FICO® Xpress Mosel solver interfaces

Using alternative solvers with Mosel

WHITEPAPER

FICO® Xpress Optimization

FICO

©2017–2026 Fair Isaac Corporation. All rights reserved. This documentation is the property of Fair Isaac Corporation ("FICO"). Receipt or possession of this documentation does not convey rights to disclose, reproduce, make derivative works, use, or allow others to use it except solely for internal evaluation purposes to determine whether to purchase a license to the software described in this documentation, or as otherwise set forth in a written software license agreement between you and FICO (or a FICO affiliate). Use of this documentation and the software described in it must conform strictly to the foregoing permitted uses, and no other use is permitted.

The information in this documentation is subject to change without notice. If you find any problems in this documentation, please report them to us in writing. Neither FICO nor its affiliates warrant that this documentation is error-free, nor are there any other warranties with respect to the documentation except as may be provided in the license agreement. FICO and its affiliates specifically disclaim any warranties, express or implied, including, but not limited to, non-infringement, merchantability and fitness for a particular purpose. Portions of this documentation and the software described in it may contain copyright of various authors and may be licensed under certain third-party licenses identified in the software, documentation, or both.

In no event shall FICO or its affiliates be liable to any person for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this documentation or the software described in it, even if FICO or its affiliates have been advised of the possibility of such damage. FICO and its affiliates have no obligation to provide maintenance, support, updates, enhancements, or modifications except as required to licensed users under a license agreement.

FICO is a registered trademark of Fair Isaac Corporation in the United States and may be a registered trademark of Fair Isaac Corporation in other countries. Other product and company names herein may be trademarks of their respective owners.

Patent(s): www.fico.com/en/patents

FICO® Xpress Mosel
Last Revised: 15 July, 2025

How to Contact the Xpress Team

Sales and Maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

Product Support

Customer Self Service Portal (online support): www.fico.com/en/product-support

Email: Support@fico.com (Please include *'Xpress'* in the subject line)

For the latest news and Xpress software and documentation updates, please visit the Xpress website at http://www.fico.com/xpress or subscribe to our mailing list.

Xpress Mosel solver interfaces

Using alternative solvers with Mosel

Y. Colombani and S. Heipcke

Xpress Optimization, FICO, 6280 Bishops Court, Birmingham Business Park, Birmingham B37 7YB, UK
http://www.fico.com/xpress

15 July, 2025

Abstract

This paper describes two approaches for integrating third party optimization solvers with the FICO® Xpress Optimization stack using the mathematical programming solver interfaces of FICO® Xpress Mosel. Integration via the low-level matrix handling interface for LP/MIP published through the Mosel Native Interface is presented by the means of an example model and fully described in the Mosel Native Interface reference documentation. The extension to the matrix handling interface published by the Mosel module *mmnl* is documented in the second section of this paper, including the full description of an example implementation. This paper also explains in detail how to integrate Mosel with any solver that supports the AMPL NL interface, using the *nlsolv* module.

Contents

1 Introduction . . . 3
2 Implementing a specific LP/MIP solver interface . . . 3
2.1 Example . . . 3
3 Implementing a specific nonlinear solver interface . . . 5
3.1 Example . . . 5
3.1.1 Mosel model using *myqxprs* . . . 5
3.1.2 Structures for passing information . . . 6
3.1.3 List of subroutines . . . 6
3.1.4 List of parameters . . . 7
3.1.5 List of types . . . 7
3.1.6 List of services . . . 8
3.1.7 Module context . . . 8
3.1.8 Interface structure . . . 8
3.1.9 Initialization function . . . 9
3.1.10 Implementation of subroutines . . . 9
3.1.11 Implementation of *mmnl* solver interface functions . . . 16
3.1.12 Implementation of services . . . 18
3.1.13 Handling optimization problems . . . 20
3.1.14 Generating names for matrix entries and matrix output . . . 21
3.2 NI library interface functions of *mmnl* . . . 23
loadprob . . . 25
buildqtls . . . 26
freeqtls . . . 27
strip . . . 28
eval . . . 29
grad . . . 30
setsolstat . . . 31

getctrnum . . . 32
getnlctrnum . . . 33
4 Using module *nlsolv* . . . 34
4.1 Example . . . 34
4.2 Configuring a solver . . . 35
4.3 Control parameters . . . 36
NL_binary . . . 37
NL_options . . . 37
NL_probname . . . 37
NL_solver . . . 37
NL_solverpath . . . 38
NL_status . . . 38
NL_verbose . . . 39
4.4 Procedures and functions . . . 39
exportnl . . . 40
getprobstat . . . 41
maximize, minimize . . . 42

1 Introduction

FICO® Xpress Mosel is designed with a modular architecture: the core of the Mosel language defines the basic types 'decision variable', 'linear constraint' and 'optimization problem' (`mpvar`, `linctr`, and `mpproblem` respectively) with suitable operators and access routines for the *formulation* of linear and mixed-integer programming problems, but it does not contain any solver. The connection to solvers is established via extensions, so-called *Mosel modules*, also refered to as *DSO* (Dynamic Shared Object)dynamic libraries implemented in C. Similarly, extensions to the basic constraint types are available through additional modules (most importantly, *mmnl* for nonlinear expressions) or *packages* (libraries implemented in Mosel).

The Xpress distribution contains readily available DSO for all Xpress Solvers (modules *mmxprs*, *mmxnlp*, *kalis*). Users wishing to use alternative Mathematical Programming (*e.g.* LP, MIP or NLP) solvers with Mosel have the choice between

- implementing their own solver module (see Sections 2 and 3), or
- connecting to a solver that supports the NL format through the module *nlsolv* (see Section 4).

It should be noted here that solvers which are not based on the Mathematical Programming paradigm such as Constraint Programming solvers always require a specific implementation via the Mosel NI.

The remainder of this document explains how to proceed if you wish to work with alternative Mathematical Programming solvers and in particular, how to use the *nlsolv* module.

2 Implementing a specific LP/MIP solver interface

The Mosel Native Interface (NI) defines a set of conventions that make it possible to extend the Mosel language with constants, subroutines, types, I/O drivers, and control parameters. The Mosel NI also publishes a special set of functionality that provides access to the matrix-based representation of optimization problems formulated using the Mosel types `mpvar`, `linctr`, `mpproblem`, that is, LP and MIP problems. These NI functions can be used for implementing interfaces to optimization solvers that are available in the form of a C/C++ library.

A minimal implementation of a solver module needs to do the following:

- Modeling functionality:
  - define subroutines to start an optimization run and retrieve solution values
  - provide access to solver parameters
  - if supported by the solver, provide support for handling multiple problems
- NI functionality:
  - implement a reset and an unload service
  - initialize the module and the required interface structures

2.1 Example

The following Mosel model highlights the modelling features that are provided through the example solver module 'myxprs'.

```
model "Problem solving and solution retrieval"
  uses "myxprs"                        ! Load the solver module

  declarations
    x,y: mpvar                         ! Some decision variables
    pb: mpproblem                      ! (Sub)problem
  end-declarations

  procedure printsol
    if getprobstat = MYXP_OPT then
      writeln("Solution: ", getobjval, ";", x.sol, ";", y.sol)
    else
      writeln("No solution found")
    end-if
  end-procedure

  Ctr1:= 3*x + 2*y <= 400
  Ctr2:= x + 3*y <= 310
  MyObj:= 5*x + 20*y

 ! Setting solver parameters
  setparam("myxp_verbose", true)       ! Display solver log
  setparam("myxp_timelimit", 10)       ! Set a time limit

 ! Solve the problem (includes matrix generation)
  maximize(MyObj)

 ! Retrieve a solver parameter
  writeln("Solver status: ", getparam("myxp_lpstatus"))
 ! Access solution information
  printsol

 ! Turn poblem into a MIP
  x is_integer; y is_integer

 ! Solve the modified problem
  maximize(MyObj)
  printsol

 ! **** Define and solve a (sub)problem ****
  with pb do
    3*x + 2*y <= 350
    x + 3*y <= 250
    maximize(5*x + 20*y)
    printsol
  end-do

end-model
```

The underlying module implementation (C program) is discussed in detail in Chapter *Implementing an LP/MIP solver interface* of the Mosel NI User Guide, including some extensions such as the implementation of a solution callback and a matrix export subroutine. The complete source can be retrieved from http://examples.xpress.fico.com. The C program needs to be compiled into the library `myxprs.dso` using the makefile that is provided along with the Mosel module examples. The resulting DSO file needs to be on the search path for DSO (*e.g.* by including its location in the environment variable `MOSEL_DSO`) and the solver libraries have to be accessible and suitably licensed in order to run the example model.

3 Implementing a specific nonlinear solver interface via *mmnl*

The Mosel module *mmnl* that provides handling of nonlinear expressions and constraints for the Mosel language also publishes a library interface in the Mosel NI format that makes it possible to implement specific DSO (such as *mmxnlp* in the Xpress distribution) for interfacing directly to solvers for nonlinear programming problems.

The example implementation discussed in the following section shows how to use the nonlinear problem handling functionality for the special case of a quadratic solver. The relevant functions of the *mmnl* inter-module interface are documented in Section 3.2.

3.1 Example: implementing an interface to a QPQC solver

The example implementation described here can be considered as an extension with handling of nonlinearities of the example documented in the chapter *Implementing an LP/MIP solver interface* of the Mosel NI User Guide. For solver-specific features that are not directly related to the handling of nonlinear constraints—such as the implementation of callback functions—the reader is therefore referred to this document.

3.1.1 Mosel model using *myqxprs*

The following example shows how to calculate the shape of a hanging chain with `N` chainlinks and fixed endpoints. In this example the objective function is linear and the constraints contain nonlinear (quadratic) terms. The modelling features that are provided through the example solver module *myqxprs* are highlighted. Features such as the setting of initial values for decision variables, the retrieval of solution values, or the type `nlctr` for nonlinear constraints are provided by the module *mmnl* that is included by our solver module (see Chapter *mmnl* of the Mosel Language Reference Manual for the documentation of *mmnl*).

```
model "catenary - myqxprs version"
 uses "myqxprs"

 parameters
   N = 100                       ! Number of chainlinks
   L = 1                         ! Difference in x-coordinates of endlinks
   H = 2*L/N                     ! Length of each link
 end-parameters

 declarations
   RN = 0..N
   x: array(RN) of mpvar         ! x-coordinates of endpoints of chainlinks
   y: array(RN) of mpvar         ! y-coordinates of endpoints of chainlinks
   PotentialEnergy: linctr       ! Objective function
   Link: array(range) of nlctr   ! Constraints
 end-declarations

 forall(i in RN) x(i) is_free
 forall(i in RN) y(i) is_free

! Objective: minimise the potential energy
 PotentialEnergy:= sum(j in 1..N) (y(j-1)+y(j))/2

! Bounds: positions of endpoints
 x(0) = 0; y(0) = 0; x(N) = L; y(N) = 0

! Constraints: positions of chainlinks
```

```
 forall(j in 1..N)
   Link(j):= (x(j)-x(j-1))^2+(y(j)-y(j-1))^2 <= H^2

! Setting start values
 forall(j in RN) setinitval(x(j), j*L/N)
 forall(j in RN) setinitval(y(j), 0)

! Configuration of the solver
 setparam("myxp_verbose", true)

! Solve the problem
 minimise(PotentialEnergy)

! Solution reporting
 if getprobstat not in {MYXP_OPT, MYXP_UNF} then
   writeln("No solution available. Solver status: ", getprobstat)
 else
   writeln("Solution: ", getobjval)
   setparam("realfmt", "%10.5f")
   forall(j in RN) writeln(getsol(x(j)), " ", getsol(y(j)))
 end-if

end-model
```

3.1.2 Structures for passing information

A minimal implementation of a solver module based on *mmnl* needs to do the following:

- Modeling functionality:
  - define subroutines to start an optimization run and retrieve solution values
  - provide access to solver parameters
  - if supported by the solver, provide support for handling multiple problems
- NI functionality:
  - implement a reset and an unload service
  - implement module dependency services for *mmnl*
  - initialize the module and the required interface structures

The next few sections (3.1.3 – 3.1.7) describe the structures that are required for exchanging information between Mosel and the external program (the solver library): these structures are communicated during the module initialization (see 3.1.9). This is followed by an explanation of the implementation of the main module routines starting in Section 3.1.10.

Beyond the minimal functionality of a solver module there are many other tasks that one might wish to expose at the Mosel language level, one such example is given in Section 3.1.14 with the `writeprob` matrix output routine and the related Mosel NI functionality for generating names based on model entity names. Another example would be the implementation of solver callback functions that allow users to interact with the problem via the Mosel language at specific points while the solver is running—please see Section *Implementing a solver callback* of the Mosel NI User Guide for the description of an implementation example.

3.1.3 List of subroutines

The minimal set of entries for the list of subroutines would be just the calls to minimization/maximization. Our implementation adds a function to retrieve the problem status

information, alternative spelling for the optimization routines, and it also provides the access routines for module control parameters that are required for the implementation of solver parameters. The optimization routines have two different implementations in our example, one for linear expressions(of type `linctr`) and a second for nonlinear expressions (the type `nlctr` defined by *mmnl*).

```
static XPRMdsofct tabfct[]=
    {
     {"",XPRM_FCT_GETPAR,XPRM_TYP_NOT,0,NULL,slvlc_getpar},
     {"",XPRM_FCT_SETPAR,XPRM_TYP_NOT,0,NULL,slvlc_setpar},
     {"getprobstat",2000,XPRM_TYP_INT,0,NULL,slvlc_getpstat},
     {"minimise",2100,XPRM_TYP_NOT,1,"c",slvlc_minim},
     {"minimize",2100,XPRM_TYP_NOT,1,"c",slvlc_minim},
     {"maximise",2101,XPRM_TYP_NOT,1,"c",slvlc_maxim},
     {"maximize",2101,XPRM_TYP_NOT,1,"c",slvlc_maxim}
     {"minimise", 2120, XPRM_TYP_NOT,1,"|nlctr|", slv_lc_nminim},
     {"minimize", 2120, XPRM_TYP_NOT,1,"|nlctr|", slv_lc_nminim},
     {"maximise", 2121, XPRM_TYP_NOT,1,"|nlctr|", slv_lc_nmaxim},
     {"maximize", 2121, XPRM_TYP_NOT,1,"|nlctr|", slv_lc_nmaxim}
    };
```

3.1.4 List of parameters

In terms of an example, we provide access to a few controls of Xpress Optimizer, and the module also shows how to implement a verbosity flag, resulting in the following list of module parameters:

```
static struct                   /* Parameters published by this module */
    {
     char *name;
     int type;
    } myxprsparams[]=
    {
     {"myxp_verbose", XPRM_TYP_BOOL|XPRM_CPAR_READ|XPRM_CPAR_WRITE},
     {"myxp_timelimit", XPRM_TYP_REAL|XPRM_CPAR_READ|XPRM_CPAR_WRITE},
     {"myxp_lpstatus", XPRM_TYP_INT|XPRM_CPAR_READ},
     {"myxp_lpobjval", XPRM_TYP_REAL|XPRM_CPAR_READ},
    };
```

The problem and LP status parameters return values are best implemented via module constants, such as:

```
static XPRMdsoconst tabconst[]=
    {
     XPRM_CST_INT("MYXP_INF",XPRM_PBINF),             /* Mosel status codes */
     XPRM_CST_INT("MYXP_OPT",XPRM_PBOPT),
     XPRM_CST_INT("MYXP_OTH",XPRM_PBOTH),
     XPRM_CST_INT("MYXP_UNF",XPRM_PBUNF),
     XPRM_CST_INT("MYXP_UNB",XPRM_PBUNB),
     XPRM_CST_INT("MYXP_LP_OPTIMAL",XPRS_LP_OPTIMAL), /* Solver status codes */
     XPRM_CST_INT("MYXP_LP_INFEAS",XPRS_LP_INFEAS),
     XPRM_CST_INT("MYXP_LP_CUTOFF",XPRS_LP_CUTOFF)
    };
```

3.1.5 List of types

The list of types has a single entry: a solver module needs to extend the Mosel type `mpproblem` with its own implementation.

```
static XPRMdsotyp tabtyp[]=
    {
     {"mpproblem.mqxp", 1, XPRM_DTYP_PROB|XPRM_DTYP_APPND, slv_pb_create,
      slv_pb_delete, NULL, NULL, slv_pb_copy}
```

```
    };
```

The following structure implements the *problem* type for our module, Mosel will maintain one instance of this type for each `mpproblem` object.

```
typedef struct SlvPb
    {
     struct SlvCtx *slctx;      /* Solver module context */
     XPRSprob xpb;
     int have;
     double *solval;            /* Structures for storing solution values */
     double *dualval;
     double *rcostval;
     double *slackval;
     XPRMcontext saved_ctx;     /* Mosel context (used by callbacks) */
     struct SlvPb *prev,*next;
    } s_slvpb;
```

A *solver module context* definition is shown below in Section 3.1.7.

3.1.6 List of services

The services `PARAM` and `PARLST` are required for the handling of module parameters, `RESET` and `UNLOAD` manage the access to the solver library. Furthermore, we need to specify the dependency on functionality provided via the NI interface of the module *mmnl* via the `DEPLST` and `IMPLST` services.

```
static XPRMdsoserv tabserv[]=
    {
     {XPRM_SRV_PARAM, (void *)slv_findparam},
     {XPRM_SRV_PARLST, (void *)slv_nextparam},
     {XPRM_SRV_RESET, (void *)slv_reset},
     {XPRM_SRV_UNLOAD, (void *)slv_quitlib}
     {XPRM_SRV_DEPLST, (void *)slv_deplist},
     {XPRM_SRV_IMPLST, (void *)slv_implst}
    };
```

The first four entries in this list are implemented via module (C library) functions (see Section 3.1.12 below for their immplementation), the module dependencies are stated via these structures:

```
static const char *slv_deplist[]={"mmnl",NULL};
static const char *slv_implst[]={"mmnl",NULL};
```

3.1.7 Module context

The module context holds the type ID for the extended `mpproblem` type, module options, and a list of references to the problems that have been created by this module:

```
typedef struct SlvCtx           /* A context for this module */
    {
     int pbid;                  /* ID of type "mpproblem.mqxp" */
     int options;               /* Runtime options */
     s_slvpb *probs;            /* List of created problems */
     void **nlctx;              /* Execution context of 'mmnl' */
    } s_slvctx;
```

3.1.8 Interface structure

The interface structure that is passed to Mosel holds the definition of the four lists (constants, subroutines, types, and services).

```
static XPRMdsointer dsointer=
    {
     sizeof(tabconst)/sizeof(XPRMdsoconst), tabconst,
     sizeof(tabfct)/sizeof(XPRMdsofct), tabfct,
     sizeof(tabtyp)/sizeof(XPRMdsotyp), tabtyp,
     sizeof(tabserv)/sizeof(XPRMdsoserv), tabserv
    };
```

In the inverse direction, the following two structures are employed for storing the API information received from Mosel and the *inter-module communication interface (IMCI)* of *mmnl*:

```
static XPRMnifct mm;       /* For storing the Mosel NI function table */
static mmnl_imci mmnl;     /* mmnl API */
```

3.1.9 Initialization function

The module initialization function performs the initialization of the solver library, communicates the module functionality (interface structures) to Mosel and retrieves the API definitions of the Mosel NI.

```
DSO_INIT myxprs_init(XPRMnifct nifct, int *interver,int *libver,
  XPRMdsointer **interf)
{
 int r;

 *interver=XPRM_NIVERS;         /* The interface version we are using */
 *libver=XPRM_MKVER(0,0,1);     /* The version of the module: 0.0.1 */
 *interf=&dsointer;             /* Our module interface structure */
 r=XPRSinit(NULL);              /* Initialize the solver */
 if((r!=0)&&(r!=32))
 {
  nifct->dispmsg(NULL,"myxprs: I cannot initialize Xpress Optimizer.\n");
  return 1;
 }
 mm=nifct;                      /* Retrieve the Mosel NI function table */
 return 0;
}
```

3.1.10 Implementation of subroutines

The first two entries of the list of subroutines concern the handling of module parameters, with the exception of `myxp_verbose` that is a setting for the module itself, all other parameters are straightforward mappings of solver control parameters published by the solver library.

```
/**** Getting a control parameter ****/
static int slv_lc_getpar(XPRMcontext ctx,void *libctx)
{
 s_slvctx *slctx;
 int n;
 double r;

 slctx=libctx;
 n=XPRM_POP_INT(ctx);
 switch(n)
 {
  case 0:
   XPRM_PUSH_INT(ctx,(slctx->options&OPT_VERBOSE)?1:0);
   break;
  case 1:
   XPRSgetdblcontrol(SLVCTX2PB(slctx)->xpb,XPRS_TIMELIMIT,&r);
   XPRM_PUSH_REAL(ctx,r);
   break;
  case 2:
```

```
   XPRSgetintattrib(SLVCTX2PB(slctx)->xpb,XPRS_LPSTATUS,&n);
   XPRM_PUSH_INT(ctx,n);
   break;
  case 3:
   XPRSgetdblattrib(SLVCTX2PB(slctx)->xpb,XPRS_LPOBJVAL,&r);
   XPRM_PUSH_REAL(ctx,r);
   break;
  default:
   mm->dispmsg(ctx,"myqxprs: Wrong control parameter number.\n");
   return XPRM_RT_ERROR;
 }
 return XPRM_RT_OK;
}

/**** Setting a control parameter ****/
static int slv_lc_setpar(XPRMcontext ctx,void *libctx)
{
 s_slvctx *slctx;
 int n;

 slctx=libctx;
 n=XPRM_POP_INT(ctx);
 switch(n)
 {
  case 0:
    slctx->options=XPRM_POP_INT(ctx)?(slctx->options|OPT_VERBOSE):(slctx->options&~OPT_VERBOSE);
    break;
  case 1:
    XPRSsetdblcontrol(SLVCTX2PB(slctx)->xpb,XPRS_TIMELIMIT,XPRM_POP_REAL(ctx));
    break;
  default:
    mm->dispmsg(ctx,"myqxprs: Wrong control parameter number.\n");
    return XPRM_RT_ERROR;
 }

 return XPRM_RT_OK;
}
```

The subroutine `getprobstat` exposes the Mosel problem status at the model level (this status value is populated after every solver run, see implementation of function `slv_optim` below).

```
static int slv_lc_getpstat(XPRMcontext ctx,void *libctx)
{
 XPRM_PUSH_INT(ctx,mm->getprobstat(ctx)&XPRM_PBRES);
 return XPRM_RT_OK;
}
```

The module functions implementing the `minimize` and `maximize` subroutines for linear and nonlinear constraints all map to the same function `slv_optim`.

```
/**** Linear objective ****/
static int slv_lc_maxim(XPRMcontext ctx,void *libctx)
{
 XPRMlinctr obj;

 obj=XPRM_POP_REF(ctx);
 return slv_optim(ctx,(s_slvctx *)libctx,OBJ_MAXIMIZE,obj,NULL);
}

static int slv_lc_minim(XPRMcontext ctx,void *libctx)
{
 XPRMlinctr obj;

 obj=XPRM_POP_REF(ctx);
 return slv_optim(ctx,(s_slvctx *)libctx,OBJ_MINIMIZE,obj,NULL);
```

```
}

/**** Nonlinear objective ****/
static int slv_lc_nmaxim(XPRMcontext ctx, void *libctx)
{
 XPRMnlctr obj;

 obj=XPRM_POP_REF(ctx);
 return slv_optim(ctx,(s_slvctx *)libctx,OBJ_MAXIMIZE,NULL,obj);
}

static int slv_lc_nminim(XPRMcontext ctx, void *libctx)
{
 XPRMnlctr obj;

 obj=XPRM_POP_REF(ctx);
 return slv_optim(ctx,(s_slvctx *)libctx,OBJ_MINIMIZE,NULL,obj);
}
```

The function `slv_optim` first clears any existing solution information (call to `slv_clearsol`), it then generates the matrix representation of the problem and loads it into the solver (`slv_loadmat`) and finally starts the actual solving process. After termination of the solver run it retrieves problem status and solution information in order to update the *mmnl* problem status information via a call to `setsolstat`.

```
/**** Optimize a problem: load it, then call the solver ****/
static int slv_optim(XPRMcontext ctx, s_slvctx *slctx, int objsense,
  XPRMlinctr obj, XPRMnlctr nlobj)
{
 int c,i;
 XPRMnlpdata nlpdata;
 s_slvpb *slpb;
 int result;
 double objval;

 slpb=SLVCTX2PB(slctx);
 slpb->saved_ctx=ctx;    /* Save current context for callbacks */
 slv_clearsol(ctx,slpb);

 /* Call function 'loadmat' to generate and load the matrix */
 nlpdata=slv_loadmat(ctx,slctx,obj,nlobj);

 if(nlpdata==NULL)
 {
  mm->dispmsg(ctx,"myqxprs: loadprob failed.\n");
  slpb->saved_ctx=NULL;
  return XPRM_RT_ERROR;
 }

 /* Set optimization direction */
 XPRSchgobjsense(slpb->xpb,(objsense==OBJ_MINIMIZE)?XPRS_OBJ_MINIMIZE:XPRS_OBJ_MAXIMIZE);

 mmnl->setsolstat(ctx,nlpdata,XPRM_PBSOL,0);  /* Solution available for callbacks */
 if((nlpdata->ngents==0) && (nlpdata->nsos==0))
 {
  /* Solve an LP problem */
  c=XPRSlpoptimize(slpb->xpb,"");
  if(c!=0)
  {
   mm->dispmsg(ctx,"myqxprs: optimisation failed.\n");
   slpb->saved_ctx=NULL;
   return XPRM_RT_ERROR;
  }

  /* Retrieve solution status */
```

```
 XPRSgetintattrib(slpb->xpb,XPRS_PRESOLVESTATE,&i);
 if(i&128)
 {
  XPRSgetdblattrib(slpb->xpb,XPRS_LPOBJVAL,&objval);
  result=XPRM_PBSOL;
 }
 else
 {
  objval=0;
  result=0;
 }
 XPRSgetintattrib(slpb->xpb,XPRS_LPSTATUS,&i);
 switch(i)
 {
  case XPRS_LP_OPTIMAL:        result|=XPRM_PBOPT; break;
  case XPRS_LP_INFEAS:         result|=XPRM_PBINF; break;
  case XPRS_LP_CUTOFF:         result|=XPRM_PBOTH; break;
  case XPRS_LP_UNFINISHED:     result|=XPRM_PBUNF; break;
  case XPRS_LP_UNBOUNDED:      result|=XPRM_PBUNB; break;
  case XPRS_LP_CUTOFF_IN_DUAL: result|=XPRM_PBOTH; break;
  case XPRS_LP_UNSOLVED:       result|=XPRM_PBOTH; break;
 }
}
else
{
 /* Solve an MIP problem */
 c=XPRSmipoptimize(slpb->xpb,"");
 if(c!=0)
 {
  mm->dispmsg(ctx,"myqxprs: optimisation failed.\n");
  slpb->saved_ctx=NULL;
  return XPRM_RT_ERROR;
 }

 /* Retrieve solution status */
 XPRSgetintattrib(slpb->xpb,XPRS_MIPSTATUS,&i);
 switch(i)
 {
  case XPRS_MIP_LP_NOT_OPTIMAL:
     objval=0;
     result=XPRM_PBUNF;
     break;
  case XPRS_MIP_LP_OPTIMAL:
     objval=0;
     result=XPRM_PBUNF;
     break;
  case XPRS_MIP_NO_SOL_FOUND:  /* Search incomplete: no solution */
     objval=0;
     result=XPRM_PBUNF;
     break;
  case XPRS_MIP_SOLUTION:     /* Search incomplete: there is a solution */
     XPRSgetdblattrib(slpb->xpb,XPRS_MIPOBJVAL,&objval);
     result=XPRM_PBUNF|XPRM_PBSOL;
     slpb->have|=HAVEMIPSOL;
     break;
  case XPRS_MIP_INFEAS:       /* Search complete: no solution */
     objval=0;
     result=XPRM_PBINF;
     break;
  case XPRS_MIP_OPTIMAL:      /* Search complete: best solution available */
     XPRSgetdblattrib(slpb->xpb,XPRS_MIPOBJVAL,&objval);
     result=XPRM_PBSOL|XPRM_PBOPT;
     slpb->have|=HAVEMIPSOL;
     break;
  case XPRS_MIP_UNBOUNDED:
     objval=0;
     result=XPRM_PBUNB;
```

```
      break;
   default:
      result=XPRM_PBOTH;
  }
  if(!(result&XPRM_PBSOL))
  {
   /* If no MIP solution try to get an LP solution */
   XPRSgetintattrib(slpb->xpb,XPRS_PRESOLVESTATE,&i);
   if(i&128)
   {
    XPRSgetdblattrib(slpb->xpb,XPRS_LPOBJVAL,&objval);
    result|=XPRM_PBSOL;
   }
  }
 }

 /* Record solution status and objective value */
 mmnl->setsolstat(ctx,nlpdata,result,objval);
 slpb->saved_ctx=NULL;
 return 0;
}
```

The module routine `slv_clearsol` frees up solution information held in the solver problem interface structures.

```
/**** Delete solution information ****/
static void slv_clearsol(XPRMcontext ctx, void *mipctx)
{
 Free(&(slpb->solval));
 Free(&(slpb->dualval));
 Free(&(slpb->rcostval));
 Free(&(slpb->slackval));
 slpb->have=0;
}

/**** Free + reset memory ****/
static void Free(void *ad)
{
 free(*(void **)ad);
 *(void **)ad=NULL;
}
```

The matrix generation routine invoked by `slv_optim` looks as follows (given that this implementation works with a quadratic solver any general nonlinear constraints or objectives are rejected). It first calls the *mmnl* subroutine `loadprob` to generate the matrix from the problem representation in Mosel. The maxtrix information is then copied into the structures that are expected by the solver's problem input routine (in our case: `XPRSloadmiqcqp`). The enumeration employs the *mmnl* functions `buildqtls`, `freeqtls`, and `strip`.

```
/**** Load the matrix into the solver ****/
static XPRMnlpdata slv_loadmat(XPRMcontext ctx, s_slvctx *slctx, XPRMlinctr obj,
  XPRMnlctr nlobj)
{
 XPRMnlpdata nlpdata;
 s_slvpb *slpb;
 static char ctcvr[7]={'N','G','L','E','R','1','2'};
 int c,r;
 char pbname[80];
 double *dqe;
 int *mqc1,*mqc2,nb_qp;
 int *qcrows,*qcnqtr,*qcmqc1,*qcmqc2;
 double *qcdqe;
 double objval;
```

```
slpb=SLVCTX2PB(slctx);
/* Call 'mmnl' function 'loadprob' to generate the matrix */
nlpdata=NULL;
c=mmnl->loadprob(ctx,*(slctx->nlctx),&nlpdata,
   NLOPT_COLWJAC|NLOPT_QUADLIN|
   ((slctx->options&OPT_VERBOSE)?NLOPT_VERBOSE:0)|
   ((slctx->options&OPT_LOADNAMES)?NLOPT_NAMES:0),
   obj,nlobj,NULL,NULL,NULL);
if(c!=0)
 return NULL;

if(nlpdata->nbqctr<nlpdata->nbnlctr)
{
 mm->dispmsg(ctx,"myqxprs: Problem contains nonlinear constraints.\n");
 mmnl->strip(ctx,nlpdata,NLSTRIP_ALL);
 return NULL;
}
if(nlpdata->objtype==1)        /* Objective is nonlinear */
{
 mm->dispmsg(ctx,"myqxprs: Objective is nonlinear.\n");
 mmnl->strip(ctx,nlpdata,NLSTRIP_ALL);
 return NULL;
}

/* Define the callback functions for exposing the solution values */
/* The solution arrays will be stored into our problem object (slpb) */
nlpdata->fctctx=slpb;
nlpdata->getsol_v=slv_getsol_v;
nlpdata->getsol_c=slv_getsol_c;

/* Xpress Optimizer uses a character code for each variable type */
for(c=0;c<nlpdata->nbctr;c++)
 nlpdata->ctrtype[c]=ctcvr[(int)(nlpdata->ctrtype[c])];

if(nlpdata->objtype==2)        /* Quadratic objective */
{
 XPRMnlqterm qtls;

 r=mmnl->buildqtls(ctx,*(slctx->nlctx),nlpdata->nlobj,1,&nb_qp,&qtls,NULL);
 if((r<0)||(r>=4))
 {
  mmnl->freeqtls(ctx,*(slctx->nlctx),qtls);
  mmnl->strip(ctx,nlpdata,NLSTRIP_ALL);
  return NULL;
 }

 if(nb_qp>0)
 {
  if((dqe=malloc((sizeof(double)+sizeof(int)*2)*nb_qp))==NULL)
  {
   mmnl->freeqtls(ctx,*(slctx->nlctx),qtls);
   mm->dispmsg(ctx,"myqxprs: Out of memory error.\n");
   mmnl->strip(ctx,nlpdata,NLSTRIP_ALL);
   return NULL;
  }
  mqc1=(int *)(dqe+nb_qp);
  mqc2=mqc1+nb_qp;
  for(c=0;c<nb_qp;c++)
  {
   mqc1[c]=mm->getvarnum(ctx,qtls[c].v1);
   mqc2[c]=mm->getvarnum(ctx,qtls[c].v2);
   dqe[c]=(qtls[c].v1==qtls[c].v2)?qtls[c].coeff*2:qtls[c].coeff;
  }
 }
 else
 {
  mqc1=mqc2=NULL;
```

```
  dqe=NULL;
 }
 mmnl->freeqtls(ctx,*(slctx->nlctx),qtls);
}
else
{
 nb_qp=0;
 mqc1=mqc2=NULL;
 dqe=NULL;
}

qcrows=qcnqtr=qcmqc1=qcmqc2=NULL;
qcdqe=NULL;
if(nlpdata->nbqctr>0)          /* Quadratic constraints */
{
 XPRMnlqterm qtls;
 int maxterms,curt;
 double rhs;

 qcrows=(int *)malloc(sizeof(int)*2*nlpdata->nbqctr);
 if(qcrows==NULL)
 {
  mm->dispmsg(ctx,"myqxprs: Out of memory error.\n");
  free(dqe);
  mmnl->strip(ctx,nlpdata,NLSTRIP_ALL);
  return NULL;
 }
 qcnqtr=qcrows+nlpdata->nbqctr;

 maxterms=0;
 curt=0;
 for(c=0;c<nlpdata->nbqctr;c++)
 {
  qcrows[c]=c;
  r=mmnl->buildqtls(ctx,*(slctx->nlctx),nlpdata->nlctr[c],1,
                                                  &(qcnqtr[c]),&qtls,&rhs);
  if((r<0)||(r>=4))            /* Should not be possible... */
  {
   mmnl->freeqtls(ctx,*(slctx->nlctx),qtls);
   free(qcdqe);
   free(qcmqc2);
   free(qcmqc1);
   free(qcrows);
   free(dqe);
   mmnl->strip(ctx,nlpdata,NLSTRIP_ALL);
   return NULL;
  }

  if(qcnqtr[c]+curt>maxterms)
  {
   int *newp1,*newp2;
   double *newp3;
   int newsize;

   newsize=maxterms+ALIGN128(qcnqtr[c]);
   newp1=newp2=NULL;
   newp3=NULL;
   if(((newp1=(int *)realloc(qcmqc1,newsize*sizeof(int)))==NULL)||
      ((newp2=(int *)realloc(qcmqc2,newsize*sizeof(int)))==NULL)||
      ((newp3=(double *)realloc(qcdqe,newsize*sizeof(double)))==NULL))
   {
    mm->dispmsg(ctx,"myqxprs: Out of memory error.\n");
    mmnl->freeqtls(ctx,*(slctx->nlctx),qtls);
    free((newp3!=NULL)?newp3:qcdqe);
    free((newp2!=NULL)?newp2:qcmqc2);
    free((newp1!=NULL)?newp1:qcmqc1);
    free(qcrows);
```

```
     free(dqe);
     mmnl->strip(ctx,nlpdata,NLSTRIP_ALL);
     return NULL;
    }
    else
    {
     qcmqc1=newp1;
     qcmqc2=newp2;
     qcdqe=newp3;
     maxterms=newsize;
    }
   }

   for(r=0;r<qcnqtr[c];r++,curt++)
   {
    qcmqc1[curt]=mm->getvarnum(ctx,qtls[r].v1);
    qcmqc2[curt]=mm->getvarnum(ctx,qtls[r].v2);
    qcdqe[curt]=(qtls[r].v1==qtls[r].v2)?qtls[r].coeff:qtls[r].coeff/2;
   }
   nlpdata->rhs[c]=-rhs;
   mmnl->freeqtls(ctx,*(slctx->nlctx),qtls);
  }
 }

 /* Load the matrix into the optimizer */
 sprintf(pbname,"xpb%p",slpb);
 r=XPRSloadmiqcqp(slpb->xpb,pbname,nlpdata->nbvar,
         nlpdata->nbctr,
         nlpdata->ctrtype,nlpdata->rhs,nlpdata->range,nlpdata->grdobj,
         nlpdata->colstart,NULL,nlpdata->cref,
         nlpdata->jac,nlpdata->dlb,nlpdata->dub,
         nb_qp,mqc1,mqc2,dqe,
         nlpdata->nbqctr,qcrows,qcnqtr,qcmqc1,qcmqc2,qcdqe,
         nlpdata->ngents,nlpdata->nsos,nlpdata->qgtype,
         nlpdata->mgcols,nlpdata->mplim,
         nlpdata->qstype,
         nlpdata->msstart,nlpdata->mscols,nlpdata->dref);
 free(qcdqe);
 free(qcmqc2);
 free(qcmqc1);
 free(qcrows);
 free(dqe);

 if(!r)
 {
  /* Set objective constant term */
  c=-1;
  objval=-nlpdata->fixobj;
  XPRSchgobj(slpb->xpb,1,&c,&objval);

 /* Release matrix information - it is no longer required */
 mmnl->strip(ctx,nlpdata,NLSTRIP_ALL);
 return nlpdata;
}
```

3.1.11 Implementation of *mmnl* solver interface functions

The following functions implement the *mmnl* solver interface functions 'getsol_v' and 'getsol_c' that are communicated to *mmnl* via the NLP problem definition interface structure of type `XPRMnlpdata` (for further detail see the documentation of `loadprob`). The 'getsol_v' and 'getsol_c' routines serve to retrieve the solution values for decision variables and constraints from the solver. These implementations retrieve the entire arrays at once and any subsequent calls return the information saved in the solver interface structures.

```
/**** Solution information for decision variables ****/
static double slv_getsol_v(XPRMcontext ctx, void *mctx, int what, int col)
{
 s_slvpb *slpb;
 int ncol;

 slpb=mctx;
 XPRSgetintattrib(slpb->xpb,XPRS_INPUTCOLS,&ncol);

 if(what)
 {
  if(!(slpb->have&HAVERCS))
  {
   if(slpb->rcostval==NULL)
   {
    if((slpb->rcostval=malloc(ncol*sizeof(double)))==NULL)
    {
     mm->dispmsg(ctx,"myqxprs: Out of memory error.\n");
     return 0;
    }
   }
   if(slpb->have&HAVEMIPSOL)  /* No rcost for a MIP => 0 */
    memset(slpb->rcostval,0,ncol*sizeof(double));
   else
    XPRSgetredcosts(slpb->xpb,NULL,slpb->rcostval,0,ncol-1);
   slpb->have|=HAVERCS;
  }
  return slpb->rcostval[col];
 }
 else
 {
  if(!(slpb->have&HAVESOL))
  {
   if(slpb->solval==NULL)
   {
    if((slpb->solval=malloc(ncol*sizeof(double)))==NULL)
    {
     mm->dispmsg(ctx,"myqxprs: Out of memory error.\n");
     return 0;
    }
   }
   XPRSgetsolution(slpb->xpb,NULL,slpb->solval,0,ncol-1);
   slpb->have|=HAVESOL;
  }
  return slpb->solval[col];
 }
}
```

```
/**** Solution information for linear constraints ****/
static double slv_getsol_c(XPRMcontext ctx, void *mctx, int what, int row)
{
 s_slvpb *slpb;
 int nrow;

 slpb=mctx;
 XPRSgetintattrib(slpb->xpb,XPRS_INPUTROWS,&nrow);

 if(what)
 {
  if(!(slpb->have&HAVEDUA))
  {
   if(slpb->dualval==NULL)
   {
    if((slpb->dualval=malloc(nrow*sizeof(double)))==NULL)
    {
     mm->dispmsg(ctx,"myqxprs: Out of memory error.\n");
     return 0;
```

```
    }
   }
   if(slpb->have&HAVEMIPSOL)  /* No dual for a MIP => 0 */
    memset(slpb->dualval,0,nrow*sizeof(double));
   else
    XPRSgetduals(slpb->xpb,NULL,slpb->dualval,0,nrow-1);
   slpb->have|=HAVEDUA;
  }
  return slpb->dualval[row];
 }
 else
 {
  if(!(slpb->have&HAVESLK))
  {
   if(slpb->slackval==NULL)
   {
    if((slpb->slackval=malloc(nrow*sizeof(double)))==NULL)
    {
     mm->dispmsg(ctx,"myqxprs: Out of memory error.\n");
     return 0;
    }
   }
   XPRSgetslacks(slpb->xpb,NULL,slpb->slackval,0,nrow-1);
   slpb->have|=HAVESLK;
  }
  return slpb->slackval[row];
 }
}
```

3.1.12 Implementation of services

The RESET service function creates a new module context if none is provided in the argument, on a subsequent call where the module context argument is populated this context will be released after freeing all data structures that may have been created via the solver library.

```
/**** Reset the myqxprs interface for a run ****/
static void *slv_reset(XPRMcontext ctx, void *libctx)
{
 XPRMdsolib dso;
 s_slvctx *slctx;

 /* End of execution: release context */
 if(libctx!=NULL)
 {
  slctx=libctx;

  /* Release all remaining problems */
  while(slctx->probs!=NULL)
  {
   slv_pb_delete(ctx,slctx,slctx->probs,-1);
  }
  free(slctx);
  return NULL;
 }
 else
 {
  /* Begin of execution: create context */
  if((slctx=malloc(sizeof(s_slvctx)))==NULL)
  {
   mm->dispmsg(ctx,"myqxprs: Out of memory error.\n");
   return NULL;
  }
  memset(slctx,0,sizeof(s_slvctx));

  /* Load interface functions and context of 'mmnl' */
```

```
  if(((dso=mm->finddso("mmnl"))==NULL)||
     ((slctx->nlctx=mm->getdsoctx(ctx,dso,(void **)(&(mmnl))))==NULL))
  {
   mm->dispmsg(ctx,"myqxprs: mmnl not found.\n");
   free(slctx);
   return NULL;
  }

  /* Record the problem ID of our problem type */
  mm->gettypeprop(ctx,mm->findtypecode(ctx,"mpproblem.mqxp"),XPRM_TPROP_PBID,
              (XPRMalltypes*)&(slctx->pbid));
  return (void *)slctx;
 }
}
```

The UNLOAD service terminates the solver library (frees up the licence) that has been initialized from the module initialization.

```
/**** Called when unloading the library ****/
static void slv_quitlib(void)
{
 if(mm!=NULL)
 {
  XPRSfree();
  mm=NULL;
 }
}
```

The implementation of the module parameter access services PARAM and PARLST is similar to those of other example modules (see Mosel NI User Guide), they are listed here for completeness' sake.

```
/**** Find a control parameter ****/
static int slv_findparam(const char *name, int *type, int why, XPRMcontext ctx,
                void *libctx)
{
 int n;

 for(n=0;n<SLV_NBPARAM;n++)
 {
  if(strcmp(name,myxprsparams[n].name)==0)
  {
   *type=myxprsparams[n].type;
   return n;
  }
 }

 return -1;
}

/**** Return the next parameter for enumeration ****/
static void *slv_nextparam(void *ref, const char **name, const char **desc,
                int *type)
{
 size_t cst;

 cst=(size_t)ref;
 if(cst>=SLV_NBPARAM)
  return NULL;
 else
 {
  *name=myxprsparams[cst].name;
  *type=myxprsparams[cst].type;
  *desc=NULL;
  return (void *)(cst+1);
 }
```

```
}
```

3.1.13 Handling optimization problems

Each Mosel model creates a default optimization problem of type `mpproblem` holding the constraints that are defined in the model. Further (sub)problems can be defined explicitly by the model developer. A solver module needs to implement an extension to the `mpproblem` type. Typically, the underlying data structure will include a reference to the solver problem representation, some status flags and structures to store solution information (see definition in Section 3.1.5 above). For our QCQP solver example with Xpress Optimizer we have implemented the type handling routines to create, delete, and copy optimization problems (for a full list of type handling routines the reader is referred to the Section 'Table of types' of the Mosel NI Reference Manual). If the underlying solver can only handle a single problem, the implementation of the 'create' routine should prevent the creation of more than one problem and a 'copy' routine is most likely not required.

The following implementation of a problem creation routine for Xpress Optimizer creates the Optimizer problem, redirects the Optimizer output onto Mosel and it also defines some logging callbacks in order to intercept a program interruption.

```
/**** Create a new "problem" ****/
static void *slv_pb_create(XPRMcontext ctx, void *libctx, void *toref, int type)
{
 s_slvctx *slctx;
 s_slvpb *slpb;
 int i;

 slctx=libctx;
 if((slpb=malloc(sizeof(s_slvpb)))==NULL)
 {
  mm->dispmsg(ctx,"myqxprs: Out of memory error.\n");
  return NULL;
 }
 memset(slpb,0,sizeof(s_slvpb));
 i=XPRScreateprob(&(slpb->xpb));
 if((i!=0)&&(i!=32))
 {
  mm->dispmsg(ctx,"myqxprs: I cannot create the problem.\n");
  free(slpb);
  return NULL;
 }
 slpb->slctx=slctx;
 /* Redirect solver messages to the Mosel streams */
 XPRSaddcbmessage(slpb->xpb,slv_cb_output,slpb,0);
 XPRSsetintcontrol(slpb->xpb,XPRS_OUTPUTLOG,1);

 /* Define log callbacks to report program interruption */
 XPRSaddcblplog(slpb->xpb,(void*)slv_cb_stopxprs,slpb,0);
 XPRSaddcbcutlog(slpb->xpb,(void*)slv_cb_stopxprs,slpb,0);
 XPRSaddcbmiplog(slpb->xpb,(void*)slv_cb_stopxprs,slpb,0);
 XPRSaddcbbarlog(slpb->xpb,(void*)slv_cb_stopxprs,slpb,0);

 if(slctx->probs!=NULL)
 {
  slpb->next=slctx->probs;
  slctx->probs->prev=slpb;
 }
 /* else we are creating the main problem */

 slctx->probs=slpb;
 return slpb;
}
```

The problem deletion routine needs to update the list of problems saved in the solver problem interface structure and it clears any solution information that might have been stored for this problem.

```
/**** Delete a "problem" ****/
static void slv_pb_delete(XPRMcontext ctx, void *libctx, void *todel, int type)
{
 s_slvctx *slctx;
 s_slvpb *slpb;

 slctx=libctx;
 slpb=todel;
 slv_clearsol(ctx,slpb);
 XPRSdestroyprob(slpb->xpb);
 if(slpb->next!=NULL)  /* Last in list */
  slpb->next->prev=slpb->prev;
 if(slpb->prev==NULL)  /* First in list */
  slctx->probs=slpb->next;
 else
  slpb->prev->next=slpb->next;
 free(slpb);
}
```

A problem is copied without duplicating the solution information.

```
/**** Copy/reset/append problems: simply clear data of the destination ****/
static int slv_pb_copy(XPRMcontext ctx, void *libctx, void *toinit, void *src,
  int ust)
{
 s_slvpb *slpb;

 slpb=toinit;
 if(XPRM_CPY(ust)<XPRM_CPY_APPEND) slv_clearsol(ctx,slpb);
 return 0;
}
```

3.1.14 Generating names for matrix entries and matrix output

While developing a solver interface it may be helpful to have the possibility of writing out a matrix representation of the problem held in the solver.

In our example implementation the matrix gets loaded into the solver through the call to optimization, so writing out the problem matrix from the solver needs to take place after this call:

```
  setparam("myxp_loadnames", true)
  maximize(MyObj)
  writeprob("mymat.lp", "l")
```

When writing out a matrix for debugging purposes one might expect to be able to match the rows and columns to the Mosel modeling entities via their respective names. By default, Mosel does not generate any names for decision variables or constraints in order to maintain a low memory footprint. This feature needs to be added explicitly into the implementation of the `slv_loadmat` routine that we have seen earlier in this chapter. After a call to the NI function `genmpnames` the function `slv_loadnames` collects the resulting names are into the corresponding data structures that are expected by the solver library (uploading names for rows, columns, and SOS separately in the case of Xpress Optimizer).

```
/**** Load the matrix into the optimizer ****/
static XPRMnlpdata slv_loadmat(XPRMcontext ctx, s_slvctx *slctx, XPRMlinctr obj,
  XPRMnlctr nlobj)
{
 /* ... load the problem via mmnl routine 'loadprob' ... */
```

```
 /* Generate names for the linear part of the problem */
 if(slctx->options&OPT_LOADNAMES)
  mm->genmpnames(ctx,MM_KEEPOBJ,NULL,0);

 /* ... load the problem matrix into the solver ... */

  /* Load names if requested */
  if(slctx->options&OPT_LOADNAMES)
   slv_loadnames(ctx,slpb,nlpdata);

 /* ... release matrix information that is no longer required... */
}

/**** Load names into the optimizer ****/
static void slv_loadnames(XPRMcontext ctx, s_slvpb *slpb, XPRMnlpdata nlpdata)
{
 char *names,*n;
 size_t totlen,totlen2;
 size_t l;
 int c,r;

 totlen=0;
 for(c=0;c<nlpdata->nbvar;c++)
 {
  l=strlen(mm->getmpname(ctx,MM_MPNAM_COL,c));
  totlen+=l+1;
 }
 totlen2=0;
 for(c=0;c<nlpdata->nb_nlctrs;c++)
 {
  l=strlen(nlpdata->names[c]);
  totlen2+=l+1;
 }
 for(;c<nlpdata->nbctr;c++)
 {
  l=strlen(mm->getmpname(ctx,MM_MPNAM_ROW,c-nlpdata->nb_nlctrs));
  totlen2+=l+1;
 }
 if(totlen<totlen2) totlen=totlen2;
 totlen2=0;
 for(c=0;c<nlpdata->nsos;c++)
 {
  l=strlen(mm->getmpname(ctx,MM_MPNAM_SOS,c));
  totlen2+=l+1;
 }
 if(totlen<totlen2) totlen=totlen2;

 if((names=malloc(totlen))==NULL)
  mm->dispmsg(ctx,"myqxprs: Not enough memory for loading the names.\n");
 else
 {
  n=names;
  for(c=0;c<nlpdata->nbvar;c++)
   n+=strlen(strcpy(n,mm->getmpname(ctx,MM_MPNAM_COL,c)))+1;
  if((r=XPRSaddnames(slpb->xpb,2,names,0,nlpdata->nbvar-1))!=0)
   mm->dispmsg(ctx,"myqxprs: Error when executing  `%s'.\n","addnames");
  if(!r && (nlpdata->nbctr>0))
  {
   n=names;
   for(c=0;c<nlpdata->nb_nlctrs;c++)
    n+=strlen(strcpy(n,nlpdata->names[c]))+1;
   for(;c<nlpdata->nbctr;c++)
    n+=strlen(strcpy(n,mm->getmpname(ctx,MM_MPNAM_ROW,c-nlpdata->nb_nlctrs)))+1;
   if((r=XPRSaddnames(slpb->xpb,1,names,0,nlpdata->nbctr-1))!=0)
    mm->dispmsg(ctx,"myqxprs: Error when executing  `%s'.\n","addnames");
  }
  if(!r && (nlpdata->nsos>0))
```

```
  {
   n=names;
   for(c=0;c<nlpdata->nsos;c++)
    n+=strlen(strcpy(n,mm->getmpname(ctx,MM_MPNAM_SOS,c)))+1;
   if((r=XPRSaddnames(slpb->xpb,3,names,0,nlpdata->nsos-1))!=0)
    mm->dispmsg(ctx,"myqxprs: Error when executing  `%s'.\n","addnames");
  }
  free(names);
 }
}
```

The loading of names is triggered from `slv_loadmat` subject to the presence of the option flag `LOADNAMES` that is set via a new module parameter `myxp_loadnames` which is declared via the following entry in the list of parameters ('myxprsparams'):

```
 {"myxp_loadnames", XPRM_TYP_BOOL|XPRM_CPAR_READ|XPRM_CPAR_WRITE}
```

The `writeprob` routine shown in the Mosel model extract at the beginning of this section needs to be declared in the list of subroutines structure ('tabfct') by adding a line like the following:

```
  {"writeprob",2103,XPRM_TYP_NOT,2,"ss",slvlc_writepb}
```

And the actual implementation in the function `slv_lc_writepb` consists of a call the the solver's matrix output function, along with some error handling such as a check for write access to the specified location:

```
static int slv_lc_writepb(XPRMcontext ctx, void *libctx)
{
 s_slvpb *slpb;
 int rts;
 char *dname,*options;
 char ename[MM_MAXPATHLEN];

 slpb=SLVCTX2PB((s_slvctx*)libctx);
 dname=MM_POP_REF(ctx);
 options=MM_POP_REF(ctx);
 if((dname!=NULL)&&            /* Make sure that the file can be created */
    (mm->pathcheck(ctx,dname,ename,MM_MAXPATHLEN,MM_RCHK_WRITE|MM_RCHK_IODRV)==0))
 {
  slpb->saved_ctx=ctx;         /* Save current context for callbacks */
  rts=XPRSwriteprob(slpb->xpb,XNLSconvstrto(XNLS_ENC_FNAME,ename,-1,NULL),options);
  slpb->saved_ctx=NULL;
  if(rts)
  {
   mm->dispmsg(ctx,"myqxprs: Error when executing  `writeprob'.\n");
   return XPRM_RT_IOERR;
  }
  else
   return XPRM_RT_OK;
 }
 else
 {
  mm->dispmsg(ctx,"myqxprs: Cannot write to '%s'.\n",dname!=NULL?dname:"");
  return XPRM_RT_IOERR;
 }
}
```

3.2 NI library interface functions of mmnl

The following list gives an overview of the functions published via the NI C library interface of *mmnl* for which we give detailed descriptions later.

`buildqtls`	Generate data required for loading a quadratic expression.	p. 26
`eval`	Evaluate a constraint or objective on the provided assignment of the decision variables.	p. 29
`freeqtls`	Release the resources allocated by `buildqtls` for a given expression.	p. 27
`getctrnum`	Get the row number of a linear constraint.	p. 32
`getnlctrnum`	Get the row number of a nonlinear constraint.	p. 33
`grad`	Evaluate the gradiant of a constraint or objective on the provided assignment of the decision variables.	p. 30
`loadprob`	Produce a matrix representation of the current problem.	p. 25
`setsolstat`	Set the problem status.	p. 31
`strip`	Release the resources allocated by `loadprob`.	p. 28

loadprob

Purpose

Produce a matrix representation of the current problem.

Synopsis

```
int loadprob(XPRMcontext ctx, void *nlctx, XPRMnlpdata *nlpd, int options,
      XPRMlinctr obj, XPRMnlctr nlobj, XPRMmpvar *extra, int
      (*preproc)(XPRMcontext ctx, void *lctx, XPRMmatrix *m), void *lctx);
```

Arguments

`ctx` Mosel's execution context

`nlctx` *mmnl* execution context

`nlpd` Reference pointer where the matrix information will be returned (must be initialized with `NULL`)

`options` Bit encoded options:

`NLOPT_VERBOSE`	display a message if the problem is found infeasible
`NLOPT_SPRSOBJ`	the objective is encoded in sparse format
`NLOPT_FULLJAC`	allocate full Jacobian (otherwise only the linear part)
`NLOPT_COLWJAC`	Jacobian represented columnwise
`NLOPT_CTRBNDS`	bounds represented by constraints
`NLOPT_RAWBNDS`	do not default lower bounds to 0
`NLOPT_QUADLIN`	generate Jacobian for linear part of quadratic constraints
`NLOPT_NAMES`	generate names for nonlinear constraints

`obj` Linear objective (or `NULL`)

`nlobj` Nonlinear objective (or `NULL`)

`extra` Array of decision variables to be added to the matrix even if they do not appear in any constraint. The last reference of the array must be `NULL`. This parameter may be `NULL`.

`preproc` reserved for future use

`lctx` reserved for future use

Return value

0 if successful, `XPRM_PBINF` if the problem was found infeasible (due to inconsistent bounds) and -1 if an error has occurred.

Further information

1. This routine converts the current problem into a matrix representation suitable for solvers. The resulting data is stored in the `s_nlpdata` datastructure, please refer to the `mmnl.h` include file for a detailed description.

2. After calling `loadprob` the structure `nlpd` should be completed by defining its fields `fctctx`, `getsol_v`, `getsol_c` and `delmat` (none of the 3 routines is mandatory, see `mmnl.h` for the function prototypes). The function `delmat` is used when a matrix previously loaded via `loadprob` becomes useless (for instance before loading an updated version of the matrix).
   If the module makes available solution values (which is done by calling the function `setprobstat`), the function `getsol_v` is called whenever Mosel requires the solution value (with second parameter 0) or the reduced cost (with second parameter 1) of the given column (third parameter: the first column has number 0).
   Similarly, `getsol_c` is used to get the slack value (second parameter is 0) or the dual value (second parameter is 1) of the given row (third parameter: the first row has number 0).

Related topics

`strip`.

buildqtls

Purpose

Generate data required for loading a quadratic expression.

Synopsis

```
int buildqtls(XPRMcontext ctx, void *nlctx, XPRMnlctr qctr, int onlyqterms,
      int *nbterm, XPRMnlqterm *qtls, double *cstt);
```

Arguments

`ctx`	Mosel's execution context
`nlctx`	*mmnl* execution context
`qctr`	Quadratric constraint or objective to process
`onlyqterms`	Ignore linear terms
`nbterm`	Reference where the number of qterms is returned
`qtls`	Reference where the list of qterms is returned
`cstt`	Reference where the constant terms is returned

Return value

0 if the expression is constant, 1 for a linear, 2 for a quadratic, 4 for a nonlinear expression and -1 in case of error

Further information

1. This function computes the list of quadratic terms of a quadratic expression. If the option `onlyqterms` is not specified then linear terms are also included in the list (the second variable reference of a linear term is `NULL`).

2. The list `qtls` is allocated by the routine: each call to `buildqtls` must be followed by a call to `freeqtls` in order to release the allocated memory.

Related topics

`freeqtls`.

freeqtls

Purpose

Release the resources allocated by `buildqtls` for a given expression.

Synopsis

```
void freeqtls(XPRMcontext ctx, void *nlctx, XPRMnlqterm qtls);
```

Arguments

`ctx` Mosel's execution context

`nlctx` *mmnl* execution context

`qtls` A list of quadratic terms returned by `buildqtls`

Related topics

`buildqtls`.

strip

Purpose

Release the resources allocated by `loadprob`.

Synopsis

```
void strip(XPRMcontext ctx, XPRMnlpdata nlpdata, int what);
```

Arguments

`ctx`	Mosel's execution context
`nlpd`	Problem information as generated by `loadprob`
`what`	Must be `NLSTRIP_ALL`

Further information

This function should be called after the problem has been loaded into the solver before running the solving procedure in order to release matrix information that is usually not required to solve a problem.

Related topics

`loadprob`.

eval

Purpose

Evaluate a constraint or objective on the provided assignment of the decision variables.

Synopsis

```
int eval(XPRMcontext ctx, XPRMnlpdata nlpdata, int nb, const double *x,
      double *val);
```

Arguments

`ctx`	Mosel's execution context
`nlpd`	Problem information as generated by `loadprob`
`nb`	Constraint number to evaluate or `NLEVAL_OBJ` for the objective, `NLEVAL_ALL` for everything, `NLEVAL_NL` for nonlinear constraints or `NLEVAL_LIN` linear constraints.
`x`	Values assigned to the decision variables
`val`	Reference that receives the result of the evaluation. In case of multiple evaluations, this must be an array large enough to store all the results

Return value

0 if successful or 1 in case of error

Further information

When the function is called with `NLEVAL_ALL` the array `val` must be of size `nlpdata->nbctr`, with `NLEVAL_NL` it requires `nlpdata->nbnlctr` entries and for `NLEVAL_LIN` it contains `nlpdata->nbctr-nlpdata->nbnlctr` values.

grad

Purpose

Evaluate the gradiant of a constraint or objective on the provided assignment of the decision variables.

Synopsis

```
int grad(XPRMcontext ctx, XPRMnlpdata nlpdata, int nb, double *grad, const
      double *x, double *val, int dense);
```

Arguments

`ctx`	Mosel's execution context
`nlpd`	Problem information as generated by `loadprob`
`nb`	Constraint number to evaluate or `NLEVAL_OBJ` for the opbjective, `NLEVAL_ALL` for everything, `NLEVAL_NL` for nonlinear constraints or `NLEVAL_LIN` linear constraints.
`grad`	Reference that receives the gradiant. In case of multiple evaluations, this must be an array large enough to store all the results
`x`	Values assigned to the decision variables
`val`	Reference that receives the result of the evaluation (or `NULL`). In case of multiple evaluations, this must be an array large enough to store all the results
`dense`	When processing a single constraint, return data in dense form

Return value

0 if successful or 1 in case of error

Further information

1. This function computes the gradient and evaluation of the objective or constraints. The `val` argument is handled like with function `eval`.

2. The `dense` option is used only when processing a single constraint or the objective: the result is returned in the form of a dense array of coefficients, the `grad` array must be of size `nlpdata->nbvar`. In all other cases the result is returned in the form of a sparse array: for the objective the size of the array must be `nlpdata->nzrobj` and the variable references have to be obtained from `nlpdata->vrefobj`. For a constraint `c` the size of the array is `nlpdata->ctrstart[c+1]-nlpdata->ctrstart[c]` and the variable references are taken from `nlpdata->vref[nlpdata->ctrstart[c]]` to `nlpdata->vref[nlpdata->ctrstart[c+1]]`.

3. If the function is used to process all constraints (`NLEVAL_ALL`) the result array is of size `nplpdata->nzr`, for linear constraints only (`NLEVAL_LIN`) it is `nplpdata->nzr_lin` and for the nonlinear constraints (`NLEVAL_NL`) it is `nplpdata->nzr-nplpdata->nzr_lin`.

setsolstat

Purpose

Set the problem status.

Synopsis

```
void setsolstat(XPRMcontext ctx, XPRMnlpdata nlpdata, int status, double
      objval);
```

Arguments

`ctx` Mosel's execution context

`nlpd` Problem information as generated by `loadprob`

`status` Bit encoded problem status:

`XPRM_PBSOL`	A solution is available,
`XPRM_PBOPT`	Optimal solution found,
`XPRM_PBUNF`	Optimization interrupted,
`XPRM_PBINF`	Problem is infeasible,
`XPRM_PBUNB`	Problem is unbounded,
`XPRM_PBOTH`	Optimization failed

`objval`

Further information

This function sets the problem status in Mosel. It is usually invoked after an optimization procedure in order to tell Mosel the result of the operation and whether a solution is available. Note that the status `XPRM_PBSOL` can be combined with the other status values (*e.g.* `XPRM_PBOPT|XPRM_PBSOL`).

getctrnum

Purpose

Get the row number of a linear constraint.

Synopsis

```
int getctrnum(XPRMcontext ctx, XPRMnlpdata nlpdata, XPRMlinctr ctr);
```

Arguments

`ctx`	Mosel's execution context
`ctr`	Reference to a linear constraint

Return value

`>= 0`	Row number of the linear constraint
`-1`	Row number not available

Further information

This function returns the row number of a linear constraint. A value of -1 is returned if no problem is available or if the constraint does not belong to the current problem.

Related topics

`getnlctrnum`.

getnlctrnum

Purpose

Get the row number of a nonlinear constraint.

Synopsis

```
int getnlctrnum(XPRMcontext ctx, XPRMnlpdata nlpdata, XPRMnlctr nlctr);
```

Arguments

`ctx` Mosel's execution context

`nlctr` Reference to a nonlinear constraint

Return value

`>= 0` Row number of the nonlinear constraint

`-1` Row number not available

Further information

This function returns the row number of a nonlinear constraint. A value of -1 is returned if no problem is available or if the constraint does not belong to the current problem.

Related topics

`getctrnum`.

4 Using module *nlsolv*

The *nlsolv* module makes it possible to export a problem in the `.nl` file format, solve it using an external solver supporting this format, and retrieve solution values back into the Mosel model. To use this module the following line must be included in the header of the Mosel model file:

```
uses 'nlsolv'
```

Using alternative solvers through *nlsolv* does not require any additional C coding, after installing a suitable solver executable the Mosel model merely needs to be configured via a couple of parameter settings.

This section explains how to setup *nlsolv* for various solvers and it also provides a documentation of the parameters and subroutines defined by this module.
For the handling of nonlinear expressions / constraints (type `nlctr`) the module *nlsolv* uses the functionality provided by the module *mmnl* that forms part of the standard Mosel distribution. All other functionality defined by *mmnl*, such as its subroutines for handling initial values, is equally available in models using *nlsolv*.
The module *nlsolv* supports the definition of multiple problems—it extends the definition of the type `mpproblem`.

4.1 Example: using nlsolv for Nonlinear Programming

The following example that calculates the shape of a hanging chain with `N` chainlinks and fixed endpoints is the same model as in Section 3.1.1. Since *nlsolv* does not include any solver, the parameter `NL_solver` needs to be set to the (external) solver that is to be used. This may be any nonlinear solver supporting the `.nl` format and providing the functionality required by the particular model. If the name of the executable is different from the name of the solver prefix, its name needs to be specified via the parameter `NL_solverpath` as shown in this example for Xpress Optimizer.

Other features shown by this model are the setting of initial values for decision variables and the retrieval of solution information at the end of the solver run—these functionalities are provided by the module *mmnl* that is included by *nlsolv* (see Chapter *mmnl* of the Mosel Language Reference Manual for the full documentation of *mmnl*).

```
model "catenary - NL version"
 uses "nlsolv"

 parameters
   SOLVER="xpress"
   SOLVERPATH="xpressasl"
   SOLVEROPTIONS=""

   N = 100                        ! Number of chainlinks
   L = 1                          ! Difference in x-coordinates of endlinks
   H = 2*L/N                      ! Length of each link
 end-parameters

 declarations
   RN = 0..N
   x: array(RN) of mpvar          ! x-coordinates of endpoints of chainlinks
   y: array(RN) of mpvar          ! y-coordinates of endpoints of chainlinks
   PotentialEnergy: linctr        ! Objective function
   Link: array(range) of nlctr    ! Constraints
 end-declarations

 forall(i in RN) x(i) is_free
 forall(i in RN) y(i) is_free

! Objective: minimise the potential energy
```

```
 PotentialEnergy:= sum(j in 1..N) (y(j-1)+y(j))/2

! Bounds: positions of endpoints
 x(0) = 0; y(0) = 0; x(N) = L; y(N) = 0

! Constraints: positions of chainlinks
 forall(j in 1..N)
   Link(j):= (x(j)-x(j-1))^2+(y(j)-y(j-1))^2 <= H^2

! Setting start values
 forall(j in RN) setinitval(x(j), j*L/N)
 forall(j in RN) setinitval(y(j), 0)

! Configuration of the solver
 setparam("nl_verbose", true)
 setparam("nl_solver", SOLVER)
 if SOLVERPATH<>"": setparam("nl_solverpath", SOLVERPATH)
 if SOLVEROPTIONS<>"": setparam("nl_options", SOLVEROPTIONS)

! Solve the problem
 minimise(PotentialEnergy)

! Solution reporting
 if getprobstat<>NL_OPT and getprobstat<>NL_UNF then
   writeln("No solution available. Solver status: ", getparam("NL_STATUS"))
 else
   writeln("Solution: ", getobjval)
   setparam("realfmt", "%10.5f")
   forall(j in RN) writeln(getsol(x(j)), " ", getsol(y(j)))
 end-if

end-model
```

4.2 Configuring a solver

Follow these steps to configure a solver:

1. Download a solver executable working with the NL format
   - the `xpressasl` driver used in the example above is available at https://ampl.com/products/solvers/linear-solvers/xpress/
     Please note that the `xpressasl` interface only supports LP, MIP, and QCQP.
   - drivers for open source solvers are available at https://ampl.com/products/solvers/open-source
2. Make sure the solver executable is on the `PATH`
   - alternatively, specify `NL_SOLVERPATH`
3. Configure your model to use the solver:
   - `NL_SOLVER` always needs to be set
   - use `NL_SOLVERPATH` if
     - the solver executable is not on the path, or
     - the executable name is different from the solver prefix in `NL_SOLVER`, or
     - the executable requires additional options (define a batch script)
   - optional: define specific solver settings in `NL_OPTIONS`

4. If using an Xpress license, set the `XPRESS` environment variable to the location of your license file. (Note that this is different from the `XPAUTH_PATH` environment variable normally used by Xpress.)

Here are some configuration examples for *nlsolv* controls under Windows:

- Xpress Optimizer (LP, MIP, QCQP)

```
setparam("NL_SOLVER", "xpress")
setparam("NL_SOLVERPATH", "xpressasl.exe")
```

- Knitro (requires a separate license, NLP)

```
setparam("NL_SOLVER", "knitro")
setparam("NL_SOLVERPATH", "knitroampl.exe")
```

- Cplex (requires a separate license, LP, MIP, QCQP)

```
setparam("NL_SOLVER", "cplex")
setparam("NL_SOLVERPATH", "cplexamp.exe")
```

- ipopt (NLP)

```
setparam("NL_SOLVER", "ipopt")
```

- cbc (NLP)

```
setparam("NL_SOLVER", "cbc")
setparam("NL_SOLVERPATH", "cbc.bat")         ! Contents:    cbc %3 -AMPL
```

- SCIP (NLP)

```
setparam("NL_SOLVER", "scip")
setparam("NL_SOLVERPATH", "scip.bat")        ! Contents:    scipampl %3 -AMPL
```

4.3 Control parameters

Via the `getparam` function and the `setparam` procedure it is possible to access the following control parameters of module *nlsolv* (the reader is reminded that parameters may be spelled with lower or upper case letters or a mix of both):

`NL_binary`	Set the NL file format.	p. 37
`NL_options`	NL solver options.	p. 37
`NL_probname`	Problem name for the NL file.	p. 37
`NL_solver`	NL solver to be used.	p. 37
`NL_solverpath`	NL solver path.	p. 38
`NL_status`	Solver status code.	p. 38
`NL_verbose`	Enable/disable message printing.	p. 39

NL_binary

Description Select ASCII or binary format for NL file.

Type Boolean, read/write

Values
`true` use binary format (default)
`false` use ASCII format

Default value `true`

Affects routines `exportnl`, `maximize`/`minimize`

Module nlsolv

NL_options

Description Set NL solver options (stopping criteria, tolerances *etc.*). These options are typically specific to a given solver.

Type String, read/write

Default value `""` (empty string)

Note Options are passed to the solver via an environment variable. The name of this variable is built from the solver name as stated by the parameter `NL_solver`. For instance for solver `"xpress"`, parameters will be stored in the environment variable `"xpress_options"`.

Affects routines `maximize`/`minimize`

Module nlsolv

NL_probname

Description Read/set the problem name used to create the `.nl` file that is passed to the solver (this name may contain a full path). If set to the empty string (default value), a unique name with a path to the temporary directory of the operating system is generated.

Type String, read/write

Default value `""` (empty string)

Affects routines `maximize`/`minimize`

Module nlsolv

NL_solver

Description Set the NL solver to be used for solving a problem.

Type String, read/write

Default value `""` (empty string)

Notes

1. If solver options are set via `NL_options`, the parameter `NL_solver` must contain the solver name prefix to be applied to the options name.
2. If `NL_solverpath` is not specified, then the *nlsolv* module tries to launch a solver executable using the name specified via `NL_solver`. In this case the solver program must be located in the current search path of the operating system (usually defined by the `PATH` environment variable).
3. Alternatively, the full path to the solver program (executable or batch file) can be set using the parameter `NL_solverpath`.

Affects routines `maximize/minimize`.

See also `NL_solverpath`, `NL_options`

Module nlsolv

NL_solverpath

Description Set the path to the NL solver to be used for solving a problem.

Type String, read/write

Default value `""` (empty string)

Notes

1. If this parameter is defined, it is used to determine the solver executable that is to be used by subsequent calls to `maximize/minimize`.
2. This parameter needs to be defined in addition to `NL_solver` when
   - the solver program cannot be located using the current search path of the operating system, or
   - the name of the solver program is different from the options prefix defined in `NL_solver`, or
   - the solver is not invoked directcly but via some batch script, for example in order to add some solver-specific command line options.

Affects routines `maximize/minimize`.

See also `NL_solver`

Module nlsolv

NL_status

Description Status code returned by the solver after its processing.

Type Integer, read only

Values

`<0`	no solver has been called yet
`<100`	solved
`<200`	solved but error likely
`<300`	problem is infeasible
`<400`	problem is unbounded
`<500`	solver stopped by some limit
`<600`	failure during processing

Default value −1 before the problem is processed

Note Refer to the solver documentation for further explanation.

Set by routines `maximize/minimize`.

See also `getprobstat`

Module nlsolv

NL_verbose

Description Enables/disables message printing by the NL module.

Type Boolean, read/write

Values
`true` Enable message printing
`false` Disable message printing

Default value `false`

Module nlsolv

4.4 Procedures and functions

The following list gives an overview of all other functions and procedures defined by *nlsolv* for which we give detailed descriptions later.

`exportnl`	Save the NL file.	p. 40
`getprobstat`	Get the optimization problem status.	p. 41
`maximize, minimize`	Maximize/minimize the current NL problem.	p. 42

exportnl

Purpose

Save the NL file.

Synopsis

```
procedure exportnl(filename:string, obj:linctr)
procedure exportnl(filename:string, nlobj:nlctr)
```

Arguments

`filename`	Name of the output file; if empty, output is printed to standard output (console)
`obj`	Linear objective function
`nlobj`	Nonlinear objective function

Example

The following example saves a problem with the nonlinear objective function `Obj` to the file `prob1.nl`, it then switches to ASCII format for displaying the problem on screen :

```
uses "nlsolv"
declarations
 Obj:nlctr
end-declarations
 ...

exportnl("prob1", Obj)
setparam("nl_binary", false)
exportnl("", Obj)
```

Further information

This procedure exports the current problem to a file, or if no file name is given (empty string `""`), prints it on screen. If the given filename has no extension, *nlsolv* appends `.nl` to it.

Module

nlsolv

getprobstat

Purpose

Get the optimization problem status.

Synopsis

```
function getprobstat:integer
```

Return value

Status of the last problem solved:

`NL_OPT`	Solved to optimality
`NL_UNF`	Unfinished
`NL_INF`	Infeasible
`NL_UNB`	Unbounded
`NL_OTH`	Unsolved or failure during processing

Example

The following procedure displays the current problem status:

```
procedure print_status
 declarations
  status: string
 end-declarations

 case getprobstat of
  NL_OPT: status:="Solved"
  NL_UNF: status:="Unfinished"
  NL_INF: status:="Infeasible"
  NL_UNB: status:="Unbounded"
  NL_OTH: status:="Failed"
  else status:="???"
 end-case

 writeln("Problem status: ", status)
end-procedure
```

Further information

More detailed information than what is provided by this function can be obtained with function `getparam`, retrieving the solver status value in `NL_status`.

Related topics

`NL_status`.

Module

nlsolv

maximize, minimize

Purpose

Maximize/minimize the current NL problem.

Synopsis

```
procedure maximize(obj:linctr)
procedure maximize(nlobj:nlctr)
procedure minimize(obj:linctr)
procedure minimize(nlobj:nlctr)
```

Arguments

`obj`	Linear objective function
`nlobj`	Nonlinear objective function

Example

The following maximizes the nonlinear objective `Obj`.

```
uses "nlsolv"
declarations
 x,y,z: mpvar
 Obj: nlctr
end-declarations
...
Obj:= x*y + z^3
maximize(Obj)
```

Further information

This procedure calls a solver to maximize/minimize the current problem (excluding all hidden constraints) using the given constraint as objective function. Before this procedure is called, a solver must have been chosen (using either parameter `NL_solver` or `NL_solverpath`).

Related topics

`NL_solver`, `NL_solverpath`.

Module

nlsolv