

Calling Syntax

Result = gclSolve(p_f, p_c, x_L, x_U, A, b_L, b_U, c_L, c_U, I, GLOBAL, PriLev)

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

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 500 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 *glcSolve* it runs independently of NLPLIB TB.

2.11.5 glbSolve

Purpose

Solve box-bounded global optimization problems.

glbSolve solves problems of the form

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

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

Calling Syntax

Result = glbSolve(Prob,varargin)

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: *PriLev*.
	x_L	Lower bounds for x, must be given to restrict the search space.
	x_U	Upper bounds for x, must be given to restrict the search space.
	p_f	Name of m-file computing the objective function $f(x)$.
	GLOBAL	Special structure field containing:
	iterations	Number of iterations, default 50.
	epsilon	Global/local weight parameter, default 10^{-4}.
	K	The Lipschitz constant. Not used.
	tolerance	Error tolerance parameter. Not used.
	If restart is chosen in the menu system, the following fields in *GLOBAL* are also used and contains information from the previous run:	
	C	Matrix with all rectangle centerpoints.
	D	Vector with distances from centerpoint to the vertices.
	L	Matrix with all rectangle side lengths in each dimension.
	F	Vector with function values.
	d	Row vector of all different distances, sorted.
	d_min	Row vector of minimum function value for each distance.
varargin	Other parameters directly sent to low level routines.	

Description of Outputs

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

Description

The global optimization routine *glbSolve* is an implementation of the DIRECT algorithm presented in [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 *glbSolve* runs a predefined number of iterations and considers the best function value found as the optimal one. It is possible for the user to **restart** *glbSolve* with the final status of all parameters from the previous run. Let's say that you have run *glbSolve* 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 *glbSolve* you must give the result of the first run as input to your next run. The m-file *glbsolve* 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.

glcSolve solves problems of the form

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

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

Calling Syntax
Result = glcSolve(Prob,varargin)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	optParam	Structure with special fields for optimization parameters, see Table 6. Fields used are: *PriLev*, *cTol*.
	x_L	Lower bounds for x, must be given to restrict the search space.
	x_U	Upper bounds for x, must be given to restrict the search space.
	A	Constraint matrix for linear constraints.
	b_L	Lower bounds on the linear constraints.
	b_U	Upper bounds on the linear constraints.
	c_L	Lower bounds on the general constraints.
	c_U	Upper bounds on the general constraints.
	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:	
	C	Matrix with all rectangle centerpoints.
	D	Vector with distances from centerpoint to the vertices.
	F	Vector with function values.
	Split	$Split(i,j)$ is the number of splits along dimension i of rectangle j.
	T	$T(i)$ is the number of times rectangle i has been trisected.
	G	Matrix with constraint values for each point.
	ignoreidx	Rectangles to be ignored in the rectangle selection procedure.
	I_L	$I\_L(i,j)$ is the lower bound for rectangle j in integer dimension $I(i)$.
	I_U	$I\_U(i,j)$ is the upper bound for rectangle j in integer dimension $I(i)$.
	feasible	Flag indicating if a feasible point has been found.
	f_min	Best function value found at a feasible point.
	s_0	s_0 is used as $s(0)$.
	s	$s(j)$ is the sum of observed rates of change for constraint j.
	t	$t(i)$ is the total number of splits along dimension i.
varargin	Other parameters directly sent to low level routines.	

Description of Outputs

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

Description

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

iniSolve.m, *endSolve.m*

2.11.7 lsSolve

Purpose

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

lsSolve solves problems of the form

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

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

Calling Syntax

Result = lsSolve(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*, *f_Low*, *LineSearch*, *LineAlg*, *xTol*, *LowIts*, *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 *LowIts* iterations. 32: Local minimum with all variables on bounds. 101: Maximum number of iterations reached. 102: Function value below given estimate.
	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 .
	Solver	Solver used.
	SolverAlgorithm	Solver algorithm used.
	Prob	Problem structure used.

Description

The prototype routine *lsSolve* 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 Hushens 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 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

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

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

Calling Syntax
Result = nlpSolve(Prob, varargin)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	optParam	Structure with special fields for optimization parameters, see Table 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.
	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 $\partial c(x)/dx$.
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.
	ExitFlag	0: Convergence. Small step. Constraints fulfilled.
		1: Infeasible problem?
		2: Maximal number of iterations reached.
	Inform	Type of convergence.
	f_0	Function value at start.
	f_k	Function value at optimum.
	g_k	Gradient value at optimum.
	H_k	Hessian value at optimum.
	x_0	Starting point.
	x_k	Optimal point.
	v_k	Lagrange multipliers.
	c_k	Value of constraints at optimum.
	cJac	Constraint Jacobian at optimum.
	xState	State of each variable, described in Table 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 *nlpSolve* 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.

qpe solves problems of the form

$$\begin{array}{ll}\min\limits_{x} & f(x) = \frac{1}{2}(x)^T F x + c^T x \\ s/t & Ax = b\end{array}$$

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

Calling Syntax

[x, lambda, QZ, RZ] = qpe(F, c, A, b)

Description of Inputs

F	Constant matrix, the Hessain.
c	Constant vector.
A	Constraint matrix for the linear constraints.
b	Right hand side vector.

Description of Outputs

x	Optimal point.
lambda	Lagrange multipliers.
QZ	The matrix Q in the QR-decomposition of F.
RZ	The matrix R in the QR-decomposition of F.

Description

The routine *qpe* solves a quadratic programming problem, restricted to equality constraints, using a null space method.

See Also
qpBiggs, *qpSolve*, *qplm*

2.11.10 qpBiggs

Purpose
Solve general quadratic programming problems.

qpBiggs solves problems of the form

$$\begin{array}{cccccc} \min\limits_{x} & f(x) & = & \frac{1}{2}(x)^T F x + c^T x & & \\ s/t & b_i & = & a_i^T x & & i = 1, 2, ..., me \\ & b_i & \leq & a_i^T x & & i = me+1, ..., m \\ & x_L & \leq & x & \leq & x_U \end{array}$$

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

Calling Syntax
[x, lambda, err, p_vec, alfa_vec] = qpBiggs(F, c, A, b, x_L, x_U, x0, me, PriLev, wait)

Description of Inputs

F	Constant matrix, the Hessain.
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 α.

Description
The implementation of *qpBiggs* 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

$$\begin{array}{cl} \min\limits_{x} & f(x) = \frac{1}{2}(x)^T F x + c^T x \\ s/t & Ax = b \end{array}$$

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

Calling Syntax
[x, lambda] = qplm(F, c, A, b)

Description of Inputs

F	Constant matrix, the Hessain.
c	Constant vector.
A	Constraint matrix for the linear constraints.
b	Right hand side vector.

Description of Outputs

x	Optimal point.
lambda	Lagrange multipliers.

Description

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

See Also

qpBiggs, *qpSolve*, *qpe*

2.11.12 qpSolve

Purpose

Solve general quadratic programming problems.

qpSolve solves problems of the form

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

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

Calling Syntax

Result = qpSolve(Prob)

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_Rank*, *MaxIter*, *wait*, *bTol* and *PriLev*.
	QP.F	Constant matrix, the Hessian.
	QP.c	Constant vector.
	A	Constraint matrix for linear constraints.
	b_L	Lower bounds on the linear constraints.
	b_U	Upper bounds on the linear constraints.
	x_L	Lower bounds on the variables.
	x_U	Upper bounds on the variables.
	x_0	Starting point.

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	Iter	Number of iterations.
	ExitFlag	0: OK, see *Inform* for type of convergence. 2: Can not find feasible starting point *x_0*. 3: Rank problems. Can not find any solution point. 4: Unbounded solution.
	Inform	If $ExitFlag > 0$, $Inform = ExitFlag$, otherwise $Inform$ show type of convergence: 0: Unconstrained solution. 1: $\lambda \geq 0$. 2: $\lambda \geq 0$. No second order Lagrange mult. estimate available. 3: λ and 2nd order Lagr. mult. positive, problem is not negative definite. 4: Negative definite problem. 2nd order Lagr. mult. positive, but releasing variables leads to the same working set.
	f_0	Function value at start.
	f_k	Function value at optimum.
	g_k	Gradient value at optimum.
	H_k	Hessian value at optimum.
	x_0	Starting point.
	x_k	Optimal point.
	v_k	Lagrange multipliers.
	xState	State of each variable, described in Table 16 .
	Solver	Solver used.
	SolverAlgorithm	Solver algorithm used.
	Prob	Problem structure used.

Description

Implements an active set strategy for Quadratic Programming. For negative definite problems it computes eigenvalues and is using directions of negative curvature to proceed. To find an initial feasible point the Phase 1 LP problem is solved calling *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

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

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

Calling Syntax

Result = sTrustR(Prob, varargin)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	optParam	Structure with special fields for optimization parameters, see Table 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 $\partial c(x)/dx$.
varargin	Other parameters directly sent to low level routines.	

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	Iter	Number of iterations.
	ExitFlag	Flag giving exit status.
	Inform	Binary code telling type of convergence: 1: Iteration points are close. 2: Projected gradient small. 4: Relative function value reduction low for *LowIts* iterations. 8: Too small trust region. 101: Maximum number of iterations reached. 102: Function value below given estimate. 103: Convergence to saddle point (eigenvalues computed).
	f_0	Function value at start.
	f_k	Function value at optimum.
	g_k	Gradient value at optimum.
	H_k	Hessian value at optimum.
	x_0	Starting point.
	x_k	Optimal point.
	v_k	Lagrange multipliers.
	c_k	Value of constraints at optimum.
	cJac	Constraint Jacobian at optimum.
	xState	State of each variable, described in Table 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*, *qpoptSOL.m*, *qpSolve.m*, *iniSolve.m*, *endSolve.m*

See Also

conSolve, *nlpSolve*, *clsSolve*

2.11.14 ucSolve

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

ucSolve solves problems of the form

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

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

Calling Syntax
Result = ucSolve(Prob, varargin)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	Solver.Alg	Solver algorithm to be run: 0: Newton. 1: Safeguarded BFGS (default). 2: Safeguarded Inverse BFGS. 3: Safeguarded Inverse DFP. 4: Safeguarded DFP. 5: Fletcher-Reeves CG. 6: Polak-Ribiere CG. 7: Fletcher conjugate descent CG-method.
	optParam	Structure with special fields for optimization parameters, see Table 6. Fields used are: *eps_f*, *eps_g*, *eps_x*, *eps_Rank*, *MaxIter*, *wait*, *size_x*, *size_f*, *f_Low*, *LineSearch*, *LineAlg*, *xTol*, *LowIts*, *method*, *PriLev* and *QN_InitMatrix*.
	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 $H(x)$.
	f_Low	Lower bound on function value.
varargin	Other parameters directly sent to low level routines.	

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	Iter	Number of iterations.
	ExitFlag	0 if convergence to local min. Otherwise errors.
	Inform	Binary code telling type of convergence: 1: Iteration points are close. 2: Projected gradient small. 4: Relative function value reduction low for *LowIts* iterations. 101: Maximum number of iterations reached. 102: Function value below given estimate. 104: Convergence to a saddle point.
	f_0	Function value at start.
	f_k	Function value at optimum.
	g_k	Gradient value at optimum.
	H_k	Hessian value at optimum.
	B_k	Quasi-Newton approximation of the Hessian at optimum.
	x_0	Starting point.
	x_k	Optimal point.
	v_k	Lagrange multipliers.
	xState	State of each variable, described in Table 16 .
	Solver	Solver used.
	SolverAlgorithm	Solver algorithm used.
	Prob	Problem structure used.

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-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 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

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

Description of Outputs

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

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

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

Description of Outputs

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

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

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

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

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

Calling Syntax

Result = LineSearch(f, g, x, p, f_0, g_0, 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.
optParam	Structure with special fields for optimization parameters, the following fields are used:
	LineAlg Type of line search algorithm, se Table 6.
	LineSearch Structure with line search parameters, see Table 14.
alphaMax	Maximal value of step length α.
alpha_1	First step in α.
pType	Type of problem:
	0 Normal problem.
	1 Nonlinear least squares.
	2 Constrained nonlinear least squares.
	3 Merit function minimization.
	4 Penalty function minimization.
PriLev	Printing level:
	$PriLev > 0$ Writes a lot of output in *LineSearch*.
	$PriLev > 3$ Writes a lot of output in *intpol2* and *intpol3*.
varargin	Other parameters directly sent to low level routines.

Description of Outputs

Result	Result structure with fields:
	alpha Optimal line search step α.
	f_alpha Optimal function value at line search step α.
	g_alpha Optimal gradient value at line search step α.
	alphaVec Vector of trial step length values.
	r_k Residual vector if Least Squares problem, otherwise empty.
	J_k Jacobian matrix if Least Squares problem, otherwise empty.
	f_k Function value at $x + \alpha p$.
	g_k Gradient value at $x + \alpha p$.
	c_k Constraint value at $x + \alpha p$.
	dc_k Constraint gradient value at $x + \alpha p$.

Description

The function *LineSearch* together with the routines *intpol2* and *intpol3* implements a modified version of a line search algorithm by Fletcher [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_{Low} on $f(\alpha)$. More precisely it is assumed that the user is prepared to accept any value of $f(\alpha)$ for which $f(\alpha) \leq f_{Low}$. For example in a nonlinear least squares problem $f_{Low} = 0$ would be appropriate.

LineSearch consists of two parts, the *bracketing phase* and the *sectioning phase*. In the bracketing phase the iterates $\alpha^{(k)}$ moves out in an increasingly large jumps until either $f \leq f_{Low}$ is detected or a bracket on an interval of acceptable points is located. The sectioning phase generates a sequence of brackets $\left[a^{(k)}, b^{(k)}\right]$ whose lengths tend to zero. Each iteration pick a new point $\alpha^{(k)}$ in $\left[a^{(k)}, b^{(k)}\right]$ by minimizing a quadratic or a cubic polynomial which interpolates $f\left(a^{(k)}\right)$, $f'\left(a^{(k)}\right)$, $f\left(b^{(k)}\right)$ and $f'\left(b^{(k)}\right)$ if it is known. The sectioning phase terminates when

$\alpha^{(k)}$ is an acceptable point.

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

Prob	Problem description structure. The following fields are used:	
	A	Constraint matrix for linear constraints.
	b_L	Lower bounds on the linear constraints.
	b_U	Upper bounds on the linear constraints.
	x_L	Lower bounds on the variables.
	x_U	Upper bounds on the variables.

Description of Outputs

Prob	Problem description structure. The following fields are changed:	
	A	Constraint matrix for linear constraints.
	b_L	Lower bounds on the linear constraints, set to NaN for redundant constraints.
	b_U	Upper bounds on the linear constraints, set to NaN for redundant constraints.
	x_L	Lower bounds on the variables.
	x_U	Upper bounds on the variables.

Description
The routine *preSolve* is an implementation of those presolve analysis techniques described by Gondzio in [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 *el_cnsts* that improves variable bounds and uses them to eliminate redundant and forcing constraints. Both *r_rw_sng* and *el_cnsts* check if empty rows appear and eliminate them if so. That is handled by the routine *emptyrow*. In *clean* the calls to *r_rw_sng* and *el_cnsts* are repeated (in given order) until no further reduction is obtained.

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

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 *solvType*.

Calling Syntax
PrintSolvers(solvType)

Description of Inputs

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

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

Examples
See Section 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

solvTypes	A vector of numbers defining which *solvType* to test.
PriLevOpt	Printing level in the solver. Default 2, short information from each iteration.
wait	Set *wait* to 1 if pause after each problem. Default 1.
PriLev	Printing level in *PrintResult*. Default 5, full information.

See Also
runtest

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 31: Menu programs and driver routines.

Function	**Description**
lpOpt	Menu program for LP problems.
lpRun	Driver routine that solves predefined LP problems.
simplex	Interactive input and solution of LP on canonical standard form.

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 *goptions* is used instead of *foptions*.

The OPERA TB routines *lpOpt*, *lpRun*, *lpSolve*, *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

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

or

```
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 diveded into three groups:

1. Numerically stable solvers.
2. Solvers used in teaching courses.
3. Other solvers.

Table 32: Numerically stable solvers for linear programming.

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

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 NLPLIB 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 NLPLIB 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

$$
\begin{array}{cc}
\min\limits_{x} & f(x) = c^T x \\
 & \\
s/t & \begin{array}{lcccl} x_L & \leq & x & \leq & x_U, \\ b_L & \leq & Ax & \leq & b_U \end{array}
\end{array}
\tag{13}
$$

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

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

Function	Description	Section	Page
lpsimp1	The Phase I simplex algorithm. Finds a basic feasible solution (bfs) using artificial variables. Calls *lpsimp2*.	3.5.13	118
lpsimp2	The Phase II revised simplex algorithm with three selection rules.	3.5.14	118
karmark	Karmakar's algorithm. Kanonical form.	3.5.8	114
lpkarma	Solves LP on equality form, by converting and calling *karmark*.	3.5.12	117

Table 34: Other solvers for linear programming.

Function	Description	Section	Page
lpdual	The dual simplex algorithm.	3.5.11	116
akarmark	Affine scaling variant of Karmarkar's algorithm.	3.5.1	108

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. Karmakar's projective algorithm (*karmark*) is developed from the description in Bazaraa et. al. [7, page 386]. There is a choice of update, either according to Bazaraa or the rule by Goldfarb and Todd [29, chap. 9]. The affine scaling variant of Karmakar'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 *lpdual* both have three rules for variable selection implemented. Bland's cycling prevention rule is the choice if fear of cycling exists. There are two variants of minimum reduced cost variable selection, the original Dantzig's rule and one which sorts the variables in increasing order in each step (the default choice). 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] (*TPsimplx*). Three simple algorithms to find a starting basic feasible solution for the transportation problem are included; the northwest corner method (*TPnw*), the minimum cost method (*TPmc*) and Vogel's approximation method (*TPvogel*). The implementation of these algorithms follows the algorithm descriptions in Winston [52, chap. 7.2]. The functions are described in Table 35.

Table 35: Routines for transportation programming.

Function	Description	Section	Page
TPnw	Find initial bfs to TP using the northwest corner method.	3.6.6	131
TPmc	Find initial bfs to TP using the minimum cost method.	3.6.5	131
TPvogel	Find initial bfs to TP using Vogel's approximation method.	3.6.7	132
TPsimplx	Implementation of the transportation simplex algorithm.	3.5.23	126

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 (*NWsimplx*). The implementation is based on Algorithm network simplex in Ahuja et. al. [3, page 415].
- Solve the symmetric traveling salesman problem using Lagrangian relaxation and the subgradient method with the Polyak rule II (*salesman*), an algorithm by Held and Karp [31].

The network programming routines are listed in Table 36.

Table 36: Routines for network programs.

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

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.

Function	Description	Section	Page
cutplane	Cutting plane method using Gomory cuts for mixed-integer programs (MIP).	3.5.3	110
mipSolve	Branch and bound algorithm for mixed-integer programs (MIP).	3.5.17	121
balas	Branch and bound algorithm for binary IP using Balas method.	3.5.2	109

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.

Function	Description	Section	Page
dpinvent	Forward recursion DP algorithm for the inventory problem.	3.5.5	111
dpknap	Forward recursion DP algorithm for the knapsack problem.	3.5.6	112

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 39: Routines for Lagrangian relaxation.

Function	**Description**	**Section**	**Page**
ksrelax	Lagrangian relaxation with knapsack subproblems.	3.5.9	115
urelax	Lagrangian relaxation with knapsack subproblems, plot result.	3.5.25	128

3.1.7 Utility Routines

Table 40 describes the low level test functions and the corresponding setup 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 40: Predefined LP test problems.

Function	**Description**
lp_prob	Initialization of **lp** test problems.
lp_f	Define the objective function for LP, $c^T x$ (for NLP solvers).
lp_g	Define the gradient function for LP, the vector c (for NLP solvers).
lp_H	Define the Hessian matrix for LP, A zero matrix (for NLP solvers).

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

Table 41: Utility routines.

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

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 type 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

$$\begin{array}{cccc} \min\limits_{x_1,x_2} & f(x_1,x_2) & = & -7x_1 - 5x_2 \\ s/t & x_1 + 2x_2 & \leq & 6 \\ & 4x_1 + x_2 & \leq & 12 \\ & x_1, x_2 & \geq & 0 \end{array} \tag{14}$$

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_min* and *x_0* as described in the problem definition description in *lp_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.

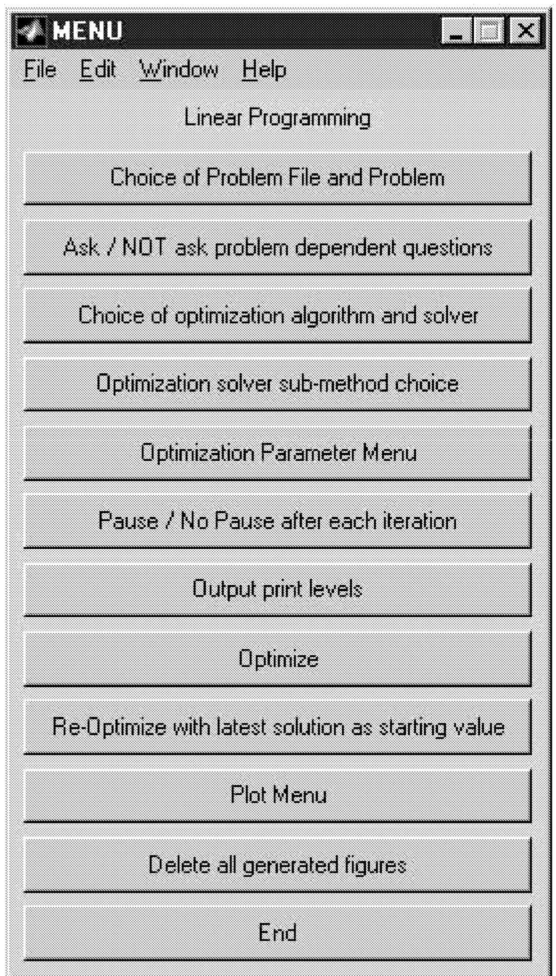


Figure 11: The main menu in *lpOpt*.

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 *lptest1* 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 *lpRun* 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

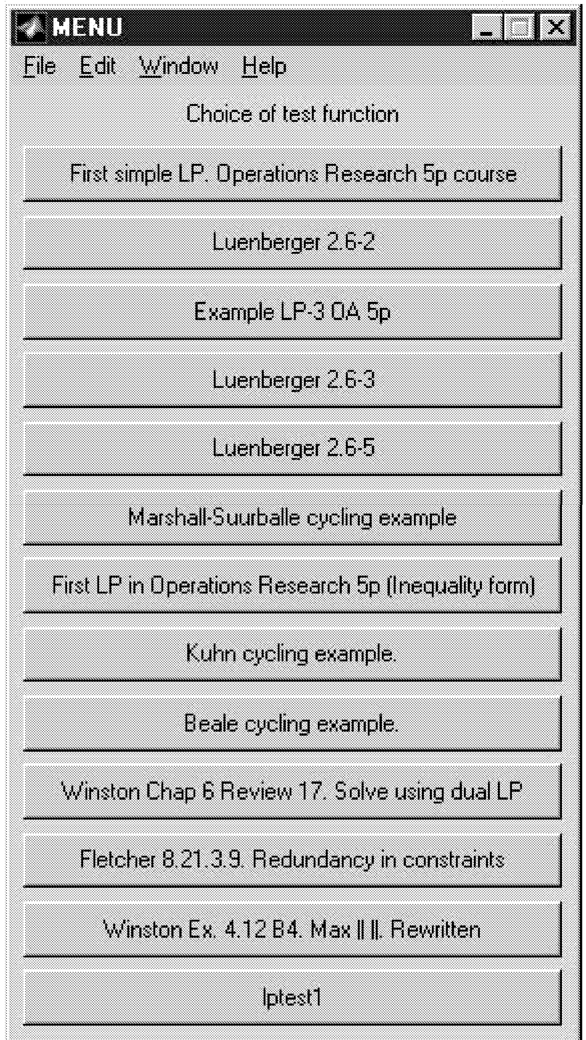


Figure 12: The problem choice menu in *lpOpt*.

straightforward way of doing it (when the problem is defined in *lp_prob.m*) is to give the following call from the Matlab prompt:

```
probNumber = 13;
Result = lpRun([], [], [], [], [], probNumber);
```

The arguments not given to *lpRun* is set to default values, see the *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 *lp*.

Then the call to *lpRun* should be:

```
Solver   = 'lp';
Prob     = probInit('lp_prob',13);
PriLev   = 0;
```

```
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 probType 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);
```

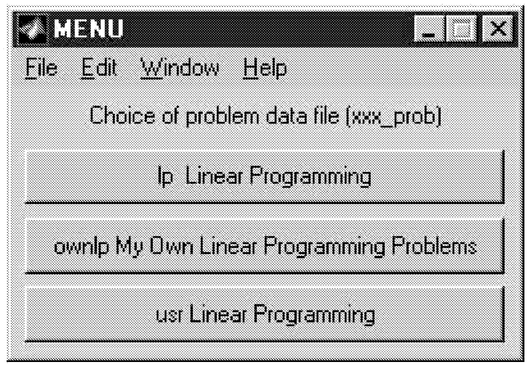


Figure 13: The problem file choice menu in *lpOpt*.

Finally, we will show how you can solve (14) by direct use of the optimization routines *lpsimp1* and *lpsimp2*.

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

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

Table 42: Test examples for linear programming.

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

Table 43: Test examples for linear programming running interior point methods.

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

3.2.2 How to Solve Transportation Programming Problems

We will as an example solve the transportation problem

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

where s is the supply vector, d is the demand vector and C is the cost matrix. See *TPsimplx* Section 3.5.23. Solving (15) by use of the routine *TPsimplx* is done by:

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

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

When neither starting base nor starting point is given as input argument *TPsimplx* calls *TPvogel* (using Vogel's approximation method) to find an initial basic feasible solution (bfs). 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 *TPsimplx*:

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

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

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

For further illustrations of how to solve transportation programming problems see the example files listed in Table 44.

3.2.3 How to Solve Network Programming Problems

In OPERA TB 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.

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

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & -1 & -1 & 0 & 1 \end{pmatrix}, c = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 4 \\ 2 \\ 4 \\ 1 \\ 3 \end{pmatrix} \tag{16}$$

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

$$P = \begin{pmatrix} 1 \\ 3 \\ 4 \\ 6 \\ 8 \\ 9 \end{pmatrix}, Z = \begin{pmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 4 \\ 3 & 2 \\ 3 & 5 \\ 4 & 5 \\ 4 & 3 \\ 5 & 4 \end{pmatrix}, c = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 4 \\ 2 \\ 4 \\ 1 \\ 3 \end{pmatrix}, T = \begin{pmatrix} 1 \\ 4 \\ 2 \\ 7 \\ 3 \\ 8 \\ 5 \\ 6 \end{pmatrix}, R = \begin{pmatrix} 1 \\ 1 \\ 3 \\ 5 \\ 7 \\ 9 \end{pmatrix} \tag{17}$$

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 = [ 1  1  0  0  0  0  0  0
     -1  0  1 -1  0  0  0  0
      0 -1  0  1  1  0 -1  0
      0  0 -1  0  0  1  1 -1
      0  0  0  0 -1 -1  0  1 ];

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

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

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

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

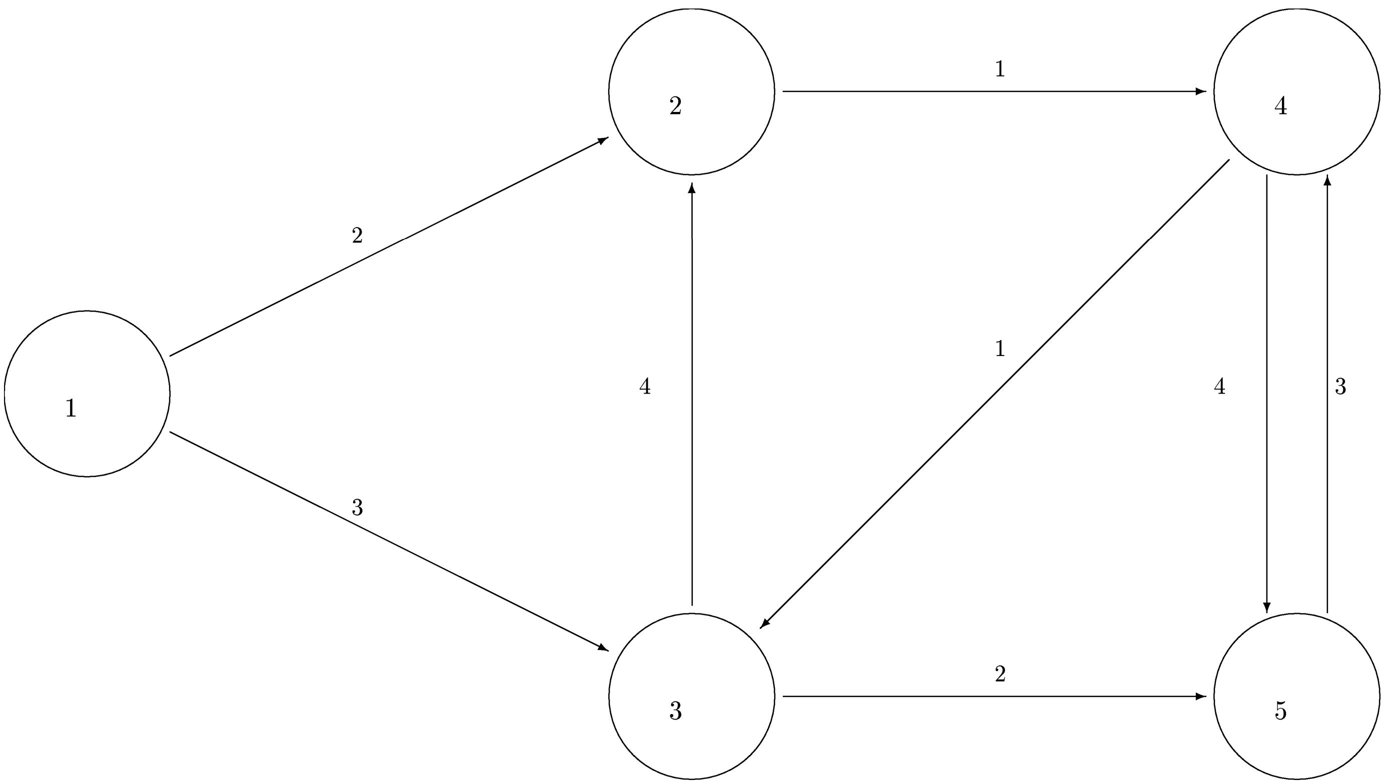


Figure 14: A network example.

requirement of the variables to be positive integers. We have chosen to use the routine *cutplane*, see Section 3.5.3.

```
A = [ 1  2
      4  1 ];
b = [ 6 12 ]';
c = [-7 -5]';
meq = 0;
optPar = lpDef;
optPar(13) = meq;
n_I = 2;

[x, B, optPar, y] = cutplane(A, b, c, optPar, [], [], n_I);
```

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.

Function	Description
exgraph	Testing network routines on simple example.
exflow	Testing several maximum flow examples.
pathflow	Pathological test example for maximum flow problems.
exflow31	Test example N31.
exmcnfp	Minimum Cost Network Flow Problem (MCNFP) example from Ahuja et. al.
ulyss16	Traveling salesman (TSP) example *Odyssey of Ulysses.* Calls *salesman.*
exulys16	TSP example *Odyssey of Ulysses*, 16 nodes. Calls *travelng.*
exulys22	TSP example *Odyssey of Ulysses*, 22 nodes. Calls *travelng.*
exgr96	TSP example *Africa Subproblem*, by Groetschel. 96 nodes. Calls *travelng.*

Table 46: Test examples for integer programming.

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

$$
\begin{array}{ll}
\max\limits_{u} & f(u) = 7u_1 + u_2 + 4u_3 \\
s/t & 2u_1 + 3u_2 + 2u_3 \le 4 \\
 & 0 \le u_1 \le 1 \\
 & 0 \le u_2 \le 1 \\
 & 0 \le u_3 \le 2 \\
 & u_j \in \mathbb{N}, j = 1,2,3
\end{array}
\tag{18}
$$

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

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

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

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

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

```
d   = [1 3 2 4]';     % Demand. N = 4;
P_s = 3;              % Setup cost $3 if u > 0
```

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

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

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

Table 47: Test examples for dynamic programming.

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

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].

$$\begin{array}{ll} \max\limits_{x} & f(x) = 16x_1 + 10x_2 + 4x_4 \\ s/t & 8x_1 + 2x_2 + x_3 + x_4 \leq 10 \\ & x_1 + x_2 \leq 1 \\ & x_3 + x_4 \leq 1 \\ & x_j \in 0/1, j = 1,2,3,4 \end{array} \tag{19}$$

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

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

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

Table 48: Test examples for Lagrangian Relaxation.

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

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^{-5} (avoiding exact zeros) or if the maximal elements are too big, a switch is made to exponential format. The exponential format uses ten characters, displaying two decimals and therefore seven matrix elements are possible to display on each row.

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

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

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 *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. $ask \geq 1$ Ask questions in *probFile*. $ask = [\,]$ If $uP = [\,]$, $ask = -1$, else $ask = 0$.
PriLev	Print level when displaying the result of the optimization in the routine *PrintResult*. See Section 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 *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 *nlplib* 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*, *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 TBwill be described.

3.5.1 akarmark

Purpose

Solve linear programming problems of the form

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

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

Calling Syntax

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

Description of Inputs

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

Description of Outputs

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

Description
The routine *akarmark* is an implementation of the affine scaling variant of Karmarkar's method as described in Bazaraa [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 *exakarma*, *exkarma*, *exkleem2*.

M-files Used
lpDef.m

See Also
lpkarma, *karmark*

3.5.2 balas

Purpose
Solve binary integer linear programming problems.

balas solves problems of the form

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

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

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

Description of Inputs

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

Description of Outputs

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

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

Algorithm
See the code in *balas.m*.

Examples
See *exbalas*.

M-files Used
lpDef.m

See Also
mipSolve, *cutplane*

3.5.3 cutplane

Purpose
Solve mixed integer linear programming problems (MIP).

cutplane solves problems of the form

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

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] = cutplane(A, b, c, optPar, x_0, B_0, n_I, PriLev)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
optPar	Optimization parameter vector, see *goptions.m*.
x_0	Starting point.
B_0	Logical vector of length n for basic variables at start.
n_I	First *n_I* x-values are integer valued.
PriLev	Printing level:
	$PriLev = 0$, no output.
	$PriLev = 1$, output of convergence results.
	$PriLev > 1$, output of each iteration.
	$PriLev > 2$, output of each step in the simplex algorithm.

Description of Outputs

x	Optimal point.
B	Optimal basic set.
optPar	Optimization parameter vector, see *goptions.m*.
y	Lagrange multipliers at the solution.

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

Algorithm
See Appendix B.2.

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

M-files Used
lpDef.m, *Phase1Simplex.m*, *Phase2Simplex.m*, *DualSolve.m*

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

3.5.4 dijkstra

Purpose
Solve the shortest path problem.

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

Description of Inputs

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

Description of Outputs

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

Description

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

Algorithm

See Appendix B.3.

Examples

See *exgraph*, *exflow31*.

See Also

labelcor, *modlabel*

Limitations

dijkstra can only solve problems with nonnegative arc lengths.

3.5.5 dpinvent

Purpose

Solve production/inventory problems of the form

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

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

Calling Syntax

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

Description of Inputs

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

Description of Outputs

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

Description

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

Algorithm

See Appendix B.4.

Examples

See *exinvent*.

3.5.6 dpknap

Purpose

Solve knapsack problems of the form

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

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

Calling Syntax

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

Description of Inputs

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

Description of Outputs

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

Description

dpknap solves knapsack problems using a forward recursion dynamic programming technique as described in [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

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

where $x, x_L, x_U \in \mathbb{R}^n$, $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b_L, b_U \in \mathbb{R}^m$.
by rewriting it into standard form as

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

and solving the dual problem

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

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

Calling Syntax
[Result] = DualSolve(Prob)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	Solver.Alg	Variable selection rule to be used: 0: Minimum reduced cost (default). 1: Bland's anti-cycling rule. 2: Minimum reduced cost. Dantzig's rule.
	QP.B	Active set *B_0* at start: $B(i) = 1$: Include variable $x(i)$ is in basic set. $B(i) = 0$: Variable $x(i)$ is set on its lower bound. $B(i) = -1$: Variable $x(i)$ is set on its upper bound.
	optParam	Structure with special fields for optimization parameters, see Table 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

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

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

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

Calling Syntax

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

Description of Inputs

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

Description of Outputs

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

Description
The routine *karmark* is an implementation of Karmakar's projective algorithm which is of polynomial complexity. The implementation uses the description in Bazaraa [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

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

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

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

Description of Inputs

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

Description of Outputs

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

Description
The routine *ksrelax* uses Lagrangian Relaxation to solve integer linear programming problems with linear inequality constraints and simple bounds on the variables. The problem is solved by relaxing all but one constraint and then solve a simple knapsack problem as a subproblem in each iteration. The algorithm is based on the presentation in Fisher [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
urelax

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	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.

Description of Outputs

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

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

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.

lpdual solves problems of the form

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

by rewriting it into standard form and solving the dual problem

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

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

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

Description of Inputs

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

Description of Outputs

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

Description

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

Algorithm

See B.11.

Examples

See *ex611a2*, *ex6rev17*.

M-files Used

lpDef.m

See Also

lpsimp1, *lpsimp2*

3.5.12 lpkarma

Purpose

Solve linear programming problems of the form

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

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

Calling Syntax

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

Description of Inputs

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

Description of Outputs

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

Description

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

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.

lpsimp1 finds a basic feasible solution to problems of the form

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

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

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

Description of Inputs

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

Description of Outputs

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

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

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.

lpsimp2 solves problems of the form

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

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

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

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
optPar	Optimization parameter vector, see *goptions.m*.
x_0	Starting point, must be a *basic feasible solution*.
B_0	Logical vector of length n for basic variables at start.

Description of Outputs

x	Optimal point.
B	Optimal basic set.
optPar	Optimization parameter vector, see *goptions.m*.
y	Lagrange multipliers.

Description
The routine *lpsimp2* implements the Phase II standard revised Simplex algorithm as formulated in Goldfarb and Todd [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 *exinled*, *excycle*, *excycle1*, *excycle2*, *excycle3*, *exKleeM*, *exfl821*, *ex412b4s*, *expertur*.

M-files Used
lpDef.m

See Also
lpsimp1, *lpdual*

Warnings
No check is done whether the given starting point is feasible or not.

3.5.15 lpSolve

Purpose
Solve general linear programming problems.

lpSolve solves problems of the form

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

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

Calling Syntax
Result = lpSolve(Prob)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	Solver.Alg	Variable selection rule to be used: 0: Minimum reduced cost. 1: Bland's rule (default). 2: Minimum reduced cost. Dantzig's rule.
	QP.B	Active set *B_0* at start: $B(i) = 1$: Include variable $x(i)$ is in basic set. $B(i) = 0$: Variable $x(i)$ is set on its lower bound. $B(i) = -1$: Variable $x(i)$ is set on its upper bound.
	optParam	Structure with special fields for optimization parameters, see Table 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.
	g_k	Gradient value at optimum, *c*.
	x_0	Starting point.
	x_k	Optimal point.
	v_k	Lagrange multipliers.
	xState	State of each variable, described in Table 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 strategi (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 II 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
[max_flow, x] = maxflow(s, t, x_U, P, Z, T, R, PriLev)

Description of Inputs

s	The starting node, the source.
t	The end node, the sink.
P	Pointer vector to start of each node in the matrix Z.
x_U	The capacity on each arc.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
T	Trace vector, points to Z with sorting order Head.
R	Pointer vector in T vector for each node.
PriLev	Printing Level: 0 Silent, 1 Print result (default).

Description of Outputs

max_flow	Maximal flow between node s and node t.
x	The flow on each arc.

Description
maxflow finds the maximum flow between two nodes in a capacitated network using the Ford-Fulkerson augmented path method. The implementation is based on the algorithm description in Luenberger [42, page 144-145].

Algorithm
See Appendix B.15.

Examples
See *exflow*, *exflow31*, *pathflow*.

3.5.17 mipSolve

Purpose
Solve mixed integer linear programming problems (MIP).

mipSolve solves problems of the form

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

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] = mipSolve(A, b, c, optPar, x_0, B_0, n_I, PriLev)

Description of Inputs

A	Constraint matrix.
b	Right hand side vector.
c	Cost vector.
optPar	Optimization parameter vector, see *goptions.m*.
x_0	Starting point.
B_0	Logical vector of length n for basic variables at start.
n_I	First *n_I* x-values are integer valued.
PriLev	Printing level: $PriLev = 0$, no output. $PriLev = 1$, output of convergence results. $PriLev > 1$, output of each iteration. $PriLev > 2$, output of each step in the simplex algorithm.

Description of Outputs

x	Optimal point.
B	Optimal basic set.
optPar	Optimization parameter vector, see *goptions.m*.
y	Lagrange multipliers at the solution.

Description

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

Algorithm

See [45, chap. 8.2] and the code in *mipSolve.m*. *Phase1Simplex* to get the solution x and

Examples

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

M-files Used

lpDef.m, *Phase1Simplex.m*, *Phase2Simplex.m*, *DualSolve.m*

See Also

cutplane, *balas*, *lpsimp1*, *lpsimp2*, *lpdual*

3.5.18 modlabel

Purpose

Solve the shortest path problem.

Calling Syntax

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

Description of Inputs

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

Description of Outputs

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

Description

The implementation of *modlabel* is based on the algorithm MODIFIED LABEL CORRECTING in [2, page 262] with the addition of the heuristic rule discussed to improve running time in practice. The rule says: Add *node* to the beginning of $LIST$ if *node* has been in $LIST$ before, otherwise add *node* at the end of $LIST$. The algorithm belongs to the class of *label correcting* methods which are applicable to networks with arbitrary arc lengths. *modlabel* requires that the network does not contain any negative directed cycle, i.e. a directed cycle whose arc lengths sum to a negative value.

Algorithm

See Appendix B.16.

Examples

See *exgraph*.

See Also

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, xmax, C, S, my, optPar] = NWsimplx(A, b, c, u, optPar)

Description of Inputs

A	Node-arc incidence matrix. A is $m \times n$.
b	Supply/demand vector of length m.
c	Cost vector of length n.
u	Arc capacity vector of length n.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
X	Optimal flow.
xmax	Upper bound on the flow.
C	Costs related to the arcs in the matrix Z.
S	Arc status at the solution: $S_i = 1$, arc i is in the optimal spanning tree. $S_i = 2$, arc i is in L (variable at lower bound). $S_i = 3$, arc i is in U (variable at upper bound).
my	Lagrangian multipliers at the solution.
optPar	Optimization parameter vector, see *goptions.m*.

Description
The implementation of the network simplex algorithm in *NWsimplx* is based on the algorithm NETWORK SIMPLEX in Ahuja et al. [3, page 415]. *NWsimplx* uses the forward and reverse star representation technique of the network, described in [3, pages 35-36].

Algorithm
See [3, page 415] and the code in *NWsimplx.m*.

Examples
See *exmcnfp*.

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

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

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

To obtain this feasible point *Phase1Simplex* solves the following Phase 1 linear programming problem,

$$
\begin{array}{ccccc}
\min\limits_{x,r_1,r_2,s_1,s_2} & f(r_1,r_2) & = & e_1^T r_1 + e_2^T r_2 & \\
s/t & x_L & \leq & x & \leq x_U \\
 & & & Ax + \begin{pmatrix} r_1 \\ s_1 \end{pmatrix} & = b_U \\
 & & & -Ax + \begin{pmatrix} r_2 \\ s_2 \end{pmatrix} & = -b_L \\
 & & & r_1, r_2, s_1, s_2 & \geq 0
\end{array} ,
$$

where $x, x_L, x_U \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$ and $b_L, b_U \in \mathbb{R}^m$. $r_1 \in \mathbb{R}^{m_1}$, $r_2 \in \mathbb{R}^{m_2}$, $s_1 \in \mathbb{R}^{m_3}$, $s_2 \in \mathbb{R}^{m_4}$, 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

Prob	Problem description structure. The following fields are used:	
	Solver.Alg	Variable selection rule used in *Phase1Simplex*: 0: Minimum reduced cost (default). 1: Bland's anti-cycling rule. 2: Minimum reduced cost. Dantzig's rule.
	optParam	Structure with special fields for optimization parameters, see Table 6. Fields used are: *PriLev*. *Phase1Simplex* is also using *MaxIter*, *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.

Description of Outputs

Result	Structure with result from optimization. The following fields are changed:	
	QP.B	The n first elements in the optimal active set. $B(i) = 1$: 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.
	ExitFlag	Exit flag from *Phase1Simplex*: 0: OK. 1: Feasible region is empty. Some nonzero artificial variables left in the base.
	Inform	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*. $Inform = Inform + 100$ if any artificial variable still in base but on zero.
	x_0	The full starting point.
	x_k	The first n variables in the solution x.
	v_k	The full set of Lagrange multipliers for the linear constraints.
	Prob	Problem structure for the Phase1 problem solved.
	Prob.x_k	The full solution vector x for the Phase 1 problem.
	Prob.QP.B	The full optimal active set for the Phase 1 problem.

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

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

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

Calling Syntax
Result = 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 *travelng* 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 TPsimplx

Purpose
Solve transportation programming problems.

TPsimplx solves problems of the form

$$
\begin{array}{llcll}
\min\limits_{x} & f(x) & = & \sum\limits_{i}^{m}\sum\limits_{j}^{n} c_{ij}x_{ij} & \\
s/t & \sum\limits_{j}^{n} x_{ij} & = & s_i & i = 1, 2, ..., m \\
 & \sum\limits_{i}^{n} x_{ij} & = & d_j & j = 1, 2, ..., n \\
 & x & \geq & 0 &
\end{array}
$$

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

Calling Syntax
[X, B, optPar, y, C] = TPsimplx(s, d, C, X, B, optPar, Penalty)

Description of Inputs

s	Supply vector.
d	Demand vector.
C	The cost matrix of linear objective function coefficients.
X	Basic Feasible Solution matrix.
B	Index (i, j) of basis found.
optPar	Optimization parameter vector, see *goptions.m*.
Penalty	If the problem is unbalanced with $\sum_i^m s_i < \sum_j^n d_j$, a dummy supply point is added with cost vector *Penalty*. If the length of *Penalty* $< n$ then the value of the first element in *Penalty* is used for the whole added cost vector. Default: Computed as $10\max(C_{ij})$.

Description of Outputs

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

Description
The routine *TPsimplx* is an implementation of the Transportation Simplex method described in Luenberger [42, chap 5.4]. In OPERA TB, three routines to find a starting basic feasible solution for the transportation problem are included; the Northwest Corner method (*TPnw*), the Minimum Cost method (*TPmc*) and Vogel's approximation method (*TPvogel*). If calling *TPsimplx* without giving a starting point then Vogel's method is used to find a starting basic feasible solution.

Algorithm
See Appendix B.20.

Examples
See *extp_bfs*, *exlu119*, *exlu119U*, *extp*.

M-files Used
TPvogel.m

See Also
TPmc, *TPnw*, *TPvogel*

Warnings
No check is done whether the given starting point is feasible or not.

3.5.24 traveling

Purpose
Solve the symmetric travelling salesman problem.

Calling Syntax
[BestTour, f_BestTour] = travelng(Z, c, 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.
optPar	Optimization parameter vector, see *goptions.m*.

Description of Outputs

BestTour	Arc list of the best tour found.
f_Besttour	Cost (total distance) of the best tour found.

Description
The routine *traveling* 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 *exgr96*, *exulys16*, *exulys22*.

M-files Used
salesman.m

See Also
salesman

3.5.25 urelax

Purpose
Solve integer linear problems of the form

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

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

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

Description of Inputs

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

Description of Outputs

x_P	Primal solution.
u	Lagrangian multipliers.
f_P	Function value at *x_P*.

Description
The routine *urelax* is a simple example of the use of Lagrangian Relaxation to solve integer linear programming problems. The problem is solved by relaxing all but one constraint and then solve a simple knapsack problem as a subproblem in each iteration. *urelax* plots the result of each iteration. OPERA TB also contains a more sophisticated routine, *ksrelax*, for solving problems of this type.

Algorithm
See Appendix B.22.

Examples
See *exip39rx*.

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] = a2frstar(A, C, U)

Description of Inputs

A	The node-arc incidence matrix. A is $m \times n$, where m is the number of arcs and n is the number of nodes.
C	Cost for each arc, n-vector.
U	Upper bounds on flow (optional).

Description of Outputs

P	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.
T	Trace vector, points to Z with sorting order Head.
R	Rewerse pointer vector in T for each node.
u	Upper bounds on flow if U is given as input, else infinity.

Description
The routine *a2frstar* converts a node-arc incidence matrix representation of a network to the forward and reverse star data storage representation as described in Ahuja et.al. [3, pages 35-36].

Examples
See *exflow*, *exflow31*, *exgraph*, *pathflow*.

3.6.2 gsearch

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

Calling Syntax
[pred, mark] = gsearch(s, P, Z, c)

Description of Inputs

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

Description of Outputs

pred	$pred(j)$ = Predecessor of node j.
mark	If $mark(j) = 1$ the node is reachable from node s.

Description
gsearch is searching for all nodes in a network which is reachable from the given source node s. The implementation is a variation of the Algorithm SEARCH in [2, pages 231-233]. The algorithm uses a depth-first search which means that it creates a path as long as possible and backs up one node to initiate a new probe when it can mark no new nodes from the tip of the path. A stack approach is used where nodes are selected from the front and added to the front.

Algorithm
See Appendix B.6.

Examples
See *exgraph*.

See Also
gsearchq

3.6.3 gsearchq

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

Calling Syntax
[pred, mark] = gsearchq(s, P, Z, c)

Description of Inputs

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

Description of Outputs

pred	$pred(j)$ = Predecessor of node j.
mark	If $mark(j) = 1$ the node is reachable from node s.

Description
gsearchq is searching for all nodes in a network which is reachable from the given source node s. The implementation is a variation of the Algorithm SEARCH in [2, pages 231-233]. The algorithm uses a breadth-first search which means that it visits the nodes in order of increasing distance from s. The distance being the minimum number of arcs in a directed path from s. A queue approach is used where nodes are selected from the front and added to the rear.

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 $n \times n$ where $C_{ij} = C_{ji}$ is the cost of arc (i,j). If there are no arc between node i and node j then set $C_{ij} = C_{ji} = \infty$. It must hold that $C_{ii} = NaN$.
Zin	List of arcs which should be forced to be included in *Z_tree*.
Zout	List of arcs which should not be allowed to be included in *Z_tree* (could also be given as NaN in C).

Description of Outputs

Z_tree	List of arcs in the minimum spanning tree.
cost	The total cost.

Description
mintree is an implementation of Kruskal's algorithm for finding a minimal spanning tree of an undirected graph. The implementation follows the algorithm description in [3, page 520-521]. It is possible to give as input, a list

of those arcs which should be forced to be included in the tree as well as a list of those arcs which should not be allowed to be included in the tree. *mintree* is called by *salesman*.

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 *extp_bfs*, *exlu119*, *exlu119U*, *extp*.

See Also
TPnw, *TPvogel*, *TPsimplx*

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 *extp_bfs*, *exlu119*, *exlu119U*, *extp*.

See Also
TPmc, *TPvogel*, *TPsimplx*

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*, *exlu119*, *exlu119U*, *extp*.

See Also
TPmc, *TPnw*, *TPsimplx*

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 $n \times 2$, where n is the number of arcs. The number of nodes m is set equal to the greatest element in Z.
C	Cost for each arc, n-vector.
U	Upper bounds on flow (optional).

Description of Outputs

P	Pointer vector to start of each node in the matrix Z.
Z	Arcs outgoing from the nodes in increasing order. $Z(:,1)$ Tail. $Z(:,2)$ Head.
c	Costs related to the arcs in the matrix Z.
T	Trace vector, points to Z with sorting order Head.
R	Rewerse pointer vector in T for each node.
u	Upper bounds on flow if U is given as input, else infinity.

Description
The routine *z2frstar* converts a table of arcs and corresponding costs in a network to the forward and reverse star data storage representation as described in Ahuja et.al. [3, pages 35-36].

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

$$\begin{array}{llllll} \min\limits_x & f(x) & & & & \\ s/t & x_L & \leq & x & \leq & x_U \\ & b_L & \leq & Ax & \leq & b_U \end{array}$$

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

Calling Syntax
[AA, bb, meq] = cpTransf(Prob, TransfType, makeEQ, LowInf)

Description of Inputs

Prob	Problem description structure. The following fields are used:	
	QP.c	Constant vector c in $c^T x$.
	A	Constraint matrix for linear constraints.
	b_L	Lower bounds on the linear constraints.
	b_U	Upper bounds on the linear constraints.
	x_L	Lower bounds on the variables.
	x_U	Upper bounds on the variables.
TransfType	Type of transformation, see the description below.	
MakeEQ	Flag, if set true, make standard form (all equalities).	
LowInf	Variables equal to $-Inf$ or variables $< LowInf$ are set to $LowInf$ before transforming the problem. Default -10^{-4}. $\vert LowInf\vert$ are limit if upper bound variables are to be used.	

Description of Outputs

AA	The expanded linear constraint matrix.
bb	The expanded upper bounds for the linear constraints.
meq	The first *meq* equations are equalities.

Description
If $TransType = 1$ the program is transformed into the form

$$\begin{array}{lll} \min\limits_x & f(x - x_L) & \\ s/t & AA(x - x_L) & \leq \; bb \\ & x - x_L & \geq \; 0 \end{array}$$

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

```
x(~x_L==x_U) = (x-x_L) + x_L(~x_L==x_U)
```

If $TransType = 2$ the program is transformed into the form

$$\begin{array}{llllll} \min\limits_x & f(x) & & & & \\ s/t & & & AA(x) & \leq & bb \\ & x_L & \leq & x & \leq & x_U \end{array}$$

where the first *meq* constraints are equalities.

If $TransType = 3$ the program is transformed into the form

$$\begin{array}{llll} \min\limits_x & f(x) & & \\ s/t & AAx & \leq & bb \\ & x & \geq & x_L \end{array}$$

where the first *meq* constraints are equalities.

4 Interfaces

4.1 The MEX-file Interface

TOMLAB is an open system with possibilities to interact with other program packages. An optimization solver implemented in Fortran or C is called from TOMLAB 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 TOMLAB.

Presently, MEX-file interfaces has been developed for six general-purpose solvers available from the Systems Optimization Laboratory, Department of Operations Research, Stanford University, Carlifornia; 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 TOMLAB 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 TOMLAB 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_fc* supplies. Also the gradient vector and the matrix of constraint normals should be supplied in one call. These parameters are returned by the routine *nlp_gdc*.

OPTIM is using a parameter vector OPTIONS of length 18, that the routine *foptions* is setting up the default values for. TOMLAB 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 TOMLAB.

Table 49: Matlab Optimization toolbox routines with a TOMLAB interface.

Function	**Type of problem solved**
constr	Constrained minimization.
leastsq	Nonlinear least squares.
fmins	Unconstrained minimization using Nelder-Mead type simplex search method.
fminu	Unconstrained minimization using gradient search.
lp	Linear programming.
qp	Quadratic programming.
nnls	Nonnegative linear least squares (no license needed).
conls	Constrained linear least squares.

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; *outsdif.dat* or *outsdif.d.* The Fortran files are compiled and linked together with the CUTE library and a solver routine. Running the binary executable, the problem is solved using the current solver. During the solution procedure, the ASCII data file *outsdif.dat* or *outsdif.d* is read.

With the CUTE distribution follows a Matlab interface. There are one gateway routine, *ctools.f*, for constrained CUTE problems, and one gateway routine, *utools.f*, for unconstrained problems. These routines are using the Matlab MEX-file interface for communication between Matlab and the compiled Fortran (or C) code. The gateway routine is compiled and linked together with the Fortran files, generated by the SIF decoder, and the Matlab MEX library to make a DLL (Dynamic Link Library) file. At run-time, Matlab calls the DLL, which will read the CUTE ASCII data file for the problem specific information. Also included in the CUTE distribution is a set of Matlab m-files that calls the gateway routine.

For the TOMLAB CUTE interface we assume that the DLLs are already built and stored in any of four predefined directories; *cutedll* for constrained problems, *cutebig* for large constrained problems, *cuteudll* for unconstrained problems, *cuteubig* for large unconstrained problems. The name of the dll is the problem name used by CUTE, e.g. *rosenbr.dll* for the Rosenbrock banana function. The ASCII data file also has a unique name, e.g. *rosenbr.dat*. The CUTE Matlab interface assumes the DLLs to be named ctools.dll and utools.dll (and the data file to be called outsdif.dat on PC). TOMLAB calls the Matlab files in the CUTE distribution, but to solve the name problem, using the m-files *ctools.m* and *utools.m* to make a call to the correct DLL file. The ASCII data file is also copied to a temporary file, with the necessary filename *outsdif.dat*, before executing the DLL.

When using the TOMLAB interface, the user either gets a menu of all DLLs in the CUTE directory chosen, or directly makes a choice of which problem to solve. 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

$$\begin{array}{cc} \min\limits_{x} & f(x) = \frac{1}{2} r^T(x) r(x) \\ s/t & a_i^T x = b_i \ , \ \ i \in E \\ & a_i^T x \geq b_i \ , \ \ i \in I \\ & x_L \leq x \leq x_U \end{array}$$

where E is the set of linear equalities and I the set of linear inequalities.
Set $k = 0$, *stop* $= 0$ and the number of consequtive zero steps, $\alpha_0 = 0$.
Set $GN_{flag} = 1$ and $A_H = I$.
Set $x^{(-1)} = \infty$, $f^{(-1)} = \infty$, $\alpha = 1$ and $\alpha_{max} = 10^{20}$.
Set $\tilde{x}_i = \max(x_i^{(0)}, x_{L_i})$ and $x_i^{(0)} = \min(\tilde{x}_i, x_{U_i})$, $i = 1, 2, ..., n$.
if $pSolve$ **then**
  Call the presolve analysis routine *preSolve*.
  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:

$$\begin{array}{cc} \min\limits_{\tilde{x}} & f(\tilde{x}) = \frac{1}{2} \tilde{x}^T B \tilde{x} - x^{(0)T} B \tilde{x} \\ s/t & a_i^T \tilde{x} = b_i \ , \ \ i \in E \\ & a_i^T \tilde{x} \geq b_i \ , \ \ i \in I \\ & x_L \leq \tilde{x} \leq x_U \end{array}$$

  where $B = diag\left(\frac{1}{\max(10^{-6}, (x_i^{(0)})^2)}\right)$ 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, $i = 1, 2, ..., n$ **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 nr_{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 λ.
    $\lambda_i = -g_i$ 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
else
if $k = 0$ **then**
Set up the initial constraint working set $W = E \cup \{i : i \in I \wedge a_i^T x = b_i\}$.
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 λ by solving the overdetermined system

$$\begin{pmatrix} A_{W^{(k)}} \\ e_i \ , \quad i \in V_L^{(k)} \\ -e_i \ , \quad i \in V_U^{(k)} \end{pmatrix}^T \lambda = g^{(k)}$$

where e_i is the ith unit row vector.
Set $\lambda_{min} = \min(\lambda)$
if $\lambda_{min} < -10^{-8}$ **then**
if only one variable is allowed to be released **then**
if λ_{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 λ_{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
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 $\tilde{A}_W = A_{ij} : i \in W \wedge j \notin V_L \cup V_U$
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 $stop = 1$.
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 $p_{full_i} = p_i$ if $i \notin V_L \cup V_U$ else set $p_{full_i} = 0$.
Compute α_1, step length estimate sent to line search routine.
if $k = 0$ **or** $\|p\| = 0$ **then**
  Set $\tilde{\alpha}_1 = 1$.
else
  Set $\tilde{\alpha}_1 = \min\left(1, -2\frac{\max\left(f^{(k-1)} - f^{(k)}, 10\epsilon_x\right)}{\tilde{g}^T p = 0}\right)$, where $\tilde{g}_i = g_i : i \notin V_L \cup V_U$.
  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 α_{max}, the maximum step α 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 $\min\limits_{0 < \alpha \le \alpha_{max}} f(x + \alpha p_{full})$.
  end if
  if $\alpha = \alpha_{max}$ **then**
    if α_{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
Set $k = k + 1$.
end while

A.1.1 Convergence criterias

- $\max\limits_i \frac{\left|x_i^{(k)} - x_i^{(k-1)}\right|}{\max\left(\left|x_i^{(k)}\right|, size_x\right)} \le \epsilon_x$ **and** $\alpha_{max} > 10^{-14}$

- $\max\left(\left|Z^T\tilde{g}^{(k)}\right| \max\left(\max_{i\notin V_L\cup V_U}\left|x_i^{(k)}\right|, size_x\right)\right) \leq \epsilon_g \max\left(abs(f^{(k)}), size_f\right)$
- $f^{(k)} \leq \epsilon_{absf} size_f$
- Relative function value reduction low for *LowIts* iterations.

A.1.2 Stop criterias

- $k \geq MaxIter$
- $f^{(k)} \leq f_{Low}$

A.1.3 Computation of Search Direction

Gauss-Newton or hybrid method if $GN_{flag} = 1$

Solve the overdetermined system $\tilde{J}Zp = -r$ 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 $Z^T\tilde{H}Zp = -Z^T\tilde{g}$ using Singular Value Decomposition with rank estimation and a subspace minimization technique.
$\tilde{H}_{ij} = H_{ij} : i, j \notin V_L \cup V_U$.

A.1.4 Update Procedure

Fletcher-Xu

Set $z = \alpha p_{full}$.
if $f^{(k)} - f^{(k+1)} \geq 0.2 f^{(k)}$ **or** $\|z\| \leq \epsilon_x$ **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 $z^T y < 0.01 z^T \left(g^{(k+1)} - g^{(k)}\right)$ **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 $H^{(k+1)} = H^{(k)} + \frac{ww^T}{z^T w} - \frac{H^{(k)} z z^T H^{(k)^T}}{z^T H^{(k)} z}$.
  end if
end if

Al-Baali-Fletcher

Set $z = \alpha p_{full}$.
if $f^{(k)} - f^{(k+1)} \geq 0.2 f^{(k)}$ **or** $\|z\| \leq \epsilon_x$ **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 $z^T y < 0.2 z^T H^{(k)} z$ **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 $GN_{flag} = 1$.
Set $H^{(k+1)} = J^{(k+1)^T} J^{(k+1)}$.
else
Set $H^{(k+1)} = H^{(k)} + \frac{w w^T}{z^T w} - \frac{H^{(k)} z z^T H^{(k)^T}}{z^T H^{(k)} z}$.
end if
end if

Huschens TSSM

Set:
$GN_{flag} = 0$,
$z = p_{full}$,
$y^\sharp = \left(J^{(k+1)} - J^{(k)} \right)^T \frac{r^{(k+1)}}{\left\| r^{(k)} \right\|}$,
$y = J^{(k+1)^T} J^{(k+1)} z + \left\| r^{(k+1)} \right\| y^\sharp$ and
$B_s = J^{(k+1)^T} J^{(k+1)} + \left\| r^{(k+1)} \right\| A_H^{(k)}$.
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 $A_H^{(k+1)} = A_H^{(k)} + \frac{\left(y^\sharp - A_H^{(k)} z \right) v^T + v \left(y^\sharp - A_H^{(k)} z \right)^T}{v^T z} - \frac{\left(y^\sharp - A_H^{(k)} z \right)^T z v v^T}{(v^T s)^2}$.
Set $H^{(k+1)} = H^{(k)} + \left\| r^{(k+1)} \right\| A_H^{(k+1)}$.

A.2 glbSolve

Set the global/local search weight parameter ϵ.
Set $C_{i1} = \frac{1}{2}$ and $L_{i1} = \frac{1}{2}$, $i = 1, 2, 3, ..., n$.
Set $F_1 = f(x)$, where $x_i = x_{L_i} + C_{i1} (x_{U_i} - x_{L_i})$, $i = 1, 2, 3, ..., n$.
Set $D_1 = \sqrt{\sum_{k=1}^{n} L_k^2}$.
Set $f_{min} = F_1$ and $i_{min} = 1$.
for $t = 1, 2, 3, ..., T$ **do**
Set $\hat{S} = \left\{ j : D_j \geq D_{i_{min}} \wedge F_j = \min_i \{ F_i : D_i = D_j \} \right\}$.
Define α and β by letting the line $y = \alpha x' + \beta$ pass through the points $(D_{i_{min}}, F_{i_{min}})$ and $\left(\max_j (D_j), \min_i \left(F_i : D_i = \max_j (D_j) \right) \right)$.
Let $\tilde{S}$ be the set of all rectangles $j \in \hat{S}$ fullfilling $F_j \leq \alpha D_j + \beta + 10^{-12}$.
Let S be the set of all rectangles in $\tilde{S}$ which lies on the convex hull defined by the points (D_j, F_j), $j \in \tilde{S}$.
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 δ 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 $\tilde{c} = c - \delta e_i$, where e_i is the ith unit vector.
Compute $\hat{f} = f(\hat{x})$ and $\tilde{f} = f(\tilde{x})$ where $\hat{x}_k = x_{L_k} + \hat{c}_k\,(x_{U_k} - x_{L_k})$ and $\tilde{x}_k = x_{L_k} + \tilde{c}_k\,(x_{U_k} - x_{L_k})$.
Set $w_i = \min(\hat{f}, \tilde{f})$.
Set $C = \begin{pmatrix} C & \hat{c} & \tilde{c} \end{pmatrix}$ and $F = \begin{pmatrix} F & \hat{f} & \tilde{f} \end{pmatrix}$.
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 $\hat{j}$ and $\tilde{j}$ be the indices that corresponds to the points $\hat{c}$ and $\tilde{c}$ above.
Set $L_{k\hat{j}} = L_{kj}$ and $L_{k\tilde{j}} = L_{kj}$, $k = 1, 2, 3, ..., n$.
Set $D_j = \sqrt{\sum_{k=1}^{n} L_{kj}^2}$.
Set $D_{\hat{j}} = D_j$ and $D_{\tilde{j}} = D_j$.
end while
end while
Set $f_{min} = \min_j(F_j)$.
Set $i_{min} = \text{argmin}\left(\frac{F_j - f_{min} + E}{D_j}\right)$, where $E = \max\left(\epsilon|f_{min}|, 10^{-8}\right)$.
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 $START = 1$, $v = START$, $w = m$ and $flag = 0$.
while $next(v) \neq START$ **or** $flag = 0$ **do**
if $next(v) = w$ **then**
Set $flag = 0$.
end if
Set $a = v$, $b = next(v)$ and $c = next(next(v))$.
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 $v = next(v)$.
else
Set $j = next(v)$.
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 $v = pred(v)$.
end if
end while
end if

A.2.2 next

if $v = m$ **then**
Set $i = 1$.
else

Set $i = v + 1$.
end if

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 $\tilde{g}_0 = g_0 (x_1 - x_0)$.
Set $c = f_1 - f_0 - \tilde{g}_0$.
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 $z_{min} = -\frac{\tilde{g}_0}{2c}$.
if $q (z_{min}) \leq q (\tilde{a})$ **and** $z_{min} \in [\tilde{a}, \tilde{b}]$ **then**
if $q (z_{min}) \leq q(\tilde{b})$ **then**
Set $\alpha = x_0 + z_{min} (x_1 - x_0)$.
else
Set $\alpha = b$.
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 $\tilde{g}_0 = g_0 (x_1 - x_0)$ and $\tilde{g}_1 = g_1 (x_1 - x_0)$.
Set $r = 3 (f_1 - f_0) - 2\tilde{g}_0 - \tilde{g}_1$.
Set $s = \tilde{g}_0 + \tilde{g}_1 - 2 (f_1 - f_0)$.
if $|s| < 10^{-12} |r|$ **then**
Call quadratic ineterpolation routine *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 $z_1 = \frac{-r+\sqrt{r^2-3s\tilde{g}_0}}{3s}$ and $z_2 = \frac{-r-\sqrt{r^2-3s\tilde{g}_0}}{3s}$.
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 $z_{min} = z_1$.
else
Set $z_{min} = z_2$.
end if
end if
else

Set $z_{min} = z_2$.
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 $\alpha = b$
end if
end if
end if

A.5 LineSearch

A.5.1 Bracketing Phase

Set $\alpha^{(0)} = 0$.
if $f'(0) = 0$ **then**
Set $\mu = \alpha_{max}$.
else
Set $\mu = \min\left(\alpha_{max}, \frac{f_{Low} - f(0)}{\rho f'(0)}\right)$.
end if
if $\mu < 0$ **then**
Set $\mu = \alpha_{max}$.
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 $k = 1, 2, 3, ...$ **do**
if $f\left(\alpha^{(k)}\right) \leq f_{Low}$ **then**
Terminate Line Search.
end if
if $f\left(\alpha^{(k)}\right) \geq f\left(\alpha^{(k-1)}\right)$ **or** $f\left(\alpha^{(k)}\right) > f(0) + \alpha^{(k)} \rho f'(0)$ **then**
if $f\left(\alpha^{(k)}\right) = f\left(\alpha^{(k-1)}\right)$ **and** $\|p\| \leq 10^{-8}$ **then**
Terminate Line Search.
end if
Set $a^{(k)} = \alpha^{(k-1)}$ and $b^{(k)} = \alpha^{(k)}$.
Set $\tilde{k} = k$.
Go to Sectioning Phase.
end if
if $\left|f'\left(\alpha^{(k)}\right)\right| \leq -\sigma f'(0)$ **then**
Set $\alpha^{(k)} = 0$.
Terminate Line Search.
end if

if $f'\left(\alpha^{(k)}\right) \geq 0$ **then**
Set $a^{(k)} = \alpha^{(k)}$ and $b^{(k)} = \alpha^{(k-1)}$.
Set $\tilde{k} = k$.
Go to Sectioning Phase.
end if
if $\mu \leq 2\alpha^{(k)} - \alpha^{(k-1)}$ **then**
Set $\alpha^{(k+1)} = \mu$.
else
Choose $\alpha^{(k+1)} \in \left[2\alpha^{(k)} - \alpha^{(k)-1)}, \min\left(\mu, \alpha^{(k)} + \tau_1\left(\alpha^{(k)} - \alpha^{(k)-1)}\right)\right)\right]$ 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 $\left|a^{(k)} - b^{(k)}\right| \leq \epsilon_1$ **then**
if $f\left(a^{(k)}\right) < f\left(b^{(k)}\right)$ **or** $f\left(a^{(k)}\right) = f\left(b^{(k)}\right) \wedge a^{(k)} > b^{(k)}$ **then**
Set $\alpha^{(k)} = a^{(k)}$.
else
Set $\alpha^{(k)} = b^{(k)}$.
end if
Terminate Line Search.
end if
Choose $\alpha^{(k)} \in \left[a^{(k)} + \tau_2\left(b^{(k)} - a^{(k)}\right), b^{(k)} - \tau_3\left(b^{(k)} - a^{(k)}\right)\right]$ using quadratic or cubic interpolation. See 2.12.1 and 2.12.2 respectively.
if $\left(a^{(k)} - \alpha^{(k)}\right) f'\left(a^{(k)}\right) \leq \epsilon_2$ **then**
Set $\alpha^{(k)} = a^{(k)}$.
Terminate Line Search.
end if
if $f\left(\alpha^{(k)}\right) > f(0) + \rho\alpha^{(k)} f'(0)$ **or** $f\left(\alpha^{(k)}\right) \geq f\left(a^{(k)}\right)$ **then**
Set $a^{(k+1)} = a^{(k)}$ and $b^{(k+1)} = \alpha^{(k)}$.
else
if $\left|f'\left(\alpha^{(k)}\right)\right| \leq -\sigma f'(0)$ **then**
Terminate Line Search.
end if
Set $a^{(k+1)} = \alpha^{(k)}$.
if $\left(b^{(k)} - a^{(k)}\right) f'\left(\alpha^{(k)}\right) \geq 0$ **then**
Set $b^{(k+1)} = a^{(k)}$.
else
Set $b^{(k+1)} = b^{(k)}$.
end if
end if
end for

A.6 lsSolve

Set $k = 0$, $stop = 0$ and the number of consequtive zero steps, $\alpha_0 = 0$.
Set $GN_{flag} = 1$ and $A_H = I$.
Set $x^{(-1)} = \infty$, $f^{(-1)} = \infty$ and $\alpha = 1$.
Set $\tilde{x}_i = \max(x_i^{(0)}, x_{L_i})$ and $x_i^{(0)} = \min(\tilde{x}_i, x_{U_i})$, $i = 1, 2, \ldots, n$.
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 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 $H^{(k)} = J^{(k)^T} J^{(k)}$.
end if
Compute first order Lagrange multiplier estimate λ.
$\lambda_i = -g_i$ 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 criterias, 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 criterias, see A.6.2.
if any stop criteria are fullfilled **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 α_1, step length estimate sent to line search routine.
if $k = 0$ **then**
  Set $\tilde{\alpha}_1 = 1$.
else
  Set $\tilde{\alpha}_1 = \min\left(1, -2\frac{\max\left(f^{(k-1)} - f^{(k)}, 10\epsilon_x\right)}{\tilde{g}^T p=0}\right)$, where $\tilde{g}_i = g_i : i \notin V_L \cup V_U$.

if $\tilde{\alpha}_1 < 0$ **then**
Change sign on $\tilde{\alpha}_1$, p_{full} and p.
end if
end if
Set $\alpha_1 = \max(0.5, \tilde{\alpha}_1)$.
Compute α_{max}, the maximum step α such that $x + \alpha p_{full}$ is feasible with respect to the variable bounds.
if $\alpha_{max} < 10^{-14}$ **then**
Set $\alpha = 0$.
else
Solve the line search problem $\min\limits_{0<\alpha\leq\alpha_{max}} f(x + \alpha p_{full})$.
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.
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

- $\max\limits_i \frac{\left|x_i^{(k)} - x_i^{(k-1)}\right|}{\max\left(\left|x_i^{(k)}\right|, size_x\right)} \leq \epsilon_x$
- $\max\limits_{i \notin V_L \cup V_U} \left(\left|g_i^{(k)}\right| \max\left(\left|x_i^{(k)}\right|, size_x\right)\right) \leq \epsilon_g \max\left(\left|f^{(k)}\right|, size_f\right)$
- $f^{(k)} \leq \epsilon_{absf} size_f$
- Relative function value reduction low for *LowIts* iterations.

A.6.2 Stop criterias

- $k \geq MaxIter$
- $f^{(k)} \leq f_{Low}$

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 $\tilde{H}p = -\tilde{g}$ using Singular Value Decomposition with rank estimation and a subspace minimization technique.
$\tilde{H}_{ij} = H_{ij} : i, j \notin V_L \cup V_U$.

A.6.4 Update Procedure

Fletcher-Xu

Set $z = \alpha p_{full}$.

if $f^{(k)} - f^{(k+1)} \geq 0.2 f^{(k)}$ **or** $\|z\| \leq \epsilon_x$ **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 $z^T y < 0.01 z^T \left(g^{(k+1)} - g^{(k)}\right)$ **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 $H^{(k+1)} = H^{(k)} + \frac{ww^T}{z^T w} - \frac{H^{(k)} z z^T H^{(k)^T}}{z^T H^{(k)} z}$.
end if
end if

Al-Baali-Fletcher

Set $z = \alpha p_{full}$.
if $f^{(k)} - f^{(k+1)} \geq 0.2 f^{(k)}$ **or** $\|z\| \leq \epsilon_x$ **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 $z^T y < 0.2 z^T H^{(k)} z$ **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 $GN_{flag} = 1$.
Set $H^{(k+1)} = J^{(k+1)^T} J^{(k+1)}$.
else
Set $H^{(k+1)} = H^{(k)} + \frac{ww^T}{z^T w} - \frac{H^{(k)} z z^T H^{(k)^T}}{z^T H^{(k)} z}$.
end if
end if

Huschens TSSM

Set:
$GN_{flag} = 0$,
$z = p_{full}$,
$y^\sharp = \left(J^{(k+1)} - J^{(k)}\right)^T \frac{r^{(k+1)}}{\left\|r^{(k)}\right\|}$,
$y = J^{(k+1)^T} J^{(k+1)} z + \left\|r^{(k+1)}\right\| y^\sharp$ and
$B_s = J^{(k+1)^T} J^{(k+1)} + \left\|r^{(k+1)}\right\| A_H^{(k)}$.
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 $A_H^{(k+1)} = A_H^{(k)} + \frac{\left(y^\sharp - A_H^{(k)} z\right) v^T + v \left(y^\sharp - A_H^{(k)} z\right)^T}{v^T z} - \frac{\left(y^\sharp - A_H^{(k)} z\right)^T z v v^T}{(v^T s)^2}$.
Set $H^{(k+1)} = H^{(k)} + \left\| r^{(k+1)} \right\| A_H^{(k+1)}$.

A.7 ucSolve

Set $k = 0$, $stop = 0$ and the number of consequtive zero steps, $\alpha_0 = 0$.
Set $\beta = 0$, $cg_{step} = 1$ and $cg_{restart} = 1$ if restart teqnique shall be used else set $cg_{restart} = 0$.
Set $x^{(-1)} = \infty$, $f^{(-1)} = \infty$, $B^{(0)} = I$ and $\alpha = 1$.
Set $\tilde{x}_i = \max(x_i^{(0)}, x_{L_i})$ and $x_i^{(0)} = \min(\tilde{x}_i, x_{U_i})$, $i = 1, 2, ..., n$.
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 nr_{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 $nr_{act} > 0$ **and** $\alpha_0 < 3$ **then**
    Compute first order Lagrange multiplier estimate λ.
    $\lambda_i = -g_i$ if $i \in V_U$, $\lambda_i = g_i$ if $i \in V_L$ and $\lambda_i = 0$ else.
    if $nr_{act} < n$ **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
    Check convergence criterias, see A.7.1.
    if any convergence criteria are fulfilled **then**
      Set $stop = 1$.
    end if
    if *stop* **and** $nr_{act} \in (0, n)$ **and** $\alpha_0 < 3$ **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 α_{max}, the maximum step α such that $x + \alpha p_{full}$ is feasible with respect to the variable bounds, where $p_{full_i} = p_i$ if $i \notin V_L \cup V_U$ else $p_{full_i} = 0$.
      Compute second order Lagrange multiplier estimate, η, for the active variables.
      $\eta = \hat{\lambda} + \hat{H} \alpha_{max} p$,
      where $\hat{\lambda} = \lambda_i : i \in V_L \cup V_U$ and $\hat{H} = H_{ij} : i \in V_L \cup V_U, j \notin V_L \cup V_U$.
      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 nr_{act}.
      end if
    end if
    if *stop* **and** $nr_{act} < n$ **then**
      Check if x is a saddle or a minimum point.
      Compute the eigenvalues of $\tilde{H}$ and let ξ be the smallest eigenvalue.

if $\xi < -10^{-12}$ **and** no eigenvector search direction was used in previous iteration **then**
Set $stop = 0$ and set p equal to the eigenvector corresponding to ξ.
Check if p is a descent direction i.e. if $\tilde{g}^T p < 0$.
If p is not a descent direction change sign on p.
end if
end if
end if
Check stop criterias, see A.7.2.
if any stop criteria are fullfilled **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
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 η.
if $\eta_i < -b_{Tol}$ **and** $x_{L_i} \neq x_{U_i}$ **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 α_1, step length estimate sent to line search routine.
if $k > 0$ **and** $\tilde{g}^T p \neq 0$ **then**
Set $\tilde{\alpha}_1 = \min\left(1, -2\frac{\max\left(f^{(k-1)} - f^{(k)}, 10\epsilon_x\right)}{\tilde{g}^T p = 0}\right)$.
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 α_{max}
if $\alpha_{max} < 10^{-14}$ **then**
Set $\alpha = 0$.
else
Solve the line search problem $\min\limits_{0<\alpha\leq\alpha_{max}} f(x + \alpha p_{full})$.
end if
else
Compute the eigenvalues and their corresponding eigenvectors of $\tilde{H}$.
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 α_{max}.
if $\alpha_{max} > 10^{-7}$ **then**
Set $\alpha_1 = 1$.
Solve the line search problem $\min\limits_{0<\alpha\leq\alpha_{max}} f(x + \alpha p_{full})$.
else

Set $\alpha = 0$.
end if
if $\alpha > 10^{-6}$ **then**
Accept the search direction p_{full} and the step length α.
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 $x^{(k+1)} = x^{(k)} + \alpha p_{full}$.
$f^{(k+1)}$ and $g^{(k+1)}$ was computed in the line search.
Depending on the chosen method, update the approximation of the Hessian, the approximation of the inverse Hessian or β, see A.7.4.
Set $k = k + 1$.
end while

A.7.1 Convergence criterias

- $\max\limits_{i} \frac{\left|x_i^{(k)} - x_i^{(k-1)}\right|}{\max\left(\left|x_i^{(k)}\right|, size_x\right)} \leq \epsilon_x$
- $\max\limits_{i \notin V_L \cup V_U} \left(\left|g_i^{(k)}\right| \max\left(\left|x_i^{(k)}\right|, size_x\right)\right) \leq \epsilon_g \max\left(\left|f^{(k)}\right|, size_f\right)$
- Relative function value reduction low for *LowIts* iterations.

A.7.2 Stop criterias

- $k \geq MaxIter$
- $f^{(k)} \leq f_{Low}$

A.7.3 Computation of Search Direction

Newton

Solve $\tilde{H}p = -\tilde{g}$ either using Singular Value Decomposition with rank estimation and a subspace minimization technique or using LU-Decomposition with or without pivoting.

Safeguarded quasi-Newton DFP or BFGS

Solve $\tilde{B}p = -\tilde{g}$ 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 $p = -\tilde{B}\tilde{g}$.

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 $z^T y < 0.2 \cdot z^T \tilde{B} z$ **then**
Set $w = \frac{0.8 z^T \tilde{B} z}{z^T \tilde{B} z - z^T y} y + \left(1 - \frac{0.8 z^T \tilde{B} z}{z^T \tilde{B} z - z^T y}\right) \tilde{B} z$.
else
Set $w = y$.
end if
if $z^T w = 0$ **then**
if $z^T \tilde{B} z \neq 0$ **then**
Set $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} - \frac{\tilde{B} z z^T \tilde{B}}{z^T \tilde{B} z}$.
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 $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} + \frac{w w^T}{z^T w} - \frac{\tilde{B} z z^T \tilde{B}}{z^T \tilde{B} z}$
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 $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} + \left(1 + \frac{y^T \tilde{B} y}{z^T y}\right) \frac{z z^T}{z^T y} - \frac{z y^T \tilde{B} + \tilde{B} y z^T}{z^T y}$.
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 $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} + \frac{w w^T}{y^T w} - \frac{\tilde{B} y y^T \tilde{B}}{y^T \tilde{B} y}$.
end if
end if

Safeguarded quasi-Newton DFP

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 $\tilde{B}^{(k+1)} = \tilde{B}^{(k)} + \left(1 + \frac{z^T \tilde{B} z}{z^T y}\right) \frac{y y^T}{z^T y} - \frac{y z^T \tilde{B} + \tilde{B} z y^T}{z^T y}$.
  end if
end if

Fletcher-Reeves CG

Set $\beta = \frac{\tilde{g}^{(k+1)^T} \tilde{g}^{(k+1)}}{\tilde{g}^{(k)^T} \tilde{g}^{(k)}}$.

Polak-Ribiere CG

Set $\beta = \frac{\left(\tilde{g}^{(k+1)^T} - \tilde{g}^{(k)^T}\right) \tilde{g}^{(k+1)}}{\tilde{g}^{(k)^T} \tilde{g}^{(k)}}$.

Fletcher conjugate descent CG

Set $\beta = -\frac{\tilde{g}^{(k+1)^T} \tilde{g}^{(k+1)}}{\tilde{g}^{(k)^T} p}$.

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 $A = \begin{pmatrix} A & b - Ax^{(0)} \end{pmatrix}$, $x^{(0)} = \begin{pmatrix} x^{(0)^T} & 1 \end{pmatrix}^T$ and $c = \begin{pmatrix} c^T & 2\sum_{j=1}^{n} |c_i| \end{pmatrix}^T$.
Set $n = n + 1$
end if
else
Set $x^{(0)} = x\_0$.
end if
Compute $L = \lceil 1 + \sum_i \sum_j \log_2(1 + |a_{ij}|) \rceil$.
Set $tol = 2^{-2L}$.
Compute $\alpha = \frac{n-1}{3n}$ and $r = \frac{1}{\sqrt{n(n-1)}}$.
Set $q = 0.97$ and $\mu = \min(10^{-12}, \frac{tol}{2n})$.
for $k = 0, 1, ..., k_{max} - 1$ **do**
Set $D = \mathrm{diag}\{x_1^{(k)}, ..., x_n^{(k)}\}$, $\hat{c} = Dc$, and $\hat{A} = AD$.
Compute dual estimate $y^{(k)}$ by solving $\hat{A}y = \hat{c} - (\mu, ..., \mu)^T$.
Compute reduced cost vector $R = c - (\frac{\mu}{x_1^{(k)}}, ..., \frac{\mu}{x_n^{(k)}})^T - A^T y$, projected gradient vector $g_p = \hat{c} - (\mu, ..., \mu)^T - \hat{A}^T y$ and search direction $d = -Dg_p$.
if $R \geq -10^{-10}$ **and** it either holds that $c^T x^{(k)} - b^T y^{(k)} < tol$ or that the function value redution is less than 10^{-14} **then**
Let W be a index set of the variables active on their lower bounds, i.e.
$W = \left\{ j : x_j^{(k)} \leq 10^{-12} \right\}$.
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 $\binom{A}{e_i, i \in W}$, where e_i is the i:th unit row vector.
Let $d_i = Z_{i1}$ 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 $\alpha = \min_{i \notin W, d_i < 0} \frac{-x_i^{(k)}}{d_i}$.
if $\{i : i \notin W, d_i < 0\} = \emptyset$ **then**
STOP, purification failed.
end if
Set $x^{(k+1)} = x^{(k)} + \alpha d$.
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 $\lambda_{max}^{inv} = \max_j \frac{d_j}{x_j^{(k)}}$.
Set $\alpha = \min \left(\frac{q}{\lambda_{max}^{inv}}, \frac{(1-q)^2}{\mu} \right)$.
Set $x^{(k+1)} = x^{(k)} + \alpha d$.
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.
end if
Set $\epsilon_1 = 10^{-12}$.
if B_0 is not given **then**
Call Phase I simplex routine *Phase1Simplex* to get the solution x and B.
if no feasible Phase I solution were found **then**
STOP.
end if
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.
if x_0 is not given **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 $x_B = x_j$, $j \in B_{idx}$.
else
Set $x = x\_0$.
end if
Call Phase II simplex routine *Phase2Simplex* to get the solution x, B and y.
for $k = 0, 1, ..., k_{max}$ **do**
Update B_{idx} and N_{idx}.
Set $x_{idx} = x_j$, $j \le n_I \wedge j \in B_{idx}$.
Set $x_{I_j} = \lfloor x_{idx_j} + \epsilon_1 \rfloor$.
Set $x_{r_j} = \max(0, x_{idx_j} - x_{I_j})$.
Determine the variable with its fractional part closest to 0.5 i.e. set $i = \text{argmin}(|x_{r_i} - 0.5|)$.
if $i = \emptyset$ **or** $x_{r_i} < \epsilon_1$ **then**
STOP, convergence.
end if
Set $c_N = A_{B_i}^{-1} A_N$, where $A_{B_i}^{-1}$ is the i:th row in A_B^{-1} and $A_N = A_{ij}$, $j \in N_{idx}$.
Set $a_j = 0$ if $j \in N_{idx}$ else set $a_j = \max(0, c_{N_j} - \lfloor c_{N_j} + \epsilon_1 \rfloor)$.
Set $A = \begin{pmatrix} A & 0^{m \times 1} \\ a & -1 \end{pmatrix}$, $b = \begin{pmatrix} b^T & x_{r_i} \end{pmatrix}^T$, $c = \begin{pmatrix} c^T & 0 \end{pmatrix}^T$ and $x = \begin{pmatrix} x^T & -x_{r_i} \end{pmatrix}^T$.
Set $B = \begin{pmatrix} B & 1 \end{pmatrix}$, $n = n + 1$ and $m = m + 1$.
Call dual simplex routine *lpdual* to get the solution x, B and y.
if the dual simplex routine failed **then**
Use Phase I and Phase II simplex routines.
end if
end for

B.3 dijkstra

Set n equal to the number of nodes.
Set $B = \{s\}$ and $T = N \backslash \{s\}$, where N is the set of all nodes.
Set $dist(s) = 0$ and $pred(s) = 0$.
for $j = 1, 2, ..., n$ **do**
if $(s, j) \in Z$ **then**
Set $dist(j) = c_{sj}$, where c_{ij} means the cost of arc (i, j).
Set $pred(j) = s$.
else
Set $dist(j) = \infty$.
end if
end for
while $B \neq N$ **do**
Let $i \in T$ be a node for which $dist(i) = \min\{dist(j) : j \in T\}$.
Set $B = B \cup \{i\}$ and $T = T \backslash \{i\}$.
for each $j : (i, j) \in Z$ **do**

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 $x_{LOW} = \min_j x_{L_j}$ and $x_{UPP} = \max_j x_{U_j}$.
Set $s = 1 + x_{UPP} - x_{LOW}$.
Set $U_{ij} = 0$, $i = 1, 2, .., s$, $j = 1, 2, ..., n$.
Set $f_i = \infty$ and $f_{P_i} = \infty$, $i = 1, 2, ..., s$.
Set $f_{P_{1+x_S - x_{LOW}}} = 0$.
for $t = 1, 2, ..., n$ **do**
Set $u_{low} = u_{L_t}$ and $u_{upp} = u_{U_t}$.
if $t = n$ **then**
Set $x_{low} = x_{upp} = x_{LAST}$.
else
Set $x_{low} = x_{L_t}$ and $x_{upp} = x_{U_t}$.
end if
for $i = x_{low}, x_{low} + 1, ..., x_{upp}$ **do**
Set $u_{min} = 0$ and $f_{min} = \infty$.
for $j = u_{low}, u_{low} + 1, ..., u_{upp}$ **do**
Set $x = i - j + d_t$.
if $x_{LOW} \leq x \leq x_{UPP}$ **then**
Set $fu = P\_s(j > 0) + P_t j + I\_s(i > 0) + I_t i + f_{P_{1+x-x_{LOW}}}$.
if $fu < f_{min}$ **then**
Set $u_{min} = j$ and $f_{min} = fu$.
end if
end if
end for
Set $f_{1+i-x_{LOW}} = f_{min}$ and $U_{1+i-x_{LOW},t} = u_{min}$.
end for
Set $f_P = f$.
end for
Set $x = x_{LAST}$.
for $t = n, n-1, ..., 1$ **do**
Set $u_t = U_{1+x-x_{LOW},t}$.
Set $x = x - u_t + d_t$.
end for
Set $f\_opt = f_{1+x_{LAST}-x_{LOW}}$.

B.5 dpknap

if u_U is not given **then**
Set $u_{U_j} = \lfloor \frac{b}{A_j} \rfloor$, $j = 1, 2, ..., n$.
else
Set $u_U = u\_U$.
end if
Set $U_{ij} = 0$, $i = 1, 2, .., b+1$, $j = 1, 2, ..., n$.
Set $f_i = 0$ and $f_{P_i} = 0$, $i = 1, 2, ..., b+1$.
for $i = 1, 2, ..., n$ **do**
for $k = 1, 2, ..., b+1$ **do**
Set $u_{max} = 0$ and $f_{max} = f_{P_k}$.
for $j = 1, 2, ..., \min(u_{U_i}, \lfloor \frac{b}{A_i} \rfloor)$ **do**

Set $x = (k-1) - A_i j$.
if $x < 0$ **then**
break for
else
if $c_i j + f_{P_{1+x}} > f_{max}$ **then**
Set $u_{max} = j$ and $f_{max} = c_i j + f_{P_{1+x}}$.
end if
end if
end for
Set $f_k = f_{max}$ and $U_{ki} = u_{max}$.
end for
Set $f_P = f$.
end for
Set $x = b + 1$.
for $k = n, n-1, ..., 1$ **do**
Set $u_k = U_{xk}$.
Set $x = x - A_k U_{xk}$.
end for
Set $f\_opt = f_{b+1}$.

B.6 gsearch

Set $pred(i) = 0$ and $mark(i) = 0$, $i = 1, 2, ..., m$.
Set $pred(s) = -1$ and $mark(s) = 1$.
Set $LIST = \{s\}$.
while $LIST \neq \emptyset$ **do**
Set i equal to the first element in $LIST$.
if there is an arc from node i to node j **and** $mark(j) = 0$ **then**
Set $mark(j) = 1$ and $pred(j) = i$.
Put j first in $LIST$.
else
Delete the first element in $LIST$.
end if
end while

B.7 gsearchq

Set $pred(i) = 0$ and $mark(i) = 0$, $i = 1, 2, ..., m$.
Set $pred(s) = -1$ and $mark(s) = 1$.
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 $mark(j) = 0$ **then**
Set $mark(j) = 1$ and $pred(j) = i$.
Put j at the end of $LIST$.
end if
end for
end while

B.8 karmark

Compute $L = \lceil 1 + \log_2(1 + \max_j |c_j|) \log_2(1 + m) + \sum_i \sum_j \log_2(1 + |a_{ij}|) \rceil$.

Set $tol = \max(2^{-L}, 10^{-0})$.

Compute $\alpha = \frac{n-1}{3n}$ and $r = \frac{1}{\sqrt{n(n-1)}}$.
Set $x^{(0)} = (\frac{1}{n}, ..., \frac{1}{n})^T$.
Set $k = 0$.
while $c^T x^{(k)} > tol$ **and** $k \leq k_{max}$ **do**
  Set $D = \text{diag}\{x_1^{(k)}, ..., x_n^{(k)}\}$, $\hat{x}^{(0)} = (\frac{1}{n}, ..., \frac{1}{n})$, $\hat{c} = Dc$ and $B = \binom{AD}{1^T}$.
  Compute $d = \left(B^T(BB^T)^{-1}B - I\right)\hat{c}$.
  if Goldfarb/Todd choice of update **then**
    Set $\alpha = 0.99$.
    Compute $\hat{x} = \hat{x}^{(0)} + \alpha\frac{d}{n\|d\|}$.
  else if Bazaraa choice of update **then**
    Compute $\hat{x} = \hat{x}^{(0)} + \alpha r\frac{d}{\|d\|}$.
  end if
  Set $x^{(k+1)} = \frac{D\hat{x}}{1^T D\hat{x}}$.
  Set $k = k + 1$.
end while
Let W be a index set of the variables active on their lower bounds, i.e. $W = \left\{j : x_j^{(k)} \leq 10^{-12}\right\}$.
Set $B = \binom{A}{1^T}$ 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 $\binom{B}{e_i, i\in W}$, where e_i is the i:th unit row vector.
  Let $d_i = Z_{i1}$ 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 $\alpha = \min\limits_{i\notin W, d_i<0} \frac{-x_i^{(k)}}{d_i}$.
  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 $x_j = 0$, $j = 1, 2, ..., n$ and $u_i = 0$, $i = 1, 2, ..., m - 1$.
Set $\hat{A}_{rj} = A_{rj}$, $j = 1, 2, ..., n$ and $\hat{b} = b_r$.
Set $\tilde{A}_{ij} = A_{ij}$ and $\tilde{b}_i = b_i$, $i \in \{1, 2, ..., m\} - \{r\}$, $j = 1, 2, ..., n$.
Set $\lambda = 2$, $fail = 0$, $f_P = 0$, $f_{D_{old}} = \infty$ and $x_P = x$.
for $k = 1, 2, ..., k_{max}$ **do**
  if $c^T x \leq -\infty$ **then**
    STOP, convergence.
  end if
  Set $\hat{c} = \tilde{A}^T u$.
  Call the knapsack problem solver *dpknap* with the parameters $\hat{A}$, $\hat{b}$, $\hat{c}$ and x_U to get the solution x and f_D.
  Set $f_D = f_D + u^T\tilde{b}$ and compute the subgradient $\tilde{g} = \tilde{b} - \tilde{A}x$.
  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 $x_P = x$.
    Set $fail = 0$ and $\lambda = 2$.
  end if
  if $f_D < f_{D_{old}}$ **then**
    Set $fail = 0$.
  else
    Set $fail = fail + 1$.
  end if
  Set $f_{D_{old}} = f_D$.
  if $fail > 0$ **then**
    Set $\lambda = \frac{1}{2}\lambda$ and $fail = 0$.
  end if

Set $s_{SQ} = \tilde{g}^T\tilde{g} + (\hat{b} - \hat{A}x)^2$.
if $\alpha \le 0$ **or** $s_{SQ} \le 10^{-14}$ **then**
STOP, convergence.
end if
Set $\alpha = \frac{\alpha}{s_{SQ}}$ and $u_i = \max(0, u_i - \alpha\tilde{g}_i)$, $i = 1, 2, ..., m-1$.
end for

B.10 labelcor

Set $dist(j) = \infty$ for each $j \in N\backslash\{s\}$, where N is the set of all nodes.
Set $dist(s) = 0$ and $pred(s) = 0$.
repeat
for all arcs $(i, j) \in Z$ **do**
if $dist(j) > dist(i) + c_{ij}$ **then**
Set $dist(j) = dist(i) + c_{ij}$.
Set $pred(j) = i$.
end if
end for
until no changes in *dist* are made

B.11 lpdual

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 $k = 1, 2, ..., k_{max}$ **do**
Compute the objective function value $\hat{f} = b^Ty$,
if $x_i \ge -10^{-10}$ 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 Blands rule or Minimal Reduced Cost rule.
Solve $A_B^T u = e_{\hat{p}}$, where $B_{\hat{p}} = p$.
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

$$\frac{-\hat{c}_{N_{\hat{q}}}}{v_{\hat{q}}} = \min \left\{ \frac{-\hat{c}_{N_j}}{v_j} : v_j < -10^{-10}, j = 1, 2, ..., n-m \right\} \stackrel{\text{def}}{=} \gamma,$$

where $q = N_{\hat{q}}$.
if $\gamma = \emptyset$ **then**
Choose

$$\frac{-\hat{c}_{N_{\hat{q}}}}{v_{\hat{q}}} = \min \left\{ \frac{-\hat{c}_{N_j}}{v_j} : v_j < 0, j = 1, 2, ..., n-m \right\} \stackrel{\text{def}}{=} \gamma.$$

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 $x_q = \alpha = \frac{x_p}{v_{\hat{q}}}$, $x_B = x_B + \alpha d$ and $x_p = 0$.
Set $B = B \cup \{q\} \backslash \{p\}$ and $N = N \cup \{p\} \backslash \{q\}$.
if $\alpha > 10^5$ **then**
Numerical difficuties, recompute x by setting $x_N = 0$ and solving $A_B x_B = b$.
end if
end for

B.12 lpkarma

Set $k = m + n + 2$ and $l = 2(m + n) + 2$.
Set $M = 3 \left[\sum_{j=1}^{n} |c_j| + \sum_{i=1}^{m} |b_i| + 2 \max_{i,j} (|c_j|, |b_i|) \right]$.
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 $B = \begin{pmatrix} \hat{B} & -v \end{pmatrix}$ and $d = \begin{pmatrix} 0^{1 \times l} & 1 \end{pmatrix}^T$.
Call *karmark* with constraint matrix B and cost vector d to get the solution $\tilde{x}$.
Set $x = (M + 1)\tilde{x}_j$, $j = 1, 2..., n$.
Set $y = (M + 1)\tilde{x}_j$, $j = n + 1, ..., n + m$.

B.13 lpsimp1

Solve the LP problem

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

with $\tilde{A} = \begin{pmatrix} A & I^{m\times m} \end{pmatrix}$, $\tilde{c} = \begin{pmatrix} 0^n & 1^{m_{eq}} & 0^{m-m_{eq}} \end{pmatrix}^T$, $x\_0 = \begin{pmatrix} 0^n & b^T \end{pmatrix}^T$ and $B\_0 = \begin{pmatrix} 0^n & 1^m \end{pmatrix}^T$.

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 $B^{(0)} = \{n-m+1, n-m+2, ..., n\}$.
  Set $N^{(0)} = \{1, 2, ..., n-m\}$.
else 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.
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 $x_B = x_j : j \in B$.
end if
for $k = 1, 2, ..., k_{max}$ **do**
  Compute the objective function value $\hat{f} = c_B^T x_B$, where $c_B = c_j : j \in B$.
  Compute shadow prices y by solving $A_B^T y = c_B$.
  Compute reduced costs $\hat{c}_N = c_N - A_N^T y$, where $A_N = A_{ij} : j \in N$ and $c_N = c_j : j \in N$.
  if $\hat{c}_N \geq -\epsilon_f$ **then**
    STOP, x is optimal.
  end if
  Choose the variable x_q 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 $P = \{i : d_i < 0\}$.
if $P = \emptyset$ **then**
The problem is unbounded, STOP.
else
Set the step length $\alpha = \frac{-x_p}{d_{\hat{p}}} = \min_{i \in P} \left(\frac{-x_{B_i}}{d_i} \right)$.
Variable x_p is to be excluded from the basis.
end if
Set $x_B = x_B + \alpha d$.
Set $x_p = 0$ and $x_q = \alpha$.
Set $B^{(k+1)} = B^{(k)} \cup \{q\} \backslash \{p\}$ and $N^{(k+1)} = N^{(k)} \cup \{p\} \backslash \{q\}$.
end for

B.15 maxflow

Set m equal to the number of nodes.
Set $x_{ij} = 0 \ \forall (i,j) \in Z$.
Set $max\_flow = 0$.
while not convergence **do**
Set $pred(i) = 0$ and $flow(i) = 0$, $i = 1, 2, ..., m$.
Set $pred(s) = -1$ and $flow(s) = \infty$.
Set $LIST = s$.
while $LIST \neq \emptyset$ **and** $pred(t) = 0$ **do**
Set i equal to the first element in $LIST$.
for all arcs (i,j) outgoing from node i **do**
if $pred(j) = 0$ **and** $x_{ij} < x\_U_{ij}$ **then**
Set $pred(j) = i$ and $flow(j) = \min(flow(i), x\_U_{ij} - x_{ij})$.
Put j at the end of $LIST$.
end if
end for
for all arcs (j,i) coming in to node i **do**
if $pred(j) = 0$ **and** $x_{ji} < x\_U_{ji}$ **then**
Set $pred(j) = i$ and $flow(j) = \min(flow(i), x_{ij})$.
Put j at the end of $LIST$.
end if
end for
Delete the first element in $LIST$.
end while
if $pred(t) > 0$ **then**
Set $j = t$ and $i = pred(t)$.
Set $x_{ij} = x_{ij} + flow(t)$.
while $i \neq s$ **do**
Set $j = i$ and $i = pred(i)$.
if $(i,j) \in Z$ **then**
Set $x_{ij} = x_{ij} + flow(t)$.
else
Set $x_{ij} = x_{ij} - flow(t)$.
end if
end while
Set $max\_flow = max\_flow + flow(t)$.
else
STOP, the maximum flow is max_flow.
end if
end while

B.16 modlabel

Set $dist(j) = \infty$ for each $j \in N \backslash \{s\}$, 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 $pred(j) = i$.
      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 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 $C_{ij} = C_{ji} = \text{Inf}$.
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
  Set $B_k = (i, j)$.
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 TPsimplx

if $\sum_i^m s_i > \sum_j^n d_j$ **then**
  Add a dummy demand point with zero cost.
else if $\sum_i^m s_i < \sum_j^n d_j$ **then**
  Add a dummy supply point with high cost.
end if
if x and B is not given **then**
  Call *TPvogel* 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 $y = \binom{u}{v}$ 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 $\hat{c}_{ij} = c_{ij} - u_i - vj$.
  Set $\hat{c}_{min} = \min(\hat{c})$.
  if $\hat{c}_{min} \geq 0$ **then**
    STOP, x is optimal.
  else
    Set $q = (q_i, q_j)$ where $\hat{c}_{q_i q_j} = \hat{c}_{min}$.
  end if
  Determine the cycle of change vector μ by solving $A\mu = b$, where $A \in \mathbb{R}^{n+m \times n+m}$ and $b \in \mathbb{R}^{n+m}$. $A_{B_{i1},i} = 1$, $A_{m+B_{i2},i} = 1$ for $i = 1, 2, ..., m+n$. $A_{m+n,m+n} = 1$ and the rest of the entries in A is zero. $b_{q_i} = -1$, $b_{q_j} = -1$ and the rest of the entries in b is zero.
  Set $\theta = \min \left\{ x_{B_{i1},B_{i,2}} : \mu_i < 0 \right\}$.
  if $\theta = \emptyset$ **then**
    STOP, the problem has an unbounded fesible region.
  else
    Set $p = (p_i, p_j)$ where $x_{p_i p_j} = \theta$.
  end if
  Set $x_{B_{i1},B_{i,2}} = x_{B_{i1},B_{i,2}} + \theta\mu_i$.
  Set $x_{q_i q_j} = \theta$.
  Set $B = B \cup \{q\} \backslash \{p\}$.
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_{c_j} 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 $j = 1, 2, ..., n$ **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_{r_i} 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 $i = 1, 2, ..., m$ **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 $\max(p_r) > \max(p_c)$ **then**
        Set $i = \text{argmax}(p_r)$.
        Choose j so that (i, j) corresponds to the smallest cost in row i of the non crossed-out entries.
      else
        Set $j = \text{argmax}(p_c)$.
        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 $\hat{A}_{rj} = A_{rj}$, $j = 1, 2, ..., n$ and $\hat{b} = b_r$.
Set $\tilde{A}_{ij} = A_{ij}$ 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 $k = 0, 1, ..., u\_max$ **do**
Set $u_i = u_i + 1$, $i = 1, 2, ..., m - 1$.
Set $\hat{c} = \tilde{A}^T u$.
Call the knapsack problem solver *dpknap* with the parameters $\hat{A}$, $\hat{b}$, $\hat{c}$ and x_U to get the solution x and f_D.
Set $f_D = f_D + u^T \tilde{b}$ and compute the subgradient $\tilde{g} = \tilde{b} - \tilde{A}x$.
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

[1] *LINGO - The Modeling Language and Optimizer.* LINDO Systems Inc., Chicago, IL, 1995.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows. In G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, editors, *Optimization*, volume 1 of *Handbooks in Operations Research and Management Science.* Elsevier/North Holland, Amsterdam, The Netherlands, 1989.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications.* Prentice-Hall Inc., Kanpur and Cambridge, 1993.

[4] M. Al-Baali and R. Fletcher. Variational methods for non-linear least squares. *J. Oper. Res. Soc.*, 36:405–421, 1985.

[5] M. Al-Baali and R. Fletcher. An efficient line search for nonlinear least-squares. *Journal of Optimization Theory and Applications*, 48:359–377, 1986.

[6] M. C. Bartholomew-Biggs. Algorithms for general constrained nonlinear optimization. Technical Report Technical Report 277, Numerical Optimisation Centre, Mathematics Division, University of Hertfordshire, 1993.

[7] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear Programming and Network Flows.* John Wiley and Sons, New York, 2nd edition, 1990.

[8] J. Bisschop and R. Entriken. *AIMMS - The Modeling System.* Paragon Decision Technology, Haarlem, The Netherlands, 1993.

[9] J. Bisschop and A. Meeraus. On the development of a general algebraic modeling system in a strategic planning environment. *Mathematical Programming Study*, 20:1–29, 1982.

[10] Mattias Björkman. Nonlinear Least Squares with Inequality Constraints. Bachelor Thesis, Department of Mathematics and Physics, Mälardalen University, Sweden, 1998. Supervised by Kenneth Holmström.

[11] I. Bongartz, A. R. Conn, N. I. M. Gould, and P. L. Toint. CUTE: Constrained and Unconstrained Testing Environment. *ACM Transactions on Mathematical Software*, 21(1):123–160, 1995.

[12] I. Bongartz, A. R. Conn, Nick Gould, and Ph. L. Toint. CUTE: Constrained and Unconstrained Testing Environment. Technical report, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, September 2 1997.

[13] Mary Ann Branch and Andy Grace. *Optimization Toolbox User's Guide.* 24 Prime Park Way, Natick, MA 01760-1500, 1996.

[14] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS - A User's Guide.* The Scientific Press, Redwood City, CA, 1988.

[15] A. R. Conn, Nick Gould, A. Sartenaer, and Ph. L. Toint. Convergence properties of minimization algorithms for convex constraints using a structured trust region. *SIAM Journal on Scientific and Statistical Computing*, 6(4):1059–1086, 1996.

[16] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. LINPACK User's Guide. *SIAM*, 1979.

[17] Erik Dotzauer and Kenneth Holmström. The TOMLAB Graphical User Interface for Nonlinear Programming. *Advanced Modeling and Optimization*, 1(2), 1999.

[18] Arne Stolbjerg Drud. Interactions between nonlinear programing and modeling systems. *Mathematical Programming, Series B*, 79:99–123, 1997.

[19] S. I. Feldman, David M. Gay, Mark W. Maimone, and N. L. Schryer. A Fortran-to-C converter. Technical Report Computing Science Technical Report No. 149, AT&T Bell Laboratories, May 1992.

[20] Marshall L. Fisher. An Application Oriented Guide to Lagrangian Relaxation. *Interfaces 15:2*, pages 10–21, March-April 1985.

[21] R. Fletcher and C. Xu. Hybrid methods for nonlinear least squares. *IMA Journal of Numerical Analysis*, 7:371–389, 1987.

[22] Roger Fletcher. *Practical Methods of Optimization*. John Wiley and Sons, New York, 2nd edition, 1987.

[23] Roger Fletcher and Sven Leyffer. Nonlinear programming without a penalty function. Technical Report NA/171, University of Dundee, 22 September 1997.

[24] R. Fourer, D. M. Gay, and B. W.Kernighan. *AMPL - A Modeling Language for Mathematical Programming*. The Scientific Press, Redwood City, CA, 1993.

[25] B. S. Garbow, J. M. Boyle, J. J. Dongara, and C. B. Moler. Matrix Eigensystem Routines-EISPACK Guide Extension. In *Lecture Notes in Computer Science*. Springer Verlag, New York, 1977.

[26] David M. Gay. Hooking your solver to AMPL. Technical report, Bell Laboratories, Lucent Technologies, Murray Hill, NJ 07974, 1997.

[27] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. *User's Guide for NPSOL (Version 4.0): A Fortran package for nonlinear programming*. Department of Operations Research, Stanford University, Stanford, CA, 1986. SOL 86-2.

[28] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, London, 1982.

[29] D. Goldfarb and M. J. Todd. Linear programming. In G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, editors, *Optimization*, volume 1 of *Handbooks in Operations Research and Management Science*. Elsevier/North Holland, Amsterdam, The Netherlands, 1989.

[30] Jacek Gondzio. Presolve analysis of linear programs prior to applying an interior point method. *INFORMS Journal on Computing*, 9(1):73–91, 1997.

[31] Michael Held and Richard M. Karp. The Traveling-Salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1:6–25, 1971.

[32] Kaj Holmberg. Heltalsprogrammering och dynamisk programmering och flöden i nätverk och kombinatorisk optimering. Technical report, Division of Optimization Theory, Linköping University, Linköping, Sweden, 1988-1993.

[33] Kenneth Holmström. The TOMLAB Optimization Environment in Matlab. *Advanced Modeling and Optimization*, 1(1):47–69, 1999.

[34] Kenneth Holmström and Mattias Björkman. The TOMLAB NLPLIB Toolbox for Nonlinear Programming. *Advanced Modeling and Optimization*, 1:70–86, 1999.

[35] Kenneth Holmström, Mattias Björkman, and Erik Dotzauer. The TOMLAB OPERA Toolbox for Linear and Discrete Optimization. *Advanced Modeling and Optimization*, 1(2), 1999.

[36] J. Huschens. On the use of product structure in secant methods for nonlinear least squares problems. *SIAM Journal on Optimization*, 4(1):108–129, February 1994.

[37] Kenneth Iverson. *A Programming Language*. John Wiley and Sons, New York, 1962.

[38] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, October 1993.

[39] Donald R. Jones. DIRECT. *Encyclopedia of Optimization*, 1999. To be published.

[40] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive Black-Box functions. *Journal of Global Optimization*, 13:455–492, 1998.

[41] P. Lindström. *Algorithms for Nonlinear Least Squares - Particularly Problems with Constraints*. PhD thesis, Inst. of Information Processing, University of Umeå, Sweden, 1983.

[42] David G. Luenberger. *Linear and Nonlinear Programming*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2nd edition, 1984.

[43] C. B. Moler. MATLAB — An Interactive Matrix Laboratory. Technical Report 369, Department of Mathematics and Statistics, University of New Mexico, 1980.

[44] Bruce A. Murtagh and Michael A. Saunders. MINOS 5.4 USER'S GUIDE. Technical Report SOL 83-20R, Revised Feb. 1995, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California 94305-4022, 1995.

[45] G. L. Nemhauser and L. A. Wolsey. Integer programming. In G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, editors, *Optimization*, volume 1 of *Handbooks in Operations Research and Management Science*. Elsevier/North Holland, Amsterdam, The Netherlands, 1989.

[46] P. C. Piela, T. G. Epperly, K. M. Westerberg, and A. W. Westerberg. ASCEND: An object-oriented computer environment for modeling and analysis: The modeling language. *Computers and Chemical Engineering*, 15:53–72, 1991.

[47] Raymond P. Polivka and Sandra Pakin. *APL: The Language and Its Usage*. Prentice Hall, Englewood Cliffs, N. J., 1975.

[48] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[49] A. Sartenaer. Automatic determination of an initial trust region in nonlinear programming. Technical Report 95/4, Department of Mathematics, Facultés Universitaires ND de la Paix, Bruxelles, Belgium, 1995.

[50] K. Schittkowski. On the Convergence of a Sequential Quadratic Programming Method with an Augmented Lagrangian Line Search Function. Technical report, Systems Optimization laboratory, Stanford University, Stanford, CA, 1982.

[51] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines - EISPACK Guide Lecture Notes in Computer Science*. Springer-Verlag, New York, 2nd edition, 1976.

[52] Wayne L. Winston. *Operations Research: Applications and Algorithms*. International Thomson Publishing, Duxbury Press, Belmont, California, 3rd edition, 1994.