Reference manual

FICO® Xpress Optimization

Xpress MATLAB Interface

Reference Manual

Release 8.4

Last update 26 September, 2017

FICO® Xpress MATLAB Interface

Deliverable Version: A

Last Revised: 26 September, 2017

Version 8.4

Contents

CHAPTER 1

Xpress Mosel MATLAB Interface

1.1 Overview

The Xpress MATLAB interface is a tool that makes Xpress optimization algorithms available directly from within the MATLAB environment, enabling users to easily define mathematical programming models and solve them with Xpress from within the MATLAB environment.

The interface for Mosel provides functions for running Mosel programs from within MATLAB and exchanging data between the Mosel models and the MATLAB environment.

1.2 Configuring MATLAB for the Xpress Mosel interface

Please refer to the "*Xpress Installation and Licensing User Guide*" for instructions on Xpress installation. The MATLAB interface does not require a separate software license.

The Xpress Mosel MATLAB Interface includes a function (`moselexec`) to run Mosel programs, a Mosel I/O Driver to exchange data with the MATLAB environment and support for using the Java Mosel classes from MATLAB.

In order to make the new functionality available in MATLAB, the Xpress `matlab` directory must be added to the *MATLAB search path*. This can be done either using the graphical *'Set Path'* dialog box or the command line. Note that this step is the same as described for the Xpress Optimizer MATLAB interface and needs to be carried only once.

For the Java Mosel classes, there are other two search paths that need to be updated: the *MATLAB Java classpath* and the *MATLAB Java libpath*.

1.2.1 Setting the MATLAB search path

The MATLAB search path can be set using the graphical interface as follows.

From the main MATLAB window, click on *File* ≫ *Set Path...*, then on the *'Add Folder'* button and select the `matlab` subfolder of your Xpress installation folder (on Windows platforms typically `'c:\xpressmp\matlab'`).

You should make this change permanent by clicking on the *'Save'* button.

It is also possible to set the search path using the MATLAB command line, with the following instructions:

```
>> addpath(fullfile(getenv('XPRESSDIR'),'/matlab'))
```

and make this permanent with the command

```
>> savepath
```

The above command uses the `XPRESSDIR` environment variable to locate your Xpress installation directory; alternatively you can also specify the path directly, as in:

```
>> addpath 'c:\xpressmp\matlab'
```

(assuming you installed Xpress on `'c:\xpressmp'`)

1.2.2 Setting the MATLAB Java class path

In order to use the Java Mosel interface in MATLAB you need to add the Java Mosel library to the MATLAB Java classpath. The library consists of a Java Archive (JAR) file located under the Xpress installation directory, in the `lib` subdirectory. MATLAB supports both a *static path* and a *dynamic path* and you can add the Mosel JAR to either one; please refer to the MATLAB documentation, section 'Bringing Java Classes into MATLAB Workspace' for more information.

In the following, we show how to add the JAR to the static path. MATLAB loads the static path from an ASCII file named `javaclasspath.txt` in your preferences folder. To view the location of the preferences folder, type `prefdir` in MATLAB. Each line in this file is the path of a folder or a jar file. You can open this file in the MATLAB editor with the following command

```
>> edit(fullfile(prefdir, '/javaclasspath.txt'))
```

then you should add the following line to this file:

```
C:\xpressmp\lib\xprm.jar
```

(assuming you installed Xpress on `'c:\xpressmp'`)

Then save the file and restart MATLAB for these changes to take effect.

Alternatively, you can run the following small MATLAB script (that you can copy & paste to the MATLAB console) to automate the above operation:

```
fjcp = fopen(fullfile(prefdir,'/javaclasspath.txt'), 'at');
fprintf(fjcp,'\n%s\n',fullfile(getenv('XPRESSDIR'),'/lib/xprm.jar'));
fclose(fjcp);
```

Again, you need to restart MATLAB for these changes to take effect.

1.2.3 Setting the MATLAB Java library path

In order to use the Java Mosel interface in MATLAB you also need to add the Mosel native library to the MATLAB Java librarypath. The native library consists of some dynamically linked files located under the Xpress installation directory, in the **`bin`** subdirectory on **Windows** and in the **`lib`** subdirectory on **Linux**. MATLAB loads the library search path from an ASCII file named `javalibrarypath.txt` in your preferences folder. To view the location of the preferences folder, type `prefdir` in MATLAB. Each line in this file is the path of a folder. You can open this file in the MATLAB editor with the following command

```
>> edit(fullfile(prefdir, '/javalibrarypath.txt'))
```

then you should add the following line to this file:

```
C:\xpressmp\bin
```

(assuming you installed Xpress for Windows on `'c:\xpressmp'`)

Then save the file and restart MATLAB for these changes to take effect.

Alternatively you can run the following small MATLAB script (that you can copy & paste to the MATLAB console) to automate the above operation:

```
fjlp = fopen(fullfile(prefdir,'/javalibrarypath.txt'), 'at');
if isunix, libdir='/lib'; else libdir='/bin'; end
fprintf(fjlp,'\n%s\n',fullfile(getenv('XPRESSDIR'),libdir));
fclose(fjlp);
```

Again, you need to restart MATLAB for these changes to take effect.

1.2.4 Verifying if the Xpress Mosel interface works

You can verify that the Xpress Mosel MATLAB interface is working properly by executing the command

```
>> moselexec -v
```

inside MATLAB. In case everything is fine you should see something like:

```
XPRESS Mosel Matlab Interface function version x.x.x
```

Similarly, for the Java interface, the command

```
>> com.dashoptimization.XPRM().getVersion
```

should print something like:

```
ans =
3.5.3
```

1.3 Running Mosel models

1.3.1 The `moselexec` function

The simplest way to run a Mosel program from MATLAB is using the `moselexec` function, as in:

```
>> moselexec burglar.mos
```

This compiles and runs the Mosel program `burglar.mos` located in the current folder (or prints an error message if the file cannot be found). You can of course specify a full path as in

```
>> moselexec C:/xpressmp/examples/mosel/Modeling/burglar.mos
```

or use the `XPRESSDIR` environment variable to point to the Xpress installation folder:

```
>> moselexec(fullfile(getenv('XPRESSDIR'),'/examples/mosel/Modeling/burglar.mos'))
```

By specifying the optional output arguments `retcode` and `exitcode`, the `moselexec` function can also return the compilation and execution result code and the program exit status, or both, for example solving this tiny example `example_m1.mos`:

```
model "example_m1"
exit(10)
end-model
```

would yield

```
>> [retcode, exitcode]=moselexec('example_m1.mos')
retcode =
          0
exitcode =
         10
```

where the value zero for `retcode` means that the program has run without errors, and `exitcode` has the value specified in the model.

Please refer to `moselexec` in the reference section for further details.

1.3.2 The I/O driver

The Mosel I/O driver for MATLAB makes it possible to exchange data between Mosel programs and the MATLAB workspace. This driver supports reading a MATLAB value as a Mosel generalized file stream, and importing and exporting data from and to MATLAB in Mosel `initializations from` and `initializations to` blocks.

Note that this driver is available only when executing Mosel programs from within the MATLAB environment.

1.3.2.1 Extended file names

Mosel uses an extend file name format to represent 'files' that can be accessed through specialized I/O drivers. The format for the MATLAB driver is

`matlab.mws:`*`expression`*

where `matlab` is the name of the Mosel module, `mws` is the name of the I/O driver name (MATLAB WorkSpace) and *`expression`* can either be a current variable name of the caller workspace, or any MATLAB expression returning a single value. In the case of a MATLAB expression, the latter will be evaluated in the caller workspace at the time of file opening. For example, the following Mosel program

```
model "example_m2"
 uses "mmsystem";
 fcopy("matlab.mws:message", "")
 writeln
end-model
```

would read the MATLAB variable `message` and print it to the MATLAB console. You can test it with the following MATLAB commands

```
>> message='Hello, World!';
>> moselexec('example_m2.mos')
Hello, World!
```

1.3.2.2 String handling

When reading a string variable, the I/O driver automatically converts it from MATLAB native 16-bit multibyte Unicode characters to the 8-bit ASCII format used by Mosel (if you prefer to convert the string using a different encoding, you can explicitly convert it to a raw byte stream beforehand with the MATLAB function `unicode2native`). If the source string is a string array that contains several rows, then these are copied, one column at a time, into a single string. Finally, if the source variable is a cell array containing strings, all strings are read successively with newline characters added at the end of each one.

It is thus possible to use a MATLAB cell array to store a Mosel program, one line per cell, and then execute it without using external files, as in the following example.

```
>> mos={
'model "example_m3"        '
' uses "mmxprs", "mmnl";   '
' declarations             '
'  a:mpvar                 '
' end-declarations         '
' minimize(a*a-5*a+10)     '
' writeln(getobjval)       '
'end-model                 '
};
>> moselexec('matlab.mws:mos')
3.75
```

1.3.2.3 *Initializations from* blocks

The `matlab.mws` I/O driver can be used in Mosel *initialization* blocks to read MATLAB values and set MATLAB variables. In this case, the *filename* should just be `"matlab.mws:"`, without any *expression*, and the *expression* can eventually be specified as the *label* associated to the *identifier* being initialized.

Consider the following Mosel program

```
model "example_m4"
 declarations
  answer: integer
  foo: real
  var: real
  today: string
  i: range
  Data: array(i) of real
 end-declarations

 initializations from "matlab.mws:"
  answer as "42"
  foo
  var as "bar"
  today as "date"
  Data as "sum(magic(foo*bar))"
 end-initializations

 writeln("answer to ultimate question: ", answer)
 writeln("foo: ", foo)
 writeln("bar: ", var)
 writeln("today: ", today)
 writeln("data: ", Data)
end-model
```

and its execution from MATLAB

```
>> foo=pi;
>> bar=exp(1);
>> moselexec('example_m4.mos');
answer to ultimate question: 42
foo: 3.14159
bar: 2.71828
today: 01-May-2014
```

```
data: [260,260,260,260,260,260,260,260]
```

Here, the *expression* used to initialize the variable `answer` is `"42"`, that is, a literal value. Variable `foo` doesn't specify an initialization label, so the default is used—the default label is the identifier itself and thus the MATLAB variable `foo` is read. The label for variable `var` explicitly says to read the MATLAB variable `bar`. The expression used to initialize `today` is the MATLAB function `date` which returns a string with today's date. And finally, `Data` is an array read from a MATLAB expression that builds a magic square of size 8 and calculates the sums of values in every column (which should be all equal in magic squares, as shown in the output).

MATLAB sparse matrices can be read into *dynamic arrays* to set only non-zero elements:

```
>> mos={
'model "example_m5"              '
' declarations                   '
'  I,J: range                    '
'  Sparse: dynamic array(I,J) of real'
' end-declarations               '
' initializations from "matlab.mws:" '
'  Sparse as "sprand(4,4,.5)"     '
' end-initializations            '
' writeln("sparse is: ", Sparse)  '
' writeln("row indices: ", I)     '
' writeln("col indices: ", J)     '
'end-model                       '
};
>> moselexec('matlab.mws:mos');
sparse is: [(1,3,0.24285),(2,1,0.917424),(2,2,0.269062),(2,3,0.7655),(4,1,0.188662)...
row indices: 1..4
col indices: 1..3
```

In the above example, `Sparse` is a 2-dimensional dynamic array containing only 6 values after initialization from a MATLAB 4 4 sparse random matrix, and index set `J` (in this execution) contains only the values 1, 2, and 3 as the matrix happened to have all zeros in column 4. Note also that array indices start from 1 which is the MATLAB convention.

1.3.2.4 *Initializations to* blocks

Mosel data can be exported to MATLAB using *initializations to* blocks. The *filename* should just be `"matlab.mws:"` in this case too, and *labels* can be used to specify MATLAB variable names to export to (if no label is specified, the name of the identifier is used). In MATLAB, these variables are set in the *caller* workspace, eventually overwriting their previous value.

The following example shows how to export a scalar value (`simplexiter`), the optimal objective and solution values into MATLAB variables. The model, `foliomat.mos`, is a modified version of the portfolio optimization example from the "*Getting Started with Xpress*" guide.

```
model "Portfolio optimization with LP - MATLAB"
 uses "mmxprs"

 declarations
  SHARES: range
  RISK: set of integer
  NA: set of integer
  RET: array(SHARES) of real
  frac: array(SHARES) of mpvar
  simplexiter: integer
 end-declarations

 initializations from "matlab.mws:"
   RISK NA RET
 end-initializations
```

```
 Return:= sum(s in SHARES) RET(s)*frac(s)
 sum(s in RISK) frac(s) <= 1/3
 sum(s in NA) frac(s) >= 0.5
 sum(s in SHARES) frac(s) = 1
 forall(s in SHARES) frac(s) <= 0.3
 maximize(Return)
 simplexiter:=getparam("XPRS_simplexiter")

 initializations to "matlab.mws:"
  simplexiter
  evaluation of getobjval as "objval"
  evaluation of array(s in SHARES) frac(s).sol as "frac"
 end-initializations
 exit(getprobstat)

end-model
```

This can be executed from MATLAB after defining RISK, NA and RET input data and results will be available as MATLAB variables as shown below.

```
>> RET  = [5 17 26 12 8 9 7 6 31 21];
>> RISK = [2 3 4 9 10];
>> NA   = [1 2 3 4];
>> [r,e]=moselexec('foliolp_mat.mos');
>> objval
objval =
   14.0667
>> frac'
ans =
    0.3000         0    0.2000         0    0.0667    0.3000 ...
```

1.3.2.5 Using MATLAB functions in Mosel

We have already seen how MATLAB functions can be called in *initializations from* blocks (including user-defined functions). Since these blocks can be used at arbitrary positions in Mosel programs, it is possible to combine this with `initializations to` blocks to load some data into MATLAB, evaluate a MATLAB function on this data and retrieve results back into Mosel. The following example shows a `fibonacci` function implemented in MATLAB and a Mosel program that also defines a fibonacci function that just calls the MATLAB one (note however that this is neither reentrant nor thread-safe).

MATLAB code (`fibonacci.m`):

```
function f=fibonacci(n)
  if n<2, f=n; return, end
  s=[0 1];
  for i=2:n, s=[s(2) sum(s)]; end
  f=s(2);
end
```

Mosel model `fib-relay.mos`:

```
model "fib_relay"
 function fibonacci(i:integer):integer
   initializations to "matlab.mws:"
     i
   end-initializations

   initializations from "matlab.mws:"
     returned as "fibonacci(i)"
   end-initializations
 end-function
```

```
 forall(i in 1..10)
   writeln("fibonacci(", i, ")=", fibonacci(i))
end-model
```

Example run:

```
>> moselexec('fib_relay.mos');
fibonacci(1)=1
fibonacci(2)=1
fibonacci(3)=2
fibonacci(4)=3
fibonacci(5)=5
...
```

1.3.2.6 Supported types

The `matlab.mws` driver supports all basic types of Mosel (boolean, integer, real, string) and the structures `set`, `range`, `list` and `array` of basic types. On the MATLAB side, the supported types are n-dimensional arrays and cell arrays of the basic numeric, logical or char classes (including sparse matrices). Only the *real* part of arrays is always used. Since MATLAB uses 1-based integer indices, Mosel arrays must also use this same convention when imported/exported to MATLAB. If necessary, data is silently casted to the appropriate type without any warning in case of truncation or loss of precision (for example when reading a Mosel integer from a fractional MATLAB double value).

1.4 Using the Java Mosel interface

1.4.1 Overview

The Java Mosel interface offers a more advanced control and interaction with Mosel than what is possible with the simple `moselexec` function. In fact, the Java Mosel interface enables the user to:

- compile source model files into binary model (bim) files
- load and unload bim files handling several models at a time
- execute models
- access the Mosel internal database through the Post Processing Interface
- manage the dynamic shared objects used by Mosel

We will show some of these functionalities in the following examples, however please refer to:

- the "*Xpress Mosel User Guide*", Chapter 14, for a brief introduction to the Java interface;
- the "*Xpress Mosel Library Reference Manual*" in JavaDoc format, for the full reference documentation of this interface;
- *MATLAB Documentation - Advanced Software Development - Call Java Libraries*, for details on using Java from MATLAB.

Furthermore, the I/O driver described in the previous section can also be used in this context.

1.4.2 Compiling and executing a model

With Java, Mosel is initialized by creating a new instance of class XPRM. In MATLAB you can either use the fully qualified class name (including the package name) as in

```
>> mosel=com.dashoptimization.XPRM;
```

or import the package and then use class names without the package name:

```
>> import com.dashoptimization.*;
>> mosel=XPRM;
```

The standard compile/load/run sequence becomes

```
>> mosel=com.dashoptimization.XPRM;
>> mosel.compile('burglar2.mos');
>> mod=mosel.loadModel('burglar2.bim');
>> mod.run;
>> mod.getResult;
```

If the model execution is embedded in a larger application it may be useful to release the resources allocated by a model after its execution. This can be done through standard finalization + garbage collection functionalities, by calling the `finalize` method on the model:

```
>> mod.finalize
```

The `mosel` object can be released in the same way (`mosel.finalize`).

1.4.3 Accessing arrays

In general, Mosel entities such as scalar variables, sets, *etc.* can be queried through the `findIdentifier` method and retrieved in the same way as described in the "*Xpress Mosel User Guide*". However, when calling Java from MATLAB, it is not possible to pass an array of a Java native type to a function and receive back in MATLAB the array as modified by the function. This would be the case, for instance, when using the `nextIndex` or `nextTEindex` methods on a Mosel array. Consider the following example that defines a (sparse) array `VALUE` with two indices of type string:

```
model example_m6
 declarations
  CITIES = {"london", "paris", "madrid", "rome", "florence"}
  ZONES = {"north", "south", "east", "west"}
  VALUE: dynamic array(CITIES,ZONES) of real
 end-declarations

 VALUE("london", "east")  := 1
 VALUE("rome",   "west")  := 2
 VALUE("paris",  "south") := 3
 VALUE("madrid", "east")  := 4
end-model
```

The array `VALUE` can be retrieved into MATLAB with the following code (`example_m6.m`):

```
value = mod.findIdentifier('VALUE');
value_iter = value.indices(true);
sets = value.getIndexSets();
while value_iter.hasNext
  indices = value_iter.next;
  fprintf(1, 'VALUE ( ');
```

```
  for i=1:size(indices,1)
    fprintf(1, '%s ', char(sets(i).get(indices(i))));
  end
  fprintf(1, ') = %g\n', value.getAsReal(indices));
end
```

Executing this script would print the Mosel array as shown below:

```
>> example_m6
VALUE ( london east ) = 1
VALUE ( madrid east ) = 4
VALUE ( paris south ) = 3
VALUE ( rome west ) = 2
```

In the above example we use the iterator `value_iter` to loop over all valued elements of the array; at each iteration we retrieve the actual numerical indices (`indices`) of the current element, their corresponding values (`sets(i).get(...)`), and the value of the current element (`value.getAsReal(indices)`).

Please note that the following alternative approach that uses `XPRMArray.nextTEIndex()`, would not work correctly in MATLAB as the call to `value.nextTEIndex(indices)` cannot update the `indices` array as in pure Java.

```
indices=value.getFirstTEIndex;
...
while value.nextTEIndex(indices)
 ...
end
```

1.4.4 Examples

The first example, `ugsol.m` is a variation of the program `ugsol.java` described in the "*Xpress Mosel User Guide*". Here the Mosel program has been embedded in a MATLAB script: the model is the same but problem data is read from MATLAB variables and the solution is exported to MATLAB. The MATLAB script compiles the Mosel program, runs it, checks the solution status and prints the solution.

```
mos={
'model Burglar_m                                                  '
' uses "mmxprs"                                                   '
' declarations                                                    '
'  WTMAX = 102                    ! Maximum weight allowed        '
'  ITEMS: range                                                   '
'  VALUE: array(ITEMS) of real    ! Value of items                '
'  WEIGHT: array(ITEMS) of real   ! Weight of items               '
'  take: array(ITEMS) of mpvar    ! 1 if we take item i; 0 otherwise    '
' end-declarations                                                '
'                                                                 '
' initializations from "matlab.mws:"                              '
'  VALUE                                                          '
'  WEIGHT                                                         '
' end-initializations                                             '
'                                                                 '
' MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)  ! Objective: max total value'
' sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX ! Weight restriction       '
' forall(i in ITEMS) take(i) is_binary       ! All variables are 0/1    '
' maximize(MaxVal)                           ! Solve the MIP-problem    '
'                                                                 '
' initializations to "matlab.mws:"                                '
'  evaluation of array(i in ITEMS) take(i).sol as "TAKE"          '
' end-initializations                                             '
'end-model                                                        '
};
```

```
ITEMS ={'camera' 'necklace' 'vase' 'picture' 'tv' 'video' 'chest' 'brick'};
VALUE =[ 15       100        90     60        40   15      10      1];
WEIGHT=[ 2        20         20     30        40   30      60      10];

mosel=com.dashoptimization.XPRM; % Initialize Mosel
mosel.compile('', 'matlab.mws:mos', 'burglar_m.bim');
mod=mosel.loadModel('burglar_m.bim');
mod.run;
if mod.getProblemStatus~=mod.PB_OPTIMAL, return, end
fprintf(1,'Objective value: %g\n', mod.getObjectiveValue); % show objective
table(ITEMS',logical(TAKE),VALUE','VariableNames',{'Item' 'Take' 'Value'})
fprintf(1,'Calculated objective: %g\n', VALUE*TAKE); % verify sol
mod.finalize
```

The second example is a variation of the portfolio optimization from the *Getting Started with Xpress* guide. The Mosel program, `foliomat2.mos`, is almost identical to the `foliodat.mos` example, modified only to use integer indices instead of string indices, and to read input from MATLAB and write results to MATLAB.

```
model "Portfolio optimization with LP"
 uses "mmxprs"                       ! Use Xpress Optimizer

 parameters
  DATAFILE= "matlab.mws:"            ! File with problem data
  MAXRISK = 1/3                      ! Max. investment into high-risk values
  MAXVAL = 0.3                       ! Max. investment per share
  MINAM = 0.5                        ! Min. investment into N.-American values
 end-parameters

 writeln("Solving for MAXRISK: ", MAXRISK)
 declarations
  SHARES: range                      ! Set of shares
  NAMES: array(SHARES) of string     ! Names of the shares
  RISK: set of integer               ! Set of high-risk values among shares
  NA: set of integer                 ! Set of shares issued in N.-America
  RET: array(SHARES) of real         ! Estimated return in investment
 end-declarations

 initializations from DATAFILE
  NAMES RISK RET NA
 end-initializations

 declarations
  frac: array(SHARES) of mpvar       ! Fraction of capital used per share
 end-declarations

! Objective: total return
 Return:= sum(s in SHARES) RET(s)*frac(s)

! Limit the percentage of high-risk values
 sum(s in RISK) frac(s) <= MAXRISK

! Minimum amount of North-American values
 sum(s in NA) frac(s) >= MINAM

! Spend all the capital
 sum(s in SHARES) frac(s) = 1

! Upper bounds on the investment per share
 forall(s in SHARES) frac(s) <= MAXVAL

! Solve the problem
 maximize(Return)

! Solution printing to a file
 writeln("Total return: ", getobjval)
 forall(s in SHARES)
```

```
  writeln(strfmt(NAMES(s),-12), ": \t", strfmt(getsol(frac(s))*100,5,2), "%")

 initializations to "matlab.mws:"
  evaluation of getobjval as "objval"
  evaluation of getprobstat=XPRS_OPT as "optsol"
  evaluation of array(s in SHARES) frac(s).sol as "frac"
 end-initializations

end-model
```

The following MATLAB script (`foliomat2.m`) first initializes input data (also deriving integer indices from strings for variables `RISK` and `NA`), it then executes the Mosel program for different values of MAXRISK, from 0.1 to 0.9 at 0.1 steps, and finally displays a couple of result tables and charts of share utilization for the different risks.

```
NAMES  ={'treasury' 'hardware' 'theater' 'telecom' 'brewery' 'highways' 'cars'    'bank'
         'software' 'electronics'};
RET    =[ 5        17    26    12    8    9        7          6           31      21 ];
DEV    =[ 0.1      19    28    22    4    3.5      5          0.5         25      16 ];
COUNTRY={'Canada' 'USA' 'USA' 'USA' 'UK' 'France' 'Germany' 'Luxemburg' 'India' 'Japan'};

RISK_N  ={'hardware' 'theater' 'telecom' 'software' 'electronics'};
NA_N    ={'treasury' 'hardware' 'theater' 'telecom'};

RISK=cellfun(@(n) strmatch(n,NAMES,'exact'), RISK_N); % find indices of high-risk shares
NA  =cellfun(@(n) strmatch(n,NAMES,'exact'), NA_N);   % find indices of N.-American shares

for m=1:9
  moselexec('foliomat2.mos',['MAXRISK=' num2str(m/10)]);
  obj(m)=objval;
  optimal(m)=optsol;
  fracm(m,:)=frac;
end

disp('Results');
disp('Estimated returns:');
disp(table([1:9]'/10,obj','VariableNames',{'MaxRisk' 'Return'}))

disp('Average share utilization:');
disp(table(NAMES',mean(fracm)','VariableNames',{'Share' 'AverageUsage'}))

ribbon(fracm)
title('Share utilization')
set(gca,'XTick',[1:size(fracm,2)])
set(gca,'XTickLabel',NAMES)
set(gca,'YDir','reverse')
set(gca,'YTick',[1:9])
set(gca,'YTickLabel',[1:9]/10)
set(gca,'ZLim',[0,max(reshape(fracm,1,[]))])
```

Running the script will yield the following results and the graphic in Figure 1.1.

```
Results
Estimated returns:
    MaxRisk    Return
    _______    ______
    0.1           0
    0.2          11
    0.3        13.3
    0.4        15.6
    0.5        17.8
    0.6        19.9
    0.7        21.1
    0.8        22.3
    0.9        23.5
Share utilization:
        Share        AverageUsage
```

```
-------------     ------------
'treasury'         0.16667
'hardware'        0.055556
'theater'          0.22222
'telecom'                0
'brewery'         0.033333
'highways'             0.2
'cars'                   0
'bank'                   0
'software'             0.2
'electronics'     0.011111
```

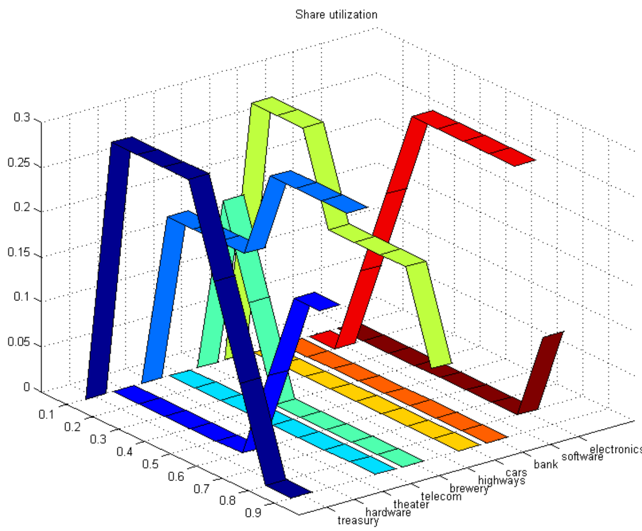

Figure 1.1: Share utilization

CHAPTER 2

Xpress Optimizer MATLAB Interface

2.1 Overview

The interface for the Optimizer provides functions for solving linear, quadratic and quadratically constrained programming problems, and the mixed integer versions of these. All optimization functions are designed to take a model description as input and produce a solution as output.

2.2 Using the Xpress for MATLAB Toolbox

Please refer to the "Xpress Installation and Licensing User Guide" for instructions on Xpress installation. The MATLAB interface does not require a separate software license.

In order to make the Xpress functions available in MATLAB, the Xpress MATLAB path must be added to the *MATLAB search path*. This can be done either using the graphical *'Set Path'* dialog box or the command line. Note that this step is the same one described for the Xpress Mosel MATLAB interface and need to be done only once.

2.2.1 Using the MATLAB graphical interface to set the search path

From the main MATLAB window, click on *File* ≫ *Set Path...*, then on the *'Add Folder'* button and select the `matlab` subfolder of your Xpress installation folder (on Windows platforms typically `'c:\xpressmp\matlab'`).

You can also make this change permanent by clicking on the *'Save'* button.

2.2.2 Using the MATLAB command line to set the search path

The command to add the Xpress interface to MATLAB search path is:

```
>> addpath 'c:\xpressmp\matlab'
```

(assuming you installed Xpress on `'c:\xpressmp'`), and this can be made permanent with the command

```
>> savepath
```

2.2.3 Verifying if Xpress works

You can verify that the Xpress MATLAB interface is working properly by executing the command

```
>> xprsver
```

inside MATLAB. In case everything is fine you should get something like:

```
FICO Xpress Optimizer 64-bit v31.01.02 (Hyper capacity)
(c) Copyright Fair Isaac Corporation 1983-2017
```

2.2.4 Interface functions

The Xpress MATLAB interface is comprised of the following functions:

- 7 optimization functions (`xprslp`, `xprsqp`, `xprsqcqp`, `xprsbip`, `xprsmip`, `xprsmiqp`, `xprsmiqcqp`)
- 2 functions to set/get controls (`xprsoptimset` and `xprsoptimget`)
- 1 function to show the Xpress version (`xprsver`)

The next section documents each of these functions. Once the MATLAB search path has been configured, the same documentation will be also directly available in MATLAB, both from the drop down menu *Help* ≫ *Product Help*, as a new Toolboxes section, and from the command line using the `help` command (*e.g.* with `help xprslp`).

2.2.5 Problem matrices

Differently from MATLAB Optimization Toolbox minimization functions, that take two distinct matrices in input: one for inequality constraints and the other for equality constraints, Xpress interface functions take only one matrix for both types of constraints plus a vector that specifies the constraint type.

Therefore, if matrices *A* and *Aeq* (with RHS, respectively, *b* and *beq*) are used to solve a linear problem with the Optimization Toolbox's `linprog` function:

```
>> x = linprog(f, A, b, Aeq, beq, lb, ub);
```

the same problem can be solved with Xpress using the commands

```
>> rtype = [repmat('L',[1 size(A,1)]) repmat('E',[1 size(Aeq,1)])];
>> x = xprslp(f, [A; Aeq], [b; beq], rtype, lb, ub);
```

where the *rtype* vector indicates that rows from matrix *A* are of type `'L'` (less than or equal) and rows from matrix *Aeq* are of type `'E'` (equalities).

2.2.6 Setting and querying controls and attributes

Optimization options can be specified with a mechanism similar to that used by the MATLAB Optimization Toolbox, that is via an options structure that specifies a list of Xpress controls and their values. See function `xprsoptimset` and the *'Control Parameters'* section of the "Xpress Optimizer Reference Manual" for more details.

The `xprsoptimset` function also handles the conversion from the Optimization Toolbox options to the corresponding Xpress options for all cases where this makes sense.

Furthermore, after calling an Xpress optimization function, it is possible to retrieve the final value of any Xpress control or attribute. The list of control and attribute names to be returned must be specified in the `'XPRSGET'` field of the option argument, separated by blanks. For example

```
>> options= xprsoptimset('XPRSGET', 'LPOBJVAL LPSTATUS')
>> [x,fval,ef,output] = xprslp(f, A, b, [], lb, ub, options);
>> fval, output.LPOBJVAL
```

```
fval =
   -78
output =
    LPOBJVAL: -78
    LPSTATUS: 1
```

It is also possible to request that the output structure be filled with all Xpress control and attribute values by setting 'XPRSGET' to 'ALL'.

In the Xpress MATLAB interface, control and attribute names are always all uppercase and without the *XPRS* prefix.

2.2.7 Special options

When calling an interface function, it is possible to pass one or more of the following additional options before the normal input arguments:

`-v`	Display the version of the called function.
`-w[flags]`	Write the problem to file; see the documentation for `XPRSwriteprob` in the Optimizer Reference Manual for more details (supported in all optimization functions).
`-s`	Save the optimizer data structures immediately before solving the problem; see the documentation for `XPRSsave` in the Optimizer Reference Manual for more details (supported in all optimization functions).

Both the `-w` and `-s` options create files in the current MATLAB directory/folder and with the same name as the name of the function being called.

For example, it is possible to export a MIP problem to a file in LP format by calling `xprsmip` with an additional `-w` option and flag `l` as follows (the file will be named `xprsmip.lp`):

```
>> x = xprsmip('-wl', f, A, b, rtype, ctype);
```

2.3 Example

In this example we solve the sample problem from MATLAB's documentation page on the `linprog` function.

The problem at hand is:

$$
\begin{array}{ll}
\text{minimize} & -5 \cdot x1 - 4 \cdot x2 - 6 \cdot x3 \\
\text{subject to} & x1 - x2 + x3 \leq 20 \\
 & 3 \cdot x1 + 2 \cdot x2 + 4 \cdot x3 \leq 42 \\
 & 3 \cdot x1 + 2 \cdot x2 \leq 30 \\
 & 0 \leq x1, 0 \leq x2, 0 \leq x3
\end{array}
$$

First, enter the coefficients

```
>> f = [-5; -4; -6];
>> A = [1 -1  1
>>      3  2  4
>>      3  2  0];
>> b = [20; 42; 30];
>> lb = zeros(3,1);
```

Next, call the Xpress linear programming function.

```
>> [x,fval,exitflag,output,lambda] = xprslp(f,A,b,'L',lb);
```

Entering `x`, `lambda.lin`, and `lambda.lower` returns the following results:

```
x =
     0.0000
    15.0000
     3.0000
lambda.lin =
     0
     1.5000
     0.5000
lambda.lower =
     1.0000
     0
     0
```

CHAPTER 3

Xpress MATLAB functions

moselexec

Purpose

Compile and run a Mosel program.

Synopsis

```
moselexec(srcfile)
moselexec(srcfile,parlist)
moselexec(srcfile,parlist,options)
retcode=moselexec(..)
[retcode, exitcode]=moselexec(..)
```

Input arguments

`srcfile`	Name of the Mosel source file to run, can be any Mosel generalized file
`parlist`	String composed of model parameter initializations separated by commas
`options`	Mosel compilation options

Output arguments

`retcode`	Compilation and execution result code	
	<0	compilation failed
	0	program executed successfully
	>0	an error occured during model execution
`exitcode`	Exit status returned by the Mosel program	

Further information

1. Compilation options are documented in the Mosel Language Reference Manual.
2. If the output argument `retcode` is omitted and execution is not successful (that is, `retcode` is non-zero), then its value is printed with a warning message (to disable this message, just add the output argument in the call).

xprsbip

Purpose

Solve binary integer programming problems with Xpress.

Synopsis

```
x = xprsbip(f,A,b,rtype,x0,options)
[x,fval,exitflag,output] = xprsbip(...)
```

Input arguments

`f`	Linear objective function vector
`A`	Matrix for linear constraints
`b`	Vector for constraints RHS
`rtype`	Character vector (string) giving the row types: `L` indicates a $\leq$ row `E` indicates a = row `G` indicates a $\geq$ row `N` indicates a free row If `rtype = []`, all rows are assumed to be of type `'L'`. If `rtype` is a single character, all constraints are assigned the corresponding type.
`x0`	Optional initial known solution used to speed-up search.
`options`	Options structure created with `optimset` or `xprsoptimset` functions. See `xprsoptimset` for more details.

Output arguments

`x`	Solution found by the optimization function. If `exitflag >0`, then `x` is a solution; otherwise, `x` is the value of the optimization routine when it terminated prematurely.
`fval`	Value of the objective function at the solution `x`.
`exitflag`	Integer identifying the reason why the optimization algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated. `1` function converged to a solution `x` (MIPSTATUS=MIP_OPTIMAL) `0` number of iterations exceeded iter limit (STOPSTATUS= STOP_ITERLIMIT) `-2` the problem is infeasible (MIPSTATUS=MIP_INFEAS) `-4` number of searched nodes exceeded limit (STOPSTATUS= STOP_NODELIMIT) `-5` search time exceeded limit (STOPSTATUS= STOP_TIMELIMIT) `-8` other stop reason, see MIPSTATUS and STOPSTATUS for details
`output`	Structure containing information about the optimization and, eventually, values of Xpress controls and attributes. See `bintprog` and the Section 2.2.6 for details.

Further information

1. This routine finds the minimum of a problem specified by

$$\begin{array}{rl} \min & f \cdot x \\ \text{s.t.} & A \cdot x \leq | = | \geq b \\ & x \in \{0, 1\} \end{array}$$

   where A is an $m \times n$ matrix; f, b, $rtype$, and $x0$ are vectors.

2. Input arguments `rtype`, `x0` and `options` can be omitted, with the condition that, if one is omitted, also all the following ones must be omitted (as in `x=xprsbip(f, A, b, rtype)`). Omitting an input argument has the same effect as passing an empty array `[]`.

3. All output arguments can be omitted too, again with the condition that, if one is omitted, also all the following ones must be omitted (as in `[x, fval]= xprsbip(f, A, b, rtype)`).

4. If the specified input bounds for a problem are inconsistent, the output `x` and `fval` are set to `[]`.

Related topics

`xprsoptimset`, `bintprog`

xprslp

Purpose

Solve linear programming problems with Xpress.

Synopsis

```
x = xprslp(f,A,b,rtype,lb,ub,options)
[x,fval,exitflag,output,lambda] = xprslp(...)
```

Input arguments

`f`	Linear objective function vector
`A`	Matrix for linear constraints
`b`	Vector for constraints RHS
`rtype`	Character vector (string) giving the row types: `L` indicates a $\leq$ row `E` indicates a = row `G` indicates a $\geq$ row `N` indicates a free row If `rtype = []`, all rows are assumed to be of type `'L'`. If `rtype` is a single character, all constraints are assigned the corresponding type.
`lb`	Lower bounds. If `lb = []` it means there are no lower bounds. If `lb` is a scalar, `x` is uniformly bounded by that scalar.
`ub`	Upper bounds. If `ub = []` it means there are no upper bounds. If `ub` is a scalar, `x` is uniformly bounded by that scalar.
`options`	Options structure created with `optimset` or `xprsoptimset` functions. See `xprsoptimset` for more details.

Output arguments

`x`	Solution found by the optimization function. If `exitflag >0`, then `x` is a solution; otherwise, `x` is the value of the optimization routine when it terminated prematurely.
`fval`	Value of the objective function at the solution `x`.
`exitflag`	Integer identifying the reason the optimization algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated. `1` function converged to a solution `x` (LPSTATUS=OPTIMAL) `0` number of iterations exceeded iter limit (LPSTATUS=UNFINISHED and STOPSTATUS=ITERLIMIT) `-2` no feasible point was found (LPSTATUS=INFEAS) `-3` problem is unbounded (LPSTATUS=UNBOUNDED) `-8` other stop reason, see LPSTATUS and STOPSTATUS for details
`output`	Structure containing information about the optimization and, eventually, values of Xpress controls and attributes. See `linprog` and the Section 2.2.6 for details.

`lambda`	Structure containing the Lagrange multipliers at the solution `x` (separated by constraint type). The fields of the structure are:
	`lower` lower bounds `lb`
	`upper` upper bounds `ub`
	`lin` linear constraints from matrix `A`

Further information

1. This routine finds the minimum of a problem specified by

$$\begin{array}{rl} \min & f \cdot x \\ \text{s.t.} & A \cdot x \leq | = | \geq b \\ & lb \leq x \leq ub \end{array}$$

   where A is an $m \times n$ matrix; f, b, $rtype$, lb, and ub are vectors.

2. Input arguments `rtype`, `lb`, `ub` and `options` can be omitted, with the condition that, if one is omitted, also all the following ones must be omitted (as in `x=xprslp(f, A, b, rtype)`). Omitting an input argument has the same effect as passing an empty array `[]`.

3. All output arguments can be omitted too, again with the condition that, if one is omitted, also all the following ones must be omitted (as in `[x, fval]= xprslp(f, A, b, rtype)`).

4. If the specified input bounds for a problem are inconsistent, the output `x` and `fval` are set to `[]`.

Related topics

`xprsoptimset`, `linprog`

xprsmip

Purpose

Solve mixed integer linear programming problems with Xpress.

Synopsis

```
x = xprsmip(f,A,b,rtype,ctype, clim,sos,lb,ub,x0,options)
[x,fval,exitflag,output] = xprsmip(...)
```

Input arguments

`f` Linear objective function vector

`A` Matrix for linear constraints

`b` Vector for constraints RHS

`rtype` Character vector (string) giving the row types:
- `L` indicates a $\leq$ row
- `E` indicates a = row
- `G` indicates a $\geq$ row
- `N` indicates a free row

If `rtype = []`, all rows are assumed to be of type `'L'`. If `rtype` is a single character, all constraints are assigned the corresponding type.

`ctype` Character vector (string) giving the column types:
- `C` (or `\0`) continuous variables
- `B` binary variables
- `I` integer variables
- `P` partial integer variables
- `S` semi-continuous variables
- `R` semi-continuous integers

If `ctype = []`, all rows are assumed to be of type `'C'`. If `ctype` is a single character, all constraints are assigned the corresponding type.

`clim` Vector containing the integer limits for the partial integer variables and lower bounds for semi-continuous and semi-continuous integer variables (column types `'P'`, `'S'`, `'R'`). Values in the positions corresponding to all other columns are ignored. `clim` is mandatory if there are any `'P'`, `'S'`, or `'R'` columns. If `clim` is a scalar, all columns are assigned to that same limit.

`sos` Struct vector defining SOS sets. The number of SOS sets is given by the number of elements in the struct. The struct must contain the following fields:
- `sos(i).type` a character indicating the SOS type, either `'1'` or `'2'`
- `sos(i).ind` numeric vector with the indices of columns in the set (column indices start from 0)
- `sos(i).wt` numeric vector with the reference row weights corresponding to the columns in the `sos(i).ind` vector. It must have the same length as `sos(i).ind`

`lb` Lower bounds. If `lb = []` it means there are no lower bounds. If `lb` is a scalar, `x` is uniformly bounded by that scalar.

`ub` Upper bounds. If `ub = []` it means there are no upper bounds. If `ub` is a scalar, `x` is uniformly bounded by that scalar.

`x0` Optional initial known solution used to speed-up search.

`options` Options structure created with `optimset` or `xprsoptimset` functions. See `xprsoptimset` for more details.

Output arguments

`x` Solution found by the optimization function. If `exitflag >0`, then `x` is a solution; otherwise, `x` is the value of the optimization routine when it terminated

	prematurely.
`fval`	Value of the objective function at the solution `x`.
`exitflag`	Integer identifying the reason the optimization algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated. `1` function converged to a solution `x` (MIPSTATUS=MIP_OPTIMAL) `0` number of iterations exceeded iter limit (STOPSTATUS= STOP_ITERLIMIT) `-2` the problem is infeasible (MIPSTATUS=MIP_INFEAS) `-4` number of searched nodes exceeded limit (STOPSTATUS= STOP_NODELIMIT) `-5` search time exceeded limit (STOPSTATUS= STOP_TIMELIMIT) `-8` other stop reason, see MIPSTATUS and STOPSTATUS for details
`output`	Structure containing information about the optimization and, eventually, values of Xpress controls and attributes. See `linprog` and the Section 2.2.6 for details.

Further information

1. This routine finds the minimum of a problem specified by

$$\begin{array}{rl} \min & f \cdot x \\ \text{s.t.} & A \cdot x \leq | = | \geq b \\ & lb \leq x \leq ub \end{array}$$

   where A is an $m \times n$ matrix; `f`, `b`, `rtype`, `ctype`, `clim`, `lb`, `ub`, and `x0` are vectors; `sos` is a struct vector.

2. Input arguments `rtype` and following can be omitted, with the condition that, if one is omitted, also all the following ones must be omitted (as in `x=xprsmip(f, A, b, rtype)`). Omitting an input argument has the same effect as passing an empty array `[]`.

3. All output arguments can be omitted too, again with the condition that, if one is omitted, also all the following ones must be omitted (as in `[x, fval]= xprsmip(f, A, b, rtype)`).

4. If the specified input bounds for a problem are inconsistent, the output `x` and `fval` are set to `[]`.

Related topics

`xprsoptimset`, `bintprog`

xprsmiqcqp

Purpose

Solve mixed integer quadratically constrained quadratic programming problems with Xpress.

Synopsis

```
x = xprsmiqcqp(H,f,A,Q,b,rtype,ctype, clim,sos,lb,ub,x0,options)
[x,fval,exitflag,output] = xprsmiqcqp(...)
```

Input arguments

`H`	Matrix for quadratic objective terms
`f`	Linear objective function vector
`A`	Matrix for the linear part of the constraints
`Q`	Cell array of length m with the $n \times n$ matrices for the quadratic terms of the constraints. If there is only one constraint ($m = 1$), then Q can be a simple double matrix instead of a cell array. For a linear constraint, the corresponding `Q{i}` matrix can be set to `[]`.
`b`	Vector for constraints RHS
`rtype`	Character vector (string) giving the row types: `L` indicates a $\leq$ row `E` indicates a = row `G` indicates a $\geq$ row `N` indicates a free row If `rtype = []`, all rows are assumed to be of type `'L'`. If `rtype` is a single character, all constraints are assigned the corresponding type.
`ctype`	Character vector (string) giving the column types: `C` (or `\0`) continuous variables `B` binary variables `I` integer variables `P` partial integer variables `S` semi-continuous variables `R` semi-continuous integers If `ctype = []`, all rows are assumed to be of type `'C'`. If `ctype` is a single character, all constraints are assigned the corresponding type.
`clim`	Vector containing the integer limits for the partial integer variables and lower bounds for semi-continuous and semi-continuous integer variables (column types `'P'`, `'S'`, `'R'`). Values in the positions corresponding to all other columns are ignored. `clim` is mandatory if there are any `'P'`, `'S'`, or `'R'` columns. If `clim` is a scalar, all columns are assigned to that same limit.

`sos`	Struct vector defining SOS sets. The number of SOS sets is given by the number of elements in the struct. The struct must contain the following fields:
	`sos(i).type` a character indicating the SOS type, either `'1'` or `'2'`
	`sos(i).ind` numeric vector with the indices of columns in the set (column indices start from 0)
	`sos(i).wt` numeric vector with the reference row weights corresponding to the columns in the `sos(i).ind` vector. It must have the same length as `sos(i).ind`

`lb`	Lower bounds. If `lb = []` it means there are no lower bounds. If `lb` is a scalar, `x` is uniformly bounded by that scalar.
`ub`	Upper bounds. If `ub = []` it means there are no upper bounds. If `ub` is a scalar, `x` is uniformly bounded by that scalar.
`x0`	Optional initial known solution used to speed-up search.

`options` Options structure created with `optimset` or `xprsoptimset` functions. See `xprsoptimset` for more details.

Output arguments

`x` Solution found by the optimization function. If `exitflag >0`, then `x` is a solution; otherwise, `x` is the value of the optimization routine when it terminated prematurely.

`fval` Value of the objective function at the solution `x`.

`exitflag` Integer identifying the reason the optimization algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated.

`1`	function converged to a solution `x` (MIPSTATUS=MIP_OPTIMAL)
`0`	number of iterations exceeded iter limit (STOPSTATUS= STOP_ITERLIMIT)
`-2`	the problem is infeasible (MIPSTATUS=MIP_INFEAS)
`-4`	number of searched nodes exceeded limit (STOPSTATUS= STOP_NODELIMIT)
`-5`	search time exceeded limit (STOPSTATUS= STOP_TIMELIMIT)
`-8`	other stop reason, see MIPSTATUS and STOPSTATUS for details

`output` Structure containing information about the optimization and, eventually, values of Xpress controls and attributes. See `quadprog` and the Section 2.2.6 for details.

Further information

1. This routine finds the minimum of a problem specified by

$$\begin{array}{ll} \min & 0.5 \cdot x' \cdot H \cdot x + f \cdot x \\ \text{s.t.} & A \cdot x + x' \cdot Q_i \cdot x \leq | = | \geq b \\ & lb \leq x \leq ub \end{array}$$

   and x in the domain specified by the `ctype`, `clim` and `sos` arguments, where `H` is an $n \times n$ matrix; `A` is an $m \times n$ matrix; `Q` is a cell array of $n \times n$ matrices; `f`, `b`, `rtype`, `ctype`, `clim`, `lb`, `ub`, and `x0` are vectors; `sos` is a struct vector.

2. Input arguments `rtype` and following can be omitted, with the condition that, if one is omitted, also all the following ones must be omitted (as in `x=xprsmiqcqp(H, f, A, Q, b, rtype)`). Omitting an input argument has the same effect as passing an empty array `[]`.

3. All output arguments can be omitted too, again with the condition that, if one is omitted, also all the following ones must be omitted (as in `[x, fval]= xprsmiqcqp(H, f, A, Q, b, rtype)`).

4. If the specified input bounds for a problem are inconsistent, the output `x` and `fval` are set to `[]`.

Related topics

`xprsoptimset`, `bintprog`, `quadprog`

xprsmiqp

Purpose

Solve mixed integer quadratic programming problems with Xpress.

Synopsis

```
x = xprsmiqp(H,f,A,b,rtype,ctype, clim,sos,lb,ub,x0,options)
[x,fval,exitflag,output] = xprsmiqp(...)
```

Input arguments

`H` Matrix for quadratic objective terms

`f` Linear objective function vector

`A` Matrix for linear constraints

`b` Vector for constraints RHS

`rtype` Character vector (string) giving the row types:
- `L` indicates a $\leq$ row
- `E` indicates a = row
- `G` indicates a $\geq$ row
- `N` indicates a free row

If `rtype = []`, all rows are assumed to be of type `'L'`. If `rtype` is a single character, all constraints are assigned the corresponding type.

`ctype` Character vector (string) giving the column types:
- `C` (or `\0`) continuous variables
- `B` binary variables
- `I` integer variables
- `P` partial integer variables
- `S` semi-continuous variables
- `R` semi-continuous integers

If `ctype = []`, all rows are assumed to be of type `'C'`. If `ctype` is a single character, all constraints are assigned the corresponding type.

`clim` Vector containing the integer limits for the partial integer variables and lower bounds for semi-continuous and semi-continuous integer variables (column types `'P'`, `'S'`, `'R'`). Values in the positions corresponding to all other columns are ignored. `clim` is mandatory if there are any `'P'`, `'S'`, or `'R'` columns. If `clim` is a scalar, all columns are assigned to that same limit.

`sos` Struct vector defining SOS sets. The number of SOS sets is given by the number of elements in the struct. The struct must contain the following fields:
- `sos(i).type` a character indicating the SOS type, either `'1'` or `'2'`
- `sos(i).ind` numeric vector with the indices of columns in the set (column indices start from 0)
- `sos(i).wt` numeric vector with the reference row weights corresponding to the columns in the `sos(i).ind` vector. It must have the same length as `sos(i).ind`

`lb` Lower bounds. If `lb = []` it means there are no lower bounds. If `lb` is a scalar, `x` is uniformly bounded by that scalar.

`ub` Upper bounds. If `ub = []` it means there are no upper bounds. If `ub` is a scalar, `x` is uniformly bounded by that scalar.

`x0` Optional initial known solution used to speed-up search.

`options` Options structure created with `optimset` or `xprsoptimset` functions. See `xprsoptimset` for more details.

Output arguments

`x` Solution found by the optimization function. If `exitflag >0`, then `x` is a solution;

	otherwise, `x` is the value of the optimization routine when it terminated prematurely.
`fval`	Value of the objective function at the solution `x`.
`exitflag`	Integer identifying the reason the optimization algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated. `1` function converged to a solution `x` (MIPSTATUS=MIP_OPTIMAL) `0` number of iterations exceeded iter limit (STOPSTATUS= STOP_ITERLIMIT) `-2` the problem is infeasible (MIPSTATUS=MIP_INFEAS) `-4` number of searched nodes exceeded limit (STOPSTATUS= STOP_NODELIMIT) `-5` search time exceeded limit (STOPSTATUS= STOP_TIMELIMIT) `-8` other stop reason, see MIPSTATUS and STOPSTATUS for details
`output`	Structure containing information about the optimization and, eventually, values of Xpress controls and attributes. See `quadprog` and the Section 2.2.6 for details.

Further information

1. This routine finds the minimum of a problem specified by

$$\begin{array}{ll} \text{min} & 0.5 \cdot x' \cdot H \cdot x + f \cdot x \\ \text{s.t.} & A \cdot x \leq | = | \geq b \\ & lb \leq x \leq ub \end{array}$$

   and x in the domain specified by the `ctype`, `clim` and `sos` arguments, where `H` is an $n \times n$ matrix; `A` is an $m \times n$ matrix; `f`, `b`, `rtype`, `ctype`, `clim`, `lb`, `ub`, and `x0` are vectors; `sos` is a struct vector.

2. Input arguments `rtype` and following can be omitted, with the condition that, if one is omitted, also all the following ones must be omitted (as in `x=xprsmiqp(H, f, A, b, rtype)`). Omitting an input argument has the same effect as passing an empty array `[]`.

3. All output arguments can be omitted too, again with the condition that, if one is omitted, also all the following ones must be omitted (as in `[x, fval]= xprsmiqp(H, f, A, b, rtype)`).

4. If the specified input bounds for a problem are inconsistent, the output `x` and `fval` are set to `[]`.

Related topics

`xprsoptimset`, `bintprog`, `quadprog`

xprsoptimget

Purpose

Retrieve Xpress optimization options values.

Synopsis

```
val = xprsoptimget(options,'param')
val = xprsoptimget(options,'param',default)
```

Arguments

`options`	optimization options structure
`param`	optimization control or attribute name

Return value

Value of the optimization control or attribute.

Example

This statement returns the value of the FEASTOL optimization control parameter in the structure called `my_options`.

```
val = xprsoptimget(my_options,'FEASTOL')
```

This statement returns the value of the FEASTOL optimization control parameter in the structure called `my_options` (as in the previous example) except that if the FEASTOL parameter is not defined, it returns the value 1e-6.

```
optnew = xprsoptimget(my_options,'FEASTOL',1e-6);)
```

Further information

1. `val = xprsoptimget(options,'param')` returns the value of the specified parameter in the optimization options structure options. The parameter name is case sensitive and must be a valid Xpress control parameter name.
2. `val = xprsoptimget(options,'param',default)` returns `default` if the specified parameter is not defined in the optimization options structure options.

Related topics

`xprsoptimset`

xprsoptimset

Purpose

Create or edit Xpress optimization options structures.

Synopsis

```
options = xprsoptimset('param1',value1,'param2',value2,...)
options = xprsoptimset
options = xprsoptimset(oldopts,'param1',value1,...)
options = xprsoptimset(oldopts,newopts)
```

Arguments

`param*`	optimization control or attribute name
`value*`	new value for the optimization option
`oldopts`	optimization options structure to copy
`newopts`	optimization options structure

Return value

A new optimization options structure.

Example

This statement creates an optimization options structure called *options* in which the FEASTOL parameter is set to 1e-8 and the MAXMIPSOL parameter is set to 10.

```
options = xprsoptimset('FEASTOL',1e-8,'MAXMIPSOL',10)
```

This statement makes a copy of the options structure called *options*, changing the value of the PRESOLVE parameter and storing new values in *optnew*.

```
optnew = xprsoptimset(options,'PRESOLVE',0);
```

This statement creates an Xpress optimization options structure with control values corresponding to the *'final'* value of the MATLAB Toolbox option Display.

```
options = xprsoptimset(optimset('Display', 'final'));
```

This statement returns an optimization options structure that contains all the parameter names and default values.

```
defaults = xprsoptimset
```

Further information

The function `xprsoptimset` creates an *options structure* that you can pass as an input argument to the Xpress optimization functions. You can use the options structure to change the default parameters for these functions.

`options = xprsoptimset('param1',value1,'param2',value2,...)`
: creates an optimization options structure called options, in which the specified parameters (`param*`) have the specified values. The parameter names are case sensitive and must be valid Xpress control parameter names.

`xprsoptimset`
: with no input returns a complete list of parameters with their default values.

`options = xprsoptimset(oldopts,'param1',value1,...)`
: creates a copy of `oldopts`, modifying or adding the specified parameters with the specified values.

`options = xprsoptimset(oldopts,newopts)`
combines an existing options structure `oldopts` with a new options structure `newopts`. Any parameters in `newopts` with nonempty values overwrite the corresponding old parameters in `oldopts`

In the last two cases, `oldopts` can be a MATLAB Toolbox option structure, in which case the following parameters are converted to the corresponding Xpress controls (others are ignored):

Display → OUTPUTLOG, MIPLOG, LPLOG
MaxIter → LPITERLIMIT
TolRLPFun → OPTIMALITYTOL
MaxTime → MAXTIME
MaxNode → MAXNODE
NodeDisplayInterval → MIPLOG
NodeSearchStrategy → NODESELECTION
TolXInteger → MIPTOL

Only options that are set to a non-empty value are taken into consideration.

Related topics

`xprsoptimget`

xprsqcqp

Purpose

Solve quadratically constrained quadratic programming problems with Xpress.

Synopsis

```
x = xprsqcqp(H,f,A,Q,b,rtype,lb,ub,options)
[x,fval,exitflag,output,lambda] = xprsqcqp(...)
```

Input arguments

`H`	Matrix for quadratic objective terms
`f`	Linear objective function vector
`A`	Matrix for the linear part of the constraints
`Q`	Cell array of length m with the $n \times n$ matrices for the quadratic terms of the constraints. If there is only one constraint ($m = 1$), then Q can be a simple double matrix instead of a cell array. For a linear constraint, the corresponding `Q{i}` matrix can be set to `[]`.
`b`	Vector for constraints RHS
`rtype`	Character vector (string) giving the row types: `L` indicates a $\leq$ row `E` indicates a = row `G` indicates a $\geq$ row `N` indicates a free row If `rtype = []`, all rows are assumed to be of type `'L'`. If `rtype` is a single character, all constraints are assigned the corresponding type.
`lb`	Lower bounds. If `lb = []` it means there are no lower bounds. If `lb` is a scalar, `x` is uniformly bounded by that scalar.
`ub`	Upper bounds. If `ub = []` it means there are no upper bounds. If `ub` is a scalar, `x` is uniformly bounded by that scalar.
`options`	Options structure created with `optimset` or `xprsoptimset` functions. See `xprsoptimset` for more details.

Output arguments

`x`	Solution found by the optimization function. If `exitflag >0`, then `x` is a solution; otherwise, `x` is the value of the optimization routine when it terminated prematurely.
`fval`	Value of the objective function at the solution `x`.
`exitflag`	Integer identifying the reason the optimization algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated. `1` function converged to a solution `x` (LPSTATUS=OPTIMAL) `0` number of iterations exceeded iter limit (LPSTATUS=UNFINISHED and STOPSTATUS=ITERLIMIT) `-2` no feasible point was found (LPSTATUS=INFEAS) `-3` problem is unbounded (LPSTATUS=UNBOUNDED) `-8` other stop reason, see LPSTATUS and STOPSTATUS for details
`output`	Structure containing information about the optimization and, eventually, values of Xpress controls and attributes. See `quadprog` and the Section 2.2.6 for details.

`lambda`	Structure containing the Lagrange multipliers at the solution `x` (separated by constraint type). The fields of the structure are:
	`lower` lower bounds `lb`
	`upper` upper bounds `ub`
	`lin` linear constraints from matrix `A`

Further information

1. This routine finds the minimum of a problem specified by

$$\begin{array}{rl} \min & 0.5 \cdot x' \cdot H \cdot x + f \cdot x \\ \text{s.t.} & A \cdot x + x' \cdot Q_i \cdot x \leq | = | \geq b \\ & lb \leq x \leq ub \end{array}$$

   where `H` is an $n \times n$ matrix; `A` is an $m \times n$ matrix; `Q` is a cell array of $n \times n$ matrices; `f`, `b`, `rtype`, `lb`, and `ub` are vectors.

2. Input arguments `rtype`, `lb`, `ub` and `options` can be omitted, with the condition that, if one is omitted, also all the following ones must be omitted (as in `x=xprsqcqp(H, f, A, Q, b, rtype)`). Omitting an input argument has the same effect as passing an empty array `[]`.

3. All output arguments can be omitted too, again with the condition that, if one is omitted, also all the following ones must be omitted (as in `[x, fval]= xprsqcqp(H, f, A, Q, b, rtype)`).

4. If the specified input bounds for a problem are inconsistent, the output `x` and `fval` are set to `[]`.

Related topics

`xprsoptimset`, `quadprog`

xprsqp

Purpose

Solve quadratic programming problems with Xpress.

Synopsis

```
x = xprsqp(H,f,A,b,rtype,lb,ub,options)
[x,fval,exitflag,output,lambda] = xprsqp(...)
```

Input arguments

`H`	Matrix for quadratic objective terms
`f`	Linear objective function vector
`A`	Matrix for linear constraints
`b`	Vector for constraints RHS
`rtype`	Character vector (string) giving the row types: `L` indicates a $\leq$ row `E` indicates a = row `G` indicates a $\geq$ row `N` indicates a free row If `rtype = []`, all rows are assumed to be of type `'L'`. If `rtype` is a single character, all constraints are assigned the corresponding type.
`lb`	Lower bounds. If `lb = []` it means there are no lower bounds. If `lb` is a scalar, `x` is uniformly bounded by that scalar.
`ub`	Upper bounds. If `ub = []` it means there are no upper bounds. If `ub` is a scalar, `x` is uniformly bounded by that scalar.
`options`	Options structure created with `optimset` or `xprsoptimset` functions. See `xprsoptimset` for more details.

Output arguments

`x`	Solution found by the optimization function. If `exitflag` >0, then `x` is a solution; otherwise, `x` is the value of the optimization routine when it terminated prematurely.
`fval`	Value of the objective function at the solution `x`.
`exitflag`	Integer identifying the reason the optimization algorithm terminated. The following lists the values of `exitflag` and the corresponding reasons the algorithm terminated. `1` function converged to a solution `x` (LPSTATUS=OPTIMAL) `0` number of iterations exceeded iter limit (LPSTATUS=UNFINISHED and STOPSTATUS=ITERLIMIT) `-2` no feasible point was found (LPSTATUS=INFEAS) `-3` problem is unbounded (LPSTATUS=UNBOUNDED) `-8` other stop reason, see LPSTATUS and STOPSTATUS for details
`output`	Structure containing information about the optimization and, eventually, values of Xpress controls and attributes. See `linprog` and the Section 2.2.6 for details.

`lambda`	Structure containing the Lagrange multipliers at the solution `x` (separated by constraint type). The fields of the structure are:
	`lower` lower bounds `lb`
	`upper` upper bounds `ub`
	`lin` linear constraints from matrix `A`

Further information

1. This routine finds the minimum of a problem specified by

$$\begin{array}{rl} \text{min} & 0.5 \cdot x' \cdot H \cdot x + f \cdot x \\ \text{s.t.} & A \cdot x \leq | = | \geq b \\ & lb \leq x \leq ub \end{array}$$

   where `H` is an $n \times n$ matrix; `A` is an $m \times n$ matrix; `f`, `b`, `rtype`, `lb`, and `ub` are vectors.

2. Input arguments `rtype`, `lb`, `ub` and `options` can be omitted, with the condition that, if one is omitted, also all the following ones must be omitted (as in `x=xprsqp(H, f, A, b, rtype)`). Omitting an input argument has the same effect as passing an empty array `[]`.

3. All output arguments can be omitted too, again with the condition that, if one is omitted, also all the following ones must be omitted (as in `[x, fval]= xprsqp(H, f, A, b, rtype)`).

4. If the specified input bounds for a problem are inconsistent, the output `x` and `fval` are set to `[]`.

Related topics

`xprsoptimset`, `quadprog`

xprsver

Purpose

Display version number for Xpress.

Synopsis

```
xprsver
```

Example

Display the version:

```
xprsver
```

MATLAB display:

```
FICO Xpress Optimizer 64-bit v21.00.02 (Hyper capacity)
(c) Copyright Fair Isaac Corporation 2010
```

Further information

This routine prints the version and release number for the Xpress software currently running.

Related topics

`xprsoptimget`, `linprog`

Appendix A

Contacting FICO

FICO provides clients with support and services for all our products. Refer to the following sections for more information.

Product support

FICO offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have purchased a FICO product and have an active support or maintenance contract. You can find support contact information on the Product Support home page (www.fico.com/support).

On the Product Support home page, you can also register for credentials to log on to FICO Online Support, our web-based support tool to access Product Support 24x7 from anywhere in the world. Using FICO Online Support, you can enter cases online, track them through resolution, find articles in the FICO Knowledge Base, and query known issues.

Please include *'Xpress'* in the subject line of your support queries.

Product education

FICO Product Education is the principal provider of product training for our clients and partners. Product Education offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support. For additional information, visit the Product Education homepage at www.fico.com/en/product-training or email producteducation@fico.com.

Product documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide. If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com.

Sales and maintenance

USA, CANADA AND ALL AMERICAS

Email: XpressSalesUS@fico.com

WORLDWIDE

Email: XpressSalesUK@fico.com

Tel: +44 207 940 8718
Fax: +44 870 420 3601

Xpress Optimization, FICO
FICO House
International Square
Starley Way
Birmingham B37 7GN
UK

Related services

Strategy Consulting: Included in your contract with FICO may be a specified amount of consulting time to assist you in using FICO Optimization Modeler to meet your business needs. Additional consulting time can be arranged by contract.

Conferences and Seminars: FICO offers conferences and seminars on our products and services. For announcements concerning these events, go to www.fico.com or contact your FICO account representative.

About FICO

FICO (NYSE:FICO) delivers superior predictive analytics solutions that drive smarter decisions. The company's groundbreaking use of mathematics to predict consumer behavior has transformed entire industries and revolutionized the way risk is managed and products are marketed. FICO's innovative solutions include the FICO® Score—the standard measure of consumer credit risk in the United States—along with industry-leading solutions for managing credit accounts, identifying and minimizing the impact of fraud, and customizing consumer offers with pinpoint accuracy. Most of the world's top banks, as well as leading insurers, retailers, pharmaceutical companies, and government agencies, rely on FICO solutions to accelerate growth, control risk, boost profits, and meet regulatory and competitive demands. FICO also helps millions of individuals manage their personal credit health through www.myfico.com. Learn more at www.fico.com. FICO: Make every decision count™.

Index