KNITRO Documentation

Release 9.1

Ziena Optimization LLC

December 03, 2014

CONTENTS

1 Introduction **3**
- 1.1 Product overview . . . 3
- 1.2 Getting KNITRO . . . 4
- 1.3 Installation . . . 4
- 1.4 Troubleshooting . . . 10
- 1.5 Release notes . . . 12

2 User guide **15**
- 2.1 Getting started . . . 15
- 2.2 Setting options . . . 25
- 2.3 Derivatives . . . 27
- 2.4 Multistart . . . 39
- 2.5 Mixed-integer nonlinear programming . . . 44
- 2.6 Complementarity constraints . . . 50
- 2.7 Algorithms . . . 59
- 2.8 Feasibility and infeasibility . . . 62
- 2.9 Parallelism . . . 63
- 2.10 The KNITRO-Tuner . . . 67
- 2.11 Termination criteria . . . 70
- 2.12 Obtaining information . . . 72
- 2.13 Callbacks . . . 80
- 2.14 Other programmatic interfaces . . . 85
- 2.15 Special problem classes . . . 87
- 2.16 Tips and tricks . . . 89
- 2.17 Bibliography . . . 91

3 Reference manual **93**
- 3.1 KNITRO / AMPL reference . . . 93
- 3.2 KNITRO / MATLAB reference . . . 105
- 3.3 Callable library reference . . . 115
- 3.4 List of output files . . . 170

This documentation is divided into three parts. The *Introduction* provides an overview of the KNITRO solver and its capabilities, and explains where to get it and how to install it. If you already have a running version of KNITRO and want to learn how to use it, you may want to skip the introduction and go directly to the *User guide*. This section provides a gentle introduction to the main features of KNITRO by means of a few simple examples. Finally, the last chapter consists of the *Reference manual*: an exhaustive description of the KNITRO API, user options, status codes and output files that are associated with the software.

CHAPTER

ONE

INTRODUCTION

This chapter contains an overview of what KNITRO can do, where to obtain it and how to get it to work. If you already have a working installation of KNITRO and know the basics of what nonlinear programming is, you may want to skip it and go directly to the next chapter, *User guide*. Otherwise, read on!

1.1 Product overview

KNITRO is an optimization software library for finding solutions of both continuous (smooth) optimization models (with or without constraints), as well as discrete optimization models with integer or binary variables (i.e. mixed integer programs). KNITRO is primarily designed for finding local optimal solutions of large-scale, continuous nonlinear problems.

The problems solved by KNITRO have the form

$$\min \ f(x) \quad \text{subject to} \quad c^L \leq c(x) \leq c^U, \ b^L \leq x \leq b^U$$

where $x \in \mathbf{R}^n$ are the unknown variables (which can be specified as continuous, binary or integer), c^L and c^U are lower and upper bounds (possibly infinite) on the general constraints, and b^L and b^U are lower and upper simple bounds (possibly infinite) on the variables. This formulation allows many types of constraints, including equalities (if $c^L = c^U$), fixed variables (if $b^L = b^U$), and both single and double-sided inequality constraints or bounded variables. Complementarity constraints may also be included. KNITRO assumes that the functions $f(x)$, and $c(x)$ are smooth, although problems with derivative discontinuities can often be solved successfully.

Although primarily designed for general, nonlinear optimization, KNITRO is efficient at solving all of the following classes of optimization problems (described in more detail in Section *Special problem classes*):

- unconstrained;
- bound constrained;
- systems of nonlinear equations;
- least squares problems, both linear and nonlinear;
- linear programming problems (LPs);
- quadratic programming problems (QPs), both convex and nonconvex;
- mathematical programs with complementarity constraints (MPCCs);
- general nonlinear (smooth) constrained problems (NLP), both convex and nonconvex;
- mixed integer linear programs (MILP) of moderate size;
- mixed integer (convex) nonlinear programs (MINLP) of moderate size.

The KNITRO package provides the following features:

- efficient and robust solution of small or large problems;
- solvers for both continuous and discrete problems;
- derivative-free, 1st derivative, and 2nd derivative options;
- option to remain feasible throughout the optimization or not;
- multi-start heuristics for trying to locate the global solution;
- both interior-point (barrier) and active-set methods;
- both iterative and direct approaches for computing steps;
- support for Windows (32-bit and 64-bit), Linux (64-bit) and Mac OS X (64-bit);
- programmatic interfaces: C/C++, Fortran, Java, Python;
- modeling language interfaces: AMPL ©, AIMMS ©, GAMS ©, MATLAB ©, MPL ©, Microsoft Excel Premium Solver ©;
- thread-safe libraries for easy embedding into application software.

1.2 Getting KNITRO

KNITRO is developed by Ziena Optimization LLC, and marketed and supported by Artelys. We have offices in Chicago, Los Angeles, Montréal and Paris. Support is provided in English or French.

Free, time-limited trial versions of KNITRO can be downloaded there:

http://www.artelys.com/knitro

Requests for information and purchase may be directed to:

info-knitro@artelys.com

For support questions related to KNITRO, send an email to:

support-knitro@artelys.com

1.3 Installation

KNITRO 9.1 is supported on the platforms described in the table below.

PLATFORM	OPERATING SYSTEM	PROCESSOR
Windows 32-bit	Windows Server 2008, Vista, Windows 7, Windows 8, Windows 8.1	AMD Duron/Intel Pentium3 or later x86 CPU
Windows 64-bit	Windows Server 2008, Vista Windows 7, Windows 8, Windows 8.1	Any AMD64 or Intel EM64T enabled 64-bit CPU
Linux 64-bit	RedHat (glibc2.5+) compatible (parallel features require OpenMP)	Any AMD64 or Intel EM64T enabled 64-bit CPU
Mac OS X 64-bit	Version 10.8 (Mountain Lion) or later	Intel EM64T enabled 64-bit CPU

For enquiries about using KNITRO on unsupported platforms, please contact Ziena or Artelys.

Listed below are the C/C++ compilers used to build KNITRO, and the Java and Fortran compilers used to test programmatic interfaces. It is usually not difficult for Ziena to compile KNITRO in a different environment (for example, it is routinely recompiled to specific versions of **gcc** on Linux). Contact us if your application requires special compilation of KNITRO.

```
> Windows (32-bit x86)
> > C/C++    > Microsoft Visual Studio C++ 10.0
> > Java:    > 1.5.0_16 from Sun
> Windows (64-bit x86_64)
> > C/C++:   > Microsoft Visual Studio C++ 10.0
> > Java:    > 1.5.0_10 from Sun
> Linux (64-bit x86_64)
> > C/C++:   > gcc/g++ 4.4.4
> > C/C++:   > gcc/g++ 4.1.2 (sequential version)
> > Java:    > 1.6.0_34 from Sun
> Mac OS X (64-bit x86_64)
> > C/C++:   > gcc/g++ 4.2.1 (XCode 4.4.0)
> > Java:    > 1.5.0
```

Note: Note that for 64-bit Linux, the parallel features in the standard KNITRO libraries require you to have versions of gcc/g++ that support OpenMP so that these libraries are compatible for linking against. If you wish to use an older version of gcc/g++ on Linux, then you can only use the *sequential* versions of the KNITRO libraries on this platform. See the `README` file provided in the `lib` directory for more information.

Instructions for installing the KNITRO package on supported platforms are given below. After installing, view the `INSTALL.txt`, `LICENSE_KNITRO.txt`, and `README.txt` files, then test the installation by running one of the examples provided with the distribution.

The KNITRO product contains example interfaces written in various programming languages under the directory `examples`. Each example consists of a main driver program coded in the given language that defines an optimization problem and invokes KNITRO to solve it. Examples also contain a makefile illustrating how to link the KNITRO library with the target language driver program.

1.3.1 Windows

The KNITRO software package for Windows is delivered as a zipped file (ending in *.zip*), or as a self-extracting executable (ending in *.exe*). For the zipped file, double-click on it and extract all contents to a new folder. For the *.exe* file, double-click on it and follow the instructions. The self-extracting executable creates start menu shortcuts and an uninstall entry in Add/Remove Programs; otherwise, the two install methods are identical.

The default installation location for KNITRO is (assuming your `HOMEDRIVE` is "C:"):

```
> C:\Program Files\Ziena
```

Unpacking will create a folder named knitro-9.1.x-z (or knitroampl-9.1.x-z for the KNITRO/AMPL solver product, or knitroMatlab-9.1.x-z for the KNITRO/MATLAB solver product). Contents of the full product distribution are the following:

- `INSTALL.txt`: a file containing installation instructions.
- `LICENSE_KNITRO.txt`: a file containing the KNITRO license agreement.
- `README.txt`: a file with instructions on how to get started using KNITRO.
- `KNITRO91-ReleaseNotes.txt`: a file containing release notes.
- `get_machine_ID`: an executable that identifies the machine ID, required for obtaining a Ziena license file.

- doc: a folder containing KNITRO documentation, including this manual.
- include: a folder containing the KNITRO header file knitro.h.
- lib: a folder containing the KNITRO library and object files: knitro_objlib.a, knitro.lib and knitro.dll, as well as any other libraries that are used with KNITRO.
- examples: a folder containing examples of how to use the KNITRO API in different programming languages (C, C++, Fortran, Java, Python). The examples\C folder contains the most extensive set (see examples\C\README.txt for details).
- knitroampl: a folder containing knitroampl.exe (the KNITRO solver for AMPL), instructions, and an example model for testing KNITRO with AMPL.
- knitromatlab: A folder containing the files needed to use the KNITRO solver for MATLAB, example models, and the instructions and explanation file README.txt.

To activate KNITRO for your computer you will need a valid **Ziena license file**. If you purchased a floating network license, then refer to the Ziena License Manager User's Manual provided in the doc folder of your distribution.

For a stand-alone, single computer license, double-click on the get_machine_ID.bat batch file provided with the distribution. This will generate a machine ID (five pairs of hexadecimal digits). Alternatively, open a DOS-like command window (click Start / Run, and then type **cmd**). Change to the directory where you unzipped the distribution, and type get_machine_ID.exe, a program supplied with the distribution to generate the machine ID.

Email the machine ID to

info-knitro@artelys.com

if purchased through Artelys. (If KNITRO was purchased through a distributor, then email the machine ID to your local distributor). Artelys (or your local distributor) will then send you a license file containing the encrypted license text string. The Ziena license manager supports a variety of ways to install licenses. The simplest procedure is to place each license file in the folder:

```
> C:\Program Files\Ziena\
```

(create the folder above if it does not exist). The license file name may be changed, but must begin with the characters ziena_lic. If this does not work, try creating a new environment variable called ZIENA_LICENSE and set it to the folder holding your license file(s). See information on setting environment variables below and refer to the Ziena License Manager User's Manual for more installation details.

Setting environment variables

In order to run KNITRO binary or executable files from anywhere on your Windows computer, as well as load dynamic libraries (or dll's) used by KNITRO at runtime, it is necessary to make sure that the PATH system environment variable is set properly on your Windows machine. In particular, you must update the system PATH environment variable so that it indicates the location of the KNITRO lib folder (containing the KNITRO provided dll's) and the knitroampl folder (or whichever folder contains the **knitroampl.exe** executable file). This can be done as follows.

- Windows 8
  - From the Windows Start screen, search for and open "Edit the system environment variables".
  - Click the Advanced tab.
  - Click Environment Variables.
  - Under System variables, edit the Path variable to add the KNITRO lib folder and knitroampl folder. Specify the whole path to these folders, and make sure to separate the paths by a semi-colon.
- Windows Vista and Windows 7

- At the Windows desktop, right-click "Computer".
- Select "Properties".
- Click on Advanced System Settings in the left pane.
- In the System Properties window select the Advanced tab.
- Click Environment Variables.
- Under System variables, edit the Path variable to add the KNITRO `lib` folder and `knitroampl` folder. Specify the whole path to these folders, and make sure to separate the paths by a semi-colon.

Note that you may need to restart your Windows machine after modifying the environment variables, for the changes to take effect. Simply logging out and relogging in is not enough. Moreover, if the `PATH` environment variable points to more than one folder that contains an executable or dll of the same name, the one that will be chosen is the one whose folder appears first in the `PATH` variable definition.

If you are using KNITRO with AMPL, you should make sure the folder containing the AMPL executable file **ampl.exe** is also added to the `PATH` variable (as well as the folder containing the **knitroampl.exe** as described above). Additionally, if you are using an external third party dll with KNITRO such as your own Basic Linear Algebra Subroutine (BLAS) libraries (see user options `blasoption` and `blasoptionlib`), or a Linear Programming (LP) solver library (see user option `lpsolver`), then you will also need to add the folders containing these dll's to the system `PATH` environment variable as described in the last step above.

If you are setting the `ZIENA_LICENSE` environment variable to activate your license, then follow the instructions above, but in the last step create a new environment variable called `ZIENA_LICENSE` and give it the value of the folder containing your Ziena license file (specify the whole path to this folder).

KNITRO for MATLAB

To use KNITRO with MATLAB, you may need to add the KNITRO/MATLAB interface files to your MATLAB path. Assuming the default installation folders were used and the KNITRODIR environment variable contains the path to the KNITRO installation directory, set by the installer or manually, the MATLAB path can be updated with the following commands in MATLAB:

```
> addpath(strcat(getenv('KNITRODIR'),'\knitromatlab'));
> savepath();
```

Alternatively, if the environment variable is not set properly, you can update the MATLAB path by calling addpath() with the full path to the KNITRO/MATLAB interface files, such as:

```
> addpath('C:\Program Files\Ziena\KNITRO 9.1.x\knitromatlab');
> savepath();
```

1.3.2 Unix (Linux, Mac OS X, Solaris)

KNITRO is supported on Linux (64-bit), Mac OS X (64-bit x86_64 on Mac OS X 10.8 or higher).

The KNITRO software package for Unix is delivered as a gzipped tar file, or as a zip file on Mac OS X. Save this file in a fresh subdirectory on your system. To unpack a gzipped tar file, type the commands:

```
> gunzip KNITRO-9.1.x-platformname.tar.gz
> tar -xvf KNITRO-9.1.x-platformname.tar
```

Unpacking will create a directory named knitro-9.1.x-z (or knitroampl-9.1.x-z for the KNITRO/AMPL solver product, or knitroMatlab-9.1.x-z for the KNITRO/MATLAB solver product). Contents of the full product distribution are the following:

- INSTALL: A file containing installation instructions.
- LICENSE_KNITRO: A file containing the KNITRO license agreement.
- README: A file with instructions on how to get started using KNITRO.
- KNITRO91-ReleaseNotes: A file containing release notes.
- get_machine_ID: An executable that identifies the machine ID, required for obtaining a Ziena license file.
- doc: A directory containing KNITRO documentation, including this manual.
- include: A directory containing the KNITRO header file knitro.h.
- lib: A directory containing the KNITRO library files: libknitro.a and libknitro.so (libknitro.dylib on Mac OS X), as well as any other libraries that can be used with KNITRO.
- examples: A directory containing examples of how to use the KNITRO API in different programming languages (C, C++, Fortran, Java, Python). The examples/C directory contains the most extensive set (see examples/C/README.txt for details).
- knitroampl: A directory containing **knitroampl** (the KNITRO solver for AMPL), instructions, and an example model for testing KNITRO with AMPL.
- knitromatlab: A folder containing the files needed to use the KNITRO solver for MATLAB, example models, and the instructions and explanation file README.

To activate KNITRO for your computer you will need a valid Ziena license file. If you purchased a floating network license, then refer to the Ziena License Manager User's Manual. For a stand-alone license, execute **get_machine_ID**, a program supplied with the distribution. This will generate a machine ID (five pairs of hexadecimal digits). Email the machine ID to

info-knitro@artelys.com

if purchased through Artelys. (If KNITRO was purchased through a distributor, then email the machine ID to your local distributor). Artelys (or your local distributor) will then send a license file containing the encrypted license text string. The Ziena license manager supports a variety of ways to install licenses. The simplest procedure is to copy each license into your HOME directory. The license file name may be changed, but must begin with the characters ziena_lic (use lower-case letters). If this does not work, try creating a new environment variable called ZIENA_LICENSE and set it to the folder holding your license file(s). See information on setting environment variables below and refer to the Ziena License Manager User's Manual for more installation details.

Setting environment variables

In order to run KNITRO binary or executable files from anywhere on your Unix computer, as well as load dynamic, shared libraries (i.e. ".so" or ".dylib" files) used by KNITRO at runtime, it is necessary to make sure that several environment variables are set properly on your machine.

In particular, you must update the PATH environment variable so that it indicates the location of the knitroampl directory (or whichever directory contains the **knitroampl** executable file). You must also update the LD_LIBRARY_PATH (DYLD_LIBRARY_PATH on Mac OS X) environment variable so that it indicates the location of the KNITRO lib directory (containing the KNITRO provided ".so" or ".dylib" shared libraries).

Setting the PATH and LD_LIBRARY_PATH (DYLD_LIBRARY_PATH on Mac OS X) environment variables on Unix systems can be done as follows. In the instructions below, replace <*file_absolute_path*> with the full path to the directory containing the KNITRO binary file (e.g. the knitroampl directory), and replace <*file_absolute_library_path*> with the full path to the directory containing the KNITRO shared object library (e.g. the KNITRO lib directory).

Linux

If you run a Unix bash shell, then type:

```
> export PATH= <file_absolute_path>:$PATH
> export LD_LIBRARY_PATH= <file_absolute_library_path>:$LD_LIBRARY_PATH
```

If you run a Unix csh or tcsh shell, then type:

```
> setenv PATH  <file_absolute_path>:$PATH
> setenv LD_LIBRARY_PATH  <file_absolute_library_path>:$LD_LIBRARY_PATH
```

Mac OS X

Determine the shell being used:

```
> echo $SHELL
```

If you run a Unix bash shell, then type:

```
> export PATH= <file_absolute_path>:$PATH
> export DYLD_LIBRARY_PATH=<file_absolute_library_path>:$DYLD_LIBRARY_PATH
```

If you run a Unix csh or tcsh shell, then type:

```
> setenv PATH  <file_absolute_path>:$PATH
> setenv DYLD_LIBRARY_PATH <file_absolute_library_path>:$DYLD_LIBRARY_PATH
```

Note that the value of the environment variable is only valid in the shell in which it was defined. Moreover, if a particular environment variable points to more than one directory that contains a binary or dynamic library of the same name, the one that will be chosen is the one whose directory appears first in the environment variable definition.

If you are using KNITRO with AMPL, you should also make sure the directory containing the AMPL executable file **ampl** is added to the `PATH` environment variable (as well as the directory containing the **knitroampl** executable file as described above). Additionally, if you are using an external third party runtime library with KNITRO such as your own Basic Linear Algebra Subroutine (BLAS) libraries (see user options `blasoption` and `blasoptionlib`), or a Linear Programming (LP) solver library (see user option `lpsolver`), then you will also need to add the directories containing these libraries to the `LD_LIBRARY_PATH` (`DYLD_LIBRARY_PATH` on Mac OS X) environment variable.

If you are setting the `ZIENA_LICENSE` environment variable to activate your license, then follow the instructions above to create a new environment variable called `ZIENA_LICENSE` and give it the value of the directory containing your Ziena license file (specify the whole path to this directory). For more installation options and general troubleshooting, read the Ziena License Manager User's Manual.

KNITRO for MATLAB

To use KNITRO with MATLAB, you may need to add the KNITRO/MATLAB interface files to your MATLAB path. Assuming the default installation folders were used and the KNITRODIR environment variable contains the path to the KNITRO installation directory, the MATLAB path can be updated with the following commands in MATLAB:

```
> addpath(strcat(getenv('KNITRODIR'),'/knitromatlab'));
> savepath();
```

Alternatively, if the environment variable is not set properly, you can update the MATLAB path by calling addpath() with the full path to the KNITRO/MATLAB interface files, such as:

```
> addpath('/home/user/knitro-9.1.x/knitromatlab');
> savepath();
```

1.4 Troubleshooting

Most issues are linked with either the calling program (such as AMPL or MATLAB) not finding the KNITRO binaries, or with KNITRO not finding the license file. These are discussed first.

1.4.1 License and PATH issues

Below is a list of steps to take if you have difficulties installing KNITRO.

- Create an environment variable `ZIENA_LICENSE_DEBUG` and set it to 1. This will enable some debug output printing that will indicate where the license manager is looking for a license file. See Section 4.1 of the Ziena License Manager User's Manual for more details on how to set the `ZIENA_LICENSE_DEBUG` environment variable and generate debugging information.
- Ensure that the user has the correct permissions for read access to all libraries and to the license file.
- Ensure that the program calling KNITRO is 32-bit (or 64-bit) when KNITRO is 32-bit (or 64-bit). As an example, you cannot use KNITRO 32-bit with MATLAB 64-bit or vice versa. This applies to the Java Virtual Machine and Python as well.
- On Windows, make sure that you are setting *system* environment variables rather than *user* environment variables, when setting environment variables for KNITRO (or, if using user environment variables, that the correct user is logged in).
- KNITRO has multiple options for installing license files. If the procedure you are trying is not working, please try an alternative procedure.
- If you have multiple KNITRO executable files or libraries of the same name on your computer, make sure that the one being used is really the one you intend to use (by making sure it appears first in the definition of the appropriate environment variable).

Please also refer to the Ziena License Manager User's Manual provided with your distribution for additional installation and troubleshooting information.

1.4.2 MATLAB issues

Below are some troubleshooting tips specific to the KNITRO/MATLAB interface.

- Ensure that the MATLAB installation calling KNITRO is 32-bit (or 64-bit) when KNITRO is 32-bit (or 64-bit). You cannot use KNITRO 32-bit with MATLAB 64-bit or vice versa.
- Make sure the KNITRO/MATLAB interface files `knitromatlab_mex.mex*`, `knitromatlab.m`, `knitromatlab_mip.m`, etc., are located in a folder/directory where they can be found by your MATLAB session. See *Installation* for more information on adding the KNITRO/MATLAB interface files to your MATLAB path.

Symbolic links, on systems that support them, are an alternative to copying / renaming the file.

1.4.3 Python interface issues

If you are using the Python interface on a Linux or Unix platform, you may need to use a Python distribution that has been compiled with the **-fopenmp** flag of the **gcc/g++** compiler in order to use the *standard* KNITRO libraries. Otherwise, you should use the *sequential* KNITRO libraries. See *Linux and Mac OS X compatibility issues* for more information.

1.4.4 Issues with LD_LIBRARY_PATH on Ubuntu

In Ubuntu, setting LD_LIBRARY_PATH directly was reported to fail; using **ldconfig** solved the problem as follows:

- Go to /etc/ld.so.conf.d/ directory;
- Create a new configuration file in this directory;
- Set all your environment variables in this file and save it;
- Execute **sudo ldconfig -v** at the prompt.

1.4.5 Loading external third party dynamic libraries

Some user options instruct KNITRO to load dynamic libraries at runtime. This will not work unless the executable can find the desired library using the operating system's load path. Usually this is done by appending the path to the directory that contains the library to an environment variable. For example, suppose the library to be loaded is in the KNITRO lib directory. The instructions below will correctly modify the load path.

- On Windows, type (assuming KNITRO 9.1.0 is installed at its default location):

  ```
  set PATH=%PATH%;C:\Program Files\Ziena\KNITRO-9.1.0-z\lib
  ```

- On Mac OS X, type (assuming KNITRO 9.1.0 is installed at /tmp):

  ```
  export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:/tmp/KNITRO-9.1.0-z/lib
  ```

- If you run a Unix bash shell, then type (assuming KNITRO 9.1.0 is installed at /tmp):

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/tmp/KNITRO-9.1.0-z/lib
  ```

- If you run a Unix csh or tcsh shell, then type (assuming KNITRO 9.1.0 is installed at /tmp):

  ```
  setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/tmp/KNITRO-9.1.0-z/lib
  ```

1.4.6 Linux and Mac OS X compatibility issues

Linux platforms sometimes generate link errors when building the programs in examples/C. Simply type "gmake" and see if the build is successful. You may see a long list of link errors similar to the following:

```
../lib/libknitro.a(.text+0x28808): In function 'ktr_xeb4':
: undefined reference to 'std::__default_alloc_template<true, 0>::deallo
cate(void*, unsigned int)'
../lib/libknitro.a(.text+0x28837): In function 'ktr_xeb4':
: undefined reference to 'std::__default_alloc_template<true, 0>::deallo
cate(void*, unsigned int)'
../lib/libknitro.a(.text+0x290b0): more undefined references to 'std::__
default_alloc_template<true, 0>::deallocate(void*, unsigned int)' follow
../lib/libknitro.a(.text+0x2a0ff): In function 'ktr_x1150':
: undefined reference to 'std::basic_string<char, std::char_traits<char>
, std::allocator<char> >::_S_empty_rep_storage'
../lib/libknitro.a(.text+0x2a283): In function 'ktr_x1150':
: undefined reference to 'std::__default_alloc_template<true, 0>::deallo
cate(void*, unsigned int)'
```

This indicates an incompatibility between the libstdc++ library on your Linux distribution and the library that KNITRO was built with. The incompatibilities may be caused by name-mangling differences between versions of the **gcc/g++** compiler, and by differences in the Application Binary Interface of the two Linux distributions. The best fix is for

Ziena to rebuild the KNITRO binaries on the same Linux distribution of your target machine (matching the Linux brand and release, and the **gcc/g++** compiler versions).

Other link errors may be seen on 64-bit Linux and Mac OS X platforms related to undefined references to "omp" or "pthread" symbols. For example, the link errors may look something like

```
undefined reference to 'pthread_setaffinity_np@GLIBC_2.3.4'
```

on Linux, or

```
Undefined symbols:
      "_GOMP_parallel_start", referenced from:
```

on Mac OS X. This implies either that the dynamic libraries needed for OpenMP (usually provided in system directories, or in the KNITRO `lib` directory for the Mac OS X distribution) are not being found, or that the version of **gcc/g++** used for linking is not compatible with the OpenMP features used in the *standard* KNITRO 9.1 libraries. To solve this issue, be sure that the `LD_LIBRARY_PATH` (`DYLD_LIBRARY_PATH` on Mac OS X) environment variable includes the KNITRO `lib` directory, or try linking against the *sequential* versions of the KNITRO libraries provided with your platform distribution on Linux. See the `README` file provided in the KNITRO `lib` directory for more information

1.4.7 Windows compatibility issues

Using the "dll" version of the KNITRO library on Windows (i.e. linking against `knitro910.lib`) is recommended and should be compatible across multiple versions of the Microsoft Visual C++ (MSVC) compiler. A static KNITRO library named `knitro910_objlib.a` is also provided. However, please be aware that this version will generally only be compatible with the same version of MSVC used to create it.

1.5 Release notes

What's new in release 9.1?

- KNITRO 9.1 introduces a new Sequential Quadratic Programming (SQP) algorithm for continuous problems. This new SQP algorithm is primarily designed for small problems, where the computational cost is dominated by function/derivative evaluations. It will often be the most efficient algorithm for such problems. See section *Algorithms* for more details.
- KNITRO 9.1 offers many enhancements to the MATLAB interface including: 1) the ability to perform parallel finite-differences; 2) the ability to perform a derivative check; 3) the ability to provide a newpoint/OutputFcn callback function; 4) a specialized interface for least-squares problems; and 5) the ability to specify initial Lagrange multipliers. See section *KNITRO / MATLAB reference* for more details.
- KNITRO 9.1 offers an efficient procedure for trying to refine the barrier solution to obtain a more precise barrier solution via the new `bar_refinement` user option.
- KNITRO 9.1 offers simplified user options for performing a derivative check (rather than performing the derivative check through API function calls). See the new user options `derivcheck`, `derivcheck_tol` and `derivcheck_type`.
- KNITRO 9.1 offers improved handling and performance on models with range constraints.
- KNITRO 9.1 offers a new option `ms_deterministic` that allows for deterministic parallel, multi-start performance when `ms_terminate` =0.
- KNITRO 9.1 allows one to print and retrieve final solution information and constraint values for mixed-integer problems (MIP).

- KNITRO 9.1 offers new API function calls for obtaining information about user options.
- KNITRO 9.1 splits the termination codes for limits (e.g. iteration and time limits) into feasible and infeasible cases. See section *Return codes* for more details.
- KNITRO 9.1 offers overall speed and robustness improvements on general NLP models and mixed integer NLP models.

Bug Fixes

- Fixed bug in the KNITRO-MATLAB interface so that the `hessian_no_f` user option is set to 0 (i.e. turned off). This bug could have adversely affected performance when there was a user-supplied hessian when using KNITRO with MATLAB in KNITRO 9.0. It is strongly recommended to upgrade to KNITRO 9.1 in this case.
- Fixed bug that caused KNITRO to terminate with a bad return code sometimes when there was an evaluation error during a finite-difference derivative computation.

CHAPTER

TWO

USER GUIDE

In this second chapter, we will take a look at a few examples that are designed to touch on the most important features of KNITRO. It is *not* meant to be an extensive reference (see *Reference manual* for that matter) but, rather, to walk you through solving your first nonlinear optimization problems with KNITRO thanks to a few simple and illustrative examples.

2.1 Getting started

KNITRO can take its input from many different calling programs and programming languages, with various levels of abstraction. There are essentially three ways to interact with KNITRO (in addition, specific interfaces for Microsoft Excel and Labview are available):

- via a modeling language like AMPL, AIMMS, GAMS, or MPL;
- via a numerical computing environment like MATLAB;
- via a programming language such as C/C++, Java, Fortran or Python.

The first two methods are usually simpler, and the first has the advantage of providing derivatives "for free" since modeling languages compute derivatives behind the scene (see Section *Derivatives*). Calling from a programming language adds some complexity but offers a very fine control over the solver's behaviour.

This section provides a hands-on example for each method, using AMPL, MATLAB and C++.

Note: KNITRO's behaviour can be controlled by *user parameters*. Depending on the interface used, user parameters will be defined by their text name such as `alg` (this would be the case in AMPL) or by programming language identifiers such as `KTR_PARAM_ALG` (that would be the case in C/C++).

2.1.1 Getting started with AMPL

AMPL overview

AMPL is a popular modeling language for optimization that allows users to represent their optimization problems in a user-friendly, readable, intuitive format. This makes the job of formulating and modeling a problem much simpler. For a description of AMPL, visit the AMPL web site at:

http://www.ampl.com/

We assume in the following that the user has successfully installed AMPL. The KNITRO/AMPL executable file **knitroampl** must be in the current directory where AMPL is started, or in a directory included in the PATH environment variable.

Inside of AMPL, to invoke the KNITRO solver type:

```
ampl: option solver knitroampl;
```

at the prompt. From then on, every time a *solve* command will be issued in AMPL, the KNITRO solver will be called.

Example AMPL model

This section provides an example AMPL model and AMPL session that calls KNITRO to solve the problem. The AMPL model is provided with KNITRO in a file called `testproblem.mod`, which is shown below.

```
# Example problem formulated as an AMPL model used
# to demonstate using KNITRO with AMPL.
# The problem has two local solutions:
#   the point (0,0,8) with objective 936.0, and
#   the point (7,0,0) with objective 951.0

# Define variables and enforce that they be non-negative.
var x{j in 1..3} >= 0;

# Objective function to be minimized.
minimize obj:
        1000 - x[1]^2 - 2*x[2]^2 - x[3]^2 - x[1]*x[2] - x[1]*x[3];

# Equality constraint.
s.t. c1: 8*x[1] + 14*x[2] + 7*x[3] - 56 = 0;

# Inequality constraint.
s.t. c2: x[1]^2 + x[2]^2 + x[3]^2 -25 >= 0;

data;

# Define initial point.
let x[1] := 2;
let x[2] := 2;
let x[3] := 2;
```

The above example displays the ease with which an optimization problem can be expressed in the AMPL modeling language.

Running the solver

Below is the AMPL session used to solve this problem with KNITRO.

```
1 ampl: reset;
2 ampl: option solver knitroampl;
3 ampl: option knitro_options "alg=2 bar_maxcrossit=2 outlev=1";
4 ampl: model testproblem.mod;
5 ampl: solve;
```

The options passed to KNITRO on line 3 above mean "use the Interior/CG algorithm" (`alg=2`), "refine the solution using the Active Set algorithm" (`bar_maxcrossit=2`) and "limit the output from KNITRO" (`outlev=1`). The meaning of KNITRO options and how to tweak them will be explained later, the point here is only to show how easy it is to control KNITRO's behavior in AMPL by using *knitro_options*. Upon receiving the `solve` command, AMPL produces the following output.

```
KNITRO 9.1.0: alg=2
bar_maxcrossit=2
outlev=1

=====================================
       Commercial Ziena License
             KNITRO 9.1.0
          Ziena Optimization
=====================================

KNITRO presolve eliminated 0 variables and 0 constraints.

algorithm:            2
bar_maxcrossit:       2
hessian_no_f:         1
outlev:               1
par_concurrent_evals: 0
KNITRO changing bar_murule from AUTO to 1.
KNITRO changing bar_initpt from AUTO to 3.
KNITRO changing bar_penaltyrule from AUTO to 2.
KNITRO changing bar_penaltycons from AUTO to 1.
KNITRO changing bar_switchrule from AUTO to 2.
KNITRO changing linsolver from AUTO to 4.

Problem Characteristics
-----------------------
Objective goal:  Minimize
Number of variables:                     3
    bounded below:                       3
    bounded above:                       0
    bounded below and above:             0
    fixed:                               0
    free:                                0
Number of constraints:                   2
    linear equalities:                   1
    nonlinear equalities:                0
    linear inequalities:                 0
    nonlinear inequalities:              1
    range:                               0
Number of nonzeros in Jacobian:          6
Number of nonzeros in Hessian:           5

EXIT: Locally optimal solution found.

Final Statistics
----------------
Final objective value               =   9.36000000000000e+002
Final feasibility error (abs / rel) =   0.00e+000 / 0.00e+000
Final optimality error  (abs / rel) =   0.00e+000 / 0.00e+000
# of iterations                    =          7
# of CG iterations                 =          8
# of function evaluations          =          8
# of gradient evaluations          =          8
# of Hessian evaluations           =          7
Total program time (secs)           =       0.039 (     0.031 CPU time)
Time spent in evaluations (secs)    =       0.000

===============================================================================
```

```
59
60 KNITRO 9.1.0: Locally optimal solution.
61 objective 936; feasibility error 0
62 7 iterations; 8 function evaluations
63 ampl:
```

The output from KNITRO tells us that the algorithm terminated successfully ("Exit: Locally optimal solution found." on line 43), that the objective value at the optimum found is about 936.0 (line 47) and that it took KNITRO about 30 milliseconds to solve the problem (line 55). More information is printed, which you do not need to understand for now; the precise meaning of the KNITRO output will be discussed in *Obtaining information*.

After solving an optimization problem, one is typically interested in information about the solution (other than simply the objective value, which we already found by looking at the KNITRO log). For instance, one may be interested in printing the value of the variables x; the AMPL *display* command does just that:

```
ampl: display x;
x [*] :=
1  0
2  0
3  8
;
```

More information about AMPL display commands can be found in the AMPL manual.

Additional examples

More examples of using AMPL for nonlinear programming can be found in Chapter 18 of the AMPL book, see the *Bibliography*.

2.1.2 Getting started with MATLAB

The KNITRO interface for MATLAB, called *knitromatlab*, is provided with your KNITRO distribution. To test whether your installation is correct, type in the expression:

```
[x fval] = knitromatlab(@(x)cos(x),1)
```

at the MATLAB command prompt. If your installation was successful, *knitromatlab* returns:

```
x = 3.1416, fval = -1.
```

If you do not get this output but an error stating that *knitromatlab* was not found, it probably means that the path has not been added to MATLAB. If KNITRO is found and called but returns an error, it probably means that no license was found. In any of these situations, please see *Troubleshooting*.

The *knitromatlab* interface

The KNITRO/MATLAB interface function is very similar to MATLAB's built-in *fmincon* function; the most elaborate form is:

```
[x,fval,exitflag,output,lambda,grad,hessian] = ...
        knitromatlab(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon, ...
        extendedFeatures,options,KNITROOptions)
```

but the simplest function call reduces to:

```
x = knitromatlab(fun,x0)
```

The *knitromatlab* function was designed to provide a similar user experience to MATLAB's *fmincon* optimization function.

The *ktrlink* interface previously provided with the MATLAB Optimization Toolbox is no longer supported. See the reference manual on using *knitrolink* instead.

First MATLAB example

Let's consider the same example as before (in section *Getting started with AMPL*), converted into MATLAB.

```
% objective to minimize
obj = @(x) 1000 - x(1)^2 - 2*x(2)^2 - x(3)^2 - x(1)*x(2) - x(1)*x(3);

% No nonlinear equality constraints.
ceq  = [];

% Specify nonlinear inequality constraint to be nonnegative
c2 =  @(x) x(1)^2 + x(2)^2 + x(3)^2 - 25;

% "nlcon" should return [c, ceq] with c(x) <= 0 and ceq(x) = 0
% so we need to negate the inequality constraint above
nlcon = @(x)deal(-c2(x), ceq);

% Initial point
x0  = [2; 2; 2];

% No linear inequality contraint ("A*x <= b")
A = [];
b = [];

% Since the equality constraint "c1" is linear, specify it here  ("Aeq*x = beq")
Aeq = [8 14 7];
beq = [56];

% lower and upper bounds
lb = zeros(3,1);
ub = [];

% solver call
x = knitromatlab(obj, x0, A, b, Aeq, beq, lb, ub, nlcon);
```

Saving this code in a file *example.m* in the current folder and issuing *example* at the MATLAB prompt produces the following output.

```
=====================================
       Commercial ZIENA LICENSE
            KNITRO 9.1.0
         Ziena Optimization
=====================================

KNITRO presolve eliminated 0 variables and 0 constraints.

algorithm:            1
gradopt:              4
hessopt:              2
honorbnds:            1
```

```
maxit:                10000
outlev:               1
par_concurrent_evals: 0
KNITRO changing bar_murule from AUTO to 4.
KNITRO changing bar_initpt from AUTO to 3.
KNITRO changing bar_penaltyrule from AUTO to 2.
KNITRO changing bar_penaltycons from AUTO to 1.
KNITRO changing bar_switchrule from AUTO to 2.
KNITRO changing linsolver from AUTO to 2.

Problem Characteristics
-----------------------
Objective goal:  Minimize
Number of variables:                     3
    bounded below:                       3
    bounded above:                       0
    bounded below and above:             0
    fixed:                               0
    free:                                0
Number of constraints:                   2
    linear equalities:                   1
    nonlinear equalities:                0
    linear inequalities:                 0
    nonlinear inequalities:              1
    range:                               0
Number of nonzeros in Jacobian:          6
Number of nonzeros in Hessian:           6

EXIT: Locally optimal solution found.

Final Statistics
----------------
Final objective value               =   9.36000000000339e+02
Final feasibility error (abs / rel) =   0.00e+00 / 0.00e+00
Final optimality error  (abs / rel) =   4.10e-07 / 2.56e-08
# of iterations                     =          9
# of CG iterations                  =          0
# of function evaluations           =         40
# of gradient evaluations           =          0
Total program time (secs)           =       0.00406 (     0.004 CPU time)
Time spent in evaluations (secs)    =       0.00189

===============================================================================


===============================================================================
```

The objective function value is the same (about 936.0) as in the AMPL example. However, even though we solved the same problem, things went quite differently behind the scenes in these two examples; as we will see in Section *Derivatives*, AMPL provides derivatives to KNITRO automatically, whereas in MATLAB the user must do it manually. Since we did not provide these derivatives, KNITRO had to approximate them. Note that with AMPL, only 8 function evaluations took place, whereas there were 40 in the MATLAB example (extra function evaluations were needed to approximate the first derivatives). On a large problem, this could have made a very significant difference in performance.

Additional examples

More examples are provided in the `knitromatlab` directory of the KNITRO distribution.

2.1.3 Getting started with the callable library

KNITRO is written in C and C++, with a well-documented application programming interface (API) defined in the file `knitro.h` provided in the installation under the `include` directory.

The KNITRO callable library is typically used to solve an optimization problem through a sequence of four basic function calls:

- `KTR_new()`: create a new KNITRO solver context pointer, allocating resources.
- `KTR_init_problem()`, for continuous problems, or `KTR_mip_init_problem()` for mixed-integer problems: load the problem definition into the KNITRO solver.
- `KTR_solve()` or `KTR_mip_solve()`: solve the problem.
- `KTR_free()`: delete the KNITRO context pointer, releasing allocated resources.

The example below shows how to use these function calls.

First example

Again, let us consider the toy example that we already solved twice, using AMPL and MATLAB. The C++ equivalent is the following.

```
#include <stdio.h>
#include <stdlib.h>
#include "knitro.h"


/* callback function that evaluates the objective
   and constraints */
int  callback (const int evalRequestCode,
        const int n,
        const int m,
        const int nnzJ,
        const int nnzH,
        const double * const x,
        const double * const lambda,
              double * const obj,
              double * const c,
              double * const objGrad,
              double * const jac,
              double * const hessian,
              double * const hessVector,
              void * userParams) {

        if (evalRequestCode == KTR_RC_EVALFC) {
                /* objective function */
                *obj    = 1000 - x[0]*x[0] - 2*x[1]*x[1] -
                          x[2]*x[2] - x[0]*x[1] - x[0]*x[2];

                /* constraints */
                c[0]    = 8*x[0] + 14*x[1] + 7*x[2] - 56;
                c[1]    = x[0]*x[0] + x[1]*x[1] + x[2]*x[2] -25;
```

```
31
32                  return(0);
33          }
34          else {
35                  printf ("Wrong evalRequestCode in callback function.\n");
36                  return(-1);
37          }
38  }
39
40  /* main */
41  int  main (int  argc, char  *argv[]) {
42          int  nStatus;
43
44          /* variables that are passed to KNITRO */
45          KTR_context *kc;
46          int n, m, nnzJ, nnzH, objGoal, objType;
47          int *cType;
48          int *jacIndexVars, *jacIndexCons;
49          double obj, *x, *lambda;
50          double *xLoBnds, *xUpBnds, *xInitial, *cLoBnds, *cUpBnds;
51          int i, j, k; // convenience variables
52
53          /*problem size and mem allocation */
54          n = 3;
55          m = 2;
56          nnzJ = n*m;
57          nnzH = 0;
58          x = (double *) malloc (n * sizeof(double));
59          lambda = (double *) malloc ((m+n) * sizeof(double));
60
61          xLoBnds      = (double *) malloc (n * sizeof(double));
62          xUpBnds      = (double *) malloc (n * sizeof(double));
63          xInitial     = (double *) malloc (n * sizeof(double));
64          cType        = (int    *) malloc (m * sizeof(int));
65          cLoBnds      = (double *) malloc (m * sizeof(double));
66          cUpBnds      = (double *) malloc (m * sizeof(double));
67          jacIndexVars = (int    *) malloc (nnzJ * sizeof(int));
68          jacIndexCons = (int    *) malloc (nnzJ * sizeof(int));
69
70          /* objective type */
71          objType = KTR_OBJTYPE_GENERAL;
72          objGoal = KTR_OBJGOAL_MINIMIZE;
73
74          /* bounds and constraints type */
75          for (i = 0; i < n; i++) {
76                  xLoBnds[i] = 0.0;
77                  xUpBnds[i] = KTR_INFBOUND;
78          }
79          for (j = 0; j < m; j++) {
80                  cType[j] = KTR_CONTYPE_GENERAL;
81                  cLoBnds[j] = 0.0;
82                  cUpBnds[j] = (j == 0 ? 0.0 : KTR_INFBOUND);
83          }
84
85          /* initial point */
86          for (i = 0; i < n; i++)
87                  xInitial[i] = 2.0;
88
```

```
89          /* sparsity pattern (here, of a full matrix) */
90          k = 0;
91          for (i = 0; i < n; i++)
92                  for (j = 0; j < m; j++) {
93                          jacIndexCons[k] = j;
94                          jacIndexVars[k] = i;
95                          k++;
96                  }
97
98          /* create a KNITRO instance */
99          kc = KTR_new();
100         if (kc == NULL)
101                 exit( -1 ); // probably a license issue
102
103         /* set options: automatic gradient and hessian matrix */
104         if (KTR_set_int_param_by_name (kc, "gradopt", KTR_GRADOPT_FORWARD) != 0)
105                 exit( -1 );
106         if (KTR_set_int_param_by_name (kc, "hessopt", KTR_HESSOPT_BFGS) != 0)
107                 exit( -1 );
108         if (KTR_set_int_param_by_name (kc, "outlev", 1) != 0)
109                 exit( -1 );
110
111         /* register the callback function */
112         if (KTR_set_func_callback (kc, &callback) != 0)
113                 exit( -1 );
114
115         /* pass the problem definition to KNITRO */
116         nStatus = KTR_init_problem (kc, n, objGoal, objType,
117                         xLoBnds, xUpBnds,
118                         m, cType, cLoBnds, cUpBnds,
119                         nnzJ, jacIndexVars, jacIndexCons,
120                         nnzH, NULL, NULL, xInitial, NULL);
121
122         /* free memory (KNITRO maintains its own copy) */
123         free (xLoBnds);
124         free (xUpBnds);
125         free (xInitial);
126         free (cType);
127         free (cLoBnds);
128         free (cUpBnds);
129         free (jacIndexVars);
130         free (jacIndexCons);
131
132         /* solver call */
133         nStatus = KTR_solve (kc, x, lambda, 0, &obj,
134                 NULL, NULL, NULL, NULL, NULL, NULL);
135
136         if (nStatus != 0)
137                 printf ("\nKNITRO failed to solve the problem, final status = %d\n",
138                         nStatus);
139         else
140                 printf ("\nKNITRO successful, objective is = %e\n", obj);
141
142         /* delete the KNITRO instance and primal/dual solution */
143         KTR_free (&kc);
144         free (x);
145         free (lambda);
146
```

```
147         return( 0 );
148 }
```

Note that the AMPL equivalent is both much shorter (only a few lines of code) and more efficient in this case since, as we mentioned before, AMPL provides automatic derivatives to KNITRO behind the scenes. To achieve the same efficiency in C, we would have to compute the derivatives manually, code them in C and input them to KNITRO using a callback similar to the one we used to define the objective and constraints. We will show how to do this in the chapter on *Derivatives*. However the callable library has the advantage of greater control (for instance, on memory usage) and allows one to embed KNITRO in a native application seamlessly.

The above example can be compiled and linked against the KNITRO callable library with a standard C compiler. Its output is the following.

```
1  =====================================
2         Commercial Ziena License
3               KNITRO 9.1.0
4            Ziena Optimization
5  =====================================
6
7  KNITRO presolve eliminated 0 variables and 0 constraints.
8
9  gradopt:              2
10 hessopt:              2
11 outlev:               1
12 presolve:             0
13 KNITRO changing algorithm from AUTO to 1.
14 KNITRO changing bar_murule from AUTO to 4.
15 KNITRO changing bar_initpt from AUTO to 3.
16 KNITRO changing bar_penaltyrule from AUTO to 2.
17 KNITRO changing bar_penaltycons from AUTO to 1.
18 KNITRO changing bar_switchrule from AUTO to 2.
19 KNITRO changing linsolver from AUTO to 2.
20 KNITRO performing finite-difference gradient computation with 1 thread.
21
22 Problem Characteristics
23 ----------------------
24 Objective goal:  Minimize
25 Number of variables:                     3
26     bounded below:                      3
27     bounded above:                      0
28     bounded below and above:            0
29     fixed:                              0
30     free:                               0
31 Number of constraints:                  2
32     linear equalities:                  0
33     nonlinear equalities:               1
34     linear inequalities:                0
35     nonlinear inequalities:             1
36     range:                              0
37 Number of nonzeros in Jacobian:         6
38 Number of nonzeros in Hessian:          6
39
40 EXIT: Locally optimal solution found.
41
42 Final Statistics
43 ----------------
44 Final objective value               =   9.36000000020000e+02
45 Final feasibility error (abs / rel) =   7.11e-15 / 5.47e-16
46 Final optimality error  (abs / rel) =   1.36e-07 / 8.51e-09
```

```
47 # of iterations                   =          9
48 # of CG iterations                =          0
49 # of function evaluations         =         40
50 # of gradient evaluations         =          0
51 Total program time (secs)         =       0.00220 (     0.002 CPU time)
52 Time spent in evaluations (secs)  =       0.00000
53
54 ===============================================================================
55
56
57 KNITRO successful, objective is = 9.360000e+02
```

Again, the solution value is the same (about 936.0), and the details of the log are slightly different. Note for instance that the log mentions "linear equalities: 0" at line 32 although the first constraint is indeed linear. KNITRO (which only knows of the objective and constraint though a callback function) cannot detect this: we should have told the solver of the constraint linearity at line 80 of the C code above, by setting the constraint type to `KTR_CONTYPE_LINEAR` instead of `KTR_CONTYPE_GENERAL`. If you go back to the AMPL example (*Getting started with AMPL*), you will see that AMPL (which has an algebraic view of the optimization problem) detected that the first constraint was linear and passed this information to KNITRO, whose log mentioned "linear equalities: 1". This shows another advantage of modeling languages over other interfaces: to some extent, they automatically detect the problem structure and inform the solver (modeling languages are actually even able to *simplify* the problem to some extent, by applying some *presolve* operations; see *AMPL presolve*).

Further information

Another chapter of this documentation will be dedicated to the callable library (*Callbacks*), more specifically to the communication mode between the solver and the user-supplied optimization problem.

The reference manual (*Callable library reference*) also contains a comprehensive documentation of the KNITRO callable library.

Finally, the file `knitro.h` contains many useful comments and can be used as an ultimate reference.

Additional examples

More C/C++ examples using the callable library are provided in the `examples/C` and `examples/C++` directories of the KNITRO distribution.

2.2 Setting options

KNITRO offers a number of user options for modifying behavior of the solver. Each option takes a value that may be an integer, double precision number, or character string. Options are usually identified by a string name (for example, `algorithm`), but programmatic interfaces also identify options by an integer value associated with a C language macro defined in the file `knitro.h`. (for example, `KTR_PARAM_ALG`). Most user options can be specified with either a numeric value or a string value.

Note: The naming convention is that user options beginning with "bar_" apply only to the barrier/interior-point algorithms; options beginning with "mip_" apply only to the mixed integer programming (MIP) solvers; options beginning with "ms_" apply only to the multi-start procedure; and options specific to the multi-algorithm procedure begin with "ma_". Options specific to parallel features begin with "par_".

2.2.1 Setting KNITRO options within AMPL

We have seen how to specify user options, for example:

```
ampl: option knitro_options "alg=2 bar_maxcrossit=2 outlev=1";
```

A complete list of KNITRO options that are available in AMPL can be shown by typing:

```
knitroampl -=
```

The output produced by this command, along with a description of each option, is provided in Section *KNITRO / AMPL reference*.

Note: When specifying multiple options, all options must be set with one `knitro_options` command as shown in the example above. If multiple `knitro_options` commands are specified in an AMPL session, only the last one will be read.

When running *knitroampl* directly with an AMPL file, user options can be set on the command line as follows:

```
knitroampl testproblem.nl maxit=100 opttol=1.0e-5
```

2.2.2 Setting KNITRO options with MATLAB

There are two ways *knitromatlab* can read user options: either using the *fmincon* format (explained in the MATLAB documentation), or using the *KNITRO options file* (explained below). If both types of options are used, KNITRO options override *fmincon* format options.

The KNITRO option file is a simple text file that contains, on each line, the name of a KNITRO option and its value. For instance, the content of the file could be:

```
algorithm    auto
bar_directinterval  10
bar_feasible no
```

Assuming that the KNITRO options file is named `knitro.opt` and is stored in the current directory, and that the fmincon-format options structure is named *KnitroOptions*, the call to *knitromatlab* would be:

```
[x fval] = ...
    knitromatlab(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,extendedFeatures,KnitroOptions, ...
                 'knitro.opt')
```

The KNITRO options file is a general mechanism to pass options to KNITRO. It can also be used with the callable library interface, but is most useful with the KNITRO/MATLAB interface for which it is the only way to set many of the available options.

2.2.3 Setting KNITRO options with the callable library

The functions for setting user options have the form:

```
int KTR_set_int_param (KTR_context *kc, int param_id, int value)
```

for setting integer valued parameters, or:

```
int KTR_set_double_param (KTR_context *kc, int param_id, double value)
```

for setting double precision valued parameters.

For example, to specify the *Interior/CG* algorithm and a tight optimality stop tolerance:

```
status = KTR_set_int_param (kc, KTR_PARAM_ALG, KTR_ALG_BAR_CG);
status = KTR_set_double_param (kc, KTR_PARAM_OPTTOL, 1.0e-8);
```

Refer to the Callable Library Reference Manual (*Changing and reading solver parameters*) for a comprehensive list.

2.2.4 The KNITRO options file

The KNITRO options file allows the user to easily change user options by editing a text file, instead of modifying application code.

Options are set by specifying a keyword and a corresponding value on a line in the options file. Lines that begin with a " #" character are treated as comments and blank lines are ignored. For example, to set the maximum allowable number of iterations to 500, you could create the following options file:

```
# KNITRO Options file
maxit             500
```

MATLAB users may simply pass the name of the KNITRO options file to *knitromatlab* as demonstrated in *Getting started with MATLAB*. When using the callable library, the options file is read into KNITRO by calling the following function before invoking `KTR_init_problem()` or `KTR_mip_init_problem()`:

```
int KTR_load_param_file (KTR_context *kc, char const *filename)
```

For example, if the options file is named `myoptions.opt`:

```
status = KTR_load_param_file (kc, "myoptions.opt");
```

The full set of options used by KNITRO in a given solve may be written to a text file through the function call:

```
int KTR_save_param_file (KTR_context *kc, char const *filename)
```

For example:

```
status = KTR_save_param_file (kc, "knitro.opt");
```

A sample options file `knitro.opt` is provided for convenience and can be found in the `examples/C` directory. Note that this file is only read by application drivers that call `KTR_load_param_file()`, such as `examples/C/callbackExample2.c`.

2.3 Derivatives

Applications should provide partial first derivatives whenever possible, to make KNITRO more efficient and more robust. If first derivatives cannot be supplied, then the application should instruct KNITRO to calculate finite-difference approximations.

First derivatives are represented by the gradient of the objective function and the Jacobian matrix of the constraints. Second derivatives are represented by the Hessian matrix, a linear combination of the second derivatives of the objective function and the constraints.

2.3.1 First derivatives

The default version of KNITRO assumes that the user can provide exact first derivatives to compute the objective function gradient and constraint gradients. It is *highly* recommended that the user provide exact first derivatives if at all possible, since using first derivative approximations may seriously degrade the performance of the code and the likelihood of converging to a solution. However, if this is not possible the following first derivative approximation options may be used.

- *Forward finite-differences* This option uses a forward finite-difference approximation of the objective and constraint gradients. The cost of computing this approximation is n function evaluations (where n is the number of variables). The option is invoked by choosing user option `gradopt` = 2.
- *Centered finite-differences* This option uses a centered finite-difference approximation of the objective and constraint gradients. The cost of computing this approximation is $2n$ function evaluations (where n is the number of variables). The option is invoked by choosing user option `gradopt` = 3. The centered finite-difference approximation is often more accurate than the forward finite-difference approximation; however, it is more expensive to compute if the cost of evaluating a function is high.

Although these finite-differences approximations should be avoided in general, they are useful to track errors: whenever the derivatives are provided by the user, it is useful to check that the differentiation (and the subsequent implementation of the derivatives) is correct. Indeed, providing derivatives that are not coherent with the function values is one of the most common errors when solving a nonlinear program. This check can be done automatically by comparing finite-differences approximations with user-provided derivatives. This is explained below (*Checking derivatives*).

2.3.2 Second derivatives

The default version of KNITRO assumes that the application can provide exact second derivatives to compute the Hessian of the Lagrangian function. If the application is able to do so and the cost of computing the second derivatives is not overly expensive, it is highly recommended to provide exact second derivatives. However, KNITRO also offers other options which are described in detail below.

- *(Dense) Quasi-Newton BFGS*:

  The quasi-Newton BFGS option uses gradient information to compute a symmetric, positive-definite approximation to the Hessian matrix. Typically this method requires more iterations to converge than the exact Hessian version. However, since it is only computing gradients rather than Hessians, this approach may be more efficient in some cases. This option stores a *dense* quasi-Newton Hessian approximation so it is only recommended for small to medium problems ($n < 1000$). The quasi-Newton BFGS option is chosen by setting user option `hessopt` = 2.

- *(Dense) Quasi-Newton SR1*:

  As with the BFGS approach, the quasi-Newton SR1 approach builds an approximate Hessian using gradient information. However, unlike the BFGS approximation, the SR1 Hessian approximation is not restricted to be positive-definite. Therefore the quasi-Newton SR1 approximation may be a better approach, compared to the BFGS method, if there is a lot of negative curvature in the problem since it may be able to maintain a better approximation to the true Hessian in this case. The quasi-Newton SR1 approximation maintains a *dense* Hessian approximation and so is only recommended for small to medium problems ($n < 1000$). The quasi-Newton SR1 option is chosen by setting user option `hessopt` = 3.

- *Finite-difference Hessian-vector product option*:

  If the problem is large and gradient evaluations are not a dominant cost, then KNITRO can internally compute Hessian-vector products using finite-differences. Each Hessian-vector product in this case requires one additional gradient evaluation. This option is chosen by setting user option `hessopt` = 4. The option is only recommended if the exact gradients are provided.

Note: This option may not be used when `algorithm` = 1 or 4 since the Interior/Direct and SQP algorithms need the full expression of the Hessian matrix (Hessian-vector products are not sufficient).

- *Exact Hessian-vector products*:

  In some cases the application may prefer to provide exact Hessian-vector products, but not the full Hessian (for instance, if the problem has a large, dense Hessian). The application must provide a routine which, given a vector *v* (stored in *hessVector*), computes the Hessian-vector product, *H*v*, and returns the result (again in *hessVector*). This option is chosen by setting user option `hessopt` = 5.

Note: This option may not be used when `algorithm` = 1 or 4 since, as mentioned above, the Interior/Direct and SQP algorithms need the full expression of the Hessian matrix (Hessian-vector products are not sufficient).

- *Limited-memory Quasi-Newton BFGS*:

  The limited-memory quasi-Newton BFGS option is similar to the dense quasi-Newton BFGS option described above. However, it is better suited for large-scale problems since, instead of storing a dense Hessian approximation, it stores only a limited number of gradient vectors used to approximate the Hessian. The number of gradient vectors used to approximate the Hessian is controlled by user option `lmsize`.

  A larger value of `lmsize` may result in a more accurate, but also more expensive, Hessian approximation. A smaller value may give a less accurate, but faster, Hessian approximation. When using the limited memory BFGS approach it is recommended to experiment with different values of this parameter (e.g. between 5 and 15).

  In general, the limited-memory BFGS option requires more iterations to converge than the dense quasi-Newton BFGS approach, but will be much more efficient on large-scale problems. The limited-memory quasi-Newton option is chosen by setting user option `hessopt` = 6.

As with exact first derivatives, exact second derivatives often provide a substantial benefit to KNITRO and it is advised to provide them whenever possible.

2.3.3 Jacobian and Hessian derivative matrices

The Jacobian matrix of the constraints is defined as

$$J(x) = \begin{bmatrix} \nabla c_0(x) & \dots & \nabla c_{m-1}(x) \end{bmatrix}$$

and the Hessian matrix of the Lagrangian is defined as

$$H(x,\lambda) = \sigma \nabla^2 f(x) + \sum_{i=0}^{m-1} \lambda_i \nabla^2 c_i(x)$$

where λ is the vector of Lagrange multipliers (dual variables), and σ is a scalar (either 0 or 1) for the objective component of the Hessian that was introduced in KNITRO 8.0.

Note: For backwards compatibility with older versions of KNITRO, the user can always assume that $\sigma = 1$ if the user option `hessian_no_f=0` (which is the default setting). However, if `hessian_no_f=1`, then KNITRO will provide a status flag to the user when it needs a Hessian evaluation indicating whether the Hessian should be evaluated with $\sigma = 0$ or $\sigma = 1$. The user must then evaluate the Hessian with the proper value of σ based on this status flag. Setting `hessian_no_f=1` and computing the Hessian with the requested value of σ may improve KNITRO efficiency in some cases. Examples of how to do this can be found in the `examples/C` directory.

Example

Assume we want to use KNITRO to solve the following problem:

$$\begin{aligned} \min\ & x_0 + x_1 x_2^3 \\ & \text{subject to:} \\ & \cos(x_0) = 0.5 \\ 3 \leq\ & x_0^2 + x_1^2 \leq 8 \\ & x_0 + x_1 + x_2 \leq 10 \\ & x_0, x_1, x_2 \geq 1. \end{aligned}$$

Rewriting in the KNITRO standard notation, we have

$$\begin{aligned} f(x) &= x_0 + x_1 x_2^3 \\ c_0(x) &= \cos(x_0) \\ c_1(x) &= x_0^2 + x_1^2 \\ c_2(x) &= x_0 + x_1 + x_2. \end{aligned}$$

Computing the Sparse Jacobian Matrix

The gradients (first derivatives) of the objective and constraint functions are given by

$$\nabla f(x) = \begin{bmatrix} 1 \\ x_2^3 \\ 3x_1x_2^2 \end{bmatrix}, \nabla c_0(x) = \begin{bmatrix} -\sin(x_0) \\ 0 \\ 0 \end{bmatrix}, \nabla c_1(x) = \begin{bmatrix} 2x_0 \\ 2x_1 \\ 0 \end{bmatrix}, \nabla c_2(x) = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

The constraint Jacobian matrix $J(x)$ is the matrix whose rows store the (transposed) constraint gradients, i.e.,

$$J(x) = \begin{bmatrix} \nabla c_0(x)^T \\ \nabla c_1(x)^T \\ \nabla c_2(x)^T \end{bmatrix} = \begin{bmatrix} -\sin(x_0) & 0 & 0 \\ 2x_0 & 2x_1 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

The values of $J(x)$ depend on the value of x and change during the solution process. The indices specifying the nonzero elements of this matrix remain constant and are set in `KTR_init_problem()` by the values of *jacIndexCons* and *jacIndexVars*.

Computing the Sparse Hessian Matrix

For the example above, the Hessians (second derivatives) of the objective function is given by

$$\nabla^2 f(x) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 3x_2^2 \\ 0 & 3x_2^2 & 6x_1x_2 \end{bmatrix},$$

and the Hessians of constraints are given by

$$\nabla^2 c_0(x) = \begin{bmatrix} -\cos(x_0) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \nabla^2 c_1(x) = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \nabla^2 c_2(x) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Scaling the objective matrix by σ, and the constraint matrices by their corresponding Lagrange multipliers and summing, we get

$$H(x,\lambda) = \begin{bmatrix} -\lambda_0 \cos(x_0) + 2\lambda_1 & 0 & 0 \\ 0 & 2\lambda_1 & \sigma 3x_2^2 \\ 0 & \sigma 3x_2^2 & \sigma 6x_1x_2 \end{bmatrix}.$$

The values of $H(x,\lambda)$ depend on the value of x and λ (and σ, which is either 0 or 1) and change during the solution process. The indices specifying the nonzero elements of this matrix remain constant and are set in `KTR_init_problem()` by the values of *hessIndexRows* and *hessIndexCols*.

2.3.4 Inputing derivatives

MATLAB users can provide the Jacobian and Hessian matrices in standard MATLAB format, either dense or sparse. See the *fmincon* documentation, http://www.mathworks.com/help/optim/ug/writing-constraints.html#brhkghv-16, for more information. Users of the callable library must provide derivatives to KNITRO in sparse format. In the above example, the number of nonzero elements *nnzJ* in *J(x)* is 6, and these arrays would be specified as follows (here in column-wise order, but the order is arbitrary) using the callable library.

```
jac[0] = -sin(x[0]);  jacIndexCons[0] = 0; jacIndexVars[0] = 0;
jac[1] = 2*x[0];      jacIndexCons[1] = 1; jacIndexVars[1] = 0;
jac[2] = 1;           jacIndexCons[2] = 2; jacIndexVars[2] = 0;
jac[3] = 2*x[1];      jacIndexCons[3] = 1; jacIndexVars[3] = 1;
jac[4] = 1;           jacIndexCons[4] = 2; jacIndexVars[4] = 1;
jac[5] = 1;           jacIndexCons[5] = 2; jacIndexVars[5] = 2;
```

Note: Even if the application does not evaluate derivatives, it must still provide a sparsity pattern for the constraint Jacobian matrix that specifies which partial derivatives are nonzero. KNITRO uses the sparsity pattern to speed up linear algebra computations. If the sparsity pattern is unknown, then the application should specify a fully dense pattern (i.e., assume all partial derivatives are nonzero).

Since the Hessian matrix will always be a symmetric matrix, KNITRO only stores the nonzero elements corresponding to the upper triangular part (including the diagonal). In the example here, the number of nonzero elements in the upper triangular part of the Hessian matrix *nnzH* is 4. The KNITRO array *hess* stores the values of these elements, while the arrays *hessIndexRows* and *hessIndexCols* store the row and column indices respectively. The order in which these nonzero elements is stored is not important. If we store them column-wise, the arrays *hess*, *hessIndexRows* and *hessIndexCols* are as follows:

```
hess[0] = -lambda[0]*cos(x[0]) + 2*lambda[1];
hessIndexRows[0] = 0;
hessIndexCols[0] = 0;

hess[1] = 2*lambda[1];
hessIndexRows[1] = 1;
hessIndexCols[1] = 1;

hess[2] = sigma*3*x[2]*x[2];
hessIndexRows[2] = 1;
hessIndexCols[2] = 2;

hess[3] = sigma*6*x[1]*x[2];
hessIndexRows[3] = 2;
hessIndexCols[3] = 2;
```

Note: In KNITRO, the array *objGrad* stores *all* of the elements of $\nabla f(x)$, while the arrays *jac*, *jacIndexCons*, and *jacIndexVars* store information concerning *only the nonzero* elements of *J(x)*. The array *jac* stores the nonzero values in *J(x)* evaluated at the current solution estimate *x*, *jacIndexCons* stores the constraint function (or row) indices corresponding to these values, and *jacIndexVars* stores the variable (or column) indices. There is no restriction on the order in which these elements are stored; however, it is common to store the nonzero elements of *J(x)* in column-wise fashion.

2.3.5 MATLAB example

Let us modify our example from *Getting started with MATLAB* so that the first derivatives are provided as well. In MATLAB, you only need to provide the derivatives for the nonlinear functions, whereas in the callable library API you need to provide the derivatives for both linear and nonlinear constraints in $J(x)$. In the example below, only the inequality constraint is nonlinear, so we only provide the derivative for this constraint.

```
function firstDer()

        function [f, g] = obj(x)
                f = 1000 - x(1)^2 - 2*x(2)^2 - x(3)^2 - x(1)*x(2) - x(1)*x(3);
                if nargout == 2
                   g = [-2*x(1) - x(2) - x(3); - 4*x(2) - x(1); -2*x(3) - x(1)];
                end
        end

        % nlcon should return [c, ceq, GC, GCeq]
        % with c(x) <= 0 and ceq(x) = 0
        function [c, ceq, GC, GCeq] = nlcon(x)
                c = -(x(1)^2 + x(2)^2 + x(3)^2 - 25);
                ceq = [];
                if nargout==4
                        GC = -([2*x(1); 2*x(2); 2*x(3)]);
                        GCeq = [];
                end
        end

        x0  = [2; 2; 2];
        A = []; b = []; % no linear inequality constraints ("A*x <= b")
        Aeq = [8 14 7]; beq = [56]; % linear equality constraints ("Aeq*x = beq")
        lb = zeros(3,1); ub = []; % lower and upper bounds

        options = optimset('GradObj', 'on', 'GradConstr', 'on');
        knitromatlab(@obj, x0, A, b, Aeq, beq, lb, ub, @nlcon, [], options);

end
```

The only difference with the derivative-free case is that the code that computes the objective function and the constraints also returns the first derivatives along with function values. The output is as follows.

```
=====================================
       Commercial Ziena License
            KNITRO 9.1.0
         Ziena Optimization
=====================================

KNITRO presolve eliminated 0 variables and 0 constraints.

algorithm:            1
```

```
hessopt:              2
honorbnds:            1
maxit:                10000
outlev:               1
KNITRO changing bar_murule from AUTO to 4.
KNITRO changing bar_initpt from AUTO to 3.
KNITRO changing bar_penaltyrule from AUTO to 2.
KNITRO changing bar_penaltycons from AUTO to 1.
KNITRO changing bar_switchrule from AUTO to 2.
KNITRO changing linsolver from AUTO to 2.

Problem Characteristics
-----------------------
Objective goal:  Minimize
Number of variables:                     3
    bounded below:                       3
    bounded above:                       0
    bounded below and above:             0
    fixed:                               0
    free:                                0
Number of constraints:                   2
    linear equalities:                   1
    nonlinear equalities:                0
    linear inequalities:                 0
    nonlinear inequalities:              1
    range:                               0
Number of nonzeros in Jacobian:          6
Number of nonzeros in Hessian:           6

EXIT: Locally optimal solution found.

Final Statistics
----------------
Final objective value               =   9.36000000000340e+02
Final feasibility error (abs / rel) =   0.00e+00 / 0.00e+00
Final optimality error  (abs / rel) =   4.16e-07 / 2.60e-08
# of iterations                     =          9
# of CG iterations                  =          0
# of function evaluations           =         10
# of gradient evaluations           =         10
Total program time (secs)           =       0.00546 (     0.008 CPU time)
Time spent in evaluations (secs)    =       0.00246

===============================================================================
```

The number of function evaluation was reduced to 10, simply by providing exact first derivatives. This small example shows the practical importance of being able to provide exact derivatives; since (unlike modeling environments like AMPL) MATLAB does not provide automatic differentiation, the user must compute these derivatives analytically and then code them manually as in the above example.

2.3.6 C/C++ example

Let us now modify our C example from *Getting started with the callable library* similarly, so as to provide first derivatives.

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include "knitro.h"


/* callback function that evaluates the objective
   and constraints */
int  callback (const int evalRequestCode,
        const int             n,
        const int             m,
        const int             nnzJ,
        const int             nnzH,
        const double * const  x,
        const double * const  lambda,
              double * const  obj,
              double * const  c,
              double * const  objGrad,
              double * const  jac,
              double * const  hessian,
              double * const  hessVector,
              void   *        userParams)
{
        if (evalRequestCode == KTR_RC_EVALFC) {
                /* objective function */
                *obj    = 1000 - x[0]*x[0] - 2*x[1]*x[1] - x[2]*x[2] - x[0]*x[1] - x[0]*x[2];

                /* constraints */
                c[0]    = 8*x[0] + 14*x[1] + 7*x[2] - 56;
                c[1]    = x[0]*x[0] + x[1]*x[1] + x[2]*x[2] -25;

                return(0);
        }
        else if (evalRequestCode == KTR_RC_EVALGA) {

                /* gradient of objective */
                objGrad[0] = -2*x[0] - x[1] - x[2];
                objGrad[1] = -4*x[1] - x[0];
                objGrad[2] = -2*x[2] - x[0];

                /* Jacobian matrix of constraints */
                jac[0] = 8;
                jac[1] = 2*x[0];
                jac[2] = 14;
                jac[3] = 2*x[1];
                jac[4] = 7;
                jac[5] = 2*x[2];

                return( 0 );
        } else {
                printf ("Wrong evalRequestCode in callback function.\n");
                return(-1);
        }
}

/* main */
int  main (int  argc, char  *argv[]) {
        int  nStatus;

        /* variables that are passed to KNITRO */
        KTR_context     *kc;
```

```
int n, m, nnzJ, nnzH, objGoal, objType;
int *cType;
int *jacIndexVars, *jacIndexCons;
double obj, *x, *lambda;
double *xLoBnds, *xUpBnds, *xInitial, *cLoBnds, *cUpBnds;
int i, j, k; // convenience variables

/*problem size and mem allocation */
n = 3;
m = 2;
nnzJ = n*m;
nnzH = 0;
x      = (double *) malloc (n     * sizeof(double));
lambda = (double *) malloc ((m+n) * sizeof(double));
xLoBnds      = (double *) malloc (n * sizeof(double));
xUpBnds      = (double *) malloc (n * sizeof(double));
xInitial     = (double *) malloc (n * sizeof(double));
cType        = (int    *) malloc (m * sizeof(int));
cLoBnds      = (double *) malloc (m * sizeof(double));
cUpBnds      = (double *) malloc (m * sizeof(double));
jacIndexVars = (int    *) malloc (nnzJ * sizeof(int));
jacIndexCons = (int    *) malloc (nnzJ * sizeof(int));

/* objective type */
objType = KTR_OBJTYPE_GENERAL;
objGoal = KTR_OBJGOAL_MINIMIZE;

/* bounds and constraints type */
for (i = 0; i < n; i++) {
        xLoBnds[i] = 0.0;
        xUpBnds[i] = KTR_INFBOUND;
}
for (j = 0; j < m; j++) {
        cType[j] = KTR_CONTYPE_GENERAL;
        cLoBnds[j] = 0.0;
        cUpBnds[j] = (j == 0 ? 0.0 : KTR_INFBOUND);
}

/* initial point */
for (i = 0; i < n; i++)
        xInitial[i] = 2.0;

/* sparsity pattern (here, of a full matrix) */
k = 0;
for (i = 0; i < n; i++)
        for (j = 0; j < m; j++) {
                jacIndexCons[k] = j;
                jacIndexVars[k] = i;
                k++;
        }

/* create a KNITRO instance */
kc = KTR_new();
if (kc == NULL)
        exit( -1 ); // probably a license issue

/* set options: exact/user-supplied gradient option */
//if (KTR_set_int_param_by_name (kc, "gradopt", KTR_GRADOPT_FORWARD) != 0)
```

```
        if (KTR_set_int_param_by_name (kc, "gradopt", KTR_GRADOPT_EXACT) != 0)
                exit( -1 );
        if (KTR_set_int_param_by_name (kc, "hessopt", KTR_HESSOPT_BFGS) != 0)
                exit( -1 );
        if (KTR_set_int_param_by_name (kc, "outlev", 1) != 0)
                exit( -1 );

        /* register the callback function */
        if (KTR_set_func_callback (kc, &callback) != 0)
                exit( -1 );
        if (KTR_set_grad_callback (kc, &callback) != 0)
                exit( -1 );

        /* pass the problem definition to KNITRO */
        nStatus = KTR_init_problem (kc, n, objGoal, objType,
                xLoBnds, xUpBnds,
                m, cType, cLoBnds, cUpBnds,
                nnzJ, jacIndexVars, jacIndexCons,
                nnzH, NULL, NULL, xInitial, NULL);

        /* free memory (KNITRO maintains its own copy) */
        free (xLoBnds);
        free (xUpBnds);
        free (xInitial);
        free (cType);
        free (cLoBnds);
        free (cUpBnds);
        free (jacIndexVars);
        free (jacIndexCons);

        /* solver call */
        nStatus = KTR_solve (kc, x, lambda, 0, &obj,
                NULL, NULL, NULL, NULL, NULL, NULL);
        if (nStatus != 0)
                printf ("\nKNITRO failed to solve the problem, final status = %d\n",
                                nStatus);
        else
                printf ("\nKNITRO successful, objective is = %e\n", obj);

        /* delete the KNITRO instance and primal/dual solution */
        KTR_free (&kc);
        free (x);
        free (lambda);

        getchar();
        return( 0 );
}
```

The callback function was simply updated to provide the derivatives, and then registered with:

```
KTR_set_grad_callback (kc, &callback)
```

Last, the `gradopt` option was set to exact/user-supplied) instead of forward finite-differences using:

```
KTR_set_int_param_by_name (kc, "gradopt", KTR_GRADOPT_EXACT)
```

Running this code produces the following output.

```
=====================================
       Commercial Ziena License
             KNITRO 9.1.0
          Ziena Optimization
=====================================

KNITRO presolve eliminated 0 variables and 0 constraints.

hessopt:              2
outlev:               1
KNITRO changing algorithm from AUTO to 1.
KNITRO changing bar_murule from AUTO to 4.
KNITRO changing bar_initpt from AUTO to 3.
KNITRO changing bar_penaltyrule from AUTO to 2.
KNITRO changing bar_penaltycons from AUTO to 1.
KNITRO changing bar_switchrule from AUTO to 2.
KNITRO changing linsolver from AUTO to 2.

Problem Characteristics
-----------------------
Objective goal:  Minimize
Number of variables:                     3
    bounded below:                       3
    bounded above:                       0
    bounded below and above:             0
    fixed:                               0
    free:                                0
Number of constraints:                   2
    linear equalities:                   0
    nonlinear equalities:                1
    linear inequalities:                 0
    nonlinear inequalities:              1
    range:                               0
Number of nonzeros in Jacobian:          6
Number of nonzeros in Hessian:           6

EXIT: Locally optimal solution found.

Final Statistics
----------------
Final objective value               =   9.36000000017290e+02
Final feasibility error (abs / rel) =   0.00e+00 / 0.00e+00
Final optimality error  (abs / rel) =   2.61e-07 / 1.63e-08
# of iterations                     =          8
# of CG iterations                  =          0
# of function evaluations           =          9
# of gradient evaluations           =          9
Total program time (secs)           =       0.00194 (     0.002 CPU time)
Time spent in evaluations (secs)    =       0.00000

===============================================================================


KNITRO successful, objective is = 9.360000e+02
```

Again, the number of function calls is reduced with respect to the derivative-free case.

Note: Automatic differentiation packages like ADOL-C and ADIFOR can help in generating code with derivatives.

These codes are an alternative to differentiating the functions manually. Another option is to use symbolic differentiation software to compute an analytical formula for the derivatives.

2.3.7 Checking derivatives

One drawback of user-supplied derivatives is the risk of error in computing or implementing the derivatives, which would result in providing KNITRO with (wrong and) incoherent information: the computed function values would not match the computed derivatives. Approximate derivatives computed by finite differences are useful to check whether user-supplied derivatives match user-supplied function evaluations.

Users of modeling languages such as AMPL need not be worried about this, since derivatives are computed automatically by the modeling software. However, for users of MATLAB and the callable library it is a good practice to check one's exact derivatives against finite differences approximations. Note that small differences between exact and finite-difference approximations are to be expected.

KNITRO offers the following user options to check for errors in the user-supplied first derivatives (i.e., the objective gradient and the Jacobian matrix).

2.3.8 Derivative Check Options

Option	Meaning
derivcheck	Specifies whether or not to enable the derivative checker
derivcheck_tol	Specifies the relative tolerance used for identifying potential errors in the derivatives
derivcheck_type	Specifies whether to use forward or central finite differences to compute the derivative check

Note that to use the derivative checker, you must set gradopt = 1 and supply callback routines that compute the objective and constraint functions and analytic first derivatives. To perform a derivative check, simply set derivcheck = 1 (by default the derivative checker is turned off). Additionally you can set derivcheck_type to specify what type of finite differencing to use for the derivative check, and derivcheck_tol to change the default relative tolerance used to detect derivative errors.

It is best to check the derivatives at different points, and to avoid points where partial derivatives happen to equal zero. If an initial point was provided by the user (e.g., via KTR_init_problem()), then KNITRO will perform the derivative check at this point. Otherwise, if no initial point is provided, KNITRO will perform the derivative check at a randomly generated point that satisfies the variable bounds. To perform a derivative check at different points, simply feed different initial points to KNITRO.

Using the example problem above, if the KNITRO derivative checker runs, and the relative differences between all the user-supplied derivatives and finite-difference derivatives satisfy the tolerance defined by derivcheck_tol, then you will see the following output:

```
-------------------------------------------------------------------------
KNITRO Derivative Check Information

Checking 1st derivatives with forward finite differences.
Derivative check performed at user-supplied initial point.
Printing relative differences >   1.0000e-06.

Maximum relative difference in the objective gradient =   0.0000e+00.
Maximum relative difference in the Jacobian           =   0.0000e+00.
Derivative check passed.
-------------------------------------------------------------------------
```

before the optimization begins. Since the derivative check passed, KNITRO will automatically proceed with the optimization using the user-supplied derivatives.

Now let us modify the objective gradient computation in the example problem above as follows:

```
/* gradient of objective */
/* objGrad[0] = -2*x[0] - x[1] - x[2]; */
objGrad[0] = -2*x[0] - x[1]; /* BUG HERE !!! */
```

Running the code again, we obtain:

```
-------------------------------------------------------------------------
KNITRO Derivative Check Information

Checking 1st derivatives with forward finite differences.
Derivative check performed at user-supplied initial point.
Printing relative differences >   1.0000e-06.

WARNING: The discrepancy for objective gradient element objGrad[0]
     exceeds the derivative check relative tolerance of 1.000000e-06.
     analytic (user-supplied) value =  -6.000000000000e+00,
     finite-difference value        =  -8.000000000000e+00,
     |rel diff| =  3.3333e-01, |abs diff| =  2.0000e+00

Maximum relative difference in the objective gradient =   3.3333e-01.
Maximum relative difference in the Jacobian           =   0.0000e+00.
Derivative check failed.
-------------------------------------------------------------------------

EXIT: Derivative check failed.
```

KNITRO is warning us that the finite difference approximation of the first coordinate of the gradient at the initial point is about -8, whereas its (supposedly) exact user-supplied value is about -6: there is a bug in our implementation of the gradient of the objective. KNITRO prints a message indicating the derivative discrepancy it found and terminates immediately with a failure message.

KNITRO also provides a separate API function `KTR_check_first_ders()` that can also be used for checking derivatives. See the *KNITRO API* section in the Reference Manual and the KNITRO header file for more information about this function.

2.4 Multistart

Nonlinear optimization problems are often nonconvex due to the objective function, constraint functions, or both. When this is true, there may be many points that satisfy the local optimality conditions. Default KNITRO behavior is to return the first locally optimal point found. KNITRO offers a simple *multi-start* feature that searches for a better optimal point by restarting KNITRO from different initial points. The feature is enabled by setting `ms_enable` = 1.

Note: In many cases the user would like to obtain the global optimum to the optimization problem; that is, the local optimum with the very best objective function value. KNITRO cannot guarantee that multi-start will find the global optimum. In general, the global optimum can only be found with special knowledge of the objective and constraint functions; for example, the functions may need to be bounded by other piece-wise convex functions. KNITRO executes with very little information about functional form. Although no guarantee can be made, the probability of finding a better local solution improves if more start points are tried.

2.4.1 Multistart algorithm

The multi-start procedure generates new start points by randomly selecting components of x that satisfy lower and upper bounds on the variables. KNITRO finds a local optimum from each start point using the same problem definition and user options. The final solution returned from `KTR_solve()` is the local optimum with the best objective function value if any local optima have been found. If no local optimum has been found, KNITRO will return the best feasible solution estimate it found. If no feasible solution estimate has been found, KNITRO will return the least infeasible point.

2.4.2 Parallel multistart

The multistart procedure can run in parallel on shared memory multi-processor machines by setting `par_numthreads` greater than 1. See *Parallelism* for more details on controlling parallel performance in KNITRO.

When the multistart procedure is run in parallel, KNITRO will produce the same sequence of initial points and solves that you see when running multistart sequentially (though, perhaps, not in the same order).

Therefore, as long as you run multistart to completion (`ms_terminate` =0) and use the deterministic option (`ms_deterministic` =1), you should visit the same initial points encountered when running multistart sequentially, and get the same final solution. By default `ms_terminate` =0 and `ms_deterministic` =1 so that the parallel multistart produces the same solution as the sequential multistart.

However, if `ms_deterministic` =0, or `ms_terminate` >0, there is no guarantee that the final solution reported by multistart will be the same when run in parallel compared to the solution when run sequentially, and even the parallel solution may change when run at different times.

2.4.3 Multistart output

For multistart, you can have output from each local solve written to a file named `knitro_ms_x.log` where "x" is the solve number by setting the option `ms_outsub=1`.

2.4.4 Multistart options

The multi-start option is convenient for conducting a simple search for a better solution point. Search time is improved if the variable bounds are made as tight as possible, confining the search to a region where a good solution is likely to be found. The user can restrict the multi-start search region without altering bounds by using the options `ms_maxbndrange` and `ms_startptrange`. The other multi-start options are the following.

Option	Meaning
`ms_deterministic`	Control whether to use deterministic multistart
`ms_enable`	Enable multistart
`ms_maxbndrange`	Maximum unbounded variable range for multistart
`ms_maxsolves`	Maximum KNITRO solves for multistart
`ms_maxtime_cpu`	Maximum CPU time for multistart, in seconds
`ms_maxtime_real`	Maximum real time for multistart, in seconds
`ms_num_to_save`	Feasible points to save from multistart
`ms_outsub`	Can write each solve to a file (parallel only)
`ms_savetol`	Tol for feasible points being equal
`ms_seed`	Initial seed for generating random start points
`ms_startptrange`	Maximum variable range for multistart
`ms_terminate`	Termination condition for multistart

The number of start points tried by multi-start is specified with the option ms_maxsolves. By default, KNITRO will try *min(200, 10*n)*, where *n* is the number of variables in the problem. Users may override the default by setting ms_maxsolves to a specific value.

The ms_maxbndrange option applies to variables unbounded in at least one direction (i.e., the upper or lower bound, or both, is infinite) and keeps new start points within a total range equal to the value of ms_maxbndrange. The ms_startptrange option applies to all variables and keeps new start points within a total range equal to the value of ms_startptrange, overruling ms_maxbndrange if it is a tighter bound. In general, use ms_startptrange to limit the multi-start search only if the initial start point supplied by the user is known to be the center of a desired search area. Use ms_maxbndrange as a surrogate bound to limit the multi-start search when a variable is unbounded.

The ms_num_to_save option allows a specific number of distinct feasible points to be saved in a file named KNITRO_mspoints.log. Each point results from a KNITRO solve from a different starting point, and must satisfy the absolute and relative feasibility tolerances. Different start points may return the same feasible point, and the file contains only distinct points. The option ms_savetol determines that two points are distinct if their objectives or any solution components (including Lagrange multipliers) are separated by more than the value of ms_savetol using a relative tolerance test. More specifically, two values x and y are considered distinct if:

$$|x - y| \geq \max(1, |x|, |y|) * \text{ms\_savetol}.$$

The file stores points in order from best objective to worst. If objectives are the same (as defined by ms_savetol), then points are ordered from smallest feasibility error to largest. The file can be read manually, but conforms to a fixed property/value format for machine reading.

Instead of using multi-start to search for a global solution, a user may want to use multi-start as a mechanism for finding any locally optimal or feasible solution estimate of a nonconvex problem and terminate as soon as one such point is found. The ms_terminate option, provides the user more control over when to terminate the multi-start procedure.

If ms_terminate = *optimal* the multi-start procedure will stop as soon as the first locally optimal solution is found or after ms_maxsolves – whichever comes first. If ms_terminate = *feasible* the multi-start procedure will instead stop as soon as the first feasible solution estimate is found or after ms_maxsolves – whichever comes first. If ms_terminate = *maxsolves*, it will only terminate after ms_maxsolves.

The option ms_seed can be used to change the seed used to generate the random initial points for multistart.

2.4.5 Multistart callbacks

The multistart procedure provides two callback utilities for the callable library API.

```
int  KNITRO_API KTR_set_ms_process_callback (KTR_context_ptr       kc,
                                             KTR_callback * const  fnPtr);

int  KNITRO_API KTR_set_ms_initpt_callback (KTR_context_ptr                kc,
                                            KTR_ms_initpt_callback * const  fnPtr);
```

The first callback can be used to perform some user task after each multistart solve and is set by calling KTR_set_ms_process_callback(). You can use the second callback to specify your own initial points for multistart instead of using the randomly generated KNITRO initial points. This callback function can be set through the function KTR_set_ms_initpt_callback().

See the *KNITRO API* section in the Reference Manual for details on setting these callback functions and the prototypes for these callback functions.

2.4.6 AMPL example

Let us consider again our AMPL example from Section *Getting started with AMPL* and run it with a different set of options:

```
ampl: reset;
ampl: option solver knitroampl;
ampl: option knitro_options "ms_enable=1 ms_num_to_save=5 ms_savetol=0.01";
ampl: model testproblem.mod;
ampl: solve;
```

The KNITRO log printed on screen changes to reflect the results of the many solver runs that were made during the multistart procedure, and the very end of this log reads:

```
Multistart stopping, reached ms_maxsolves limit.

MULTISTART: Best locally optimal point is returned.
EXIT: Locally optimal solution found.

Final Statistics
----------------
Final objective value               =   9.35999999745429e+02
Final feasibility error (abs / rel) =   1.44e-07 / 3.83e-10
Final optimality error  (abs / rel) =   6.48e-07 / 4.28e-08
# of iterations                     =        476
# of CG iterations                  =        180
# of function evaluations           =        573
# of gradient evaluations           =        506
# of Hessian evaluations            =        486
Total program time (secs)           =       0.15680 (     0.027 CPU time)

================================================================================

KNITRO 9.1.0: Locally optimal solution.
objective 935.9999997; feasibility error 1.44e-07
476 iterations; 573 function evaluations
```

which shows that many more functions calls were made than without multistart. A file `knitro_mspoints.txt` was also created, whose content reads:

```
// KNITRO 9.1.0 Multi-start Repository for feasible points.
// Each point contains information about the problem and the point.
// Points are sorted by objective value, from best to worst.


// Next feasible point.
numVars = 3
numCons = 2
objGoal = MINIMIZE
obj =   9.3600000342420878e+02
knitroStatus = 0
localSolveNumber = 1
feasibleErrorAbsolute =   0.00e+00
feasibleErrorRelative =   0.00e+00
optimalityErrorAbsolute =   2.25e-07
optimalityErrorRelative =   1.41e-08
x[0] =   2.0511214409048425e-07
x[1] =   4.1077619358921463e-08
x[2] =   7.9999996834308824e+00
```

```
lambda[0] =  -4.5247620510168322e-08
lambda[1] =   2.2857143915699769e+00
lambda[2] =  -1.0285715141992103e+01
lambda[3] =  -3.2000001143071813e+01
lambda[4] =  -2.1985040913238130e-07

// Next feasible point.
numVars = 3
numCons = 2
objGoal = MINIMIZE
obj =   9.5100000269458542e+02
knitroStatus = 0
localSolveNumber = 2
feasibleErrorAbsolute =   0.00e+00
feasibleErrorRelative =   0.00e+00
optimalityErrorAbsolute =   3.67e-07
optimalityErrorRelative =   2.62e-08
x[0] =   6.9999996377946481e+00
x[1] =   7.4479065893720198e-08
x[2] =   2.6499084231411754e-07
lambda[0] =  -6.3891336872934633e-08
lambda[1] =   1.7500001368019027e+00
lambda[2] =  -2.1791026695882249e-07
lambda[3] =  -1.7500002055167382e+01
lambda[4] =  -5.2500010586300956e+00
```

In addition to the known solution with value 936 that we had already found with a single solver run, we discover another local minimum with value 951 that we had never seen before. In this case, the new solution is not as good as the first one, but sometimes running the multistart algorithm significantly improves the objective function value with respect to single-run optimization.

2.4.7 MATLAB example

In order to run the multistart algorithm in MATLAB, we must pass the relevant set of options to KNITRO via the KNITRO options file. Let us create a simple text file named `knitro.opt` with the following content:

```
ms_enable 1
ms_num_to_save 5
ms_savetol 0.01
hessopt 2
```

(the last line *hessopt 2* is necessary to remind KNITRO to use approximate second derivatives, since we are not providing the exact hessian). Then let us run our MATLAB example from Section *MATLAB example* again, where the call to *knitromatlab* has been replaced with:

```
knitromatlab(@obj, x0, A, b, Aeq, beq, lb, ub, @nlcon, [], options, 'knitro.opt');
```

and where the `knitro.opt` file was placed in the current directory so that MATLAB can find it. The KNITRO log looks simlar to what we observed with AMPL.

2.4.8 C example

The C example can also be easily modified to enable multistart by adding the following lines before the call to `KTR_solve()`:

```
// multistart
if (KTR_set_int_param_by_name (kc, "ms_enable", 1) != 0)
exit( -1 );
if (KTR_set_int_param_by_name (kc, "ms_num_to_save", 5) != 0)
exit( -1 );
if (KTR_set_double_param_by_name (kc, "ms_savetol", 0.01) != 0)
exit( -1 );
```

Again, running this example we get a KNITRO log that looks similar to what we observed with AMPL.

2.5 Mixed-integer nonlinear programming

KNITRO provides tools for solving optimization models (both linear and nonlinear) with binary or integer variables. The KNITRO mixed integer programming (MIP) code offers two algorithms for mixed-integer nonlinear programming (MINLP). The first is a nonlinear branch and bound method and the second implements the hybrid Quesada-Grossman method for convex MINLP.

The KNITRO MINLP code is designed for convex mixed integer programming and is only a heuristic for nonconvex problems. The MINLP code also handles mixed integer linear programs (MILP) of moderate size.

Note: The KNITRO MIP tools do not currently handle special ordered sets (SOS's) or semi-continuous variables.

2.5.1 AMPL example

Using MINLP features in AMPL is very simple: one only has to declare variables as integer in the AMPL model. In our toy example, let us modify the declaration of variable x as follows:

```
var x{j in 1..3} >= 0 integer;
```

and then run the example again. The KNITRO log now mentions 3 integer variables, and displays additional statistics related to the MIP solve.

```
=====================================
       Commercial Ziena License
             KNITRO 9.1.0
          Ziena Optimization
=====================================

mip_debug:            1
mip_outinterval:      1
mip_outsub:           1
KNITRO changing mip_method from AUTO to 1.
KNITRO changing mip_rootalg from AUTO to 1.
KNITRO changing mip_lpalg from AUTO to 3.
KNITRO changing mip_branchrule from AUTO to 2.
KNITRO changing mip_selectrule from AUTO to 2.
KNITRO changing mip_rounding from AUTO to 3.
KNITRO changing mip_heuristic from AUTO to 1.
KNITRO changing mip_pseudoinit from AUTO to 1.

Problem Characteristics
-----------------------
Objective goal:  Minimize
Number of variables:                    3
```

```
    bounded below:                      3
    bounded above:                      0
    bounded below and above:            0
    fixed:                              0
    free:                               0
Number of binary variables:             0
Number of integer variables:            3
Number of constraints:                  2
    linear equalities:                  1
    nonlinear equalities:               0
    linear inequalities:                0
    nonlinear inequalities:             1
    range:                              0
Number of nonzeros in Jacobian:         6
Number of nonzeros in Hessian:          5

KNITRO detected 0 GUB constraints
KNITRO derived 0 knapsack covers after examining 0 constraints
KNITRO solving root node relaxation

   Node    Left    Iinf     Objective         Best relaxatn  Best incumbent
  ------  ------  ------  --------------     --------------  --------------
*      1       0       0    9.360000e+02       9.360000e+02    9.360000e+02

EXIT: Optimal solution found.

Final Statistics for MIP
------------------------
Final objective value               =   9.36000000000000e+02
Final integrality gap (abs / rel)   =  0.00e+00 /   0.00e+00 ( 0.00%)
# of nodes processed                =        1
# of subproblems solved             =        2
Total program time (secs)           =       0.00829 (     0.007 CPU time)
Time spent in evaluations (secs)    =       0.00018

===============================================================================

KNITRO 9.1.0: Locally optimal solution.
objective 936; integrality gap 0
1 nodes; 2 subproblem solves
```

Note that this example is not particularly interesting since the two solutions we know for the continuous version of this problem are already integer "by chance". As a consequence, the MINLP solve returns the same solution as the continuous solve. However, if for instance you change the first constraint to:

```
s.t. c1: 8*x[1] + 14*x[2] + 7*x[3] - 50 = 0;
```

instead of:

```
s.t. c1: 8*x[1] + 14*x[2] + 7*x[3] - 56 = 0;
```

you will observe that the integer solution differs from the continuous one.

2.5.2 MATLAB example

To use the MINLP features in MATLAB, one must use the function *knitromatlab_mip*, rather than *knitromatlab*. The function signature is very similar to *knitromatlab*, but three additional argument arrays are used. The most elaborate form is:

```
[x,fval,exitflag,output,lambda,grad,hessian] =
        knitromatlab_mip(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,...
            xType,objFnType,cineqFnType,extendedFeatures,options,KNITROOptions)
```

The array *xType* sets the variable types and must be the same length as *x0* if it is used. Continuous, integer, and binary variables are set with 0, 1, and 2, respectively. Passing an empty array, [], is equivalent to an array of all zeros.

The scalar *objFnType* sets the objective function type. Convex, nonconvex, and uncertain objectives are set with 0, 1, and 2, respectively. Passing an empty array, [], is equivalent to passing an array of twos.

The array *cineqFnType* sets the inequality constraint function types and its length must be the same as the number of inequality constraints. Linear constraints are known to be convex, and nonlinear equality constraints are known to be nonconvex, so they are not included in the array. Convex, nonconvex, and uncertain inequality constraints are set with 0, 1, and 2, respectively. Passing an empty array, [], is equivalent to passing an array of twos.

Modifying the toy example in MATLAB to use integer variables can be done as follows:

```
xType = [2;2;2];
objFnType = 1;
cineqFnType = 1;

%modify the solver call
x = knitromatlab_mip(obj, x0, A, b, Aeq, beq, lb, ub, ...
         nlcon, xType, objFnType, cineqFnType);
```

2.5.3 C example

A MIP problem is defined and solved via the callable library interface using the API functions `KTR_mip_init_problem()` and `KTR_mip_solve()`.

The signature of `KTR_mip_init_problem()` is the following.

```
int  KNITRO_API KTR_mip_init_problem (
        KTR_context_ptr kc,
        const int             n,
        const int             objGoal,
        const int             objType,
        const int             objFnType,
        const int    * const  xType,
        const double * const  xLoBnds,
        const double * const  xUpBnds,
        const int             m,
        const int    * const  cType,
        const int    * const  cFnType,
        const double * const  cLoBnds,
        const double * const  cUpBnds,
        const int             nnzJ,
        const int    * const  jacIndexVars,
        const int    * const  jacIndexCons,
        const int             nnzH,
        const int    * const  hessIndexRows,
        const int    * const  hessIndexCols,
        const double * const  xInitial,
        const double * const  lambdaInitial);
```

The only differences with `KTR_init_problem()` are

```
const int              objFnType,
const int    * const   xType,
...
const int    * const   cFnType,
```

where *objFnType* sets the objective function type (convex, nonconvex or uncertain), *xType* sets the variable type (binary, integer or continuous) and *cFnType* sets the constraint function type (same choices as for the objective function).

The signature of `KTR_mip_solve()` is exactly the same as for `KTR_solve()`.

In order to turn our C toy example into a MINLP probelm, it thus suffices to replace the call to `KTR_init_problem()` with

```
/* in the declarations */
int objFnType = KTR_FNTYPE_NONCONVEX;
int *xType;
int *cFnType;

/* allocate and fill in the arrays */
xType = (int *) malloc (n * sizeof(int));
cFnType = (int *) malloc (m * sizeof(int));
xType[0] = KTR_VARTYPE_INTEGER;
xType[1] = KTR_VARTYPE_INTEGER;
xType[2] = KTR_VARTYPE_INTEGER;
cFnType[0] = KTR_FNTYPE_CONVEX;
cFnType[1] = KTR_FNTYPE_NONCONVEX;

/* call to KTR_mip_init_problem */
nStatus = KTR_mip_init_problem (
        kc, n, objGoal, objType,
        objFnType, xType,
        xLoBnds, xUpBnds,
        m, cType, cFnType, cLoBnds, cUpBnds,
        nnzJ, jacIndexVars, jacIndexCons,
        nnzH, NULL, NULL, xInitial, NULL);

/* free memory */
free(xType);
free(cFnType);
```

and the call to `KTR_solve()` by a call to `KTR_mip_solve()` with the same arguments. The KNITRO log will look similar to what we observed in the AMPL example above. Again, this example is quite unusual in the sense that the continuous solution is already integer, which of course is not the case in general.

2.5.4 MINLP options

Many user options are provided for the MIP features to tune performance, including options for branching, node selection, rounding and heuristics for finding integer feasible points. User options specific to the MIP tools begin with *mip_*. It is recommended to experiment with several of these options as they often can make a significant difference in performance.

Name	Meaning
mip_branchrule	MIP branching rule
mip_debug	MIP debugging level (0=none, 1=all)
mip_gub_branch	Branch on GUBs (0=no, 1=yes)
mip_heuristic	MIP heuristic search
mip_heuristic_maxit	MIP heuristic iteration limit
mip_implications	Add logical implications (0=no, 1=yes)
mip_integer_tol	Threshold for deciding integrality
mip_integral_gap_abs	Absolute integrality gap stop tolerance
mip_integral_gap_rel	Relative integrality gap stop tolerance
mip_knapsack	Add knapsack cuts (0=no, 1=ineqs, 2=ineqs+eqs)
mip_lpalg	LP subproblem algorithm
mip_maxnodes	Maximum nodes explored
mip_maxsolves	Maximum subproblem solves
mip_maxtime_cpu	Maximum CPU time in seconds for MIP
mip_maxtime_real	Maximum real in seconds time for MIP
mip_method	MIP method (0=auto, 1=BB, 2=HQG)
mip_outinterval	MIP output interval
mip_outlevel	MIP output level
mip_outsub	Enable MIP subproblem output
mip_pseudoinit	Pseudo-cost initialization
mip_rootalg	Root node relaxation algorithm
mip_rounding	MIP rounding rule
mip_selectrule	MIP node selection rule
mip_strong_candlim	Strong branching candidate limit
mip_strong_level	Strong branching tree level limit
mip_strong_maxit	Strong branching iteration limit
mip_terminate	Termination condition for MIP

If finding any integer feasible point is your highest priority, you should set the mip_heuristic option to search for an integer feasible point before beginning the branch and bound procedure (by default no heuristics are applied).

2.5.5 Branching priorities

It is also possible to specify branching priorities in KNITRO. Priorities must be positive numbers (variables with non-positive values are ignored). Variables with higher priority values will be considered for branching before variables with lower priority values. When priorities for a subset of variables are equal, the branching rule is applied as a tiebreaker.

Branching priorities in AMPL

Branching priorities for integer variables can be specified in AMPL use the AMPL suffixes feature as shown below.

```
...
ampl: var x{j in 1..3} >= 0 integer;
...
ampl: suffix priority IN, integer, >=0, <=9999;
ampl: let x[1].priority := 5;
ampl: let x[2].priority := 1;
ampl: let x[3].priority := 10;
```

See the AMPL documentation for more information on the ".priority " suffix.

Branching priorities in the callable library API

Branching priorities for integer variables can be specified through the callable library interface using the function shown below.

```
int KTR_mip_set_branching_priorities (      KTR_context_ptr kc,
                                      const int * const     xPriorities);
```

The array *xPriorities* has length *"n"*, where *n* is the number of variables. Values for continuous variables are ignored. KNITRO makes a local copy of all inputs, so the application may free memory after the call. This routine must be called after calling `KTR_mip_init_problem()` and before calling `KTR_mip_solve()`.

2.5.6 MINLP callbacks

The KNITRO MINLP procedure provides a user callback utility that can be used in the callable library API to perform some user task after each node is processed in the branch-and-bound tree. This callback function is set by calling the following function:

```
int  KNITRO_API KTR_set_mip_node_callback (KTR_context_ptr       kc,
                                           KTR_callback * const  fnPtr);
```

See the *KNITRO API* section in the Reference Manual for details on setting this callback function and the prototype for this callback function.

2.5.7 Determining convexity

Knowing whether or not a function is convex may be useful in methods for mixed integer programming as linearizations derived from convex functions can be used as outer approximations of those constraints. These outer approximations are useful in computing lower bounds. The callable library for the mixed integer programming API allows for the user to specify whether or not the problem functions (objective and constraints) are convex or not. If unknown, they can be marked as such.

A function f is convex if for any two points x and y, we have

$$f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y), \text{ for all } \alpha \in [0,1].$$

In identifying the objective or constraints as convex, we are assuming a problem form where the objective is being minimized and the constraints are all formulated as "less than or equal to" constraints. If we are maximizing or looking at "greater than or equal to" constraints, then the objective or constraint should be labeled as convex if its negation is convex.

More specifically, the objective function $f(x)$ should be marked as convex if when minimizing $f(x)$ satisfies the above convexity condition, or if when maximizing $-f(x)$ satisfies it. A constraint $c_i(x)$ should be labeled as convex if:

- c^L is infinite, c^U is finite and $c_i(x)$ satisfies the convexity condition; or
- c^L is finite, c^U is infinite and $-c_i(x)$ satisfies the convexity condition; or
- $c_i(x)$ is linear.

All linear functions are convex. Any nonlinear equality constraint is nonconvex.

The MIP solvers in KNITRO are designed for convex problems (problems where the objective and all the constraints are convex). If one or more functions are nonconvex, these solvers are only heuristics and may terminate at non-optimal points. The continuous solvers in KNITRO can handle either convex or nonconvex models. However, for nonconvex models, they may converge to local (rather than global) optimal solutions.

2.5.8 Additional examples

Examples for solving a MINLP problem using the MATLAB, C, and Java interfaces are provided with the distribution in the knitromatlab and examples directories.

2.6 Complementarity constraints

A complementarity constraint enforces that two variables are *complementary* to each other; i.e., that the following conditions hold for scalar variables *x* and *y*:

$$x \cdot y = 0, \quad x \geq 0, \quad y \geq 0.$$

The condition above is sometimes expressed more compactly as

$$0 \leq x \perp y \geq 0.$$

Intuitively, a complementarity constraint is a way to model a constraint that is combinatorial in nature since, for example, the complementary conditions imply that either *x* or *y* must be 0 (both may be 0 as well).

Without special care, these type of constraints may cause problems for nonlinear optimization solvers because problems that contain these types of constraints fail to satisfy constraint qualifications that are often assumed in the theory and design of algorithms for nonlinear optimization. For this reason, we provide a special interface in KNITRO for specifying complementarity constraints. In this way, KNITRO can recognize these constraints and handle them with special care internally.

Note: The complementarity features of KNITRO are not available through all interfaces. Currently, they are accessible only to users of the callable library, the MATLAB interface, and some modeling environments such as AMPL.

If a modeling language does not allow you to specifically identify and express complementarity constraints, then these constraints must be formulated as regular constraints and KNITRO will not perform any specializations.

Note: There are various ways to express complementarity conditions, but the complementarity features in the KNITRO callable library API and MATLAB API require you to specify the complementarity condition as two non-negative variables complementary to each other as shown above. Any complementarity condition can be written in this form.

2.6.1 Example

This problem is taken from J.F. Bard, *Convex two-level optimization*, *Mathematical Programming* 40(1), 15-27, 1988.

Assume we want to solve the following MPEC with KNITRO.

$$\begin{aligned}
\min \quad & f(x) = (x_0 - 5)^2 + (2x_1 + 1)^2 \\
\text{subject to:} \quad & \\
& c_0(x) = 2(x_1 - 1) - 1.5x_0 + x_2 - 0.5x_3 + x_4 = 0 \\
& c_1(x) = 3x_0 - x_1 - 3 \geq 0 \\
& c_2(x) = -x_0 + 0.5x_1 + 4 \geq 0 \\
& c_3(x) = -x_0 - x_1 + 7 \geq 0 \\
& c_1(x) \cdot x_2 = 0 \\
& c_2(x) \cdot x_3 = 0 \\
& c_3(x) \cdot x_4 = 0 \\
& x_i \geq 0 \quad \forall i = 0, \ldots 4.
\end{aligned}$$

Observe that complementarity constraints appear. Expressing this in compact notation, we have:

$$\begin{aligned}
\min \quad & f(x) = (x_0 - 5)^2 + (2x_1 + 1)^2 \\
\text{subject to:} \quad & \\
& 2(x_1 - 1) - 1.5x_0 + x_2 - 0.5x_3 + x_4 = 0 \quad (c_0) \\
& c_1(x) = 3x_0 - x_1 - 3 \\
& c_2(x) = -x_0 + 0.5x_1 + 4 \\
& c_3(x) = -x_0 - x_1 + 7 \\
& 0 \leq c_1(x) \perp x_2 \geq 0 \\
& 0 \leq c_2(x) \perp x_3 \geq 0 \\
& 0 \leq c_3(x) \perp x_4 \geq 0 \\
& x_0, x_1 \geq 0.
\end{aligned}$$

Since KNITRO requires that complementarity constraints be written as two variables complementary to each other, we must introduce slack variables (x_5, x_6, x_7) and re-write the problem as follows:

$$\begin{aligned}
\min \quad & f(x) = (x_0 - 5)^2 + (2x_1 + 1)^2 \\
\text{subject to:} \quad & \\
& 2(x_1 - 1) - 1.5x_0 + x_2 - 0.5x_3 + x_4 = 0 \quad (c_0) \\
& 3x_0 - x_1 - 3 - x_5 = 0 \quad (c_1) \\
& -x_0 + 0.5x_1 + 4 - x_6 = 0 \quad (c_2) \\
& -x_0 - x_1 + 7 - x_7 = 0 \quad (c_3) \\
& 0 \leq x_5 \perp x_2 \geq 0 \\
& 0 \leq x_6 \perp x_3 \geq 0 \\
& 0 \leq x_7 \perp x_4 \geq 0 \\
& x_i \geq 0, \quad \forall i = 0, \ldots 7..
\end{aligned}$$

The problem is now in a form suitable for KNITRO.

2.6.2 Complementarity constraints in AMPL

Complementarity constraints should be modeled using the AMPL *complements* command; e.g.,:

```
0 <= x complements y => 0;
```

The KNITRO callable library API and MATLAB API require that complementarity constraints be formulated as one variable complementary to another variable (both non-negative). However, in AMPL (beginning with KNITRO 8.0), you can express the complementarity constraints in any form allowed by AMPL. AMPL will then translate the complementarity constraints automatically to the form required by KNITRO.

Be aware that the AMPL presolver sometimes removes complementarity constraints. Check carefully that the problem definition reported by KNITRO includes all complementarity constraints, or switch off the AMPL presolver by setting option *presolve* to 0, if you don't want the AMPL presolver to modify the problem.

2.6.3 Complementarity constraints in MATLAB

Complementarity constraints can be specified through two fields of the *extendedFeatures* structure. The fields *ccIndexList1* and *ccIndexList2* contain the pairs of indices of variables that are complementary to each other.

Note: Variables which are specified as complementary should be specified to have a lower bound of 0 through the variable lower bound array *lb*.

2.6.4 Complementarity constraints with the callable library

Complementarity constraints can be specified in KNITRO through a call to the function `KTR_addcompcons()` which has the following prototype:

```
int  KNITRO_API KTR_addcompcons (KTR_context_ptr    kc,
                                 const int          numCompConstraints,
                                 const int * const  indexList1,
                                 const int * const  indexList2);
```

In addition to *kc* which is a pointer to a structure which holds all the relevant information about a particular problem instance, the arguments are:

- *numCompConstraints*, the number of complementarity constraints to be added to the problem (i.e., the number of pairs of variables which are complementary to each other).
- **indexList1* and **indexList2*, two arrays of length *numCompConstraints* specifying the variable indices for the first and second sets of variables in the pairs of complementary variables.

Note: The call to *KTR_addcompcons()* must occur after the call to *KTR_init_problem()*, but before the first call to `KTR_solve()`.

Note: Variables which are specified as complementary through the special *KTR_addcompcons()* functions should be specified to have a lower bound of 0 through the KNITRO lower bound array *xLoBnds*.

2.6.5 AMPL example

The AMPL model for our toy problem above is the following.

```
# Variables
var x{j in 0..7} >= 0;

# Objective function
minimize obj:
```

```
                (x[0]-5)^2 + (2*x[1]+1)^2;

# Constraints
s.t. c0: 2*(x[1]-1) - 1.5*x[0] + x[2] - 0.5*x[3] + x[4] = 0;
s.t. c1: 3*x[0] - x[1] - 3 - x[5] = 0;
s.t. c2: -x[0] + 0.5*x[1] + 4 - x[6] = 0;
s.t. c3: -x[0] - x[1] + 7 - x[7] = 0;
s.t. c4: 0 <= x[5] complements x[2] >= 0;
s.t. c5: 0 <= x[6] complements x[3] >= 0;
s.t. c6: 0 <= x[7] complements x[4] >= 0;
```

Running it through AMPL, we get the following output.

```
======================================
       Commercial Ziena License
            KNITRO 9.1.0
          Ziena Optimization
======================================

No start point provided -- KNITRO computing one.

KNITRO presolve eliminated 0 variables and 0 constraints.

hessian_no_f:         1
par_concurrent_evals: 0
The problem is identified as an MPEC.
KNITRO changing algorithm from AUTO to 1.
KNITRO changing bar_murule from AUTO to 4.
KNITRO changing bar_initpt from AUTO to 3.
KNITRO changing bar_penaltyrule from AUTO to 2.
KNITRO changing bar_penaltycons from AUTO to 1.
KNITRO changing bar_switchrule from AUTO to 1.
KNITRO changing linsolver from AUTO to 2.

Problem Characteristics
-----------------------
Objective goal:  Minimize
Number of variables:                     11
    bounded below:                        8
    bounded above:                        0
    bounded below and above:              0
    fixed:                                0
    free:                                 3
Number of constraints:                    7
    linear equalities:                    7
    nonlinear equalities:                 0
    linear inequalities:                  0
    nonlinear inequalities:               0
    range:                                0
Number of complementarities:              3
Number of nonzeros in Jacobian:          20
Number of nonzeros in Hessian:            2

  Iter      Objective      FeasError   OptError    ||Step||    CGits
--------  --------------  ----------  ----------  ----------  -------
       0    2.811162e+01   1.548e+00
       9    1.700000e+01   4.118e-12   1.256e-07   6.806e-05        0

EXIT: Locally optimal solution found.
```

```
Final Statistics
----------------
Final objective value               =   1.70000001674276e+01
Final feasibility error (abs / rel) =   4.12e-12 / 2.66e-12
Final optimality error  (abs / rel) =   1.26e-07 / 1.57e-08
# of iterations                     =          9
# of CG iterations                  =          7
# of function evaluations           =         10
# of gradient evaluations           =         10
# of Hessian evaluations            =          9
Total program time (secs)           =       0.00207 (     0.002 CPU time)
Time spent in evaluations (secs)    =       0.00004

===============================================================================

KNITRO 9.1.0: Locally optimal solution.
objective 17.000000167427565; feasibility error 4.12e-12
9 iterations; 10 function evaluations
```

KNITRO received our three complementarity constraints correctly (*"Number of complementarities: 3"*) and converged successfully (*"Locally optimal solution found"*).

2.6.6 MATLAB example

The following functions can be used in MATLAB to solve the same example as is shown for AMPL.

```
function exampleMPEC1

Jpattern = [];

Hpattern = sparse(zeros(8));
Hpattern(1,1) = 1;
Hpattern(2,2) = 1;

options = optimset('Algorithm', 'active-set', 'Display','iter', ...
 'GradObj','off','GradConstr','on', ...
 'JacobPattern',Jpattern,'Hessian','user-supplied','HessPattern',Hpattern, ...
 'HessFcn',@hessfun,'MaxIter',1000, ...
 'TolX', 1e-15, 'TolFun', 1e-8, 'TolCon', 1e-8);

A = []; b = [];
Aeq = [-1.5  2   1 -0.5 1  0  0  0;
        3   -1   0  0   0 -1  0  0;
       -1    0.5 0  0   0  0 -1  0;
       -1   -1   0  0   0  0  0 -1];
beq = [2 3 -4 -7];
lb = zeros(8,1);
ub = Inf*ones(8,1);
x0 = zeros(8,1);

extendedFeatures.ccIndexList1 = [6 7 8];
extendedFeatures.ccIndexList2 = [3 4 5];

[x,fval,exitflag,output,lambda] = ...
    knitromatlab(@objfun,x0,A,b,Aeq,beq,lb,ub,@constfun,extendedFeatures,options, ...
                 'knitro.opt');
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [f,g] = objfun(x)

f = (x(1)-5)^2 + (2*x(2)+1)^2;

if nargout > 1
 g = zeros(8,1);
 g(1) = 2*(x(1)-5);
 g(2) = 4*(2*x(2)+1);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [c,ceq,Gc,Gceq]= constfun(x)

c = [];
ceq=[];
Gc = [];
Gceq=[];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [H]= hessfun(x,lambda)

H=sparse(zeros(8));

H(1,1) = 2;
H(2,2) = 4;
```

Running this file will produce the following output from KNITRO.

```
======================================
       Commercial Ziena License
             KNITRO 9.1.0
          Ziena Optimization
======================================

KNITRO presolve eliminated 0 variables and 0 constraints.

bar_initpt:           1
hessian_no_f:         1
par_concurrent_evals: 0
par_blasnumthreads:   1
The problem is identified as an MPEC.
KNITRO changing algorithm from AUTO to 1.
KNITRO changing bar_murule from AUTO to 4.
KNITRO changing bar_penaltyrule from AUTO to 2.
KNITRO changing bar_penaltycons from AUTO to 1.
KNITRO changing bar_switchrule from AUTO to 1.
KNITRO changing linsolver from AUTO to 2.
KNITRO shifted start point to satisfy presolved bounds (8 variables).

Problem Characteristics
-----------------------
Objective goal:  Minimize
Number of variables:                    8
    bounded below:                      8
    bounded above:                      0
```

```
    bounded below and above:             0
    fixed:                               0
    free:                                0
Number of constraints:                   4
    linear equalities:                   4
    nonlinear equalities:                0
    linear inequalities:                 0
    nonlinear inequalities:              0
    range:                               0
Number of complementarities:             3
Number of nonzeros in Jacobian:         14
Number of nonzeros in Hessian:           2

  Iter      Objective      FeasError   OptError    ||Step||    CGits
--------  --------------  ----------  ----------  ----------  -------
       0    2.496050e+01   4.030e+00
      10    1.700000e+01   2.880e-10   3.366e-06   7.923e-07        0

EXIT: Locally optimal solution found.

Final Statistics
----------------
Final objective value               =   1.70000000019849e+01
Final feasibility error (abs / rel) =   2.88e-10 / 7.15e-11
Final optimality error  (abs / rel) =   3.37e-06 / 4.21e-07
# of iterations                     =         10
# of CG iterations                  =          5
# of function evaluations           =         13
# of gradient evaluations           =         11
# of Hessian evaluations            =         10
Total program time (secs)           =       0.00307 (     0.004 CPU time)
Time spent in evaluations (secs)    =       0.00152

===============================================================================
```

2.6.7 C example

The same example can be implemented using the callable library. Arrays *indexList1* and *indexList2* are used to specify the list of complementarities and the `KTR_addcompcons()` function is called to register the list.

```
#include <stdio.h>
#include <stdlib.h>
#include "knitro.h"


/* callback function that evaluates the objective
   and constraints */
int  callback (const int             evalRequestCode,
                     const int             n,
                     const int             m,
                     const int             nnzJ,
                     const int             nnzH,
                     const double * const  x,
                     const double * const  lambda,
                           double * const  obj,
                           double * const  c,
                           double * const  objGrad,
```

```
                           double * const  jac,
                           double * const  hessian,
                           double * const  hessVector,
                           void   *        userParams)
{
    if (evalRequestCode == KTR_RC_EVALFC) {
        /* objective function */
        *obj = (x[0]-5)*(x[0]-5) + (2*x[1]+1)*(2*x[1]+1);

        /* constraints */
        c[0] = 2*(x[1]-1) - 1.5*x[0] + x[2] - 0.5*x[3] + x[4];
        c[1] = 3*x[0] - x[1] - 3 -x[5];
        c[2] = -x[0] + 0.5*x[1] + 4 -x[6];
        c[3] = -x[0] - x[1] + 7 - x[7];

        return(0);
    } else {
        printf ("Wrong evalRequestCode in callback function.\n");
        return(-1);
    }
}

/* main */
int  main (int  argc, char  *argv[]) {
    int  nStatus;

    /* variables that are passed to KNITRO */
    KTR_context    *kc;
    int            n, m, numCompConstraints, nnzJ, nnzH, objGoal, objType;
    int            *cType, *indexList1, *indexList2;
    int            *jacIndexVars, *jacIndexCons;
    double         obj, *x, *lambda;
    double         *xLoBnds, *xUpBnds, *xInitial, *cLoBnds, *cUpBnds;
    int            i, j, k; // convenience variables

    /*problem size and mem allocation */
    n = 8;                  /* number of variables */
    m = 4;                  /* number of regular constraints */
    numCompConstraints = 3; /* number of complementarity constraints */
    nnzJ = n*m;
    nnzH = 0;
    x      = (double *) malloc (n     * sizeof(double));
    lambda = (double *) malloc ((m+n) * sizeof(double));
    xLoBnds      = (double *) malloc (n * sizeof(double));
    xUpBnds      = (double *) malloc (n * sizeof(double));
    xInitial     = (double *) malloc (n * sizeof(double));
    cType        = (int    *) malloc (m * sizeof(int));
    cLoBnds      = (double *) malloc (m * sizeof(double));
    cUpBnds      = (double *) malloc (m * sizeof(double));
    jacIndexVars = (int    *) malloc (nnzJ * sizeof(int));
    jacIndexCons = (int    *) malloc (nnzJ * sizeof(int));
    indexList1   = (int    *) malloc (numCompConstraints * sizeof(int));
    indexList2   = (int    *) malloc (numCompConstraints * sizeof(int));

    /* objective type */
    objType = KTR_OBJTYPE_GENERAL;
    objGoal = KTR_OBJGOAL_MINIMIZE;
```

```
/* bounds and constraints type */
for (i = 0; i < n; i++) {
    xLoBnds[i] = 0.0;
    xUpBnds[i] = KTR_INFBOUND;
}
for (j = 0; j < m; j++) {
    cType[j] = KTR_CONTYPE_GENERAL;
    cLoBnds[j] = 0.0;
    cUpBnds[j] = 0.0;
}

/* complementarities */
indexList1[0] = 2;   indexList2[0] = 5;
indexList1[1] = 3;   indexList2[1] = 6;
indexList1[2] = 4;   indexList2[2] = 7;

/* sparsity pattern (here, of a full matrix) */
k = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++) {
        jacIndexCons[k] = j;
        jacIndexVars[k] = i;
        k++;
    }

/* create a KNITRO instance */
kc = KTR_new();
if (kc == NULL)
    exit( -1 ); // probably a license issue

/* set options: automatic gradient and hessian matrix */
if (KTR_set_int_param_by_name (kc, "gradopt", 3) != 0)
    exit( -1 );
if (KTR_set_int_param_by_name (kc, "hessopt", 6) != 0)
    exit( -1 );

/* register the callback function */
if (KTR_set_func_callback (kc, &callback) != 0)
    exit( -1 );

/* pass the problem definition to KNITRO */
nStatus = KTR_init_problem (kc, n, objGoal, objType,
                            xLoBnds, xUpBnds,
                            m, cType, cLoBnds, cUpBnds,
                            nnzJ, jacIndexVars, jacIndexCons,
                            nnzH, NULL, NULL, xInitial, NULL);

/* declare complementarities */
KTR_addcompcons (kc, numCompConstraints, indexList1, indexList2);

/* free memory (KNITRO maintains its own copy) */
free (xLoBnds);
free (xUpBnds);
free (xInitial);
free (cType);
free (cLoBnds);
free (cUpBnds);
free (jacIndexVars);
```

```
    free (jacIndexCons);
    free (indexList1);
    free (indexList2);

    /* solver call */
    nStatus = KTR_solve (kc, x, lambda, 0, &obj,
        NULL, NULL, NULL, NULL, NULL, NULL);

    if (nStatus != 0)
        printf ("\nKNITRO failed to solve the problem, final status = %d\n",
                nStatus);
    else
        printf ("\nKNITRO successful, objective is = %e\n", obj);

    /* delete the KNITRO instance and primal/dual solution */
    KTR_free (&kc);
    free (x);
    free (lambda);

    getchar();
    return( 0 );
}
```

Running this code produces an output similar to what we obtained with AMPL for the same problem. More function evaluations are made since, for simplicity, we did not provide first derivatives and failed to notify KNITRO that the constraints are linear. The final objective value is however the same:

```
KNITRO successful, objective is = 1.700000e+001
```

2.7 Algorithms

KNITRO implements four state-of-the-art interior-point and active-set methods for solving continuous, nonlinear optimization problems. Each algorithm possesses strong convergence properties and is coded for maximum efficiency and robustness. However, the algorithms have fundamental differences that lead to different behavior on nonlinear optimization problems. Together, the four methods provide a suite of different ways to attack difficult problems.

We encourage the user to try all algorithmic options to determine which one is more suitable for the application at hand.

2.7.1 Overview

This section only describes the four algorithms implemented in KNITRO in very broad terms. For details, please see the *Bibliography*.

- Interior/Direct algorithm

  Interior-point methods (also known as barrier methods) replace the nonlinear programming problem by a series of barrier subproblems controlled by a barrier parameter. Interior-point methods perform one or more minimization steps on each barrier subproblem, then decrease the barrier parameter and repeat the process until the original problem has been solved to the desired accuracy. The Interior/Direct method computes new iterates by solving the primal-dual KKT matrix using direct linear algebra. The method may temporarily switch to the Interior/CG algorithm, described below, if it encounters difficulties.

- Interior/CG algorithm

This method is similar to the Interior/Direct algorithm. It differs mainly in the fact that the primal-dual KKT system is solved using a projected conjugate gradient iteration. This approach differs from most interior-point methods proposed in the literature. A projection matrix is factorized and the conjugate gradient method is applied to approximately minimize a quadratic model of the barrier problem. The use of conjugate gradients on large-scale problems allows KNITRO to utilize exact second derivatives without explicitly forming or storing the Hessian matrix.

- Active Set algorithm

  Active set methods solve a sequence of subproblems based on a quadratic model of the original problem. In contrast with interior-point methods, the algorithm seeks active inequalities and follows a more exterior path to the solution. KNITRO implements a sequential linear-quadratic programming (SLQP) algorithm, similar in nature to a sequential quadratic programming method but using linear programming subproblems to estimate the active set. This method may be preferable to interior-point algorithms when a good initial point can be provided; for example, when solving a sequence of related problems. KNITRO can also "crossover" from an interior-point method and apply Active Set to provide highly accurate active set and sensitivity information.

- Sequential Quadratic Programming (SQP) algorithm

  The SQP method in KNITRO is an active-set method that solves a sequence of quadratic programming (QP) subproblems to solve the problem. This method is primarily designed for small to medium scale problems with expensive function evaluations – for example, problems where the function evaluations involve performing expensive black-box simulations and/or derivatives are computed via finite-differencing. The SQP iteration is expensive since it involves solving a QP subproblem. However, it often converges in the fewest number of function/gradient evaluations, which is why this method is often preferable for situations where the evaluations are the dominant cost of solving the model.

Note: For mixed integer programs (MIPs), KNITRO provides two variants of the branch and bound algorithm that rely on the previous four algorithms to solve the continuous (relaxed) subproblems. The first is a standard branch and bound implementation, while the second is specialized for convex, mixed integer nonlinear problems.

2.7.2 Algorithm choice

- Automatic

  By default, KNITRO automatically tries to choose the best algorithm for a given problem based on problem characteristics.

  However, we strongly encourage you to experiment with all the algorithms as it is difficult to predict which one will work best on any particular problem.

- Interior/Direct

  This algorithm often works best, and will automatically switch to Interior/CG if the direct step is suspected to be of poor quality, or if negative curvature is detected. Interior/Direct is recommended if the Hessian of the Lagrangian is ill-conditioned. The Interior/CG method in this case will often take an excessive number of conjugate gradient iterations. It may also work best when there are dependent or degenerate constraints. Choose this algorithm by setting user option `algorithm` = 1.

  We encourage you to experiment with different values of the bar_murule option when using the Interior/Direct or Interior/CG algorithm. It is difficult to predict which update rule will work best on a problem.

  Note: Since the Interior/Direct algorithm in KNITRO requires the explicit storage of a Hessian matrix, this algorithm only works with Hessian options (`hessopt`) 1, 2, 3, or 6. It may not be used with Hessian options 4 or 5 (where only Hessian-vector products are performed) since they do not supply a full Hessian matrix.

- Interior/CG

  This algorithm is well-suited to large problems because it avoids forming and factorizing the Hessian matrix. Interior/CG is recommended if the Hessian is large and/or dense. It works with all Hessian options. Choose this algorithm by setting user option `algorithm` = 2.

  We encourage you to experiment with different values of the `bar_murule` *option when using the Interior/Direct or Interior/CG algorithm. It is difficult to predict which update rule will work best on a problem.*

- Active Set:

  This algorithm is fundamentally different from interior-point methods. The method is efficient and robust for small and medium-scale problems, but is typically less efficient than the Interior/Direct and Interior/CG algorithms on large-scale problems (many thousands of variables and constraints). Active Set is recommended when "warm starting" (i.e., when the user can provide a good initial solution estimate, for example, when solving a sequence of closely related problems). This algorithm is also best at rapid detection of infeasible problems. Choose this algorithm by setting user option `algorithm` = 3.

- SQP

  This algorithm is best suited to small problems where the function and derivative evaluations are the dominant cost. Like the active-set method above, this method can converge quickly when a good initial solution estimate is provided.

  Choose this algorithm by setting user option `algorithm` = 4.

  Note: Since the SQP algorithm in KNITRO currently requires the explicit storage of a Hessian matrix, this algorithm only works with Hessian options (`hessopt`) 1, 2, 3, or 6. It may not be used with Hessian options 4 or 5 (where only Hessian-vector products are performed) since they do not supply a full Hessian matrix.

- Multi Algorithm:

  This option runs all four algorithms above either sequentially or in parallel. It can be selected by setting user option `algorithm` = 5 and is explained in more detail below.

2.7.3 Multiple algorithms

Setting user option `algorithm` = 5 (`KTR_ALG_MULTI`), allows you to easily run all four KNITRO algorithms. The algorithms will run either sequentially or in parallel depending on the setting of `par_numthreads` (see *Parallelism*).

The user option `ma_terminate` controls how to terminate the multi-algorithm ("ma") procedure. If `ma_terminate` = 0, the procedure will run until all four algorithms have completed (if multiple optimal solution are found, KNITRO will return the one with the best objective value). If `ma_terminate` = 1, the procedure will terminate as soon as the first local optimal solution is found. If `ma_terminate` = 2, the procedure will stop at the first feasible solution estimate. If `ma_terminate` = 3, the procedure will stop as soon as any of the algorithms terminate for any reason. If you are not sure which algorithm works best for your application, a recommended strategy is to set `algorithm` = 5 with `ma_terminate` = 1 (this is particularly advantageous if it can be done in parallel).

The user options `ma_maxtime_cpu` and `ma_maxtime_real` place overall time limits on the total multi-algorithm procedure while the options `maxtime_cpu` and `maxtime_real` impose time limits for each algorithm solve.

The output from each algorithm can be written to a file named `knitro_ma_x.log` where "x" is the algorithm number by setting the option `ma_outsub` =1.

2.7.4 Crossover

Interior-point (or barrier) methods are a powerful tool for solving large-scale optimization problems. However, one drawback of these methods is that they do not always provide a clear picture of which constraints are active at the solution. In general they return a less exact solution and less exact sensitivity information. For this reason, KNITRO offers a *crossover* feature in which the interior-point method switches to the Active Set method at the interior-point solution estimate, in order to "clean up" the solution and provide more exact sensitivity and active set information.

The crossover procedure is controlled by the `bar_maxcrossit` user option. If this parameter is greater than 0, then KNITRO will attempt to perform `bar_maxcrossit` Active Set crossover iterations after the interior-point method has finished, to see if it can provide a more exact solution. This can be viewed as a form of post-processing. If `bar_maxcrossit` is not positive, then no crossover iterations are attempted.

The crossover procedure will not always succeed in obtaining a more exact solution compared with the interior-point solution. If crossover is unable to improve the solution within `bar_maxcrossit` crossover iterations, then it will restore the interior-point solution estimate and terminate. If `outlev` is greater than one, KNITRO will print a message indicating that it was unable to improve the solution. For example, if `bar_maxcrossit` = 3 and the crossover procedure did not succeed, the message will read:

```
Crossover mode unable to improve solution within 3 iterations.
```

In this case, you may want to increase the value of `bar_maxcrossit` and try again. If KNITRO determines that the crossover procedure will not succeed, no matter how many iterations are tried, then a message of the form

```
Crossover mode unable to improve solution.
```

will be printed.

The extra cost of performing crossover is problem dependent. In most small or medium scale problems, the crossover cost is a small fraction of the total solve cost. In these cases it may be worth using the crossover procedure to obtain a more exact solution. On some large scale or difficult degenerate problems, however, the cost of performing crossover may be significant. It is recommended to experiment with this option to see whether improvement in the exactness of the solution is worth the additional cost.

2.8 Feasibility and infeasibility

This section deals with the issue of infeasibility or inability to converge to a feasible solution, and with options offered by KNITRO to ensure that the iterates taken from the initial points to the solution remain feasible. This can be useful when, for instance, certain functions are not defined outside a given domain and the user wants to prevent the algorithm from evaluating these functions at certains points.

2.8.1 Infeasibility

KNITRO is a solver for finding *local* solutions to general nonlinear, possibly nonconvex problems. Just as KNITRO may converge to a local solution that is not the global solution, it is also possible for a nonlinear optimization solver to converge to a *locally* infeasible point or *infeasible* stationary point on nonconvex problems. That is, even if the user's model is feasible, a nonlinear solver can converge to a point where the model is locally infeasible. At this point, a move in any direction will increase some measure of infeasibility and thus a local solver cannot make any further progress from such a point. Just as only a global optimization solver can guarantee that it will locate the globally optimal solution, only a global solver can also avoid the possibility of converging to these locally infeasible points.

If your problem is nonconvex and the KNITRO termination message indicates that it has converged to an infeasible point, then you should try running KNITRO again from a different starting point (preferably one close to the feasible region). Alternatively, you can use the KNITRO multi-start feature which will automatically try to run KNITRO several times from different starting points, to try to avoid getting stuck at locally infeasible points.

If you are using one of the interior-point algorithms in KNITRO, and KNITRO is struggling to find a feasible point, you can try different settings for the `bar_feasible` user option to place special emphasis on obtaining feasibility, as follows.

2.8.2 Feasibility options

KNITRO offers an option `bar_feasible` that can force iterates to stay feasible with respect to inequality constraints or can place special emphasis on trying to get feasible.

If `bar_feasible` = 1 or `bar_feasible` = 3 KNITRO will seek to generate iterates that satisfy the inequalities by switching to a *feasible mode* of operation, which alters the manner in which iterates are computed. The option does not enforce feasibility with respect to *equality* constraints, as this would impact performance too much.

In order to enter feasible mode, the initial point must satisfy all the inequalities to a sufficient degree; if not, KNITRO may generate infeasible iterates and does not switch to the feasible mode until a sufficiently feasible point is found (with respect the the inequalities). We say *sufficient* satisfaction occurs at a point *x* if it is true for all inequalities that:

$$c^L + tol \leq c(x) \leq c^U - tol$$

The constant *tol*>0 is determined by the option `bar_feasmodetol`; its default value is 1.0e-4. Feasible mode becomes active once an iterate *x* satisfies this condition for all inequality constraints. If the initial point satisfies this condition, then every iterate will be feasible with respect to the inequalities.

KNITRO can also place special emphasis on *getting* feasible (with respect to all constraints) through the option `bar_feasible`. If `bar_feasible` = 2 or `bar_feasible` = 3, KNITRO will first place special emphasis on getting feasible before working on optimality. This option is not always guaranteed to accelerate the finding of a feasible point. However, it may do a better job of obtaining feasibility on difficult problems where the default version struggles.

Note: This option can only be used with the Interior/Direct and Interior/CG algorithms.

2.8.3 Honor bounds mode

In some applications, the user may want to enforce that the initial point and all subsequent iterates satisfy the simple bounds:

$$b^L \leq x \leq b^U.$$

For instance, if the objective function or a nonlinear constraint function is undefined at points outside the bounds, then the bounds should be enforced at all times.

By default, KNITRO enforces bounds on the variables only for the initial start point and the final solution (`honorbnds` = 2). To enforce satisfaction at all iterates, set `honorbnds` = 1. To allow execution from an initial point that violates the bounds, set `honorbnds` = 0.

2.9 Parallelism

KNITRO offers several features to exploit parallel computations on shared memory multi-processor machines. These features are implemented using OpenMP.

Note: The parallel features offered through KNITRO are not available through all interfaces. Check with your modeling language vendor to see if these features are included. The parallel features are included in the AMPL

interface and through the callable library (provided you use the *callback* mode). Parallel features are also available through the MATLAB interface, but some may be less efficient in this environment.

Note: Parallel features in KNITRO require use of the *callback* interface (with the exception of parallel BLAS and the parallel MKL PARDISO linear solver, which can be used in either *callback* or *reverse communication* mode).

KNITRO offers the following parallel features:

2.9.1 Parallel Finite-Difference Gradients

As described in *Derivatives*, if you are unable to provide the exact first derivatives, KNITRO offers the option to approximate first derivatives using either a forward or central finite-difference approach, by setting the option `gradopt`. KNITRO will compute these finite difference gradient values in parallel if the user specifies that KNITRO should use multiple threads through the option `par_numthreads` (see below). This parallel feature only applies to first derivative finite-difference evaluations.

Note: In the KNITRO-MATLAB interface, the parallel finite-difference feature is controlled by the *UseParallel* MATLAB option, rather than the KNITRO `par_numthreads` option. See *KNITRO / MATLAB reference* for more information.

2.9.2 Parallel Multistart

The multistart procedure described in *Multistart* can run in parallel by setting `par_numthreads` to use multiple threads.

When the multistart procedure is run in parallel, KNITRO will produce the same sequence of initial points and solves that you see when running multistart sequentially (though, perhaps, not in the same order).

Therefore, as long as you run multistart to completion (`ms_terminate` =0) and use the deterministic option (`ms_deterministic` =1), you should visit the same initial points encountered when running multistart sequentially, and get the same final solution. By default `ms_terminate` =0 and `ms_deterministic` =1 so that the parallel multistart produces the same solution as the sequential multistart.

However, if `ms_deterministic` =0, or `ms_terminate` >0, there is no guarantee that the final solution reported by multistart will be the same when run in parallel compared to the solution when run sequentially, and even the parallel solution may change when run at different times.

2.9.3 Parallel Algorithms

If the user option `alg` is set to `multi`, then KNITRO will run all four algorithms (see *Algorithms*). When `par_numthreads` is set to use multiple threads, the four KNITRO algorithms will run in parallel. The termination of the parallel algorithms procedure is controlled by the user option `ma_terminate`. See *Algorithms* for more details on the multi algorithm procedure.

2.9.4 Parallel Basic Linear Algebra Subroutine (BLAS)

The KNITRO algorithms - in particular the interior-point/barrier algorithms - rely heavily on BLAS operations (e.g. dot products of vectors, dense matrix-matrix and matrix-vector products, etc.). For large-scale problems, these operations may often take 35%-50% of the overall solution time, and sometimes more.

These operations can be computed in parallel using multiple threads by setting the user option par_blasnumthreads >1 (by default par_blasnumthreads =1). This option is currently only active when using the default Intel BLAS (blasoption =1) provided with KNITRO.

2.9.5 Parallel Sparse Linear System Solves

The primary computational cost each iteration in the KNITRO interior-point algorithms is the solution of a linear system of equations. The linsolver user option specifies the linear system, solver to use. You can use the multi-threaded Intel MKL PARDISO solver in KNITRO by choosing linsolver =6. By default the Intel MKL PARDISO solver will use one thread, however, it can solve linear systems in parallel by choosing par_lsnumthreads >1 (in combination with linsolver =6).

Note: Generally you should not use BOTH parallel BLAS and a parallel linear solver as they may conflict with each other. If par_blasnumthreads >1 one should set par_lsnumthreads =1 and vise versa.

2.9.6 Parallel Options

Option	Meaning
par_blasnumthreads	Specifies the number of threads to use for parallel BLAS (when blasoption =1)
par_lsnumthreads	Specifies the number of threads to use for parallel linear system solves (when linsolver =6)
par_numthreads	Specifies the number of threads to use for all parallel features
par_concurrent_evals	Whether or not to allow concurrent evaluations

The user option par_blasnumthreads is used to determine the number of threads KNITRO can use for parallel BLAS computations. This option is only active when using the default Intel BLAS (blasoption =1). The domain specific par_blasnumthreads, will override the general thread setting specified by par_numthreads for BLAS operations.

The user option par_lsnumthreads is used to determine the number of threads KNITRO can use for parallel linear system solves. This option is only active when using the default Intel MKL PARDISO linear solver (linsolver =6). The domain specific par_lsnumthreads, will override the general thread setting specified by par_numthreads for linear system solve operations.

The user option par_numthreads is used to determine the number of threads KNITRO can use for all parallel computations. KNITRO will decide how to apply the threads. If par_numthreads > 0, then the number of threads is determined by the value of par_numthreads. If par_numthreads = 0, then the number of threads is determined by the value of the environment variables OMP_NUM_THREADS. If par_numthreads = 0 and OMP_NUM_THREADS is not set, then the number of threads to use will be automatically deteremined by OpenMP. If par_numthreads < 0, KNITRO will run in sequential mode.

The user option par_concurrent_evals determines whether or not the user provided callback functions used for function and derivative evaluations can take place concurrently in parallel (for possibly different values of "x"). If it is not safe to have concurrent evaluations, then setting par_concurrent_evals =0, will put these evaluations in a critical region so that only one evaluation can take place at a time. If par_concurrent_evals =1 then concurrent evaluations are allowed when KNITRO is run in parallel, and it is the responsibility of the user to ensure that these evaluations are stable.

Preventing concurrent evaluations will decrease the efficiency of the parallel features, particularly when the evaluations are expensive or there are many threads and these evaluations create a bottleneck.

2.9.7 AMPL example

Let us consider again our AMPL example from Section *Getting started with AMPL* and run it with the parallel multi algorithm procedure. We specify that KNITRO should run in parallel with four threads (one for each algorithm):

```
ampl: reset;
ampl: option solver knitroampl;
ampl: option knitro_options "alg=5 ma_terminate=0 par_numthreads=4";
ampl: model testproblem.mod;
ampl: solve;
```

The KNITRO log printed to the screen shows the results of each algorithm (one per line):

```
======================================
       Commercial Ziena License
             KNITRO 9.1.0
          Ziena Optimization
======================================

KNITRO presolve eliminated 0 variables and 0 constraints.

algorithm:            5
hessian_no_f:         1
ma_terminate:         0
par_concurrent_evals: 0
par_numthreads:       4

Problem Characteristics
-----------------------
Objective goal:  Minimize
Number of variables:                    3
    bounded below:                      3
    bounded above:                      0
    bounded below and above:            0
    fixed:                              0
    free:                               0
Number of constraints:                  2
    linear equalities:                  1
    nonlinear equalities:               0
    linear inequalities:                0
    nonlinear inequalities:             1
    range:                              0
Number of nonzeros in Jacobian:         6
Number of nonzeros in Hessian:          5

KNITRO running multiple algorithms in parallel with 4 threads.

   Alg      Status    Objective      FeasError   OptError   Real Time
--------  --------- -------------  ----------  ---------- ----------
       2         0    9.360000e+02   0.000e+00   1.945e-07      0.001
       1         0    9.360000e+02   6.738e-08   6.614e-08      0.002
       3         0    9.360000e+02   0.000e+00   0.000e+00      0.005
       4         0    9.360000e+02   0.000e+00   0.000e+00      0.012
Multiple algorithms stopping, all solves have completed.

EXIT: Locally optimal solution found.

Final Statistics
----------------
```

```
Final objective value               =   9.35999997829394e+02
Final feasibility error (abs / rel) =   6.74e-08 / 5.18e-09
Final optimality error  (abs / rel) =   6.61e-08 / 4.13e-09
# of iterations                     =         16
# of CG iterations                  =         12
# of function evaluations           =         24
# of gradient evaluations           =         20
# of Hessian evaluations            =         16
Total program time (secs)           =       0.01480 (     0.026 CPU time)

================================================================================

KNITRO 9.1.0: Locally optimal solution.
objective 935.9999978; feasibility error 6.74e-08
16 iterations; 24 function evaluations
```

As can be seen, all four KNITRO algorithms solve the problem and find the same local solution. However, the two interior-point algorithms (`alg=1` and `2`) are the fastest.

2.9.8 C example

As an example, the C example can also be easily modified to enable parallel multi-algorithms by adding the following lines before the call to `KTR_init_problem()`:

```
// parallelism
if (KTR_set_int_param_by_name (kc, "algorithm", KTR_ALG_MULTI) != 0)
exit( -1 );
if (KTR_set_int_param_by_name (kc, "ma_terminate", 0) != 0)
exit( -1 );
if (KTR_set_int_param_by_name (kc, "par_numthreads", 4) != 0)
exit( -1 );
```

Again, running this example we get a KNITRO log that looks simlar to what we observed with AMPL.

2.10 The KNITRO-Tuner

The KNITRO-Tuner can help you identify some non-default options settings that may improve performance on a particular model or set of models. This section desribes how to use the KNITRO-Tuner.

2.10.1 Default Tuning

If you are unsure about what KNITRO options should be tuned to try to improve performance, then you can simply run the default KNITRO-Tuner by setting the option `tuner` =1, when running KNITRO on your model. This will cause KNITRO to automatically run your model with a variety of automatically determined option settings, and report some statistics at the end. Any KNITRO options that have been set in the usual way will remain fixed throughout the tuning procedure.

2.10.2 Custom Tuning

If you have some ideas about which KNITRO options you want to tune, then you can tell KNITRO which options you want it to tune (as well as specify the values for particular options that you want KNITRO to explore). This can be

done by specifying a Tuner options file. A Tuner options file is a simple text file that is similar to a standard KNITRO options file (see *Setting options* for details on how to define a standard KNITRO options file).

A Tuner options file differs from a standard KNITRO options file in a few ways:

1. You can define multiple values (separated by spaces) for each option. This tells KNITRO the values you want it to explore.
2. You can specify an option name without any values. This will tell KNITRO to explore all possible option values for that option. This only works for options that have a finite set of possible option value settings.
3. A Tuner options file is loaded through the API function `KTR_load_tuner_file()` if using the callable library API (procedures for loading a Tuner options file for other environments are demonstrated in the examples below).

All possible combinations of options/values specified in a Tuner options file will be explored by KNITRO, while any KNITRO options that have been set in the usual way will remain fixed throughout the tuning procedure.

An example of using the KNITRO-Tuner and defining a Tuner options file is provided in `examples/C` in the KNITRO distribution. Below is the Tuner options file from that example.

```
# This file is used to specify the options and option values
# that will be systematically explored by the KNITRO-Tuner
# in "tunerExample.c".  One can specify the specific option
# values to be explored by a particular option (as with
# "bar_directinterval" and "pivot" below). If just the option
# name is listed (as with "algorithm" and "bar_murule"),
# then all values for that option will be explored (only for
# options that have a finite number of integer values).
# http://ziena.com/documentation.html

algorithm
bar_directinterval 0 1 10
bar_murule
pivot              1e-8 1e-14
```

This options file tells the KNITRO-Tuner to explore all possible option values for the `algorithm` and `bar_murule` options, while exploring three values (0, 1 and 10) for the `bar_directinterval` option and two values (1e-8 and 1e-14) for the `pivot` option.

2.10.3 Tuner Output

The Tuner output, by default, provides a summary line of output for each solve during the tuning process indicating the results of that particular solve. When the Tuner completes all solves, it reports the non-default option settings for the fastest solve. Perhaps more insightful, however, is a summary table of statistics provided by the Tuner at the end of the solve. For example, in the example provided in `examples/C`, we may see something like this:

```
Summary Statistics
-------------------------------------------------------------------------
                                          Percent   Average     Average
    Option Name          Value      #Runs  Optimal  #FuncEvals     Time
--------------------  ----------  --------  -------  ----------  ----------
  bar_directinterval           0        24   100.00        12.2       0.001
  bar_directinterval           1        24   100.00         7.9       0.001
  bar_directinterval          10        24   100.00         7.9       0.001
--------------------  ----------  --------  -------  ----------  ----------
          bar_murule           1        12   100.00         8.7       0.001
          bar_murule           2        12   100.00         7.5       0.001
          bar_murule           3        12   100.00         9.7       0.001
```

```
          bar_murule           4        12   100.00         9.5       0.001
          bar_murule           5        12   100.00        10.3       0.001
          bar_murule           6        12   100.00        10.5       0.001
--------------------  ----------  --------  -------  ----------  ----------
               pivot    1.00e-08        38   100.00         9.1       0.001
               pivot    1.00e-14        38   100.00         9.1       0.001
--------------------  ----------  --------  -------  ----------  ----------
           algorithm           1        36   100.00        12.7       0.002
           algorithm           2        36   100.00         6.0       0.001
           algorithm           3         2   100.00         5.0       0.002
           algorithm           4         2   100.00         3.0       0.011
-------------------------------------------------------------------------
```

This table indicates the option values explored, the number of Tuner runs for each option value, the percentage of those runs where it found an optimal solution, the average number of function evaluations (in the cases where it found an optimal solution), and the average time (in the cases where it found an optimal solution). In this particular example, the model tested is very small, so the solution times are generally near 0.

This summary table provides some global view of which option settings may be preferable. For example, the table above suggests that `algorithm` =2 may be preferable for models of this type since it (on average) requires a little less time to find an optimal solution. Although if function evaluations were the dominant cost, then `algorithm` =4 might be preferable. The table also suggests that perhaps the non-default setting `bar_murule` =2 should be used, since it requires, on average, the fewest number of function evaluations to converge, although other values are only slightly worse.

More detailed output can be obtained through non-default settings of `tuner_outsub`. In particular, if `tuner_outsub` =1, then a summary file called `knitro_tuner_summary.log` is created in the current folder/directory. Each line of this file shows the option settings used and the summary results with these settings. Additionally, if `tuner_outsub` =2, the individual output file for each tuner solve is created in a file called `knitro_tuner_*.log`, where * is the corresponding solve number.

2.10.4 Tuner Options

The following options may be used to customize the performance of the KNITRO-Tuner.

Option	Meaning
`tuner`	Enable Tuner
`tuner_maxtime_cpu`	Maximum CPU time for Tuner, in seconds
`tuner_maxtime_real`	Maximum real time for Tuner, in seconds
`tuner_optionsfile`	Specify location/name of Tuner options file
`tuner_outsub`	Output additional Tuner subproblem solve information
`tuner_terminate`	Termination condition for Tuner

The following examples show how to load a Tuner options file in various environments.

2.10.5 AMPL example

When using KNITRO/AMPL, you can specify the location/name of a Tuner options file through the `tuner_optionsfile` option as shown below.

```
ampl: option knitro_options "tuner=1 tuner_optionsfile='tuner-explore.opt'";
```

2.10.6 MATLAB example

In KNITRO/MATLAB, the only way to enable the KNITRO-Tuner and specify the location of a Tuner options file is through a standard KNITRO options file. For example, the following KNITRO options file, passed as the last argument to *knitromatlab* would enable the Tuner and load the Tuner options file `tuner-explore.opt` assumed to exist in the current folder/directory.

```
# Example KNITRO options file used to enable the Tuner
# and load a Tuner options file in KNITRO/MATLAB.

tuner 1
tuner_optionsfile  tuner-explore.opt
```

2.10.7 C example

In the callable library interface, a Tuner options file can be loaded through the `KTR_load_tuner_file()` API function.

```
/*---- TURN ON THE KNITRO-TUNER */
if (KTR_set_int_param (kc, KTR_PARAM_TUNER, KTR_TUNER_ON) != 0)
    exit( -1 );

/*---- LOAD TUNER OPTIONS FILE "tuner-explore.opt". */
if (KTR_load_tuner_file (kc, "tuner-explore.opt") != 0)
    exit( -1 );
```

2.11 Termination criteria

This section describes the stopping tests used by KNITRO to declare (local) optimality, and the corresponding user options that can be used to enforce more or less stringent tolerances in these tests.

2.11.1 Continuous problems

The first-order conditions for identifying a locally optimal solution are:

$$\nabla_x \mathcal{L}(x,\lambda) = \nabla f(x) + \sum_{i=0}^{m-1} \lambda_i^c \nabla c_i(x) + \sum_{j=0}^{n-1} \lambda_j^b = 0 \tag{1}$$

$$\lambda_i^c \min[(c_i(x) - c_i^L), (c_i^U - c_i(x))] = 0, \quad i = 0, \ldots, m-1 \tag{2}$$

$$\lambda_j^b \min[(x_j - b_j^L), (b_j^U - x_j)] = 0, \quad j = 0, \ldots, n-1 \tag{2b}$$

$$c_i^L \ \leq \ c_i(x) \leq c_i^U, \quad i = 0, \ldots, m-1 \tag{3}$$

$$b_j^L \ \leq \ x_j \leq b_j^U, \quad j = 0, \ldots, n-1 \tag{3b}$$

$$\lambda_i^c \geq 0, \quad i \in \mathcal{I},\ c_i^L \text{ infinite },\ c_i^U \text{ finite} \tag{4}$$

$$\lambda_i^c \leq 0, \quad i \in \mathcal{I},\ c_i^U \text{ infinite },\ c_i^L \text{ finite} \tag{4b}$$

$$\lambda_j^b \geq 0, \quad j \in \mathcal{B},\ b_j^L \text{ infinite },\ b_j^U \text{ finite} \tag{5}$$

$$\lambda_j^b \leq 0, \quad j \in \mathcal{B},\ b_j^U \text{ infinite },\ b_j^L \text{ finite }. \tag{5b}$$

Here $\mathcal{I}$ and $\mathcal{B}$ represent the sets of indices corresponding to the general inequality constraints and (non-fixed) variable bound constraints respectively. In the conditions above, λ_i^c is the Lagrange multiplier corresponding to constraint

$c_i(x)$, and λ_j^b is the Lagrange multiplier corresponding to the simple bounds on the variable x_j. There is exactly one Lagrange multiplier for each constraint and variable. The Lagrange multiplier may be restricted to take on a particular sign depending on whether the corresponding constraint (or variable) is upper bounded or lower bounded, as indicated by (4)-(5). If the constraint (or variable) has both a finite lower and upper bound, then the appropriate sign of the multiplier depends on which bound (if either) is binding (active) at the solution.

In KNITRO we define the feasibility error *FeasErr* at a point x^k to be the maximum violation of the constraints (3), (3b), i.e.,

$$\text{FeasErr} = \max_{i=0,\ldots,m-1, j=0,\ldots,n-1} (0, (c_i^L - c_i(x^k)), (c_i(x^k) - c_i^U), (b_j^L - x_j^k), (x_j^k - b_j^U)),$$

while the optimality error (*OptErr*) is defined as the maximum violation of the first three conditions (1)-(2b).

The remaining conditions on the sign of the multipliers (4)-(5b) are enforced explicitly throughout the optimization.

In order to take into account problem scaling in the termination test, the following scaling factors are defined

$$\tau_1 = \max(1, (c_i^L - c_i(x^0)), (c_i(x^0) - c_i^U), (b_j^L - x_j^0), (x_j^0 - b_j^U)),$$
$$\tau_2 = \max(1, \|\nabla f(x^k)\|_\infty)$$

where x^0 represents the initial point.

For unconstrained problems, the scaling factor τ_2 is not effective since $\|\nabla f(x^k)\|_\infty \to 0$ as a solution is approached. Therefore, for unconstrained problems only, the following scaling is used in the termination test

$$\hat{\tau}_2 = \max(1, \min(|f(x^k)|, \|\nabla f(x^0)\|_\infty))$$

in place of τ_2.

KNITRO stops and declares *locally optimal solution found* if the following stopping conditions are satisfied:

$$\text{FeasErr} \leq \min(\tau_1 * \texttt{feastol}, \texttt{feastol\_abs}) \quad \text{(stop1)}$$
$$\text{OptErr} \leq \min(\tau_2 * \texttt{opttol}, \texttt{opttol\_abs}) \quad \text{(stop2)}$$

where `feastol`, `opttol`, `feastol_abs`, and `opttol_abs` are constants defined by user options.

Note: Please be aware that the *min* function in *(stop1)-(stop2)* was a *max* function in versions of KNITRO previous to KNITRO 9.0, and the default values for the user option tolerances were also changed. The changes were made to prevent cases where KNITRO might declare optimality with very large absolute errors (but small relative errors), or incorrectly declare optimality on unbounded models.

This stopping test is designed to give the user much flexibility in deciding when the solution returned by KNITRO is accurate enough. By default, KNITRO uses a scaled stopping test, while also enforcing that some minimum absolute tolerances for feasibility and optimality are satisfied. One can use a purely absolute stopping test by setting `feastol_abs` <= `feastol` and `opttol_abs` <= `opttol`.

Note that the optimality conditions *(stop2)* apply to the problem being solved internally by KNITRO. If the user option `scale` is set to *yes* then the problem objective and constraint functions may first be scaled before the problem is sent to KNITRO for the optimization. In this case, the optimality conditions apply to the scaled form of the problem. If the accuracy achieved by KNITRO with the default settings is not satisfactory, the user may either decrease the tolerances described above, or try setting `scale` = *no*.

Complementarity constraints

The feasibility error for a complementarity constraint is measured as $\min(x_1, x_2)$ where x_1 and x_2 are non-negative variables that are complementary to each other. The tolerances defined by *(stop1)* are used for determining feasibility of complementarity constraints.

Constraint specific feasibility tolerances

By default KNITRO applies the same feasibility stopping tolerances `feastol` / `feastol_abs` to all constraints. However, it is possible for you to define an (absolute) feasibility tolerance for each individual constraint in case you want to customize how feasible the solution needs to be with respect to each individual constraint.

This can be done using the callable library API function `KTR_set_feastols()`, which allows you to define an array of tolerances for the general constraints, the variable bounds and any complementarity constraints. Please see section *KNITRO API* for more details on this API function. When using the AMPL modeling language, the same feature can be used by defining the AMPL input suffixes *cfeastol* and *xfeastol* for each constraint or variable in your model.

2.11.2 Discrete or mixed integer problems

Algorithms for solving optimization problems where one or more of the variables are restricted to take on only discrete values, proceed by solving a sequence of continuous relaxations, where the discrete variables are *relaxed* such that they can take on any continuous value.

The best *global* solution of these relaxed problems, $f(x_R)$, provides a lower bound on the optimal objective value for the original problem (upper bound if maximizing). If a feasible point is found for the relaxed problem that satisfies the discrete restrictions on the variables, then this provides an upper bound on the optimal objective value of the original problem (lower bound if maximizing). We will refer to these feasible points as *incumbent* points and denote the objective value at an incumbent point by $f(x_I)$. Assuming all the continuous subproblems have been solved to global optimality (if the problem is convex, all local solutions are global solutions), an optimal solution of the original problem is verified when the lower bound and upper bound are equal.

KNITRO declares optimality for a discrete problem when the gap between the best (i.e., largest) lower bound $f^*(x_R)$ and the best (i.e., smallest) upper bound $f^*(x_I)$ is less than a threshold determined by the user options, `mip_integral_gap_abs` and `mip_integral_gap_rel`. Specifically, KNITRO declares optimality when either

$$f^*(x_I) - f^*(x_R) \leq \texttt{mip\_integral\_gap\_abs},$$

or

$$f^*(x_I) - f^*(x_R) \leq \texttt{mip\_integral\_gap\_rel} * \max(1, |f^*(x_I)|),$$

where `mip_integral_gap_abs` and `mip_integral_gap_rel` are typically small positive numbers.

Since these termination conditions assume that the continuous subproblems are solved to global optimality and KNITRO only finds local solutions of nonconvex, continuous optimization problems, they are only reliable when solving convex, mixed integer problems. The integrality gap $f^*(x_I) - f^*(x_R)$ should be non-negative although it may become slightly negative from roundoff error, or if the continuous subproblems are not solved to sufficient accuracy. If the integrality gap becomes largely negative, this may be an indication that the model is nonconvex, in which case KNITRO may not converge to the optimal solution, and will be unable to verify optimality (even if it claims otherwise).

2.12 Obtaining information

In addition to the KNITRO log that is printed on screen, information about the computation performed by KNITRO is available in the form of various function calls. This section explains how this information can be retrieved and interpreted.

2.12.1 KNITRO output for continuous problems

This section describes KNITRO outputs at various levels for continuous problems. We examine the output that results from running `examples/C/callbackExample2.c` with full output to solve `problemHS15.c`.

Note: If `outlev=0` then all printing of output is suppressed. If `outlev` is positive, then KNITRO prints information about the solution of your optimization problem either to standard output (`outmode` = *screen*), to a file named `knitro.log` (`outmode` = *file*), or to both (`outmode` = *both*). The option `outdir` controls the directory where output files are created (if any are) and the option `outappend` controls whether output is appended to existing files.

Display of Nondefault Options

KNITRO first prints the banner displaying the Ziena license type and version of KNITRO that is installed. It then lists all user options which are different from their default values If nothing is listed in this section then it must be that all user options are set to their default values. Lastly, KNITRO prints messages that describe how it resolved user options that were set to automatic values. For example, if option `algorithm` = *auto*, then KNITRO prints the algorithm that it chooses.

```
=====================================
       Commercial Ziena License
             KNITRO 9.1.0
          Ziena Optimization
=====================================

KNITRO presolve eliminated 0 variables and 0 constraints.

hessian_no_f:         1
outlev:               6
KNITRO changing algorithm from AUTO to 1.
KNITRO changing bar_murule from AUTO to 4.
KNITRO changing bar_initpt from AUTO to 3.
KNITRO changing bar_penaltyrule from AUTO to 2.
KNITRO changing bar_penaltycons from AUTO to 1.
KNITRO changing bar_switchrule from AUTO to 2.
KNITRO changing linsolver from AUTO to 2.
```

In the example above, it is indicated that we are using a more verbose output level (`outlev` = 6) instead of the default value (`outlev` = 2). KNITRO chose algorithm 1 (Interior/Direct), and then automatically determined some other options related to the algorithm.

Display of problem characteristics

KNITRO next prints a summary description of the problem characteristics including the number and type of variables and constraints and the number of nonzero elements in the Jacobian matrix and Hessian matrix.

```
Problem Characteristics
-----------------------
Objective goal:    Minimize
Number of variables:                  2
        bounded below:                0
        bounded above:                1
        bounded below and above:      0
        fixed:                        0
        free:                         1
```

```
Number of constraints:                2
        linear equalities:            0
        nonlinear equalities:         0
        linear inequalities:          0
        nonlinear inequalities:       2
        range:                        0
Number of nonzeros in Jacobian:       4
Number of nonzeros in Hessian:        3
```

Display of Iteration Information

Next, if `outlev` is greater than 2, KNITRO prints columns of data reflecting detailed information about individual iterations during the solution process. An iteration is defined as a step which generates a new solution estimate (i.e., a successful step).

If `outlev` = 2, summary data is printed every 10 iterations, and on the final iteration. If `outlev` = 3, summary data is printed every iteration. If `outlev` = 4, the most verbose iteration information is printed every iteration.

```
  Iter     fCount     Objective      FeasError   OptError    ||Step||    CGits
--------  --------  --------------  ----------  ----------  ----------  -------
       0         1    9.090000e+02   3.000e+00
       1         2    7.992179e+02   2.859e+00   2.191e+01   7.245e-02        0
       2         3    1.865455e+01   9.076e-01   3.917e+01   2.197e+00        0
       3        11    3.211028e+02   8.846e-01   6.751e+00   1.336e+00        8
       4        12    1.437527e+01   5.021e-01   6.570e-01   1.162e+00        2
       5        13    3.543851e+01   3.873e-01   3.873e-01   2.008e-01        0
       6        14    1.144533e+02   2.197e-01   5.820e-01   4.076e-01        0
       7        15    2.342032e+02   7.408e-02   7.408e-02   4.123e-01        0
       8        16    3.011424e+02   6.019e-03   3.302e-02   1.904e-01        0
       9        17    3.064931e+02   9.852e-06   1.490e-04   1.470e-02        0
      10        18    3.065000e+02   0.000e+00   7.038e-10   1.970e-05        0
```

The meaning of each column is described below.

- **Iter**: iteration number.
- **fCount**: the cumulative number of function evalutions. (This information is only printed if `outlev` is greater than 3).
- **Objective**: the value of the objective function at the current iterate.
- **FeasError**: a measure of the feasibility violation at the current iterate
- **OptError**: a measure of the violation of the Karush-Kuhn-Tucker (KKT) (first-order) optimality conditions (not including feasibility) at the current iterate.
- **Step**: the 2-norm length of the step (i.e., the distance between the new iterate and the previous iterate).
- **CGits**: the number of Projected Conjugate Gradient (CG) iterations required to compute the step.

Display of termination status

At the end of the run a termination message is printed indicating whether or not the optimal solution was found and if not, why KNITRO stopped. The termination message typically starts with the word "EXIT". If KNITRO was successful in satisfying the termination test, the message will look as follows:

```
EXIT: Locally optimal solution found.
```

Display of Final Statistics

Following the termination message, a summary of some final statistics on the run are printed. Both relative and absolute error values are printed.

```
Final Statistics
----------------
Final objective value               =   3.06500000082417e+02
Final feasibility error (abs / rel) =   0.00e+00 / 0.00e+00
Final optimality error  (abs / rel) =   7.04e-10 / 4.82e-11
# of iterations                     =         10
# of CG iterations                  =         10
# of function evaluations           =         18
# of gradient evaluations           =         11
# of Hessian evaluations            =         10
Total program time (secs)           =       0.00390 (     0.002 CPU time)
Time spent in evaluations (secs)    =       0.00001
```

Display of solution vector and constraints

If `outlev` equals 5 or 6, the values of the solution vector are printed after the final statistics. If `outlev` equals 6, the final constraint values are also printed, and the values of the Lagrange multipliers (or dual variables) are printed next to their corresponding constraint or bound.

```
Constraint Vector                Lagrange Multipliers
-----------------                --------------------
c[       0] =   1.00000000001e+00,   lambda[       0] =  -7.00000000020e+02
c[       1] =   4.50000000072e+00,   lambda[       1] =  -1.69345778120e-08

Solution Vector
---------------
x[       0] =   4.99999999956e-01,   lambda[       2] =   1.75099999991e+03
x[       1] =   2.00000000019e+00,   lambda[       3] =   0.00000000000e+00
```

Debugging / profiling information

KNITRO can produce additional information which may be useful in debugging or analyzing performance. If `outlev` is positive and `debug` = 1, then multiple files named `kdbg_*.log` are created which contain detailed information on performance. If `outlev` is positive and `debug` = 2, then KNITRO prints information useful for debugging program execution. The information produced by `debug` is primarily intended for developers, and should not be used in a production setting.

Intermediate iterates

Users can generate a file containing iterates and/or solution points with option `newpoint`. The output file is called `knitro_newpoint.log`.

2.12.2 KNITRO output for discrete problems

This section describes KNITRO outputs at various levels for discrete or mixed integer problems. We examine the output that results from running `examples/C/callbackMINLP_static` to solve `problemMINLP.c`.

Note: When outlev is positive, the options mip_outlevel, mip_debug, mip_outinterval and mip_outsub control the amount and type of MIP output generated as described below.

KNITRO first prints the banner displaying the Ziena license type and version of KNITRO that is installed. It then lists all user options which are different from their default values. If nothing is listed in this section then it must be that all user options are set to their default values. Lastly, KNITRO prints messages that describe how it resolved user options that were set to automatic values. For example, if option mip_branchrule = *auto*, then KNITRO prints the branching rule that it chooses.

```
=====================================
       Commercial Ziena License
             KNITRO 9.1.0
          Ziena Optimization
=====================================

mip_method: 1
mip_outinterval: 1
KNITRO changing mip_rootalg from AUTO to 1.
KNITRO changing mip_lpalg from AUTO to 3.
KNITRO changing mip_branchrule from AUTO to 2.
KNITRO changing mip_selectrule from AUTO to 2.
KNITRO changing mip_rounding from AUTO to 3.
KNITRO changing mip_heuristic from AUTO to 1.
KNITRO changing mip_pseudoinit from AUTO to 1.
```

In the example above, it is indicated that we are using mip_method = 1 which is the standard branch and bound method, and that we are printing output information at every node since mip_outinterval = 1. It then determined seven other options related to the MIP method.

Display of Problem Characteristics

KNITRO next prints a summary description of the problem characteristics including the number and type of variables and constraints and the number of nonzero elements in the Jacobian matrix and Hessian matrix (if providing the exact Hessian).

If no initial point is provided by the user, KNITRO indicates that it is computing one. KNITRO also prints the results of any MIP preprocessing to detect special structure and indicates which MIP method it is using.

```
Problem Characteristics
-----------------------
Objective goal:    Minimize
        Number of variables:                          6
        bounded below:                                0
        bounded above:                                0
        bounded below and above:                      6
        fixed:                                        0
        free:                                         0
Number of binary variables:                           3
Number of integer variables:                          0
Number of constraints:                                6
        linear equalities:                            0
        nonlinear equalities:                         0
        linear inequalities:                          4
        nonlinear inequalities:                       2
        range:                                        0
Number of nonzeros in Jacobian:                      16
Number of nonzeros in Hessian:                        3
```

```
No start point provided -- KNITRO computing one.

KNITRO detected 1 GUB constraints
KNITRO derived 0 knapsack covers after examining 3 constraints
KNITRO solving root node relaxation
KNITRO MIP using Branch and Bound method
```

Display of Node Information

Next, if mip_outlevel = 1, KNITRO prints columns of data reflecting detailed information about individual nodes during the solution process. If mip_outlevel = 2, the accumulated time is printed at each node also. The frequency of this node information is controlled by the mip_outinterval parameter. For example, if mip_outinterval = 100, this node information is printed only for every 100th node (printing output less frequently may save significant CPU time in some cases). In the example below, mip_outinterval = 1, so information about every node is printed (without the accumulated time).

```
   Node     Left    Iinf     Objective         Best relaxatn   Best incumbent
  ------   ------  ------  --------------     --------------   --------------
       1        0       2    7.592845e-01       7.592845e-01
       2        1       1    5.171320e+00       7.592845e-01
*      2        1                         r                     7.671320e+00
*      3        2       0    6.009759e+00 f     5.171320e+00     6.009759e+00
       4        1            1.000000e+01 pr    5.171320e+00     6.009759e+00
       5        0            7.092732e+00 pr    6.009759e+00     6.009759e+00
```

The meaning of each column is described below.

- **Node:** the node number. If an integer feasible point was found at a given node, then it is marked with a star (*).
- **Left:** the current number of active nodes left in the branch and bound tree.
- **Iinf:** the number of integer infeasible variables at the current node solution.
- **Objective:** the value of the objective function at the solution of the relaxed subproblem solved at the current node. If the subproblem was infeasible or failed, this is indicated. Additional symbols may be printed at some nodes if the node was pruned (*pr*), integer feasible (*f*), or an integer feasible point was found through rounding (*r*).
- **Best relaxatn:** the value of the current best relaxation (lower bound on the solution if minimizing).
- **Best incumbent:** the value of the current best integer feasible point (upper bound on the solution if minimizing).

Display of Termination Status

At the end of the run a termination message is printed indicating whether or not the optimal solution was found and if not, why KNITRO stopped. The termination message typically starts with the word "EXIT". If KNITRO was successful in satisfying the termination test, the message will look as follows:

```
EXIT: Optimal solution found.
```

See the reference manual (*Return codes*) for a list of possible termination messages and a description of their meaning and the corresponding value returned by KTR_mip_solve().

Display of Final Statistics

Following the termination message, a summary of some final statistics on the run are printed.

```
Final Statistics for MIP
------------------------
Final objective value                =    6.00975890892825e+00
Final integrality gap (abs / rel)    =    0.00e+00 /     0.00e+00 (0.00%)
# of nodes processed                 =    5
# of subproblems solved              =    6
Total program time (secs)            =    0.09930 (0.099 CPU time)
Time spent in evaluations (secs)     =    0.00117
```

Display of Solution Vector and Constraints

If `outlev` equals 5 or 6, the values of the solution vector are printed after the final statistics. If `outlev` equals 6, the constraint values at the solution are also printed.

```
Solution Vector
---------------
x[0] =   1.30097589089e+00
x[1] =   0.00000000000e+00
x[2] =   1.00000000000e+00
x[3] =   0.00000000000e+00      (binary variable)
x[4] =   1.00000000000e+00      (binary variable)
x[5] =   0.00000000000e+00      (binary variable)
```

KNITRO can produce additional information which may be useful in debugging or analyzing MIP performance. If `outlev` is positive and `mip_debug` = 1, then the file named `kdbg_mip.log` is created which contains detailed information on the MIP performance. In addition, if `mip_outsub` = 1, this file will contain extensive output for each subproblem solve in the MIP solution process. The information produced by `mip_debug` is primarily intended for developers, and should not be used in a production setting.

2.12.3 Getting information programmatically

Important solution information from KNITRO is either made available as output from the call to `KTR_solve()` or `KTR_mip_solve()`, or can be retrieved through special function calls.

The `KTR_solve()` and `KTR_mip_solve()` functions return the final value of the objective function in *obj*, the final (primal) solution vector in the array *x* and the final values of the Lagrange multipliers (or dual variables) in the array *lambda*. The solution status code is given by the return value from `KTR_solve()` or `KTR_mip_solve()`.

In addition, information related to the final statistics can be retrieved through the following function calls. The precise meaning of each option is described in the reference manual (*KNITRO user options*).

```
int KTR_get_number_FC_evals (const KTR_context_ptr  kc);
int KTR_get_number_GA_evals (const KTR_context_ptr  kc);
int KTR_get_number_H_evals (const KTR_context_ptr  kc);
int KTR_get_number_HV_evals (const KTR_context_ptr  kc);
int KTR_get_solution (const KTR_context_ptr kc,
                      int     * const status,
                      double  * const obj,
                      double  * const x,
                      double  * const lambda);
int KTR_get_constraint_values (const KTR_context_ptr kc, double * const c);
```

Continuous problems

```
int KTR_get_number_iters (const KTR_context_ptr  kc);
int KTR_get_number_cg_iters (const KTR_context_ptr  kc);
```

```
double KTR_get_abs_feas_error (const KTR_context_ptr  kc);
double KTR_get_rel_feas_error (const KTR_context_ptr  kc);
double KTR_get_abs_opt_error (const KTR_context_ptr  kc);
double KTR_get_rel_opt_error (const KTR_context_ptr  kc);
int KTR_get_objgrad_values (const KTR_context_ptr kc, double * const objGrad);
int KTR_get_jacobian_values (const KTR_context_ptr kc, double * const jac);
int KTR_get_hessian_values (const KTR_context_ptr kc, double * const hess);
```

Discrete or mixed integer problems

```
int KTR_get_mip_num_nodes (const KTR_context_ptr  kc);
int KTR_get_mip_num_solves (const KTR_context_ptr  kc);
double KTR_get_mip_abs_gap (const KTR_context_ptr  kc);
double KTR_get_mip_rel_gap (const KTR_context_ptr  kc);
double  KTR_get_mip_incumbent_obj (const KTR_context_ptr  kc);
int  KTR_get_mip_incumbent_x (const KTR_context_ptr kc, double * const x);
double  KTR_get_mip_relaxation_bnd (const KTR_context_ptr  kc);
double  KTR_get_mip_lastnode_obj (const KTR_context_ptr  kc);
```

2.12.4 User-defined names in KNITRO output

By default KNITRO uses *x* for variable names and *c* for constraint names in the output. However, the user can define more meaningful and customized names for the objective function, the variables and the constraint functions through the the API function `KTR_set_names()` when using the callable library API.

When using the AMPL modeling language, you can have KNITRO output objective function, variable and constraint names specified in the AMPL model by issuing the following command in the AMPL session:

```
option knitroampl_auxfiles rc;
```

2.12.5 Suppressing all output in AMPL

Even when setting the options:

```
ampl: option solver_msg 0;
ampl: option KNITRO_options "outlev=0";
```

in an AMPL session, AMPL will still print some basic information like the solver name and non-default user option settings to the screen. In order to suppress all AMPL and KNITRO output you must change your AMPL solve commands to something like:

```
ampl: solve >scratch-file;
```

where scratch-file is the name of some temporary file where the unwanted output can be sent. Under Unix, "solve >/dev/null" automatically throws away the unwanted output and under Windows, "solve > NUL" does the same.

2.12.6 AMPL solution information through suffixes

Some KNITRO solution information can be retrieved and displayed through AMPL using AMPL suffixes defined for KNITRO. In particular, when solving a MIP using KNITRO/AMPL, the best relaxation bound and the incumbent solution can be displayed using the *relaxbnd* and *incumbent* suffixes. For example, if the objective function is named *obj*, then:

```
ampl: display obj.relaxbnd;
```

give the current relaxation bound and:

```
ampl: display obj.incumbent;
```

will give the current incumbent solution (if one exists).

2.12.7 AMPL presolve

AMPL will often perform a reordering of the variables and constraints defined in the AMPL model. The AMPL presolver may also simplify the form of the problem by eliminating certain variables or constraints. The output printed by KNITRO corresponds to the reordered, reformulated problem. To view final variable and constraint values in the original AMPL model, use the AMPL display command after KNITRO has completed solving the problem.

It is possible to correlate KNITRO variables and constraints with the original AMPL model. You must type an extra command in the AMPL session:

```
option knitroampl_auxfiles rc;
```

and set KNITRO option `presolve_dbg` = 2. Then the solver will print the variables and constraints that KNITRO receives, with their upper and lower bounds, and their AMPL model names. The extra AMPL command causes the model names to be passed to the KNITRO/AMPL solver.

The output below is obtained with the example file `testproblem.mod` supplied with the distribution. The center column of variable and constraint names are those used by KNITRO, while the names in the right-hand column are from the AMPL model:

```
ampl: model testproblem.mod;
ampl: option solver knitroampl;
ampl: option knitroampl_auxfiles rc;
ampl: option knitro_options "presolve_dbg=2 outlev=0";

KNITRO 9.1: presolve_dbg=2
outlev=0
----- AMPL problem for KNITRO -----
Objective name:  obj
   0.000000e+00  <=  x[   0]  <=     1.000000e+20  x[1]
   0.000000e+00  <=  x[   1]  <=     1.000000e+20  x[2]
   0.000000e+00  <=  x[   2]  <=     1.000000e+20  x[3]

   2.500000e+01  <=  c[   0]  <=     1.000000e+20  c2
   5.600000e+01  <=  c[   1]  <=     5.600000e+01  c1
-----------------------------------
KNITRO 9.1: Locally optimal solution found.
objective 9.360000e+02; feasibility error 7.105427e-15
6 major iterations; 7 function evaluations
```

2.13 Callbacks

KNITRO needs to evaluate the objective function and constraints (function values and ideally, their derivatives) at various points along the optimization process. In order to pass this information to the solver, you need to provide a handle to a user-defined function that performs the necessary computation. This is referred to as a *callback*.

Callbacks in KNITRO require you to supply several function pointers that KNITRO calls when it needs new function, gradient or Hessian values, or to execute a user-provided *newpoint* routine. For convenience, the function, gradient and Hessian callback routines take the same list of arguments.

If your callback requires additional parameters, you are encouraged to create a structure containing them and pass its address as the *userParams* pointer. KNITRO does not modify or dereference the *userParams* pointer, so it is safe to use for this purpose.

The C language prototypes for the KNITRO callback functions used for evaluations and the *newpoint* feature are defined in `knitro.h`. The prototype used for evaluations is:

```
typedef int KTR_callback (
        const int             evalRequestCode,
        const int             n,
        const int             m,
        const int             nnzJ,
        const int             nnzH,
        const double * const  x,
        const double * const  lambda,
              double * const  obj,
              double * const  c,
              double * const  objGrad,
              double * const  jac,
              double * const  hessian,
              double * const  hessVector,
              void   *        userParams);
```

while the prototype used for the *newpoint* callback is:

```
typedef int KTR_newpt_callback (
        KTR_context_ptr       kc,
        const int             n,
        const int             m,
        const int             nnzJ,
        const double * const  x,
        const double * const  lambda,
        const double          obj,
        const double * const  c,
        const double * const  objGrad,
        const double * const  jac,
              void   *        userParams);
```

The *evalRequestCode* input indicates which callback utility KNITRO would like you to perform and can take on any of the following values:

- KTR_RC_EVALFC (1): Evaluate functions *f(x)* (objective) and *c(x)* (constraints).
- KTR_RC_EVALGA (2): Evaluate gradient of *f(x)* and the constraint Jacobian matrix.
- KTR_RC_EVALH (3): Evaluate the Hessian $H(x, \lambda)$.
- KTR_RC_EVALHV (7): Evaluate the Hessian $H(x, \lambda)$ times a vector.
- KTR_RC_EVALH_NO_F (8): Evaluate the Hessian $H(x, \lambda)$ *without the objective component included*.
- KTR_RC_EVALHV_NO_F (9): Evaluate the Hessian $H(x, \lambda)$ times a vector *without the objective component included*.

See the *Derivatives* section for details on how to compute the Jacobian and Hessian matrices in a form suitable for KNITRO.

The callback functions for evaluating the functions, gradients and Hessian or for performing some newpoint task, are set as described below. Each user callback routine should return an *int* value of 0 if successful, or a negative value to indicate that an error occurred during execution of the user-provided function.

```
/* This callback should modify "obj" and "c". */
int KTR_set_func_callback (KTR_context_ptr kc, KTR_callback * func);

/* This callback should modify "objGrad" and "jac". */
int KTR_set_grad_callback (KTR_context_ptr kc, KTR_callback * func);

/* This callback should modify "hessian" or "hessVector",
   depending on the value of "evalRequestCode". */
int KTR_set_hess_callback (KTR_context_ptr kc, KTR_callback * func);

/* This callback should modify nothing. */
int KTR_set_newpt_callback (KTR_context_ptr kc, KTR_newpt_callback * func);
```

Note: To enable *newpoint* callbacks, set `newpoint` = *user*. These should only be used for continuous problems.

KNITRO also provides a special callback function for output printing. By default KNITRO prints to *stdout* or a `knitro.log` file, as determined by the `outmode` option. Alternatively, you can define a callback function to handle all output. This callback function can be set as shown below

```
int KTR_set_puts_callback (KTR_context_ptr kc, KTR_puts * puts_func);
```

The prototype for the KNITRO callback function used for handling output is

```
typedef int KTR_puts (char * str, void * user);
```

In addition to the callbacks defined above, KNITRO makes additional callbacks available to the user for features such as multi-start and MINLP. Please see a complete list and description of KNITRO callback functions in the *KNITRO API* section in the Reference Manual.

2.13.1 Example

Consider the following nonlinear optimization problem from the Hock and Schittkowski test set.

$$\min\ 100 - (x_2 - x_1^2)^2 + (1 - x_1)^2$$
$$1 \leq x_1 x_2,\ 0 \leq x_1 + x_2^2,\ x_1 \leq 0.5.$$

This problem is coded as `examples/C/problemHS15.c`.

Note: The KNITRO distribution comes with several C language programs in the directory *examples/C*. The instructions in `examples/C/README.txt` explain how to compile and run the examples. This section overviews the coding of driver programs using the callback interface, but the working examples provide more complete detail.

Every driver starts by allocating a new KNITRO solver instance and checking that it succeeded (`KTR_new()` might return NULL if the Ziena license check fails):

```
#include "knitro.h"

/*... Include other headers, define main() ...*/

KTR_context     *kc;
```

```
/*... Declare other local variables ...*/

/*---- CREATE A NEW KNITRO SOLVER INSTANCE. */
kc = KTR_new();
if (kc == NULL) {
        printf ("Failed to find a Ziena license.\n");
        return( -1 );
}
```

The next task is to load the problem definition into the solver using `KTR_init_problem()`. The problem has 2 unknowns and 2 constraints, and it is easily seen that all first and second partial derivatives are generally nonzero. The code below captures the problem definition and passes it to KNITRO:

```
/*---- DEFINE PROBLEM SIZES. */
n = 2;
m = 2;
nnzJ = 4;
nnzH = 3;

/*... allocate memory for xLoBnds, xUpBnds, etc. ...*/

/*---- DEFINE THE OBJECTIVE FUNCTION AND VARIABLE BOUNDS. */
objType = KTR_OBJTYPE_GENERAL;
objGoal = KTR_OBJGOAL_MINIMIZE;
xLoBnds[0] = -KTR_INFBOUND;
xLoBnds[1] = -KTR_INFBOUND;
xUpBnds[0] = 0.5;
xUpBnds[1] = KTR_INFBOUND;

/*---- DEFINE THE CONSTRAINT FUNCTIONS. */
cType[0] = KTR_CONTYPE_QUADRATIC;
cType[1] = KTR_CONTYPE_QUADRATIC;
cLoBnds[0] = 1.0;
cLoBnds[1] = 0.0;
cUpBnds[0] = KTR_INFBOUND;
cUpBnds[1] = KTR_INFBOUND;

/*---- PROVIDE FIRST DERIVATIVE STRUCTURAL INFORMATION. */
jacIndexCons[0] = 0;
jacIndexCons[1] = 0;
jacIndexCons[2] = 1;
jacIndexCons[3] = 1;
jacIndexVars[0] = 0;
jacIndexVars[1] = 1;
jacIndexVars[2] = 0;
jacIndexVars[3] = 1;

/*---- PROVIDE SECOND DERIVATIVE STRUCTURAL INFORMATION. */
hessIndexRows[0] = 0;
hessIndexRows[1] = 0;
hessIndexRows[2] = 1;
hessIndexCols[0] = 0;
hessIndexCols[1] = 1;
hessIndexCols[2] = 1;

/*---- CHOOSE AN INITIAL START POINT. */
xInitial[0] = -2.0;
xInitial[1] =   1.0;
```

```
/*---- INITIALIZE KNITRO WITH THE PROBLEM DEFINITION. */
nStatus = KTR_init_problem (kc, n, objGoal, objType,
        xLoBnds, xUpBnds,
        m, cType, cLoBnds, cUpBnds,
        nnzJ, jacIndexVars, jacIndexCons,
        nnzH, hessIndexRows, hessIndexCols,
        xInitial, NULL);
        if (nStatus != 0)
                        /*... an error occurred ...*/

        /*... free xLoBnds, xUpBnds, etc. ...*/
```

Assume for simplicity that the user writes three routines for computing problem information. In `examples/C/problemHS15.c` these are named *computeFC*, *computeGA*, and *computeH*.

```
/*-----------------------------------------------------------------*/
/*         FUNCTION callbackEvalFC                                 */
/*-----------------------------------------------------------------*/
/** The signature of this function matches KTR_callback in knitro.h.
 *    Only "obj" and "c" are modified.
 */
int callbackEvalFC (
            const int               evalRequestCode,
            const int               n,
            const int               m,
            const int               nnzJ,
            const int               nnzH,
            const double * const    x,
            const double * const    lambda,
            double * const          obj,
            double * const          c,
            double * const          objGrad,
            double * const          jac,
            double * const          hessian,
            double * const          hessVector,
            void   *                userParams)
{
        if (evalRequestCode != KTR_RC_EVALFC)
            {
            printf ("*** callbackEvalFC incorrectly called with eval code %dn",
                            evalRequestCode);
            return( -1 );

        /*---- IN THIS EXAMPLE, CALL THE ROUTINE IN problemDef.h. */
        *obj = computeFC (x, c);
        return( 0 );

/*-----------------------------------------------------------------*/
/*         FUNCTION callbackEvalGA                                 */
/*-----------------------------------------------------------------*/
/** The signature of this function matches KTR_callback in knitro.h.
 *    Only "objGrad" and "jac" are modified.
 */

        /*... similar implementation to callbackEvalFC ...*/

/*-----------------------------------------------------------------*/
/*         FUNCTION callbackEvalH                                  */
```

```
/*-----------------------------------------------------------------*/
/** The signature of this function matches KTR_callback in knitro.h.
 *    Only "hessian" is modified.
 */

        /*... similar implementation to callbackEvalFC ...*/
```

To write a driver program using callback mode, simply wrap each evaluation routine in a function that matches the *KTR_callback()* prototype defined in `knitro.h`. Note that all three wrappers use the same prototype. This is in case the application finds it convenient to combine some of the evaluation steps, as demonstrated in `examples/C/callbackExample2.c`.

Back in the main program each wrapper function is registered as a callback to KNITRO, and then `KTR_solve()` is invoked to find the solution:

```
/*---- REGISTER THE CALLBACK FUNCTIONS THAT PERFORM PROBLEM EVALS.
 *---- THE HESSIAN CALLBACK ONLY NEEDS TO BE REGISTERED FOR SPECIFIC
 *---- HESSIAN OPTIONS (E.G., IT IS NOT REGISTERED IF THE OPTION FOR
 *---- BFGS HESSIAN APPROXIMATIONS IS SELECTED).
 */
if (KTR_set_func_callback (kc, &callbackEvalFC) != 0)
        exit( -1 );
if (KTR_set_grad_callback (kc, &callbackEvalGA) != 0)
        exit( -1 );
if ((nHessOpt == KTR_HESSOPT_EXACT) || (nHessOpt == KTR_HESSOPT_PRODUCT)) {
        if (KTR_set_hess_callback (kc, &callbackEvalHess) != 0)
                exit( -1 );

/*---- SOLVE THE PROBLEM     */
nStatus = KTR_solve (kc, x, lambda, 0, &obj,
        NULL, NULL, NULL, NULL, NULL, NULL);
if (nStatus != KTR_RC_OPTIMAL)
        printf ("KNITRO failed to solve the problem, final status = %dn", nStatus);

/*---- DELETE THE KNITRO SOLVER INSTANCE. */
KTR_free (&kc);
```

Note: KNITRO also offers a reverse communication mode as an alternative to callbacks for languages, such as Fortran 77, that do not support function handles. An example of how to use the reverse communication mode can be found in the example `examples/C/reverseCommExample.c`. Reverse communication is a legacy mode and new users should use callbacks instead as not all new features are made available through the reverse communication mode. Support for the reverse communication interface will be dropped in the near future.

2.14 Other programmatic interfaces

This chapter discusses interfaces to C++, Java, Fortran and Python offered by the KNITRO callable library.

2.14.1 KNITRO in a C++ application

Calling KNITRO from a C++ application follows the same outline as a C application. The distribution provides a C++ general test harness in the directory `examples/C++`. In the example, optimization problems are coded as subclasses of an abstract interface and compiled as separate shared objects. A main driver program dynamically loads a problem and sets up callback mode so KNITRO can invoke the particular problem's evaluation methods. The main driver can

also use KNITRO to conveniently check partial derivatives against finite-difference approximations. It is easy to add more test problems to the test environment.

2.14.2 KNITRO in a Java application

Calling KNITRO from a Java application follows the same outline as a C application. The optimization problem must be defined in terms of arrays and constants that follow the KNITRO API, and then the Java version of `KTR_init_problem()` / `KTR_mip_init_problem()` is called. Java *int* and *double* types map directly to their C counterparts. Having defined the optimization problem, the Java version of `KTR_solve()` or `KTR_mip_solve()` is called in reverse communications mode.

The KNITRO distribution provides a Java Native Interface (JNI) wrapper for the KNITRO callable library functions defined in `knitro.h`. The Java API loads `libJNI-knitro.dll`, a JNI-enabled form of the KNITRO binary (on Unix the file is called `lib/libJNI-knitro.so`; on Mac OS X it is `lib/libJNI-knitro.jnilib`). In this way Java applications can create a KNITRO solver instance and call Java methods that execute KNITRO functions. The JNI form of KNITRO is thread-safe, which means that a Java application can create multiple instances of a KNITRO solver in different threads, each instance solving a different problem. This feature might be important in an application that is deployed on a web server.

To write a Java application, take a look at the sample programs in `examples/Java`. The call sequence for using KNITRO is almost exactly the same as C applications that call `knitro.h` functions with a `KTR_context` reference. In Java, an instance of the class *KnitroJava* takes the place of the context reference. The sample programs compile by linking with the Java API class file delivered in the `examples/Java/knitrojava.jar` archive. This archive also contains the source code for *KnitroJava*. Examine it directly to see the full set of methods supplied with the Java API (not all methods are used in the sample programs). To extract the source code, type the command:

```
jar xf knitrojava.jar
```

and look for `com/ziena/knitro/KnitroJava.java`.

The sample programs closely mirror the structural form of the C reverse communications example.

The KNITRO Java API is compiled with Java release 1.5. However, the code does not make use of advanced 1.5 features (for example, there are no generics) and runs equally well on Java release 1.4.

2.14.3 KNITRO in a Fortran application

Calling KNITRO from a Fortran application follows the same outline as a C application. The optimization problem must be defined in terms of arrays and constants that follow the KNITRO API, and then the Fortran version of `KTR_init_problem()` is called. Fortran integer and double precision types map directly to C *int* and *double* types.

Fortran applications require wrapper functions written in C to (1) isolate the *KTR_context* structure, which has no analog in unstructured Fortran, (2) convert C function names into names recognized by the Fortran linker, and (3) renumber array indices to start from zero (the C convention used by KNITRO) for applications that follow the Fortran convention of starting from one. The wrapper functions can be called from Fortran with exactly the same arguments as their C language counterparts, except for the omission of the *KTR_context* argument.

An example Fortran program and set of C wrappers is provided in `examples/Fortran`. The example loads the matrix sparsity of the optimization problem with indices that start numbering from zero, and therefore requires no conversion from the Fortran convention of numbering from one. The C wrappers provided are sufficient for the simple example, but do not implement all the functionality of the KNITRO callable library. Users are free to write their own C wrapper routines, or extend the example wrappers as needed.

2.14.4 KNITRO in a Python application

Calling KNITRO from a Python application follows the same outline as a C application, with the same methods. C *int* and *double* types are automatically mapped into their Python counterparts (*int* and *float*). C arrays are automatically mapped into Python list types. C pointers are automatically mapped into single element lists (used in particular for recovering objective function values). Methods that accept *NULL* values in C also accept *None* values in Python.

KNITRO accepts empty Python lists for C pointer arguments that are used as output. In this case, the output value will automatically be appended to the list provided. *None* can also be provided if an output value is not required.

The optimization problem must be defined in terms of Python lists and constants that follows the C KNITRO API, and then the Python version of `KTR_init_problem()` / `KTR_mip_init_problem()` is called. Having defined the optimization problem, the Python version of `KTR_solve()` or `KTR_mip_solve()` is called in callback mode or in reverse communications mode. All Python methods have the same function prototype as in C, with the exception of `KTR_init_problem()` / `KTR_mip_init_problem()`, which do not require arguments *m*, *nnzJ*, *nnzH*. These arguments are automatically inferred from the lengths of the other list arguments.

The KNITRO distribution provides a Python wrapper for the KNITRO callable library functions defined in `knitro.h`. The Python API loads directly `knitro.dll` (`libknitro.so` on Unix; `libknitro.dylib` on Mac OS X). In this way Python applications can create a KNITRO solver instance and call Python methods that execute KNITRO functions. The Python form of KNITRO is thread-safe, which means that a Python application can create multiple instances of a KNITRO solver in different threads, each instance solving a different problem. This feature might be important in an application that is deployed on a web server. To write a Python application, take a look at the sample programs in `examples/Python`. The call sequence for using KNITRO is almost exactly the same as C applications that call `knitro.h` functions with a *KTR_context_ptr* object.

Python functions can be declared as callbacks for KNITRO as long as they follow the corresponding callback function prototypes (defined in `knitro.h`). The arguments passed by KNITRO to these callbacks functions are mapped as follows. C *int* and *double* types are automatically mapped into their Python counterparts (*int* and *float*). C arrays and pointers are automatically mapped into *KTR_array* objects. The class *KTR_array* is used to wrap C-style *int* and *double* arrays, which makes it transparent to the user to perform the following operations: *getitem*, *setitem*, *getslice*, *setslice*, *getlength*, *getrepresentation*, *getstring*. These classes allows KNITRO users to interact directly with the content of the C pointer or array, and avoid unnecessary copies from C to Python and from Python to C. It is possible to modify the array content but not its size, as KNITRO users are not expected to manually create *KTR_array* objects. Although the Python language makes it unnecessary, objects can be passed to the callback function through the *userParams* argument.

The sample programs can be run directly from the command line after installing `knitro.pyc` (or make sure that `knitro.pyc` is in the folder containing the example source file). The sample programs provided closely mirror the structural form of the C callback and reverse communication examples. The KNITRO Python API has been developed with Python releases 2.6.6 and 2.7.3.

2.15 Special problem classes

The following sections describe specializations in KNITRO to deal with particular classes of optimization problems. We also provide guidance on how to best set user options and model your problem to get the best performance out of KNITRO for particular types of problems.

2.15.1 Linear programming problems (LPs)

A linear program (LP) is an optimization problem where the objective function and all the constraint functions are linear.

KNITRO has built in specializations for efficiently solving LPs. However, KNITRO is unable to automatically detect whether or not a problem is an LP. In order for KNITRO to detect that a problem is an LP, you must specify this by setting the value of *objType* to `KTR_OBJTYPE_LINEAR` and all values of the array *cType* to `KTR_CONTYPE_LINEAR` in the function call to `KTR_init_problem()`. If this is not done, KNITRO will not apply special treatment to the LP and will typically be less efficient in solving the LP.

2.15.2 Quadratic programming problems (QPs)

A quadratic program (QP) is an optimization problem where the objective function is quadratic and all the constraint functions are linear.

KNITRO has built in specializations for efficiently solving QPs. However, KNITRO is unable to automatically detect whether or not a problem is a QP. In order for KNITRO to detect that a problem is a QP, you must specify this by setting the value of *objType* to `KTR_OBJTYPE_QUADRATIC` and all values of the array *cType* to `KTR_CONTYPE_LINEAR` in the function call to `KTR_init_problem()`. If this is not done, KNITRO will not apply special treatment to the QP and will typically be less efficient in solving the QP.

Typically, these specializations will only help on convex QPs.

2.15.3 Systems of nonlinear equations

KNITRO is effective at solving systems of nonlinear equations. To solve a square system of nonlinear equations using KNITRO one should specify the nonlinear equations as equality constraints and specify the objective function as zero (i.e., *f(x)=0*).

If KNITRO is converging to a stationary point for which the nonlinear equations are not satisfied, the multi-start option may help in finding a solution by trying different starting points.

2.15.4 Least squares problems

There are two ways of using KNITRO for solving problems in which the objective function is a sum of squares of the form:

$$f(x) = r_1(x)^2 + r_2(x)^2 + ... + r_q(x)^2.$$

If the value of the objective function at the solution is not close to zero (the large residual case), the least squares structure of f can be ignored and the problem can be solved as any other optimization problem. Any of the KNITRO options can be used.

On the other hand, if the optimal objective function value is expected to be small (small residual case) then KNITRO can implement the Gauss-Newton or Levenberg-Marquardt methods which only require first derivatives of the residual functions, $r_j(x)$, and yet converge rapidly.

To do so, the user need only define the approximate Hessian of f to be equal to

$$2J(x)^T J(x)$$

where $J(x)$ is the Jacobian matrix of the residual functions $r_j(x)$ at x. The Gauss-Newton and Levenberg-Marquardt approaches consist of using this approximate value for the Hessian and ignoring the remaining term.

KNITRO will behave like a Gauss-Newton method by setting `algorithm` = 1, and will be very similar to the classical Levenberg-Marquardt method when `algorithm` = 2.

See *Nonlinear Least Squares* for an implementation in *knitromatlab*.

2.15.5 Complementarity constraints (MPCCs)

As we have seen in *Complementarity constraints*, a mathematical program with complementarity (or equilibrium) constraints (also know as an MPCC or MPEC) is an optimization problem which contains a particular type of constraint referred to as a complementarity constraint. A complementarity constraint is a constraint that enforces that two variables x_1 and x_2 are *complementary* to each other, i.e. that the following conditions hold:

$$x_1 x_2 = 0, x_1 \geq 0, x_2 \geq 0.$$

These constraints sometimes occur in practice and deserve special handling. See *Complementarity constraints* for details on how to use complementarity constraints with KNITRO.

2.15.6 Global optimization

KNITRO is designed for finding locally optimal solutions of continuous optimization problems. A local solution is a feasible point at which the objective function value at that point is as good or better than at any "nearby" feasible point. A globally optimal solution is one which gives the best (i.e., lowest if minimizing) value of the objective function out of all feasible points. If the problem is *convex* all locally optimal solutions are also globally optimal solutions. The ability to guarantee convergence to the global solution on large-scale *nonconvex* problems is a nearly impossible task on most problems unless the problem has some special structure or the person modeling the problem has some special knowledge about the geometry of the problem. Even finding local solutions to large-scale, nonlinear, nonconvex problems is quite challenging.

Although KNITRO is unable to guarantee convergence to global solutions it does provide a *multi-start* heuristic that attempts to find multiple local solutions in the hopes of locating the global solution. See *Multistart*.

2.15.7 Mixed integer programming (MIP)

KNITRO provides tools for solving optimization models (both linear and nonlinear) with binary or integer variables. See the dedicated chapter *Mixed-integer nonlinear programming* for a discussion on this topic.

2.16 Tips and tricks

This last chapter contains some rules of the thumb to improve efficiency or solve memory issues.

2.16.1 Option tuning for efficiency

- If you are unsure how to set non-default options, or which user options to play with, simply running your model with the setting `tuner` =1 will cause the KNITRO-Tuner to run many instances of your model with a variety of option settings, and report some statistics and recommendations on what non-default option settings may improve performance on your model. Often significant performance improvements may be made by choosing non-default option settings. See *The KNITRO-Tuner* for more details.
- The most important user option is the choice of which continuous nonlinear optimization algorithm to use, which is specified by the `algorithm` option. Please try all four options as it is often difficult to predict which one will work best, or try using the *multi* option (`algorithm=5`). In particular the Active Set algorithms may often work best for small problems, problems whose only constraints are simple bounds on the variables, or linear programs. The interior-point algorithms are generally preferable for large-scale problems.
- Perhaps the second most important user option setting is the `hessopt` user option that specifies which Hessian (or Hessian approximation) technique to use. If you (or the modeling language) are not providing the exact Hessian to KNITRO, then you should experiment with different values here.

- One of the most important user options for the interior-point algorithms is the `bar_murule` option, which controls the handling of the barrier parameter. It is recommended to experiment with different values for this user option if you are using one of the interior-point solvers in KNITRO.
- If you are using the Interior/Direct algorithm and it seems to be taking a large number of conjugate gradient (CG) steps (as evidenced by a non-zero value under the CGits output column header on many iterations), then you should try a small value for the `bar_directinterval` user option (e.g., 0-2). This option will try to prevent KNITRO from taking an excessive number of CG steps. Additionally, if there are solver iterations where KNITRO slows down because it is taking a very large number of CG iterations, you can try enforcing a maximum limit on the number of CG iterations per algorithm iteration using the `maxcgit` user option.
- The `linsolver` option can make a big difference in performance for some problems. For small problems (particularly small problems with dense Jacobian and Hessian matrices), it is recommended to try the *qr* setting, while for large problems, it is recommended to try the *hybrid*, *ma27*, *ma57* and *mklpardiso* settings to see which is fastest. When using either the *hybrid*, *qr*, *ma57*, or *mklpardiso* setting for the `linsolver` option it is *highly* recommended to use the Intel MKL BLAS (`blasoption` = 1) provided with KNITRO or some other optimized BLAS as this can result in significant speedups compared to the internal KNITRO BLAS (`blasoption` = 0).
- When solving mixed integer problems (MIPs), if KNITRO is struggling to find an integer feasible point, then you should try different values for the `mip_heuristic` option, which will try to find an integer feasible point before beginning the branch and bound process. Other important MIP options that can significantly impact the performance of KNITRO are the `mip_method`, `mip_branchrule`, and `mip_selectrule` user options, as well as the `algorithm` option which will determine the KNITRO algorithm to use to solve the nonlinear, continuous subproblems generated during the branch and bound process.

2.16.2 Memory issues

If you receive a KNITRO termination message indicating that there was not enough memory on your computer to solve the problem, or if your problem appears to be running very slow because it is using nearly all of the available memory on your computer system, the following are some recommendations to try to reduce the amount of memory used by KNITRO.

- Experiment with different algorithms. Typically the Interior/Direct algorithm is chosen by default and uses the most memory. The Interior/CG and Active Set algorithms usually use much less memory. In particular if the Hessian matrix is large and dense and using most of the memory, then the Interior/CG method may offer big savings in memory. If the constraint Jacobian matrix is large and dense and using most of the memory, then the Active Set algorithm may use much less memory on your problem.
- If much of the memory usage seems to come from the Hessian matrix, then you should try different Hessian options via the `hessopt` user option. In particular `hessopt` settings *finite_diff*, *product*, and *lbfgs* use the least amount of memory.
- Try different linear solver options in KNITRO via the `linsolver` user option. Sometimes even if your problem definition (e.g. Hessian and Jacobian matrix) can be easily stored in memory, the sparse linear system solvers inside KNITRO may require a lot of extra memory to perform and store matrix factorizations. If your problem size is relatively small you can try `linsolver` setting *qr*. For large problems you should try both *ma27* and *ma57* settings as one or the other may use significantly less memory. In addition, using a smaller `pivot` user option value may reduce the amount of memory needed for the linear solver.

2.17 Bibliography

For a detailed description of the algorithm implemented in *Interior/CG* see Byrd et al., 1999 [1] and for the global convergence theory see Byrd et al., 2000 [2]. The method implemented in *Interior/Direct* is described in Waltz et al., 2006 [3]. The *Active Set* algorithm is described in Byrd et al., 2004 [4] and the global convergence theory for this algorithm is in Byrd et al., 2006a [5]. A summary of the algorithms and techniques implemented in the KNITRO software product is given in Byrd et al., 2006b [6]. We also recommend the following papers: Byrd et al., 2003 [7], Fourer et al., 2003 [8], Hock and Schittkowski, 1981 [9], and Nocedal and Wright, 1999 [10].

To solve linear systems arising at every iteration of the algorithm, KNITRO may utilize routines *MA27* or *MA57* [11], a component package of the Harwell Subroutine Library.

http://www.cse.clrc.ac.uk/activity/HSL

In addition, the *Active Set* algorithm in KNITRO may make use of the COIN-OR *Clp* linear programming solver module. The version used in KNITRO may be downloaded from

http://www.ziena.com/clp.html

[1] R. H. Byrd, M. E. Hribar, and J. Nocedal, **"An interior point algorithm for large scale nonlinear programming**, *SIAM Journal on Optimization*, 9(4):877–900, 1999.

[2] R. H. Byrd, J.-Ch. Gilbert, and J. Nocedal, **"A trust region method based on interior point techniques for nonlinear programming"**, *Mathematical Programming*, 89(1):149–185, 2000.

[3] R. A. Waltz, J. L. Morales, J. Nocedal, and D. Orban, **"An interior algorithm for nonlinear optimization that combines line search and trust region steps**, *Mathematical Programming A*, 107(3):391–408, 2006.

[4] R. H. Byrd, N. I. M. Gould, J. Nocedal, and R. A. Waltz, **"An algorithm for nonlinear optimization using linear programming and equality constrained subproblems**, *Mathematical Programming, Series B*, 100(1):27–48, 2004.

[5] R. H. Byrd, N. I. M. Gould, J. Nocedal, and R. A. Waltz, **"On the convergence of successive linear-quadratic programming algorithms**, *SIAM Journal on Optimization*, 16(2):471–489, 2006.

[6] R. H. Byrd, J. Nocedal, and R.A. Waltz, **"KNITRO: An integrated package for nonlinear optimization**, In G. di Pillo and M. Roma, editors, *Large-Scale Nonlinear Optimization*, pages 35–59. Springer, 2006.

[7] R. H. Byrd, J. Nocedal, and R. A. Waltz, **"Feasible interior methods using slacks for nonlinear optimization**, *Computational Optimization and Applications*, 26(1):35–61, 2003.

[8] R. Fourer, D. M. Gay, and B. W. Kernighan, **"AMPL: A Modeling Language for Mathematical Programming, 2nd Ed.**, Brooks/Cole – Thomson Learning, 2003.

[9] Hock, W. and Schittkowski, K. **" Test Examples for Nonlinear Programming Codes**, volume 187 of *Lecture Notes in Economics and Mathematical Systems*, Springer-Verlag, 1981.

[10] J. Nocedal and S. J. Wright, **" Numerical Optimization**, *Springer Series in Operations Research*, Springer, 1999.

[11] Harwell Subroutine Library, **"A catalogue of subroutines (HSL 2002)**, AEA Technology, Harwell, Oxfordshire, England, 2002.

CHAPTER

THREE

REFERENCE MANUAL

The reference manual contains a comprehensive list of KNITRO's functions, user options, predefined constants, and related files. The *KNITRO / AMPL reference* contains a short description of KNITRO options for AMPL and a link to the corresponding option in the callable library, with its detailed description.

3.1 KNITRO / AMPL reference

A complete list of available KNITRO options can always be shown by typing:

```
knitroampl -=
```

in a terminal, which produces the following output.

```
alg                     Algorithm (0=auto, 1=direct, 2=cg, 3=active, 4=sqp, 5=multi)
algorithm               Algorithm (0=auto, 1=direct, 2=cg, 3=active, 4=sqp, 5=multi)
bar_directinterval      Frequency for trying to force direct steps
bar_feasible            Emphasize feasibility
bar_feasmodetol         Tolerance for entering stay feasible mode
bar_initmu              Initial value for barrier parameter
bar_initpt              Barrier initial point strategy for slacks/multipliers
bar_maxbacktrack        Maximum number of linesearch backtracks
bar_maxcrossit          Maximum number of crossover iterations
bar_maxrefactor         Maximum number of KKT refactorizations allowed
bar_murule              Rule for updating the barrier parameter
bar_penaltycons         Apply penalty method to constraints
bar_penaltyrule         Rule for updating the penalty parameter
bar_refinement          Whether to refine barrier solution
bar_relaxcons           Whether to relax constraints
bar_switchrule          Rule for barrier switching alg
blasoption              Which BLAS/LAPACK library to use
blasoptionlib           Name of dynamic BLAS/LAPACK library
cplexlibname            Name of dynamic CPLEX library
debug                   Debugging level (0=none, 1=problem, 2=execution)
delta                   Initial trust region radius
derivcheck              Whether to use derivative checker
derivcheck_tol          Relative tolerance for derivative checker
derivcheck_type         Derivative checker type (1=forward, 2=central)
feastol                 Feasibility stopping tolerance
feastol_abs             Absolute feasibility tolerance
feastolabs              Absolute feasibility tolerance
gradopt                 Gradient computation method
hessopt                 Hessian computation method
honorbnds               Enforce satisfaction of the bounds
```

```
infeastol             Infeasibility stopping tolerance
linsolver             Which linear solver to use
linsolver_ooc         Use out-of-core option?
lmsize                Number of limited-memory pairs stored for LBFGS
lpsolver              LP solver used by Active Set algorithm
ma_maxtime_cpu        Maximum CPU time when 'alg=multi', in seconds
ma_maxtime_real       Maximum real time when 'alg=multi', in seconds
ma_outsub             Enable subproblem output when 'alg=multi'
ma_terminate          Termination condition when option 'alg=multi'
maxcgit               Maximum number of conjugate gradient iterations
maxit                 Maximum number of iterations
maxtime_cpu           Maximum CPU time in seconds, per start point
maxtime_real          Maximum real time in seconds, per start point
mip_branchrule        MIP branching rule
mip_debug             MIP debugging level (0=none, 1=all)
mip_gub_branch        Branch on GUBs (0=no, 1=yes)
mip_heuristic         MIP heuristic search
mip_heuristic_maxit   MIP heuristic iteration limit
mip_implications      Add logical implications (0=no, 1=yes)
mip_integer_tol       Threshold for deciding integrality
mip_integral_gap_abs  Absolute integrality gap stop tolerance
mip_integral_gap_rel  Relative integrality gap stop tolerance
mip_knapsack          Add knapsack cuts (0=no, 1=ineqs, 2=ineqs+eqs)
mip_lpalg             LP subproblem algorithm
mip_maxnodes          Maximum nodes explored
mip_maxsolves         Maximum subproblem solves
mip_maxtime_cpu       Maximum CPU time in seconds for MIP
mip_maxtime_real      Maximum real in seconds time for MIP
mip_method            MIP method (0=auto, 1=BB, 2=HQG)
mip_outinterval       MIP output interval
mip_outlevel          MIP output level
mip_outsub            Enable MIP subproblem output
mip_pseudoinit        Pseudo-cost initialization
mip_rootalg           Root node relaxation algorithm
mip_rounding          MIP rounding rule
mip_selectrule        MIP node selection rule
mip_strong_candlim    Strong branching candidate limit
mip_strong_level      Strong branching tree level limit
mip_strong_maxit      Strong branching iteration limit
mip_terminate         Termination condition for MIP
ms_deterministic      Use deterministic multistart
ms_enable             Enable multistart
ms_maxbndrange        Maximum unbounded variable range for multistart
ms_maxsolves          Maximum KNITRO solves for multistart
ms_maxtime_cpu        Maximum CPU time for multistart, in seconds
ms_maxtime_real       Maximum real time for multistart, in seconds
ms_num_to_save        Feasible points to save from multistart
ms_outsub             Enable subproblem output for parallel multistart
ms_savetol            Tol for feasible points being equal
ms_seed               Seed for multistart random generator
ms_startptrange       Maximum variable range for multistart
ms_terminate          Termination condition for multistart
newpoint              Use newpoint feature
objno                 objective number: 0 = none, 1 = first (default),
                      2 = second (if _nobjs > 1), etc.
objrange              Objective range
objrep                Whether to replace
                        minimize obj: v;
```

```
                      with
                        minimize obj: f(x)
                      when variable v appears linearly
                      in exactly one constraint of the form
                        s.t. c: v >= f(x);
                      or
                        s.t. c: v == f(x);
                      Possible objrep values:
                      0 = no
                      1 = yes for v >= f(x) (default)
                      2 = yes for v == f(x)
                      3 = yes in both cases
optionsfile           Name/location of KNITRO options file if provided
opttol                Optimality stopping tolerance
opttol_abs            Absolute optimality tolerance
opttolabs             Absolute optimality tolerance
outappend             Append to output files (0=no, 1=yes)
outdir                Directory for output files
outlev                Control printing level
outmode               Where to direct output (0=screen, 1=file, 2=both)
par_blasnumthreads    Number of parallel threads for BLAS
par_lsnumthreads      Number of parallel threads for linear solver
par_numthreads        Number of parallel threads
pivot                 Initial pivot tolerance
presolve              KNITRO presolver level
presolve_dbg          KNITRO presolver debugging level
presolve_tol          KNITRO presolver tolerance
qpcheck               whether to check for a QP: 0 = no, 1 (default) = yes
relax                 whether to ignore integrality: 0 (default) = no, 1 = yes
scale                 Automatic scaling option
soc                   Second order correction options
timing                Whether to report problem I/O and solve times:
                        0 (default) = no
                        1 = yes, on stdout
tuner                 Enables KNITRO Tuner
tuner_maxtime_cpu     Maximum CPU time when 'tuner=on', in seconds
tuner_maxtime_real    Maximum real time when 'tuner=on', in seconds
tuner_optionsfile     Name/location of Tuner options file if provided
tuner_outsub          Enable subproblem output when 'tuner=on'
tuner_terminate       Termination condition when 'tuner=on'
version               Report software version
wantsol               solution report without -AMPL: sum of
                        1 ==> write .sol file
                        2 ==> print primal variable values
                        4 ==> print dual variable values
                        8 ==> do not print solution message
xpresslibname         Name of dynamic Xpress library
xtol                  Stepsize stopping tolerance
```

These options are detailed below.

3.1.1 KNITRO options in AMPL

- **alg**: optimization algorithm used (default 0). See `algorithm`.

Value	Description
0	let KNITRO choose the algorithm
1	Interior/Direct (barrier) algorithm
2	Interior/CG (barrier) algorithm
3	Active Set algorithm
5	Run multiple algorithms

- **bar_directinterval**: frequency for trying to force direct steps (default 10). See `bar_directinterval`.
- **bar_feasible**: whether feasibility is given special emphasis (default 0). See `bar_feasible`.

Value	Description
0	no special emphasis on feasibility
1	iterates must honor inequalities
2	emphasize first getting feasible before optimizing
3	implement both options 1 and 2 above

- **bar_feasmodetol**: tolerance for entering stay feasible mode (default 1.0e-4). See `bar_feasmodetol`.
- **bar_initmu**: initial value for barrier parameter (default 1.0e-1). See `bar_initmu`.
- **bar_initpt**: initial settings of x (if not set by user), slacks and multipliers for barrier algorithms (default 0). See `bar_initpt`.

Value	Description
0	let KNITRO choose the initial point strategy
1	initialization strategy 1 (x unaffected if initialized by user)
2	initialization strategy 2 (x unaffected if initialized by user)
3	initialization strategy 3 (x unaffected if initialized by user)

- **bar_maxbacktrack**: maximum number of linesearch backtracks (default 3). See `bar_maxbacktrack`.
- **bar_maxcrossit**: maximum number of allowable crossover iterations (default 0). See `bar_maxcrossit`.
- **bar_maxrefactor**: maximum number of KKT refactorizations allowed (default -1). See `bar_maxrefactor`.
- **bar_murule**: barrier parameter update rule (default 0). See `bar_murule`.

Value	Description
0	let KNITRO choose the barrier update rule
1	monotone decrease rule
2	adaptive rule based on complementarity gap
3	probing rule (Interior/Direct only)
4	safeguarded Mehrotra predictor-corrector type rule
5	Mehrotra predictor-corrector type rule
6	rule based on minimizing a quality function

- **bar_penaltycons**: technique for penalizing constraints in the barrier algorithms (default 0). See `bar_penaltycons`.

Value	Description
0	let KNITRO choose the strategy
1	do not apply penalty approach to any constraints
2	apply a penalty approach to all general constraints

- **bar_penaltyrule**: penalty parameter rule for step acceptance (default 0). See `bar_penaltyrule`.

Value	Description
0	let KNITRO choose the strategy
1	use single penalty parameter approach
2	use more tolerant, flexible strategy

- **bar_refinement**: specify whether to refine barrier solution for more precision (default 0). See bar_refinement.

Value	Description
0	do not apply refinement phase
1	try to refine the barrier solution

- **bar_relaxcons**: specify whether to relax constraints in the barrier algorithms (default 2). See bar_relaxcons.

Value	Description
0	do not relax constraints
1	relax equality constraints only
2	relax inequality constraints only
3	relax all constraints

- **bar_switchrule**: controls technique for switching between feasibility phase and optimality phase in the barrier algorithms (default 0). See bar_switchrule.

Value	Description
0	let KNITRO determine the switching procedure
1	never switch to feasibility phase
2	allow switches to feasibility phase
3	use more aggressive switching rule

- **blasoption**: specify the BLAS/LAPACK function library to use (default 1). See blasoption.

Value	Description
0	use KNITRO built-in functions
1	use Intel Math Kernel Library functions
2	use the dynamic library specified with blasoptionlib

- **blasoptionlib**: specify the BLAS/LAPACK function library if using blasoption=2. See blasoptionlib.
- **debug**: enable debugging output (default 0). See debug.

Value	Description
0	no extra debugging
1	print info to debug solution of the problem
2	print info to debug execution of the solver

- **delta**: initial trust region radius scaling (default 1.0e0). See delta.
- **feastol**: feasibility termination tolerance (relative) (default 1.0e-6). See feastol.
- **feastol_abs**: feasibility termination tolerance (absolute) (default 1.0e-3). See feastol_abs.
- **gradopt**: gradient computation method (default 1). See gradopt.

Value	Description
1	use exact gradients
2	compute forward finite-difference approximations
3	compute centered finite-difference approximations

- **hessopt**: Hessian (Hessian-vector) computation method (default 1). See hessopt.

Value	Description
1	use exact Hessian derivatives
2	use dense quasi-Newton BFGS Hessian approximation
3	use dense quasi-Newton SR1 Hessian approximation
4	compute Hessian-vector products by finite diffs
5	compute exact Hessian-vector products
6	use limited-memory BFGS Hessian approximation

- **honorbnds**: allow or not bounds to be violated during the optimization (default 2). See `honorbnds`.

Value	Description
0	allow bounds to be violated during the optimization
1	enforce bounds satisfaction of all iterates
2	enforce bounds satisfaction of initial point

- **infeastol**: tolerance for declaring infeasibility (default 1.0e-8). See `infeastol`.
- **linsolver**: linear system solver to use inside KNITRO (default 0). See `linsolver`.

Value	Description
0	let KNITRO choose the linear system solver
1	(not currently used; same as 0)
2	use a hybrid approach; solver depends on system
3	use a dense QR method (small problems only)
4	use HSL MA27 sparse symmetric indefinite solver
5	use HSL MA57 sparse symmetric indefinite solver
6	use Intel MKL PARDISO sparse symmetric indefinite solver

- **linsolver_ooc**: solve linear system out-of-core (default 0). See `linsolver_ooc`.

Value	Description
0	do not solve linear systems out-of-core
1	invoke Intel MKL PARDISO out-of-core option sometimes (only when `linsolver` = 6)
2	invoke Intel MKL PARDISO out-of-core option always (only when `linsolver` = 6)

- **lmsize**: number of limited-memory pairs stored in LBFGS approach (default 10). See `lmsize`.
- **lpsolver**: LP solver in Active Set algorithm (default 1). See `lpsolver`.

Value	Description
1	use internal LP solver
2	use ILOG-CPLEX LP solver (requires a valid CPLEX license; specify library location with `cplexlibname`)
3	use FICO-Xpress LP solver (requires a valid Xpress license; specify library location with `xpresslibname`)

- **ma_maxtime_cpu**: maximum CPU time in seconds before terminating for the multi-algorithm (`alg=5`) procedure (default 1.0e8). See `ma_maxtime_cpu`.
- **ma_maxtime_real**: maximum real time in seconds before terminating for the multi-algorithm (`alg=5`) procedure (default 1.0e8). See `ma_maxtime_real`.
- **ma_outsub**: enable writing algorithm output to files for the multi-algorithm (`alg=5`) procedure (default 0). See `ma_outsub`.

Value	Description
0	do not write detailed algorithm output to files
1	write detailed algorithm output to files named `knitro_ma_*.log`

- **ma_terminate**: termination condition for multi-algorithm (`alg=5`) procedure (default 1). See `ma_terminate`.

Value	Description
0	terminate after all algorithms have completed
1	terminate at first local optimum
2	terminate at first feasible solution

- **maxcgit**: maximum allowable conjugate gradient (CG) iterations (default 0). See `maxcgit`.

Value	Description
0	let KNITRO set the number based on the problem size
n	maximum of *n > 0* CG iterations per minor iteration

- **maxit**: maximum number of iterations before terminating (default 0). See `maxit`.

Value	Description
0	let KNITRO set the number based on the problem
n	maximum limit of *n > 0* iterations

- **maxtime_cpu**: maximum CPU time in seconds before terminating (default 1.0e8). See `maxtime_cpu`.
- **maxtime_real**: maximum real time in seconds before terminating (default 1.0e8). See `maxtime_real`.
- **mip_branchrule**: MIP branching rule (default 0). See `mip_branchrule`.

Value	Description
0	let KNITRO choose the branching rule
1	most-fractional branching
2	pseudo-cost branching
3	strong branching

- **mip_debug**: MIP debugging level (default 0). See `mip_debug`.

Value	Description
0	no MIP debugging output
1	print MIP debugging information

- **mip_gub_branch**: Branch on GUBs (default 0). See `mip_gub_branch`.

Value	Description
0	do not branch on GUB constraints
1	allow branching on GUB constraints

- **mip_heuristic**: heuristic search approach (default 0). See `mip_heuristic`.

Value	Description
0	let KNITRO decide whether to apply a heuristic
1	do not apply any heuristic
2	use feasibility pump heuristic
3	use MPEC heuristic

- **mip_heuristic_maxit**: heuristic search iteration limit (default 100). See `mip_heuristic_maxit`.
- **mip_implications**: add logical implications (default 1). See `mip_implications`.

Value	Description
0	do not add constraints from logical implications
1	add constraints from logical implications

- **mip_integer_tol**: threshold for deciding integrality (default 1.0e-8). See `mip_integer_tol`.
- **mip_integral_gap_abs**: absolute integrality gap stop tolerance (default 1.0e-6). See `mip_integral_gap_abs`.

- **mip_integral_gap_rel**: relative integrality gap stop tolerance (default 1.0e-6). See mip_integral_gap_rel.
- **mip_knapsack**: add knapsack cuts(default 1). See mip_knapsack.

Value	Description
0	do not add knapsack cuts
1	add knapsack inequality cuts only
2	add knapsack inequality and equality cuts

- **mip_lpalg**: LP subproblem algorithm (default 0). See mip_lpalg.

Value	Description
0	let KNITRO decide the LP algorithm
1	Interior/Direct (barrier) algorithm
2	Interior/CG (barrier) algorithm
3	Active Set (simplex) algorithm

- **mip_maxnodes**: maximum nodes explored (default 100000). See mip_maxnodes.
- **mip_maxsolves**: maximum subproblem solves (default 200000). See mip_maxsolves.
- **mip_maxtime_cpu**: maximum CPU time in seconds for MIP (default 1.0e8). See mip_maxtime_cpu.
- **mip_maxtime_real**: maximum real time in seconds for MIP (default 1.0e8). See mip_maxtime_real.
- **mip_method**: MIP method (default 0). See mip_method.

Value	Description
0	let KNITRO choose the method
1	branch and bound method
2	hybrid method for convex nonlinear models

- **mip_outinterval**: MIP node output interval (default 10). See mip_outinterval.
- **mip_outlevel**: MIP output level (default 1). See mip_outlevel.
- **mip_outsub**: enable MIP subproblem debug output (default 0). See mip_outsub.
- **mip_pseudoinit**: method to initialize pseudo-costs (default 0). See mip_pseudoinit.

Value	Description
0	let KNITRO choose the method
1	use average value
2	use strong branching

- **mip_rootalg**: root node relaxation algorithm (default 0). See mip_rootalg.

Value	Description
0	let KNITRO decide the root algorithm
1	Interior/Direct (barrier) algorithm
2	Interior/CG (barrier) algorithm
3	Active Set algorithm
4	SQP algorithm

- **mip_rounding**: MIP rounding rule (default 0). See mip_rounding.

Value	Description
0	let KNITRO choose the rounding rule
1	do not attempt rounding
2	use fast heuristic
3	apply rounding solve selectively
4	apply rounding solve always

- **mip_selectrule**: MIP node selection rule (default 0). See `mip_selectrule`.

Value	Description
0	let KNITRO choose the node select rule
1	use depth first search
2	use best bound node selection
3	use a combination of depth first and best bound

- **mip_strong_candlim**: strong branching candidate limit (default 10). See `mip_strong_candlim`.
- **mip_strong_level**: strong branching level limit (default 10). See `mip_strong_level`.
- **mip_strong_maxit**: strong branching subproblem iteration limit (default 1000). See `mip_strong_maxit`.
- **mip_terminate**: termination condition for MIP (default 0). See `mip_terminate`.

Value	Description
0	terminate at optimal solution
1	terminate at first integer feasible solution

- **ms_deterministic**: whether to use a deterministic version of multi-start (default 1). See `ms_deterministic`.

Value	Description
0	multithreaded multi-start is non-deterministic
1	multithreaded multi-start is deterministic (when `ms_terminate` = 0)

- **ms_enable**: multi-start feature (default 0). See `ms_enable`.

Value	Description
0	multi-start disabled
1	multi-start enabled

- **ms_maxbndrange**: maximum range to vary unbounded x when generating start points (default 1.0e3). See `ms_maxbndrange`.
- **ms_maxsolves**: maximum number of start points to try during multi-start (default 0). See `ms_maxsolves`.

Value	Description
0	let KNITRO set the number based on problem size
n	try exactly $n > 0$ start points

- **ms_maxtime_cpu**: maximum CPU time for multi-start, in seconds (default 1.0e8). See `ms_maxtime_cpu`.
- **ms_maxtime_real**: maximum real time for multi-start, in seconds (default 1.0e8). See `ms_maxtime_real`.
- **ms_num_to_save**: number of feasible points to save in `knitro_mspoints.log` (default 0). See `ms_num_to_save`.
- **ms_outsub**: enable writing output from subproblem solves to files for parallel multi-start (default 0). See `ms_outsub`.

Value	Description
0	do not write subproblem output to files
1	write detailed subproblem output to files named `knitro_ms_*.log`

- **ms_savetol**: tolerance for feasible points to be considered distinct (default 1.0e-6). See `ms_savetol`.
- **ms_seed**: seed value used to generate random initial points in multi-start; should be a non-negative integer (default 0). See `ms_seed`.
- **ms_startptrange**: maximum range to vary all x when generating start points (default 1.0e20). See `ms_startptrange`.
- **ms_terminate**: termination condition for multi-start (default 0). See `ms_terminate`.

Value	Description
0	terminate after ms_maxsolves
1	terminate at first local optimum (if before ms_maxsolves)
2	terminate at first feasible solution (if before ms_maxsolves)

- **newpoint**: how to save new points found by the solver. (default 0). See `newpoint`.

Value	Description
0	no action
1	save the latest new point to file `knitro_newpoint.log`
2	append all new points to file `knitro_newpoint.log`

- **objrange**: maximum allowable objective function magnitude (default 1.0e20). See `objrange`.
- **optionsfile**: path that specifies the location of a KNITRO options file if used.
- **opttol**: optimality termination tolerance (relative) (default 1.0e-6). See `opttol`.
- **opttol_abs**: optimality termination tolerance (absolute) (default 1.0e-3). See `opttol_abs`.
- **outappend**: append output to existing files (default 0). See `outappend`.

Value	Description
0	do not append
1	do append

- **outdir**: directory where output files are created. See `outdir`.
- **outlev**: printing output level (default 2). See `outlev`.

Value	Description
0	no printing
1	just print summary information
2	print basic information every 10 iterations
3	print basic information at each iteration
4	print all information at each iteration
5	also print final (primal) variables
6	also print final Lagrange multipliers (sensitivies)

- **outmode**: KNITRO output redirection (default 0). See `outmode`.

Value	Description
0	direct KNITRO output to standard out (e.g., screen)
1	direct KNITRO output to the file `knitro.log`
2	print to both the screen and file `knitro.log`

- **par_blasnumthreads**: specify the number of threads to use for BLAS (default 1). See `par_blasnumthreads`.

Value	Description
1	for any non-positive value
n	use *n > 0* threads

- **par_lsnumthreads**: specify the number of threads to use for linear system solves (default 1). See `par_lsnumthreads`.

Value	Description
1	for any non-positive value
n	use *n > 0* threads

- **par_numthreads**: specify the number of threads to use for all parallel features (default 1). See `par_numthreads`.

Value	Description
0	determine by environment variable $OMP_NUM_THREADS
n	use *n* > *0* threads

- **pivot**: initial pivot threshold for matrix factorizations (default 1.0e-8). See `pivot`.
- **presolve**: enable KNITRO presolver (default 1). See `presolve`.

Value	Description
0	do not use KNITRO presolver
1	use the KNITRO presolver

- **presolve_dbg**: presolve debug output (default 0).

Value	Description
0	no debugging information
2	print the KNITRO problem with AMPL model names

- **presolve_tol**: tolerance used by KNITRO presolver to remove variables and constraints (default 1.0e-6). See `presolve_tol`.
- **scale**: automatic scaling (default 1). See `scale`.

Value	Description
0	do not scale the problem
1	perform automatic scaling of functions

- **soc**: 2nd order corrections (default 1). See `soc`.

Value	Description
0	do not allow second order correction steps
1	selectively try second order correction steps
2	always try second order correction steps

- **tuner**: Invoke KNITRO-Tuner (default 0). See `tuner`.

Value	Description
0	tuner disabled
1	tuner enabled

- **tuner_maxtime_cpu**: maximum CPU time in seconds before terminating the KNITRO-Tuner (`tuner=1`) procedure (default 1.0e8). See `tuner_maxtime_cpu`.
- **tuner_maxtime_real**: maximum real time in seconds before terminating the KNITRO-Tuner (`tuner=1`) procedure (default 1.0e8). See `tuner_maxtime_real`.
- **tuner_optionsfile**: path that specifies the location of a KNITRO-Tuner (`tuner=1`) options file if used.
- **tuner_outsub**: enable writing additional Tuner subproblem solve output to files for the KNITRO-Tuner (`tuner=1`) procedure (default 0). See `tuner_outsub`.

Value	Description
0	do not write detailed algorithm output to files
1	write summary solve output to a file named `knitro_tuner_summary.log`
2	write detailed algorithm output to files named `knitro_tuner_*.log`

- **tuner_terminate**: termination condition for KNITRO-Tuner (`tuner=1`) procedure (default 0). See `tuner_terminate`.

Value	Description
0	terminate after all solves have completed
1	terminate at first local optimum
2	terminate at first feasible solution

- **xtol**: stepsize termination tolerance (default 1.0e-15). See `xtol`.

3.1.2 Return codes

Upon completion, KNITRO displays a message and returns an exit code to AMPL. If KNITRO found a solution, it displays the message:

```
Locally optimal solution found
```

with exit code of zero; the exit code can be seen by typing:

```
ampl: display solve_result_num;
```

If a solution is not found, then KNITRO returns a non-zero return code from the table below:

Value	Description
0	Locally optimal solution found.
100	Current solution estimate cannot be improved. Nearly optimal.
101	Relative change in feasible solution estimate < xtol.
102	Current feasible solution estimate cannot be improved.
200	Convergence to an infeasible point. Problem may be locally infeasible.
201	Relative change in infeasible solution estimate < xtol.
202	Current infeasible solution estimate cannot be improved.
203	Multistart: No primal feasible point found.
204	Problem determined to be infeasible with respect to constraint bounds.
205	Problem determined to be infeasible with respect to variable bounds.
300	Problem appears to be unbounded.
400	Iteration limit reached. Current point is feasible.
401	Time limit reached. Current point is feasible.
403	MIP: All nodes have been explored. Integer feasible point found.
404	MIP: Integer feasible point found.
405	MIP: Subproblem solve limit reached. Integer feasible point found.
406	MIP: Node limit reached. Integer feasible point found.
410	Iteration limit reached. Current point is infeasible.
411	Time limit reached. Current point is infeasible.
413	MIP: All nodes have been explored. No integer feasible point found.
415	MIP: Subproblem solve limit reached. No integer feasible point found.
416	MIP: Node limit reached. No integer feasible point found.
501	LP solver error.
502	Evaluation error.
503	Not enough memory.
504	Terminated by user.
505	Input or other API error.
506	Internal KNITRO error.
507	Unknown termination.
508	Illegal objno value.

For more information on return codes, see *Return codes*.

3.1.3 AMPL suffixes defined for KNITRO

Suffix Name	Description
priority	Specify branch priorities for MIP (input)
cfeastol	Specify individual constraint feasibility tolerances (input)
xfeastol	Specify individual variable bound feasibility tolerances (input)
relaxbnd	Retrieve the best relaxation bound for MIP (output)
incumbent	Retrieve the incumbent solution for MIP (output)

3.2 KNITRO / MATLAB reference

Usage of *knitromatlab* is described here.

3.2.1 What is *knitromatlab*?

knitromatlab is the interface used to call KNITRO from the MATLAB environment.

knitromatlab's syntax is similar to MATLAB's built-in optimization function *fmincon*. The main differences are:

- *knitromatlab* has additional input arguments for additional features and options.
- There is a separate function, *knitromatlab_mip*, to solve mixed-integer nonlinear programs.
- *knitromatlab* does not require the Optimization Toolbox.
- Many returned flags and messages differ, because they are returned directly from KNITRO libraries.

3.2.2 Syntax

The most elaborate form is

```
[x,fval,exitflag,output,lambda,grad,hessian] = ...
    knitromatlab(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,extendedFeatures,options,knitroOpts)
```

but the simplest function call reduces to:

```
x = knitromatlab(fun,x0)
```

Any of the following forms may be used:

```
x = knitromatlab(fun,x0)
x = knitromatlab(fun,x0,A,b)
x = knitromatlab(fun,x0,A,b,Aeq,beq)
x = knitromatlab(fun,x0,A,b,Aeq,beq,lb,ub)
x = knitromatlab(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
x = knitromatlab(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,extendedFeatures)
x = knitromatlab(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,extendedFeatures,options)
x = knitromatlab(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,extendedFeatures,options,knitroOpts)
[x,fval] = knitromatlab(...)
[x,fval,exitflag] = knitromatlab(...)
[x,fval,exitflag,output] = knitromatlab(...)
[x,fval,exitflag,output,lambda,] = knitromatlab(...)
[x,fval,exitflag,output,lambda,grad] = knitromatlab(...)
[x,fval,exitflag,output,lambda,grad,hessian] = knitromatlab(...)
```

```
x = knitromatlab_mip(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,xType)
x = knitromatlab_mip(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,xType,objFnType)
x = knitromatlab_mip(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,xType,objFnType,cineqFnType)
x = knitromatlab_mip(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,xType,objFnType,cineqFnType, ...
        extendedFeatures)
x = knitromatlab_mip(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,xType,objFnType,cineqFnType, ...
        extendedFeatures,options)
x = knitromatlab_mip(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,xType,objFnType,cineqFnType, ...
        extendedFeatures,options,knitroOpts)
[x,fval] = knitromatlab_mip(...)
[x,fval,exitflag] = knitromatlab_mip(...)
[x,fval,exitflag,output] = knitromatlab_mip(...)
[x,fval,exitflag,output,lambda,] = knitromatlab_mip(...)
[x,fval,exitflag,output,lambda,grad] = knitromatlab_mip(...)
[x,fval,exitflag,output,lambda,grad,hessian] = knitromatlab_mip(...)
```

An additional function, *knitrolink*, may be used in place of the old *ktrlink* interface. *knitrolink* has the same input and output arguments as *ktrlink*, but it is equivalent to using *knitromatlab* with an empty array for the value of *extendedFeatures*.

3.2.3 Input Arguments

Input Argument	Description
fun	The function to be minimized. *fun* accepts a vector *x* and returns a scalar *f*, the objective function evaluated at *x*. If exact gradients are used, an additional vector with the objective gradient should be returned.
x0	The initial point vector.
A	Linear inequality constraint coefficient matrix.
b	Linear inequality constraint upper bound vector.
Aeq	Linear equality constraint coefficient matrix.
beq	Linear equality constraint right-hand side vector.
lb	Variable lower bound vector.
ub	Variable upper bound vector.
nonlcon	The function that computes the nonlinear inequality and equality constraints. *nonlcon* accepts a vector *x* and returns two vectors, the values of the nonlinear inequality functions at *x* and the values of the nonlinear equality functions at *x*. If exact gradients are used, two additional matrices should be returned with the gradients for the nonlinear inequality functions and the gradients for the nonlinear equality functions.
extended-Features	The structure used to define other, extended modeling features of KNITRO. Currently it is used for complementarity constraints, the output function, parallel finite differencing, initial lambda values, and some Hessian and Jacobian information. The two comlementarity constraint fields are *extendedFeatures.ccIndexList1* and *extendedFeatures.ccIndexList2* which contain the variable index lists for variables complementary to each other. The same index may not appear more than once in the lists. The *lambdaInitial* field allows the user to specify the initial lambda values in a structure with a different field for each constraint type. The fields are *ineqlin* for linear inequality constraints, *eqlin* for linear equality constraints, *ineqnonlin* for nonlinear inequality constraints, *eqnonlin* for nonlinear equality constraints, *upper* for upper bounds of variables, and *lower* for lower bounds of variables. If only some of the fields are defined, the missing fields will be filled with zeros. If none of the fields are defined, KNITRO will compute an initial value. The other six available fields are *JacobPattern*, *HessPattern*, *HessFcn*, *HessMult*, *OutputFcn*, and *UseParallel*. They have the same properties as the options set by *optimset*.
options	The options structure set with *optimset*.
knitroOpts	The text file with KNITRO options.

The objective function, *fun*, can be specified as a function handle for a MATLAB function, as in

```
x = knitromatlab(@objFunc,x0)
```

with a function

```
function [f,g] = objFunc(x)
    f = x^2;
    g = 2*x;
```

or as a function handle for an anonymous function:

```
x = knitromatlab(@(x)x^2,x0)
```

The constraint function, *nonlcon*, is similar, but it returns at least two vectors, *c* and *ceq*. It may additionally return two matrices, the gradient matrix of the nonlinear inequality constraints, and the gradient matrix of the nonlinear equality constraints. The third and fourth arguments are only needed when *GradConstr* is 'on' or *gradopt* is set to 1. See http://www.mathworks.com/help/optim/ug/nonlinear-constraints-with-gradients.html for more details.

3.2.4 Output Arguments

Output Argument	Description
x	The optimal solution vector.
fval	The optimal solution objective value.
exitflag	Integer identifying the reason for termination of the algorithm.
output	Structure containing information about the optimization.
lambda	Structure containing the Lagrange multipliers at the solution with a different field for each constraint type.
grad	Gradient vector at *x*.
hessian	Hessian matrix at *x*. For notes on when this option is available, see `KTR_get_hessian_values()`.

3.2.5 Output Structure Fields

Output Argument	Field	Description
output	*iterations*	Number of iterations.
output	*funcCount*	Number of function evaluations.
output	*constrviolation*	Maximum of constraint violations.
output	*firstorderopt*	Measure of first-order optimality.
lambda	*lower*	Lower bounds
lambda	*upper*	Upper bounds
lambda	*ineqlin*	Linear inequalities
lambda	*eqlin*	Linear equalities
lambda	*ineqnonlin*	Nonlinear inequalities
lambda	*eqnonlin*	Nonlinear equalities

3.2.6 Setting Options

knitromatlab takes up to two options inputs. The first is in *fmincon* format, using *optimset*, and the second is a KNITRO options text file. Because the full version of *optimset* requires a MATLAB Optimization Toolbox license, *HessFcn*, *HessMult*, *HessPattern*, *JacPattern*, *OutputFcn*, and *UseParallel* can also be used with the *extendedFeatures* structure. All the other options have equivalent ways of being set in the KNITRO options text file. If a KNITRO options text file is specified, unspecified options will still use the default option values from the *fmincon* format options. Settings from the KNITRO options text file and the *extendedFeatures* structure will take precedence over settings made in the MATLAB options structure. Note that options set with the *optimoptions* function are not compatible with *knitromatlab* functions.

To use KNITRO options, create an options text file as described for the callable library, and include the file as the 12th argument in the call to *knitromatlab*, or the 15th argument in the call to *knitromatlab_mip*.

Options Structure Example:

```
options = optimset('Algorithm', 'interior-point', 'Display','iter', ...
        'GradObj','on','GradConstr','on', ...
        'JacobPattern',Jpattern,'Hessian','user-supplied','HessPattern',Hpattern, ...
        'HessFcn',@hessfun,'MaxIter',1000, ...
        'TolX', 1e-15, 'TolFun', 1e-8, 'TolCon', 1e-8, 'UseParallel', true);
[x,fval,exitflag,output,lambda,grad,hess] = ...
        knitromatlab(@objfun,x0,A,b,Aeq,beq,lb,ub,@constfun,[],options,[]);
```

The example above shows how to set options using the MATLAB options structure. The example below shows how to set the same options using *extendedFeatures* and a KNITRO options file.

Options File Example:

```
extendedFeatures.JacobPattern = Jpattern;
extendedFeatures.HessPattern = Hpattern;
extendedFeatures.HessFcn = @hessfun;
extendedFeatures.UseParallel = true;
[x,fval,exitflag,output,lambda,grad,hess] = ...
        knitromatlab(@objfun,x0,A,b,Aeq,beq,lb,ub,@constfun,extendedFeatures,[], ...
        'knitro.opt');
```

knitro.opt:

```
algorithm direct    # Equivalent to setting 'Algorithm' to 'interior-point'
outlev iter_verbose # Equivalent to setting 'Display' to 'iter'
gradopt exact       # Equivalent to setting 'GradObj' to 'on' and 'GradConstr' to 'on'
hessopt exact       # Equivalent to setting 'Hessian' to 'on'
maxit 1000          # Equivalent to setting 'MaxIter' to 1000
xtol 1e-15          # Equivalent to setting 'TolX' to 1e-15
opttol 1e-8         # Equivalent to setting 'TolFun' to 1e-8
feastol 1e-8        # Equivalent to setting 'TolCon' to 1e-8
```

3.2.7 Options

Option	Equivalent KNITRO Option	Description
Algorithm	*algorithm*	The optimization algorithm: *'interior-point'*, *'active-set'*, or *'sqp'*. Default: *'interior-point'*
AlwaysHonorConstraints	*honorbnds*	Bounds are satisfied at every iteration if set to the default *'bounds'*. They are not necessarily satisfied if set to *'none'*.
DerivativeCheck	*derivcheck*	Check the value of the user-provided exact gradients at a random point against the finite difference gradients. If the difference is not within the specified tolerance, KNITRO will stop execution and display the violation. May be set to *'off'* (default) or *'on'*. The default relative tolerance is 1e-6, but can be changed with the derivcheck_tol option in the KNITRO options file. The finite difference method is set by *FinDiffType*, and is set to '*forward*' by default.
Display	*outlev*	**Level of display** • *'off'* or *'none'* displays no output. • *'iter'* displays information for each iteration, and gives the default exit message. • *'iter-detailed'* displays information for each iteration, and gives the technical exit message. • *'notify'* displays output only if the function does not converge, and gives the default exit message. • *'notify-detailed'* displays output only if the function does not converge, and gives the technical exit message. • *'final'* (default) displays just the final output, and gives the default exit message. • *'final-detailed'* displays just the final output, and gives the technical exit message.
FinDiffType	*gradopt*, *derivcheck_type*	The finite difference type is either *'forward'* (default) or *'central'*. *'central'* takes twice as many function evaluations and may violate bounds during evaluation, but is usually more accurate. When exact derivatives are used and *DerivativeCheck* is used, this option sets the finite difference type to *'forward'* (default) or *'central'* to compare with the exact derivatives.

Option	Equivalent KNITRO Option	Description
GradConstr	*gradopt*	**Gradient for nonlinear constraint functions.** • *'off'* (default) sets the algorithm to use finite differences to estimate the gradients of nonlinear constraints. • *'on'* sets the algorithm to expect exact gradients of the nonlinear constraints in the third and fourth constraint function outputs, as described for nonlcon in the Input Arguments section. To use sparse gradients, the sparsity pattern must be set with the *JacobPattern* option.
GradObj	*gradopt*	**Gradient for nonlinear objective function.** • *'off'* (default) sets the algorithm to use finite differences to estimate the gradients of the objective function. • *'on'* sets the algorithm to expect exact gradients of the objective in the second objective function outputs, as decribed for fval in the Input Arguments section.
HessFcn		Function handle to the user-supplied Hessian. Default: []
Hessian	*hessopt*	Sets the Hessian option for KNITRO. Default: *'bfgs'*
HessMult		Handle to a user-supplied function that returns a Hessian-times-vector product. Default: []
HessPattern		Sparsity pattern of Hessian. Default: *'sparse(ones(n))'*
InitBarrierParam	*bar_initmu*	Initial barrier value. Default: *0.1*
InitTrustRegionRadius	*delta*	Initial radius of the trust region. Default: *'sqrt(n)'*
JacobPattern		Sparsity pattern of the Jacobian of the nonlinear constraint matrix. It can be used for finite-differencing or for user-supplied gradients. Default: *'sparse(ones(Jrows,Jcols))'*
MaxIter	*maxit*	Maximum number of iterations allowed. Default: *10000*
MaxProjCGIter	*maxcgit*	Tolerance for the number of projected conjugate gradient iterations. Default: '2*(n-numberOfEqualities)'
ObjectiveLimit	*objrange*	Specifies the extreme limits of the objective function for purposes of determining unboundedness. If the magnitude of the objective function becomes greater than objrange for a feasible iterate, then the problem is determined to be unbounded and KNITRO proceeds no further. Default: *1.0e20*
OutputFcn	*newpoint* KTR_set_newpt_callback	Handle to a user-supplied function that is called after every iteration of the algorithm and returns a boolean indicating if the algorithm should stop. See more details below. Default: []

Option	Equivalent KNITRO Option	Description
ScaleProblem	*scale*	The default value of *'obj-and-con'* allows KNITRO to scale the objective and constraint functions based on their values at the initial point. Setting the option to *'none'* disables scaling If scaling is performed, all internal computations, including the stopping tests, are based on the scaled values.
SubproblemAlgorithm	*algorithm*	Determines how the iteration step is calculated. The default option is *'ldl-factorization'*, which is usually faster than the alternative, *'cg'* (conjugate gradient). Conjugate gradient may be faster for large problems with dense Hessians.
TolCon	*feastol*	Termination tolerance on the constraint violation. Default: *1.0e-6*
TolFun	*opttol*	Termination tolerance on the function value. Default: *1.0e-6*
TolX	*xtol*	Termination tolerance on *x*. Default: *1.0e-6*
UseParallel		Boolean indicating if parallel finite differences will be used. It has no effect when exact gradients are used or if the Parallel Computing Toolbox is not installed. The KNITRO option "*par_numthreads*" does not have an effect on parallel finite differences in MATLAB. The MATLAB command "*parpool(n)*" will set the number of workers to the minimum of *n* and the maximum number allowed, which can be set in the cluster profile. If the parallel pool is not started before *knitromatlab* is run, it will start one with the default number of workers set by MATLAB, as long as the Parallel Pool preferences allow automatically creating a parallel pool. Default: *false*

3.2.8 Output Function

The output function can be assigned with

```
extendedFeatures.OutputFcn = @outputfun;
```

or in the options structure with

```
options = optimset('OutputFcn',@outputfun);
```

where the function is defined as

```
function stop = outputfun(x,optimValues,state);
```

Only the value of *stop* can be set to *true* or *false*. Setting it to *true* will terminate KNITRO.

The inputs to the function cannot be modified. The inputs include the current point *x*, the structure *optimValues*, and the *state*. Since KNITRO only calls the function after every iteration, the value of *state* will always be 'iter'. The optimValues structure contains the following fields:

3.2.9 optimValues Fields

optimValues Field	Description
lambda	Structure containing the Lagrange multipliers at the solution with a different field for each constraint type.
fval	The objective value at x.
c	The nonlinear inequality constraint values at x.
ceq	The nonlinear equality constraint values at x.
gradient	The gradient vector at x.
cineqjac	The nonlinear inequality constraint Jacobian matrix.
ceqjac	The nonlinear equality constraint Jacobian matrix.

Note that setting *newpoint* to any value other than 3 in the KNITRO options file will take precedence over *OutputFcn*. Note that the nonlinear constraint Jacobian matrices are given with the variables as the rows and constraints as the columns, the transpose of *JacobPattern*.

3.2.10 Sparsity Pattern for Nonlinear Constraints

The sparsity pattern for the constraint Jacobian is a matrix, which is passed as the *JacobPattern* option. *JacobPattern* is only for the nonlinear constraints, with one row for each constraint. The nonlinear inequalities, in order, make up the first rows, and the nonlinear equalities, in order, are in the rows after that. Gradients for linear constraints are not included in this matrix, since the sparsity pattern is known from the linear coefficient matrices.

All that matters for the matrix is whether the values are zero or not zero for each entry. A nonzero value indicates that a value is expected from the gradient function. A MATLAB sparse matrix may be used, which may be more efficient for large sparse matrices of constraints.

The gradients of the constraints returned by the nonlinear constraint function and those used in the newpoint function have the transpose of the Jacobian pattern, i.e., *JacobPattern* has a row for each nonlinear constraint and a column for each variable, while the gradient matrices (one for inequalities and one for equalities) have a column for each constraint and a row for each variable.

3.2.11 Hessians

The Hessian is the matrix of the second derivative of the Lagrangian, as in

http://www.mathworks.com/help/optim/ug/fmincon.html#brh002z

The matrix H can be given as a full or sparse matrix of the upper triangular or whole matrix pattern.

If *HessMult* is used, then the Hessian-vector-product of the Hessian and a vector supplied by KNITRO at that iteration is returned.

3.2.12 Backwards Compatibility

The *ktrlink* interface previously provided by the MATLAB Optimization Toolbox function is no longer supported. The interface function *knitrolink* can be used in its place with the same function signature, but it has the same effect as using *knitromatlab* with an empty matrix as the *extendedFeatures* argument. Users are encouraged to use *knitromatlab*, since *knitrolink* may be removed from future versions.

3.2.13 Nonlinear Least Squares

There is a special function, *knitromatlab_lsqnonlin*, for using KNITRO to solve nonlinear least squares problems. It behaves similarly to the *lsqnonlin* function in the MATLAB Optimization Toolbox. Note that the extendedFeatures structure is not an input argument of *lsqnonlin*, but is the argument before *options* in *knitromatlab_lsqnonlin*. If the structure is not used, an empty matrix, [], should be used in its place.

The most elaborate form is:

```
[x,resnorm,residual,exitflag,output,lambda,jacobian] = ...
              knitromatlab_lsqnonlin(fun,x0,lb,ub,extendedFeatures,options)
```

but the simplest function call reduces to:

```
x = knitromatlab_lsqnonlin(fun,x0)
```

3.2.14 Input Arguments for *knitromatlab_lsqnonlin*

Input Argument	Description
fun	The function whose sum of squares is minimized. *fun* accepts a vector *x* and returns a vector *F*, the values of the functions evaluated at *x*. If exact gradients are used, an additional matrix should be returned with the Jacobian for the function at *x*. Unlike the user-supplied Jacobian for *knitromatlab*, the entries *J(i,j)* of the Jacobian for *knitromatlab_lsqnonlin* represent the partial derivative of the function component *i* with respect to variable *j*.
x0	The initial point vector.
lb	Variable lower bound vector.
ub	Variable upper bound vector.
extendedFeatures	The structure used to define other, extended modeling features of KNITRO. It is similar to the extendedFeatures input to *knitromatlab*, but currently it is only used for the *JacobPattern* (rows are the function components and columns are the variables) and *OutputFcn* fields.
options	The options structure set with *optimset*. See details below.

The options available in the *options* structure are the same as those available in *knitromatlab*, except the differences noted here.

- *GradConstr*, *HessFcn*, *Hessian*, *HessMult*, and *HessPattern* are not available.
- *JacobPattern* is the Jacobian for the function, where rows are the function components and columns are the variables.
- *OutputFcn* inputs refer to the transformed problem, but *x* still refers to the current point.
- *Algorithm* may be set to *'interior-point'* (default) to use the Gauss-Newton method, or *'levenberg-marquardt'* to use the Levenberg-Marquardt method.

knitromatlab_lsqnonlin does not use Hessian information or options provided by the user, but uses the approximation shown in *Least squares problems*. Passing extra parameters and the KNITRO options file are not currently available for this function.

3.2.15 Note on exit flags

The returned exit flags will correspond with KNITRO's return code, rather than matching *fmincon*'s exit flags.

3.2.16 Return codes

Upon completion, KNITRO displays a message and returns an exit code to MATLAB. In the example above KNITRO found a solution, so the message was:

```
Locally optimal solution found
```

with the return value of exitflag set to 0.

If a solution is not found, then KNITRO returns one of the following:

Value	Description
0	Locally optimal solution found.
-100	Current solution estimate cannot be improved. Nearly optimal.
-101	Relative change in feasible solution estimate < xtol.
-102	Current feasible solution estimate cannot be improved.
-200	Convergence to an infeasible point. Problem may be locally infeasible.
-201	Relative change in infeasible solution estimate < xtol.
-202	Current infeasible solution estimate cannot be improved.
-203	Multistart: No primal feasible point found.
-204	Problem determined to be infeasible.
-205	Problem determined to be infeasible.
-300	Problem appears to be unbounded.
-400	Iteration limit reached. Current point is feasible.
-401	Time limit reached. Current point is feasible.
-403	MIP: All nodes have been explored. Integer feasible point found.
-404	MIP: Integer feasible point found.
-405	MIP: Subproblem solve limit reached. Integer feasible point found.
-406	MIP: Node limit reached. Integer feasible point found.
-410	Iteration limit reached. Current point is infeasible.
-411	Time limit reached. Current point is infeasible.
-413	MIP: All nodes have been explored. No integer feasible point found.
-415	MIP: Subproblem solve limit reached. No integer feasible point found.
-416	MIP: Node limit reached. No integer feasible point found.
-501	LP solver error.
-502	Evaluation error.
-503	Not enough memory.
-504	Terminated by user.
-505	Input or other API error.
-506	Internal KNITRO error.
-507	Unknown termination.
-508	Illegal objno value.

For more information on return codes, see *Return codes*.

3.3 Callable library reference

The various objects offered by the callable library API are listed here. The file `knitro.h` is also a good source of information, and the ultimate reference.

3.3.1 KNITRO API

All functions offered by the KNITRO callable library are listed here.

Creating and destroying solver objects

KTR_new()

```
KTR_context_ptr  KNITRO_API KTR_new (void);
```

This function must be called first. It returns a pointer to an object (the KNITRO "context pointer") that is used in all other calls. If you enable KNITRO with the Ziena floating network license handler, then this call also checks out a license and reserves it until KTR_free() is called with the context pointer, or the program ends. The contents of the context pointer should never be modified by a calling program. Returns NULL on error.

KTR_new_puts()

```
KTR_context_ptr  KNITRO_API KTR_new_puts (KTR_puts * const  fnPtr,
                                          void     * const  userParams);
```

This function is similar to KTR_new(), but also takes an argument that sets a "put string" callback function to handle output generated by the KNITRO solver, and a pointer for passing user-defined data. See KTR_set_puts_callback() for more information. Returns NULL on error.

Call KTR_new() or KTR_new_puts() first. Either returns a pointer to the solver object that is used in all other KNITRO API calls. A new KNITRO license is acquired and held until KTR_free() has been called, or until the calling program ends.

KTR_free()

```
int  KNITRO_API KTR_free (KTR_context_ptr *  kc_handle);
```

This function should be called last and will free the context pointer. The address of the context pointer is passed so that KNITRO can set it to NULL after freeing all memory. This prevents the application from mistakenly calling KNITRO functions after the context pointer has been freed. Returns 0 if OK, nonzero if error.

Changing and reading solver parameters

Parameters cannot be set after KNITRO begins solving; ie, after the KTR_solve() function is called. They may be set again after calling KTR_restart().

Note: The gradopt and hessopt user options must be set before calling KTR_init_problem() or KTR_mip_init_problem(), and cannnot be changed after calling these functions.

All methods return 0 if OK, nonzero if there was an error. In most cases, parameter values are not validated until KTR_init_problem() or KTR_solve() is called.

KTR_reset_params_to_defaults()

```
int  KNITRO_API KTR_reset_params_to_defaults (KTR_context_ptr  kc);
```

Reset all parameters to default values.

KTR_load_param_file()

```
int  KNITRO_API KTR_load_param_file
    (KTR_context_ptr  kc, const char * const  filename);
```

Set all parameters specified in the given file.

KTR_save_param_file()

```
int  KNITRO_API KTR_save_param_file
    (KTR_context_ptr  kc, const char * const  filename);
```

Write all current parameter values to a file.

KTR_set_int_param_by_name()

```
int  KNITRO_API KTR_set_int_param_by_name
    (KTR_context_ptr  kc, const char * const  name, const int  value);
```

Set an integer valued parameter using its string name.

KTR_set_char_param_by_name()

```
int  KNITRO_API KTR_set_char_param_by_name
    (KTR_context_ptr  kc, const char * const  name, const char * const  value);
```

Set a character valued parameter using its string name.

KTR_set_double_param_by_name()

```
int  KNITRO_API KTR_set_double_param_by_name
    (KTR_context_ptr  kc, const char * const  name, const double  value);
```

Set a double valued parameter using its string name.

KTR_set_int_param()

```
int  KNITRO_API KTR_set_int_param
    (KTR_context_ptr  kc, const int  param_id, const int  value);
```

Set an integer valued parameter using its integer identifier (see *KNITRO user options*).

KTR_set_char_param()

```
int  KNITRO_API KTR_set_char_param
    (KTR_context_ptr  kc, const int  param_id, const char * const  value);
```

Set a character valued parameter using its integer identifier (see *KNITRO user options*).

KTR_set_double_param()

```
int  KNITRO_API KTR_set_double_param
    (KTR_context_ptr  kc, const int  param_id, const double  value);
```

Set a double valued parameter using its integer identifier (see *KNITRO user options*).

KTR_get_int_param_by_name()

```
int  KNITRO_API KTR_get_int_param_by_name
    (KTR_context_ptr  kc, const char * const  name, int * const  value);
```

Get an integer valued parameter using its string name.

KTR_get_double_param_by_name()

```
int  KNITRO_API KTR_get_double_param_by_name
         (KTR_context_ptr  kc, const char * const  name, double * const  value);
```

Get a double valued parameter using its string name.

KTR_get_int_param()

```
int  KNITRO_API KTR_get_int_param
          (KTR_context_ptr  kc, const int  param_id, int * const  value);
```

Get an integer valued parameter using its integer identifier (see *KNITRO user options*).

KTR_get_double_param()

```
int  KNITRO_API KTR_get_double_param
          (KTR_context_ptr  kc, const int  param_id, double * const  value);
```

Get a double valued parameter using its integer identifier (see *KNITRO user options*).

KTR_get_param_name()

```
int KNITRO_API KTR_get_param_name
    (      KTR_context_ptr  kc,
     const int  param_id,
           char * const  param_name,
           const size_t  output_size);
```

Sets the string param_name to the name of parameter indexed by integer identifier param_id (see *KNITRO user options*) and returns 0. Returns an error if param_id does not correspond to any parameter, or if the parameter output_size (the size of char array param_name) is less than the size of the parameter's description.

KTR_get_param_doc()

```
int KNITRO_API KTR_get_param_doc
    (      KTR_context_ptr kc,
     const int  param_id,
           char * const  description,
     const size_t  output_size);
```

Sets the string description to the description of the parameter indexed by integer identifier param_id (see *KNITRO user options*) and its possible values and returns 0. Returns an error if param_id does not correspond to any parameter, or if the parameter output_size (the size of char array description) is less than the size of the parameter's description.

KTR_get_param_type()

```
int KNITRO_API KTR_get_param_type
    (       KTR_context_ptr  kc,
     const int  param_id,
           int * const  param_type);
```

Sets the int * param_type to the parameter type of parameter indexed by integer identifier param_id (see *KNITRO user options*). Possible values are KTR_PARAMTYPE_INT, KTR_PARAMTYPE_FLOAT, KTR_PARAMTYPE_STRING. Returns an error if param_id does not correspond to any parameter.

KTR_get_num_param_values()

```
int KNITRO_API KTR_get_num_param_values
    (      KTR_context_ptr  kc,
     const int  param_id,
           int * const  num_param_values);
```

Set the int * num_param_values to the number of possible parameter values for parameter indexed by integer identifier param_id and returns 0. If there is not a finite number of possible values, num_param_values will be zero. Returns an error if param_id does not correspond to any parameter.

KTR_get_param_value_doc()

```
int KNITRO_API KTR_get_param_value_doc
    (      KTR_context_ptr  kc,
     const int  param_id,
     const int  value_id,
           char * const  param_value_string,
     const size_t  output_size);
```

Set string param_value_string to the description of parameter value indexed by [param_id][value_id]. Returns an error if param_id does not correspond to any parameter, or if value_id is greater than the number of possible parameter values, or if there are not a finite number of possible parameter values, or if the parameter output_size (the size of char array param_value_string) is less than the size of the parameter's description.

KTR_get_param_id()

```
int KNITRO_API KTR_get_param_id
    (      KTR_context_ptr  kc,
     const char * const  name,
           int * const  param_id);
```

Gets the integer value corresponding to the parameter name input and copies it into param_id input. Returns zero if successful and an error code otherwise.

KTR_get_release()

```
void  KNITRO_API KTR_get_release(const int  length, char * const  release);
```

Copy the KNITRO release name into *release*. This variable must be preallocated to have *length* elements, including the string termination character. For compatibility with future releases, please allocate at least 15 characters.

KTR_load_tuner_file()

```
int  KNITRO_API KTR_load_tuner_file
    (KTR_context_ptr  kc, const char * const  filename);
```

Similar to `KTR_load_param_file()` but specifically allows user to specify a file of options (and option values) to explore for the KNITRO-Tuner (see *The KNITRO-Tuner*).

KTR_set_feastols()

```
int  KNITRO_API KTR_set_feastols
    (      KTR_context_ptr kc,
     const double * const  cFeasTols,
     const double * const  xFeasTols,
     const double * const  ccFeasTols);
```

Set an array of absolute feasibility tolerances (one for each constraint and variable) to use for the termination tests. The user options `KTR_PARAM_FEASTOL` / `KTR_PARAM_FEASTOLABS` define a single tolerance that is applied equally to every constraint and variable. This API function allows the user to specify separate feasibility termination tolerances for each constraint and variable. Values specified through this function will override the value determined by `KTR_PARAM_FEASTOL` / `KTR_PARAM_FEASTOLABS`. The tolerances should be positive values. If a non-positive value is specified, that constraint or variable will use the standard tolerances based on `KTR_PARAM_FEASTOL` / `KTR_PARAM_FEASTOLABS`. Array `cFeasTols` has length *m*, array `xFeasTols` has length *n*, and array `ccFeasTols` has length *ncc*, where *ncc* is the number of complementarity constraints added through `KTR_addcompcons()`. The regular constraints are considered to be satisfied when:

```
c[i] - cUpBnds[i] <= cFeasTols[i]  for all i=1..m, and
cLoBnds[i] - c[i] <= cFeasTols[i]  for all i=1..m.
```

The variables are considered to be satisfied when:

```
x[i] - xUpBnds[i] <= xFeasTols[i]  for all i=1..n, and
xLoBnds[i] - x[i] <= xFeasTols[i]  for all i=1..n.
```

The complementarity constraints are considered to be satisfied when:

```
min(x1_i, x2_i) <= ccFeasTols[i]  for all i=1..ncc,
```

where x1 and x2 are the arrays of complementary pairs. If there are no regular (or complementarity) constraints set `cFeasTols=NULL` (or `ccFeasTols=NULL`). If `cFeasTols/xFeasTols/ccFeasTols=NULL`, then the standard tolerances will be used. KNITRO makes a local copy of all inputs, so the application may free memory after the call. This routine must be called after calling `KTR_init_problem()` / `KTR_mip_init_problem()` and after any calls to `KTR_addcompcons()`. It must be called before calling `KTR_solve()` / `KTR_mip_solve()`. Returns 0 if OK, nonzero if error.

KTR_set_names()

```
int  KNITRO_API KTR_set_names
    (      KTR_context_ptr kc,
     const char   * const  objName,
           char   * const  varNames[],
           char   * const  conNames[]);
```

Set names for model components passed in by the user/modeling language so that KNITRO can internally print out these names. KNITRO makes a local copy of all inputs, so the application may free memory after the call. This routine must be called after calling `KTR_init_problem()` / `KTR_mip_init_problem()` and before calling `KTR_solve()` / `KTR_mip_solve()`. Returns 0 if OK, nonzero if error.

Problem modification

KTR_addcompcons()

```
int  KNITRO_API KTR_addcompcons (KTR_context_ptr    kc,
                                 const int          numCompConstraints,
                                 const int * const  indexList1,
                                 const int * const  indexList2);
```

This function adds complementarity constraints to the problem. It must be called after `KTR_init_problem()` and before `KTR_solve()`. The two lists are of equal length, and contain matching pairs of variable indices. Each pair defines a complementarity constraint between the two variables. The function can be called more than once to accumulate a long list of complementarity constraints in KNITRO's internal problem definition. Returns 0 if OK, or a negative value on error.

KTR_chgvarbnds()

```
int  KNITRO_API KTR_chgvarbnds (      KTR_context_ptr          kc,
                                const double          * const  xLoBnds,
                                const double          * const  xUpBnds);
```

This function prepares KNITRO to re-optimize the current problem after modifying the variable bounds from a previous solve. The arrays *xLoBnds* and *xUpBnds* have the same meaning as in `KTR_init_problem()` and must be specified completely. This function must be called after `KTR_init_problem()` and precedes a call to `KTR_solve()`. Returns 0 if OK, nonzero if error.

Solving

Problem structure is passed to KNITRO using `KTR_init_problem()`. Functions `KTR_solve()` and `KTR_mip_solve()` have the same parameter list. Function `KTR_solve()` should be used for models where

all the variables are continuous, while KTR_mip_solve() should be used for models with one or more binary or integer variables.

Applications must provide a means of evaluating the nonlinear objective, constraints, first derivatives, and (optionally) second derivatives. (First derivatives are also optional, but highly recommended.) If the application provides callback functions for making evaluations, then a single call to KTR_solve() will return the solution. Alternatively, the application can employ a reverse communications driver. In this case, KTR_solve() returns a status code whenever it needs evaluation data (see examples/C/reverseCommExample.c).

The typical calling sequence is:

```
KTR_new
KTR_init_problem
KTR_set_xxx_param (set any number of parameters)
KTR_solve (a single call, or a reverse communications loop)
KTR_free
```

Calling sequence if the same problem is to be solved again, with different parameters, a different start point, or a change to the bounds on the variables:

```
KTR_new
KTR_init_problem
KTR_set_xxx_param (set any number of parameters)
KTR_solve (a single call, or a reverse communications loop)
KTR_restart (if changing the initial point or some user parameters)
KTR_chgvarbnds (if modifying variable bounds)
KTR_set_xxx_param (set any number of parameters)
KTR_solve (a single call, or a reverse communications loop)
KTR_free
```

Note: KTR_set_xxx_param() may also be called before KTR_init_problem() (and gradopt and hessopt *must* be set before KTR_init_problem() and remain constant).

API

KTR_init_problem()

```
int  KNITRO_API KTR_init_problem (KTR_context_ptr       kc,
                                  const int             n,
                                  const int             objGoal,
                                  const int             objType,
                                  const double * const  xLoBnds,
                                  const double * const  xUpBnds,
                                  const int             m,
                                  const int    * const  cType,
                                  const double * const  cLoBnds,
                                  const double * const  cUpBnds,
                                  const int             nnzJ,
                                  const int    * const  jacIndexVars,
                                  const int    * const  jacIndexCons,
                                  const int             nnzH,
                                  const int    * const  hessIndexRows,
                                  const int    * const  hessIndexCols,
                                  const double * const  xInitial,
                                  const double * const  lambdaInitial);
```

These functions pass the optimization problem definition to KNITRO, where it is copied and stored internally until KTR_free() is called. Once initialized, the problem may be solved any number of times with different user options or initial points (see the KTR_restart() call below). Array arguments passed to KTR_init_problem() or KTR_mip_init_problem() are not referenced again and may be freed or reused if desired. In the description below, some programming macros are mentioned as alternatives to fixed numeric constants; e.g., KTR_OBJGOAL_MINIMIZE. These macros are defined in knitro.h. Returns 0 if OK, nonzero if error.

Arguments:

- *kc* is the KNITRO context pointer. Do not modify its contents.
- *n* is a scalar specifying the number of variables in the problem; i.e., the length of *x*.
- *objGoal* is the optimization goal (see KTR_OBJGOAL_MINIMIZE, KTR_OBJGOAL_MAXIMIZE).
- *objType* is a scalar that describes the type of objective function *f(x)* (see KTR_OBJTYPE_GENERAL, KTR_OBJTYPE_LINEAR, KTR_OBJTYPE_QUADRATIC).
- *xLoBnds* is an array of length *n* specifying the lower bounds on *x*. *xLoBnds[i]* must be set to the lower bound of the corresponding i-th variable x_i. If the variable has no lower bound, set *xLoBnds[i]* to be -KTR_INFBOUND. For binary variables, set *xLoBnds[i]=0*.
- *xUpBnds* is an array of length *n* specifying the upper bounds on *x*. *xUpBnds[i]* must be set to the upper bound of the corresponding i-th variable. If the variable has no upper bound, set *xUpBnds[i]* to be KTR_INFBOUND. For binary variables, set *xUpBnds[i]=1*.

Note: If *xLoBnds* or *xUpBnds* are NULL, then KNITRO assumes all variables are unbounded in that direction.

- *m* is a scalar specifying the number of constraints *c(x)*.
- *cType* is an array of length *m* that describes the types of the constraint functions *c(x)* (see KTR_CONTYPE_GENERAL, KTR_CONTYPE_LINEAR, KTR_CONTYPE_QUADRATIC).
- *cLoBnds* is an array of length *m* specifying the lower bounds on the constraints *c(x)* . *cLoBnds[i]* must be set to the lower bound of the corresponding *i*-th constraint. If the constraint has no lower bound, set *cLoBnds[i]* to be -KTR_INFBOUND. If the constraint is an equality, then *cLoBnds[i]* should equal *cUpBnds[i]*.
- *cUpBnds* is an array of length *m* specifying the upper bounds on the constraints *c(x)* . *cUpBnds[i]* must be set to the upper bound of the corresponding *i*-th constraint. If the constraint has no upper bound, set *cUpBnds[i]* to be KTR_INFBOUND. If the constraint is an equality, then *cLoBnds[i]* should equal *cUpBnds[i]*.
- *nnzJ* is a scalar specifying the number of nonzero elements in the sparse constraint Jacobian.
- *jacIndexVars* is an array of length *nnzJ* specifying the variable indices of the constraint Jacobian nonzeros. If *jacIndexVars[i]=j*, then *jac[i]* refers to the *j*-th variable, where *jac* is the array of constraint Jacobian nonzero elements passed in the call to KTR_solve().

  jacIndexCons[i] and *jacIndexVars[i]* determine the row numbers and the column numbers, respectively, of the nonzero constraint Jacobian element *jac[i]*.

Note: C array numbering starts with index 0. Therefore, the *j*-th variable x_j maps to array element x[j], and $0 \leq j < n$.

- *jacIndexCons* is an array of length *nnzJ* specifying the constraint indices of the constraint Jacobian nonzeros. If *jacIndexCons[i]=k*, then *jac[i]* refers to the *k*-th constraint, where *jac* is the array of constraint Jacobian nonzero elements passed in the call to KTR_solve().

  jacIndexCons[i] and *jacIndexVars[i]* determine the row numbers and the column numbers, respectively, of the nonzero constraint Jacobian element *jac[i]*.

Note: C array numbering starts with index 0. Therefore, the k-th constraint c_k maps to array element c[k], and $0 \leq k < m$.

- *nnzH* is a scalar specifying the number of nonzero elements in the sparse Hessian of the Lagrangian. Only nonzeros in the upper triangle (including diagonal nonzeros) should be counted.

Note: If user option `hessopt` is not set to `KTR_HESSOPT_EXACT`, then Hessian nonzeros will not be used. In this case, set *nnzH=0*, and pass NULL pointers for *hessIndexRows* and *hessIndexCols*.

- *hessIndexRows* is an array of length *nnzH* specifying the row number indices of the Hessian nonzeros.

  hessIndexRows[i] and *hessIndexCols[i]* determine the row numbers and the column numbers, respectively, of the nonzero Hessian element *hess[i]*, where *hess* is the array of Hessian elements passed in the call `KTR_solve()`.

Note: Row numbers are in the range *0 , ... , n - 1*.

- *hessIndexCols* is an array of length *nnzH* specifying the column number indices of the Hessian nonzeros.

  hessIndexRows[i] and *hessIndexCols[i]* determine the row numbers and the column numbers, respectively, of the nonzero Hessian element *hess[i]*, where *hess* is the array of Hessian elements passed in the call to `KTR_solve()`.

Note: Column numbers are in the range *0 , ... , n - 1*.

- *xInitial* is an array of length *n* containing an initial guess of the solution vector *x*. If the application prefers to let KNITRO make an initial guess, then pass a NULL pointer for *xInitial*.
- *lambdaInitial* is an array of length *m+n* containing an initial guess of the Lagrange multipliers for the constraints *c(x)* and bounds on the variables *x*. The first *m* components of *lambdaInitial* are multipliers corresponding to the constraints specified in *c(x)*, while the last *n* components are multipliers corresponding to the bounds on *x*. If the application prefers to let KNITRO make an initial guess, then pass a NULL pointer for *lambdaInitial*.

`KTR_solve``()`

```
int  KNITRO_API KTR_solve ( KTR_context_ptr       kc,
                            double * const  x,
                            double * const  lambda,
                      const int             evalStatus,
                            double * const  obj,
                      const double * const  c,
                            double * const  objGrad,
                            double * const  jac,
                      const double * const  hess,
                            double * const  hessVector,
                            void   * const  userParams);
```

Arguments:

- *kc* is the KNITRO context pointer. Do not modify its contents.
- *x* is an array of length *n* output by KNITRO. If `KTR_solve()` returns `KTR_RC_OPTIMAL`, then *x* contains the solution.

Reverse communications mode: upon return, *x* contains the value of unknowns at which KNITRO needs more problem information. For continuous problems, if user option `newpoint` is set to `KTR_NEWPOINT_USER` and `KTR_solve()` returns `KTR_RC_NEWPOINT`, then *x* contains a newly accepted iterate, but not the final solution.

- *lambda* is an array of length *m*+*n* output by KNITRO. If `KTR_solve()` returns zero, then *lambda* contains the multiplier values at the solution. The first *m* components of *lambda* are multipliers corresponding to the constraints specified in *c(x)*, while the last *n* components are multipliers corresponding to the bounds on *x*.

  Reverse communications mode: upon return, *lambda* contains the value of multipliers at which KNITRO needs more problem information.

- *evalStatus* is a scalar input to KNITRO used only in reverse communications mode. A value of zero means the application successfully computed the problem information requested by KNITRO at *x* and *lambda*. A nonzero value means the application failed to compute problem information (e.g., if a function is undefined at the requested value *x*). Set to 0 for callback mode.

- *obj* is a scalar holding the value of *f(x)* at the current *x*. If `KTR_solve()` returns `KTR_RC_OPTIMAL`, then *obj* contains the value of the objective function *f(x)* at the solution.

Note: Reverse communications mode: if `KTR_solve()` returns `KTR_RC_EVALFC`, then *obj* must be filled with the value of *f(x)* computed at *x* before `KTR_solve()` is called again.

- *c* is an array of length *m* used only in reverse communications mode. If `KTR_solve()` returns `KTR_RC_EVALFC`, then *c* must be filled with the value of *c(x)* computed at *x* before `KTR_solve()` is called again. Set to NULL for callback mode.

- *objGrad* is an array of length *n* used only in reverse communications mode. If `KTR_solve()` returns `KTR_RC_EVALGA`, then *objGrad* must be filled with the value of $\nabla f(x)$ computed at *x* before `KTR_solve()` is called again. Set to NULL for callback mode.

- *jac* is an array of length *nnzJ* used only in reverse communications mode. If `KTR_solve()` returns `KTR_RC_EVALGA`, then *jac* must be filled with the constraint Jacobian *J(x)* computed at *x* before `KTR_solve()` is called again. Entries are stored according to the sparsity pattern defined in `KTR_init_problem()`. Set to NULL for callback mode.

Note: If `gradopt` is set to compute finite differences for first derivatives, then `KTR_solve()` will modify *objGrad* and *jac*; otherwise, these arguments are not modified.

- *hess* is an array of length *nnzH* used only in reverse communications mode, and only if option `hessopt` is set to `KTR_HESSOPT_EXACT`. If `KTR_solve()` returns `KTR_RC_EVALH`, then hess must be filled with the Hessian of the Lagrangian computed at *x* and *lambda* before `KTR_solve()` is called again. Entries are stored according to the sparsity pattern defined in `KTR_init_problem()`. Set to NULL for callback mode.

- *hessVector* is an array of length *n* used only in reverse communications mode, and only if option `hessopt` is set to `KTR_HESSOPT_PRODUCT`. If `KTR_solve()` returns `KTR_RC_EVALHV`, then the Hessian of the Lagrangian at *x* and *lambda* should be multiplied by *hessVector*, and the result placed in *hessVector* before `KTR_solve()` is called again. Set to NULL for callback mode.

- *userParams* is a pointer to a structure used only in callback mode. The pointer is provided so the application can pass additional parameters needed for its callback routines. If the application needs no additional parameters, then pass a NULL pointer.

The return value of `KTR_solve()` and `KTR_mip_solve()` specifies the final exit code from the optimization process. If the return value is 0 (`KTR_RC_OPTIMAL`) or negative, then KNITRO has finished solving. In reverse communications mode the return value may be positive, in which case it specifies a request for additional problem

information, after which the application should call KNITRO again. A detailed description of the possible return values is given in *Return codes*.

KTR_restart()

```
int  KNITRO_API KTR_restart (KTR_context_ptr       kc,
                             const double * const  xInitial,
                             const double * const  lambdaInitial);
```

This function can be called to start another KTR_solve() sequence after making small modifications. The problem structure cannot be changed (e.g., KTR_init_problem() cannot be called between KTR_solve() and KTR_restart()). However, user options (with the exception of gradopt and hessopt) can be modified, and a new initial value can be passed with KTR_restart(). KNITRO parameter values are not changed by this call. The sample program examples/C/restartExample.c uses KTR_restart() to solve the same problem from the same start point, but each time changing the interior point bar_murule option to a different value. Returns 0 if OK, nonzero if error.

Note: If output to a file is enabled, this will erase the current file.

KTR_mip_init_problem()

```
int  KNITRO_API KTR_mip_init_problem( KTR_context_ptr       kc,
                                      const int             n,
                                      const int             objGoal,
                                      const int             objType,
                                      const int             objFnType,
                                      const int    * const  xType,
                                      const double * const  xLoBnds,
                                      const double * const  xUpBnds,
                                      const int             m,
                                      const int    * const  cType,
                                      const int    * const  cFnType,
                                      const double * const  cLoBnds,
                                      const double * const  cUpBnds,
                                      const int             nnzJ,
                                      const int    * const  jacIndexVars,
                                      const int    * const  jacIndexCons,
                                      const int             nnzH,
                                      const int    * const  hessIndexRows,
                                      const int    * const  hessIndexCols,
                                      const double * const  xInitial,
                                      const double * const  lambdaInitial);
```

See KTR_init_problem() above. The only difference is the addition of the following arguments.

- *objFnType* is a scalar that describes the convexity status of the objective function *f(x)* (MIP only; see KTR_FNTYPE_UNCERTAIN, KTR_FNTYPE_CONVEX, KTR_FNTYPE_NONCONVEX).
- *xType* is an array of length *n* that describes the types of variables *x* (MIP only; see KTR_VARTYPE_CONTINUOUS, KTR_VARTYPE_INTEGER, KTR_VARTYPE_BINARY).
- *cFnType* is an array of length *m* that describes the convexity status of the constraint functions *c(x)* (MIP only; see KTR_FNTYPE_UNCERTAIN, KTR_FNTYPE_CONVEX, KTR_FNTYPE_NONCONVEX).

Returns 0 if OK, nonzero if error.

KTR_mip_set_branching_priorities()

```
int  KNITRO_API KTR_mip_set_branching_priorities(KTR_context_ptr    kc,
                                                 const int * const  xPriorities);
```

This function can be used to set the branching priorities for integer variables when using the MIP features in KNITRO. Priorities must be positive numbers (variables with non-positive values are ignored). Variables with higher priority values will be considered for branching before variables with lower priority values. When priorities for a subset of variables are equal, the branching rule is applied as a tiebreaker. Array *xPriorities* has length *n*, and values for continuous variables are ignored. KNITRO makes a local copy of all inputs, so the application may free memory after the call. This routine must be called after calling `KTR_mip_init_problem()` and before calling `KTR_mip_solve()`. Returns 0 if OK, nonzero if error.

KTR_mip_solve()

```
int  KNITRO_API KTR_mip_solve( KTR_context_ptr       kc,
                               double * const  x,
                               double * const  lambda,
                         const int             evalStatus,
                               double * const  obj,
                               double * const  c,
                               double * const  objGrad,
                               double * const  jac,
                               double * const  hess,
                               double * const  hessVector,
                               void   * const  userParams);
```

Call KNITRO to solve the MIP problem, similar to `KTR_solve()`. If the application provides callback functions for evaluating the function, constraints, and derivatives, then a single call to `KTR_mip_solve()` returns the solution. Otherwise, KNITRO operates in reverse communications mode and returns a status code that may request another call.

Returns one of the status codes "KTR_RC_*" (see *Return codes*).

KTR_set_findiff_relstepsizes()

```
int  KNITRO_API KTR_set_findiff_relstepsizes
    (      KTR_context_ptr kc,
     const double * const  relStepSizes);
```

Set an array of relative stepsizes to use for the finite-difference gradient/Jacobian computations when using finite-difference first derivatives. Finite-difference step sizes "delta" in KNITRO are computed as:

```
delta[i] = relStepSizes[i]*max(abs(x[i]),1);
```

The default relative step sizes for each component of *x* are sqrt(eps) for forward finite differences, and eps^(1/3) for central finite differences. Use this function to overwrite the default values. Array `relStepSizes` has length *n* and all values should be non-zero. If `relStepSizes` is set to NULL, then default KNITRO values will be used. KNITRO makes a local copy of all inputs, so the application may free memory after the call. This routine must be called after calling `KTR_init_problem()` and before calling `KTR_solve()`. Returns 0 if OK, nonzero if error.

Callbacks

To solve a nonlinear optimization problem, KNITRO needs the application to supply information at various trial points. The KNITRO C language API has two modes of operation for obtaining problem information: *callback* and *reverse communication*. With callback mode the application provides C language function pointers that KNITRO may call to evaluate the functions, gradients, and Hessians. With reverse communication, the function `KTR_solve()` (or `KTR_mip_solve()`) returns one of the constants listed below to tell the application what it needs, and then waits to be called again with the new problem information. KNITRO specifies a trial point with a new vector of variable values x, and sometimes a corresponding vector of Lagrange multipliers λ.

For simplicity, the callback functions

- KTR_set_func_callback

- KTR_set_grad_callback
- KTR_set_hess_callback
- KTR_set_ms_process_callback
- KTR_set_mip_node_callback

(described in detail below) all use the same `KTR_callback()` function prototype defined here.

```
typedef int KTR_callback (const int             evalRequestCode,
                          const int             n,
                          const int             m,
                          const int             nnzJ,
                          const int             nnzH,
                          const double * const  x,
                          const double * const  lambda,
                                double * const  obj,
                                double * const  c,
                                double * const  objGrad,
                                double * const  jac,
                                double * const  hessian,
                                double * const  hessVector,
                                void   *        userParams);
```

At a trial point, KNITRO may ask the application to:

- evaluate $f(x)$ and $c(x)$ at x (`KTR_RC_EVALFC`).
- evaluate $\nabla f(x)$ and $\nabla c(x)$ at x (`KTR_RC_EVALGA`).
- evaluate the Hessian matrix of the problem at x and λ normally (`KTR_RC_EVALH`), or without the objective component included (`KTR_RC_EVALH_NO_F`).
- evaluate the Hessian matrix times a vector v at x and λ normally (`KTR_RC_EVALHV`), or without the objective component included (`KTR_RC_EVALHV_NO_F`).

The constants KTR_RC_* are return codes defined in `knitro.h` and listed in *Return codes*.

The argument *lambda* is not defined when requesting `KTR_RC_EVALFC` or `KTR_RC_EVALGA`. Usually, applications define three callback functions, one for `KTR_RC_EVALFC`, one for `KTR_RC_EVALGA`, and one for `KTR_RC_EVALH` / `KTR_RC_EVALHV`. It is possible to combine `KTR_RC_EVALFC` and `KTR_RC_EVALGA` into a single function, because x changes only for an `KTR_RC_EVALFC` request. This is advantageous if the application evaluates functions and their derivatives at the same time. Pass the same callback function in `KTR_set_func_callback()` and `KTR_set_grad_callback()`, have it populate *obj*, *c*, *objGrad*, and *jac* for an `KTR_RC_EVALFC` request, and do nothing for an `KTR_RC_EVALGA` request. Do not combine `KTR_RC_EVALFC` and `KTR_RC_EVALGA` if `hessopt` = `KTR_HESSOPT_FINITE_DIFF`, because the finite difference Hessian changes x and calls `KTR_RC_EVALGA` without calling `KTR_RC_EVALFC` first. It is not possible to combine `KTR_RC_EVALH` / `KTR_RC_EVALHV` because *lambda* changes after the `KTR_RC_EVALFC` call.

The *userParams* argument is an arbitrary pointer passed from the KNITRO `KTR_solve()` call to the callback. It should be used to pass parameters defined and controlled by the application, or left null if not used. KNITRO does not modify or dereference the *userParams* pointer.

Callbacks should return 0 if successful, a negative error code if not. Possible unsuccessful (negative) error codes for the "func", "grad", and "hess" callback functions include `KTR_RC_CALLBACK_ERR` (for generic callback errors), and `KTR_RC_EVAL_ERR` (for evaluation errors, e.g log(-1)).

In addition, for the "func", "newpoint", "ms_process" and "mip_node" callbacks, the user may set the `KTR_RC_USER_TERMINATION` return code to force KNITRO to terminate based on some user-defined condition.

`KTR_set_func_callback()`

```
int  KNITRO_API KTR_set_func_callback (KTR_context_ptr       kc,
                                       KTR_callback * const  fnPtr);
```

Set the callback function that evaluates *obj* and *c* at *x*. It may also evaluate *objGrad* and *jac* if KTR_RC_EVALFC and KTR_RC_EVALGA are combined into a single call. Do not modify *hessian* or *hessVector*.

KTR_set_grad_callback()

```
int  KNITRO_API KTR_set_grad_callback (KTR_context_ptr       kc,
                                       KTR_callback * const  fnPtr);
```

Set the callback function that evaluates *objGrad* and *jac* at *x*. It may do nothing if KTR_RC_EVALFC and KTR_RC_EVALGA are combined into a single call. Do not modify *hessian* or *hessVector*.

KTR_set_hess_callback()

```
int  KNITRO_API KTR_set_hess_callback (KTR_context_ptr       kc,
                                       KTR_callback * const  fnPtr);
```

Set the callback function that evaluates second derivatives at (*x*, *lambda*). If *evalRequestCode* equals KTR_RC_EVALH, then the function must return nonzeros in *hessian*. If it equals KTR_RC_EVALHV, then the function multiplies second derivatives by *hessVector* and returns the product in *hessVector*. Do not modify *obj*, *c*, *objGrad*, or *jac*.

KTR_set_newpt_callback()

```
typedef int KTR_newpt_callback (KTR_context_ptr       kc,
                                const int             n,
                                const int             m,
                                const int             nnzJ,
                                const double * const  x,
                                const double * const  lambda,
                                const double          obj,
                                const double * const  c,
                                const double * const  objGrad,
                                const double * const  jac,
                                      void   *        userParams);

int  KNITRO_API KTR_set_newpt_callback (KTR_context_ptr             kc,
                                        KTR_newpt_callback * const  fnPtr);
```

Set the callback function that is invoked after KNITRO computes a new estimate of the solution point (i.e., after every major iteration). The function should not modify any KNITRO arguments. Argument *kc* is the context pointer for the current problem being solved inside KNITRO (either the main single-solve problem, or a subproblem when using multi-start, Tuner, etc.). This can then be used to call KNITRO functions to get problem information from within the callback. Arguments *x* and *lambda* contain the new point and values. Arguments *obj* and *c* contain objective and constraint values at *x*, and *objGrad* and *jac* contain the objective gradient and constraint Jacobian at *x*.

KTR_set_ms_process_callback()

```
int  KNITRO_API KTR_set_ms_process_callback (KTR_context_ptr       kc,
                                             KTR_callback * const  fnPtr);
```

This callback function is for multistart (MS) problems only. Set the callback function that is invoked after KNITRO finishes processing a multistart solve. The function should not modify any KNITRO arguments. Arguments *x* and *lambda* contain the solution from the last solve. Arguments *obj* and *c* contain objective and constraint values at *x*. First and second derivative arguments are not currently defined and should not be examined.

KTR_set_mip_node_callback()

```
int  KNITRO_API KTR_set_mip_node_callback (KTR_context_ptr       kc,
                                           KTR_callback * const  fnPtr);
```

This callback function is for mixed integer (MIP) problems only. Set the callback function that is invoked after KNITRO finishes processing a node on the branch-and-bound tree (i.e., after a relaxed subproblem solve in the branch-and-bound procedure). The function should not modify any KNITRO arguments. Arguments *x* and *lambda* contain the solution from the node solve. Arguments *obj* and *c* contain objective and constraint values at *x*. First and second derivative arguments are not currently defined and should not be examined.

KTR_set_ms_initpt_callback()

```
typedef int  KTR_ms_initpt_callback (const int             nSolveNumber,
                                     const int             n,
                                     const int             m,
                                     const double * const  xLoBnds,
                                     const double * const  xUpBnds,
                                           double * const  x,
                                           double * const  lambda,
                                           void   * const  userParams);


int  KNITRO_API KTR_set_ms_initpt_callback (KTR_context_ptr                kc,
                                            KTR_ms_initpt_callback * const  fnPtr);
```

This callback allows applications to define a routine that specifies an initial point before each local solve in the multi-start procedure. On input, arguments *x* and *lambda* are the randomly generated initial points determined by KNITRO, which can be overwritten by the user. The argument *nSolveNumber* is the number of the multistart solve. Return 0 if successful, a negative error code if not. Use KTR_set_ms_initpt_callback to set this callback function.

KTR_set_puts_callback()

```
typedef int  KTR_puts (const char * const  str,
                       void * const  userParams);

int  KNITRO_API KTR_set_puts_callback (KTR_context_ptr   kc,
                                       KTR_puts * const  fnPtr);
```

Applications can set a "put string" callback function to handle output generated by the KNITRO solver. By default KNITRO prints to *stdout* or a file named `knitro.log`, as determined by `KTR_PARAM_OUTMODE`. The `KTR_puts()` function takes a *userParams* argument which is a pointer passed directly from `KTR_solve()`. Note that *userParams* will be a NULL pointer until defined by an application call to `KTR_new_puts()` or `KTR_solve()`. The `KTR_puts()` function should return the number of characters that were printed.

Reading solution properties

KTR_get_number_FC_evals()

```
int  KNITRO_API KTR_get_number_FC_evals (const KTR_context_ptr  kc);
```

Return the number of function evaluations requested by `KTR_solve()`. A single request evaluates the objective and all constraint functions. Returns a negative number if there is a problem with *kc*.

KTR_get_number_GA_evals()

```
int  KNITRO_API KTR_get_number_GA_evals (const KTR_context_ptr  kc);
```

Return the number of gradient evaluations requested by `KTR_solve()`. A single request evaluates first derivatives of the objective and all constraint functions. Returns a negative number if there is a problem with *kc*.

KTR_get_number_H_evals()

```
int  KNITRO_API KTR_get_number_H_evals (const KTR_context_ptr  kc);
```

Return the number of Hessian evaluations requested by KTR_solve(). A single request evaluates second derivatives of the objective and all constraint functions. Returns a negative number if there is a problem with *kc*.

KTR_get_number_HV_evals()

```
int  KNITRO_API KTR_get_number_HV_evals (const KTR_context_ptr  kc);
```

Return the number of Hessian-vector products requested by KTR_solve(). A single request evaluates the product of the Hessian of the Lagrangian with a vector submitted by KNITRO. Returns a negative number if there is a problem with *kc*.

KTR_get_number_iters()

```
int  KNITRO_API KTR_get_number_iters (const KTR_context_ptr  kc);
```

Return the number of iterations made by KTR_solve(). Returns a negative number if there is a problem with *kc*. For continuous problems only.

KTR_get_number_cg_iters()

```
int  KNITRO_API KTR_get_number_cg_iters (const KTR_context_ptr  kc);
```

Return the number of conjugate gradients (CG) iterations made by KTR_solve(). Returns a negative number if there is a problem with *kc*. For continuous problems only.

KTR_get_abs_feas_error()

```
double  KNITRO_API KTR_get_abs_feas_error (const KTR_context_ptr  kc);
```

Return the absolute feasibility error at the solution. Returns a negative number if there is a problem with *kc*. For continuous problems only.

KTR_get_rel_feas_error()

```
double  KNITRO_API KTR_get_rel_feas_error (const KTR_context_ptr  kc);
```

Return the relative feasibility error at the solution. Returns a negative number if there is a problem with *kc*. For continuous problems only.

KTR_get_abs_opt_error()

```
double  KNITRO_API KTR_get_abs_opt_error (const KTR_context_ptr  kc);
```

Return the absolute optimality error at the solution. Returns a negative number if there is a problem with *kc*. For continuous problems only.

KTR_get_rel_opt_error()

```
double  KNITRO_API KTR_get_rel_opt_error (const KTR_context_ptr  kc);
```

Return the relative optimality error at the solution. Returns a negative number if there is a problem with *kc*. For continuous problems only.

KTR_get_solution()

```
int  KNITRO_API KTR_get_solution (const KTR_context_ptr kc,
                                  int     * const status,
                                  double  * const obj,
                                  double  * const x,
                                  double  * const lambda);
```

Return the solution status, objective, primal and dual variables. The status and objective value scalars are returned as pointers that need to be de-referenced to get their values. The arrays *x* and *lambda* must be allocated by the user. Returns 0 if call is successful; <0 if there is an error.

KTR_get_constraint_values()

```
int  KNITRO_API KTR_get_constraint_values (const KTR_context_ptr kc,
                                           double  * const c);
```

Return the values of the constraint vector in *c*. The array *c* must be allocated by the user. Returns 0 if call is successful; <0 if there is an error.

KTR_get_objgrad_values()

```
int  KNITRO_API KTR_get_objgrad_values (const KTR_context_ptr kc,
                                        double  * const objGrad);
```

Return the values of the objective gradient vector in *objGrad*. The array *objGrad* must be allocated by the user. It is a dense array of dimension "n" (where "n" is the number of variables in the problem). Returns 0 if call is successful; <0 if there is an error. For continuous problems only.

KTR_get_jacobian_values()

```
int  KNITRO_API KTR_get_jacobian_values (const KTR_context_ptr kc,
                                         double  * const jac);
```

Return the values of the constraint Jacobian in *jac*. The Jacobian values returned correspond to the non-zero sparse Jacobian indices provided by the user in KTR_init_problem(). The array *jac* must be allocated by the user. Returns 0 if call is successful; <0 if there is an error. For continuous problems only.

KTR_get_hessian_values()

```
int  KNITRO_API KTR_get_hessian_values (const KTR_context_ptr kc,
                                        double  * const hess);
```

Return the values of the Hessian (or possibly Hessian approximation) in *hess*. This routine is currently only valid if 1 of the 2 following cases holds:

1. KTR_HESSOPT_EXACT (presolver on or off), or;
2. KTR_HESSOPT_BFGS or KTR_HESSOPT_SR1, but only with the KNITRO presolver off (i.e. KTR_PRESOLVE_NONE).

In all other cases, either KNITRO does not have an internal representation of the Hessian (or Hessian approximation), or the internal Hessian approximation corresponds only to the presolved problem form and may not be valid for the original problem form. In these cases *hess* is left unmodified, and the routine has return code 1.

Note that in case 2 above (KTR_HESSOPT_BFGS or KTR_HESSOPT_SR1) the values returned in *hess* are the upper triangular values of the dense quasi-Newton Hessian approximation stored row-wise. There are ((n*n - n)/2 + n) such values (where "n" is the number of variables in the problem. These values may be quite different from the values of the exact Hessian.

When KTR_HESSOPT_EXACT (case 1 above) the Hessian values returned correspond to the non-zero sparse Hessian indices provided by the user in KTR_init_problem().

The array *hess* must be allocated by the user. Returns 0 if call is successful; 1 if *hess* was not set because KNITRO does not have a valid Hessian for the model stored; <0 if there is an error. For continuous problems only.

KTR_get_mip_num_nodes()

```
int  KNITRO_API KTR_get_mip_num_nodes (const KTR_context_ptr kc);
```

Return the number of nodes processed in the MIP solve. Returns a negative number if there is a problem with *kc*.

KTR_get_mip_num_solves()

```
int  KNITRO_API KTR_get_mip_num_solves (const KTR_context_ptr kc);
```

Return the number of continuous subproblems processed in the MIP solve. Returns a negative number if there is a problem with *kc*.

KTR_get_mip_abs_gap()

```
double  KNITRO_API KTR_get_mip_abs_gap (const KTR_context_ptr kc);
```

Return the final absolute integrality gap in the MIP solve. Returns KTR_INFBOUND if no incumbent (i.e., integer feasible) point found. Returns KTR_RC_BAD_KCPTR if there is a problem with *kc*.

KTR_get_mip_rel_gap()

```
double  KNITRO_API KTR_get_mip_rel_gap (const KTR_context_ptr kc);
```

Return the final absolute integrality gap in the MIP solve. Returns KTR_INFBOUND if no incumbent (i.e., integer feasible) point found. Returns KTR_RC_BAD_KCPTR if there is a problem with *kc*.

KTR_get_mip_incumbent_obj()

```
double  KNITRO_API KTR_get_mip_incumbent_obj (const KTR_context_ptr kc);
```

Return the objective value of the MIP incumbent solution. Returns KTR_INFBOUND if no incumbent (i.e., integer feasible) point found. Returns KTR_RC_BAD_KCPTR if there is a problem with *kc*.

KTR_get_mip_relaxation_bnd()

```
double  KNITRO_API KTR_get_mip_relaxation_bnd (const KTR_context_ptr kc);
```

Return the value of the current MIP relaxation bound. Returns KTR_RC_BAD_KCPTR if there is a problem with *kc*.

KTR_get_mip_lastnode_obj()

```
double  KNITRO_API KTR_get_mip_lastnode_obj (const KTR_context_ptr kc);
```

Return the objective value of the most recently solved MIP node subproblem. Returns KTR_RC_BAD_KCPTR if there is a problem with *kc*.

KTR_get_mip_incumbent_x()

```
int  KNITRO_API KTR_get_mip_incumbent_x (const KTR_context_ptr kc,
                                         double * const x);
```

Return the MIP incumbent solution in *x* if one exists. Returns 1 if incumbent solution exists and call is successful; 0 if no incumbent (i.e., integer feasible) exists and leaves *x* unmodified; <0 if there is an error.

Checking derivatives

KTR_check_first_ders()

```
int  KNITRO_API KTR_check_first_ders (const KTR_context_ptr  kc,
                                      double * const   x,
                                     const int               finiteDiffMethod,
                                     const double            absThreshold,
                                     const double            relThreshold,
                                     const int               evalStatus,
                                     const double            obj,
                                     const double * const    c,
```

```
                                    const double * const   objGrad,
                                    const double * const   jac,
                                          void   *         userParams);
```

Compare the application's analytic first derivatives to a finite difference approximation at *x*. The objective and all constraint functions are checked.

Returns one of the status codes "KTR_RC_*" (see *Return codes*).

Arguments:

- *x* is the input (length *n*) point at which to check derivatives
- *finiteDiffMethod* is the finite differences method to use (see KTR_PARAM_GRADOPT).
- *absThreshold* sets the absolute tolerance: print when |*estimate* - *analytic*| > *threshold*.
- *relThreshold* sets the relative tolerance: print when |*estimate* - *analytic*| > *threshold* * *scale* where *scale* = *max(1,* |*analytic*|*)*.
- *evalStatus* is the evaluation status.
- *obj* is the objective at *x*.
- *c* (length *m*) the constraints vector at *x*.
- *objGrad* (length *n*) is the analytic gradient at *x*.
- *jac* (length *nnzJ*) is the analytic constraint Jacobian at *x*.
- *userParams* is the user structure passed directly to application callback.

Note that derivatives can also be checked simply by setting the user option KTR_PARAM_DERIVCHECK (see *Derivatives*). This API function is mainly provided for backwards compatibility.

Problem definition defines

KTR_OBJGOAL

```
#define KTR_OBJGOAL_MINIMIZE    0
#define KTR_OBJGOAL_MAXIMIZE    1
```

Possible objective goals for the solver (*objGoal* in KTR_init_problem()).

KTR_OBJTYPE

```
#define KTR_OBJTYPE_GENERAL    0
#define KTR_OBJTYPE_LINEAR     1
#define KTR_OBJTYPE_QUADRATIC  2
```

Possible values for the objective type (*objType* in KTR_init_problem()).

KTR_CONTYPE

```
#define KTR_CONTYPE_GENERAL    0
#define KTR_CONTYPE_LINEAR     1
#define KTR_CONTYPE_QUADRATIC  2
```

Possible values for the constraint type (*cType* in KTR_init_problem()).

KTR_VARTYPE

```
#define KTR_VARTYPE_CONTINUOUS  0
#define KTR_VARTYPE_INTEGER     1
#define KTR_VARTYPE_BINARY      2
```

Possible values for the variable type (*xType* in `KTR_mip_init_problem()`).

KTR_FNTYPE

```
#define KTR_FNTYPE_UNCERTAIN   0
#define KTR_FNTYPE_CONVEX      1
#define KTR_FNTYPE_NONCONVEX   2
```

Possible values for the objective and constraint functions (fnType in `KTR_mip_init_problem()`).

3.3.2 Return codes

The solution status return codes are organized as follows.

- 0: the final solution satisfies the termination conditions for verifying optimality.
- -100 to -199: a feasible approximate solution was found.
- -200 to -299: KNITRO terminated at an infeasible point.
- -300: the problem was determined to be unbounded.
- -400 to -499: KNITRO terminated because it reached a pre-defined limit (-40x codes indicate that a feasible point was found before reaching the limit, while -41x codes indicate that no feasible point was found before reaching the limit).
- -500 to -599: KNITRO terminated with an input error or some non-standard error.

A more detailed description of individual return codes and their corresponding termination messages is provided below.

KTR_RC_OPTIMAL

```
#define KTR_RC_OPTIMAL                    0   /*-- OPTIMAL CODE */
```

Locally optimal solution found. KNITRO found a locally optimal point which satisfies the stopping criterion. If the problem is convex (for example, a linear program), then this point corresponds to a globally optimal solution.

KTR_RC_NEAR_OPT

```
#define KTR_RC_NEAR_OPT                -100 /*-- FEASIBLE CODES */
```

Primal feasible solution estimate cannot be improved. It appears to be optimal, but desired accuracy in dual feasibility could not be achieved. No more progress can be made, but the stopping tests are close to being satisfied (within a factor of 100) and so the current approximate solution is believed to be optimal.

KTR_RC_FEAS_XTOL

```
#define KTR_RC_FEAS_XTOL               -101
```

Primal feasible solution; the optimization terminated because the relative change in the solution estimate is less than that specified by the parameter `xtol`. To try to get more accuracy one may decrease `xtol`. If `xtol`

is very small already, it is an indication that no more significant progress can be made. It's possible the approximate feasible solution is optimal, but perhaps the stopping tests cannot be satisfied because of degeneracy, ill-conditioning or bad scaling.

KTR_RC_FEAS_NO_IMPROVE

```
#define KTR_RC_FEAS_NO_IMPROVE          -102
```

Primal feasible solution estimate cannot be improved; desired accuracy in dual feasibility could not be achieved. No further progress can be made. It's possible the approximate feasible solution is optimal, but perhaps the stopping tests cannot be satisfied because of degeneracy, ill-conditioning or bad scaling.

KTR_RC_FEAS_FTOL

```
#define KTR_RC_FEAS_FTOL                -103
```

KTR_RC_INFEASIBLE

```
#define KTR_RC_INFEASIBLE               -200 /*-- INFEASIBLE CODES */
```

Convergence to an infeasible point. Problem may be locally infeasible. If problem is believed to be feasible, try multistart to search for feasible points. The algorithm has converged to an infeasible point from which it cannot further decrease the infeasibility measure. This happens when the problem is infeasible, but may also occur on occasion for feasible problems with nonlinear constraints or badly scaled problems. It is recommended to try various initial points with the multi-start feature. If this occurs for a variety of initial points, it is likely the problem is infeasible.

KTR_RC_INFEAS_XTOL

```
#define KTR_RC_INFEAS_XTOL              -201
```

Terminate at infeasible point because the relative change in the solution estimate is less than that specified by the parameter `xtol`. To try to find a feasible point one may decrease `xtol`. If `xtol` is very small already, it is an indication that no more significant progress can be made. It is recommended to try various initial points with the multi-start feature. If this occurs for a variety of initial points, it is likely the problem is infeasible.

KTR_RC_INFEAS_NO_IMPROVE

```
#define KTR_RC_INFEAS_NO_IMPROVE        -202
```

Current infeasible solution estimate cannot be improved. Problem may be badly scaled or perhaps infeasible. If problem is believed to be feasible, try multistart to search for feasible points. If this occurs for a variety of initial points, it is likely the problem is infeasible.

KTR_RC_INFEAS_MULTISTART

```
#define KTR_RC_INFEAS_MULTISTART        -203
```

Multistart: no primal feasible point found. The multi-start feature was unable to find a feasible point. If the problem is believed to be feasible, then increase the number of initial points tried in the multi-start feature and also perhaps increase the range from which random initial points are chosen.

KTR_RC_INFEAS_CON_BOUNDS

```
#define KTR_RC_INFEAS_CON_BOUNDS        -204
```

The constraint bounds have been determined to be infeasible.

KTR_RC_INFEAS_VAR_BOUNDS

```
#define KTR_RC_INFEAS_VAR_BOUNDS        -205
```

The variable bounds have been determined to be infeasible.

KTR_RC_UNBOUNDED

```
#define KTR_RC_UNBOUNDED                -300 /*-- UNBOUNDED CODE */
```

Problem appears to be unbounded. Iterate is feasible and objective magnitude is greater than `objrange`. The objective function appears to be decreasing without bound, while satisfying the constraints. If the problem really is bounded, increase the size of the parameter `objrange` to avoid terminating with this message.

KTR_RC_ITER_LIMIT_FEAS

```
#define KTR_RC_ITER_LIMIT_FEAS          -400 /*-- LIMIT EXCEEDED CODES (FEASIBLE) */
```

The iteration limit was reached before being able to satisfy the required stopping criteria. A feasible point was found. The iteration limit can be increased through the user option `maxit`.

KTR_RC_TIME_LIMIT_FEAS

```
#define KTR_RC_TIME_LIMIT_FEAS          -401
```

The time limit was reached before being able to satisfy the required stopping criteria. A feasible point was found. The time limit can be increased through the user options `maxtime_cpu` and `maxtime_real`.

KTR_RC_MIP_EXH_FEAS

```
#define KTR_RC_MIP_EXH_FEAS             -403
```

All nodes have been explored. An integer feasible point was found. The MIP optimality gap has not been reduced below the specified threshold, but there are no more nodes to explore in the branch and bound tree. If the problem is convex, this could occur if the gap tolerance is difficult to meet because of bad scaling or roundoff errors, or there was a failure at one or more of the subproblem nodes. This might also occur if the problem is nonconvex. In this case, KNITRO terminates and returns the best integer feasible point found.

KTR_RC_MIP_TERM_FEAS

```
#define KTR_RC_MIP_TERM_FEAS            -404
```

Terminating at first integer feasible point. KNITRO has found an integer feasible point and is terminating because the user option `mip_terminate` is set to "feasible".

KTR_RC_MIP_SOLVE_LIMIT_FEAS

```
#define KTR_RC_MIP_SOLVE_LIMIT_FEAS     -405
```

Subproblem solve limit reached. An integer feasible point was found. The MIP subproblem solve limit was reached before being able to satisfy the optimality gap tolerance. The subproblem solve limit can be increased through the user option mip_maxsolves.

KTR_RC_MIP_NODE_LIMIT_FEAS

```
#define KTR_RC_MIP_NODE_LIMIT_FEAS      -406
```

Node limit reached. An integer feasible point was found. The MIP node limit was reached before being able to satisfy the optimality gap tolerance. The node limit can be increased through the user option mip_maxnodes.

KTR_RC_ITER_LIMIT_INFEAS

```
#define KTR_RC_ITER_LIMIT_INFEAS        -410 /*-- LIMIT EXCEEDED CODES (INFEASIBLE) */
```

The iteration limit was reached before being able to satisfy the required stopping criteria. No feasible point was found. The iteration limit can be increased through the user option maxit.

KTR_RC_TIME_LIMIT_INFEAS

```
#define KTR_RC_TIME_LIMIT_INFEAS        -411
```

The time limit was reached before being able to satisfy the required stopping criteria. No feasible point was found. The time limit can be increased through the user options maxtime_cpu and maxtime_real.

KTR_RC_MIP_EXH_INFEAS

```
#define KTR_RC_MIP_EXH_INFEAS           -413
```

All nodes have been explored. No integer feasible point was found. The MIP optimality gap has not been reduced below the specified threshold, but there are no more nodes to explore in the branch and bound tree. If the problem is convex, this could occur if the gap tolerance is difficult to meet because of bad scaling or roundoff errors, or there was a failure at one or more of the subproblem nodes. This might also occur if the problem is nonconvex.

KTR_RC_MIP_SOLVE_LIMIT_INFEAS

```
#define KTR_RC_MIP_SOLVE_LIMIT_INFEAS   -415
```

Subproblem solve limit reached. No integer feasible point was found. The MIP subproblem solve limit was reached before being able to satisfy the optimality gap tolerance. The subproblem solve limit can be increased through the user option mip_maxsolves.

KTR_RC_MIP_NODE_LIMIT_INFEAS

```
#define KTR_RC_MIP_NODE_LIMIT_INFEAS    -416
```

Node limit reached. No integer feasible point was found. The MIP node limit was reached before being able to satisfy the optimality gap tolerance. The node limit can be increased through the user option mip_maxnodes.

KTR_RC_CALLBACK_ERR

```
#define KTR_RC_CALLBACK_ERR             -500 /*-- OTHER FAILURES */
```

Callback function error. This termination value indicates that an error (i.e., negative return value) occurred in a user provided callback routine.

KTR_RC_LP_SOLVER_ERR

```
#define KTR_RC_LP_SOLVER_ERR            -501
```

LP solver error. This termination value indicates that an unrecoverable error occurred in the LP solver used in the active-set algorithm preventing the optimization from continuing.

KTR_RC_EVAL_ERR

```
#define KTR_RC_EVAL_ERR                 -502
```

Evaluation error. This termination value indicates that an evaluation error occurred (e.g., divide by 0, taking the square root of a negative number), preventing the optimization from continuing.

KTR_RC_OUT_OF_MEMORY

```
#define KTR_RC_OUT_OF_MEMORY            -503
```

Not enough memory available to solve problem. This termination value indicates that there was not enough memory available to solve the problem.

KTR_RC_USER_TERMINATION

```
#define KTR_RC_USER_TERMINATION         -504
```

KNITRO has been terminated by the user.

Other codes

```
#define KTR_RC_OPEN_FILE_ERR       -505
#define KTR_RC_BAD_N_OR_F          -506 /*-- PROBLEM DEFINITION ERROR */
#define KTR_RC_BAD_CONSTRAINT      -507 /*-- PROBLEM DEFINITION ERROR */
#define KTR_RC_BAD_JACOBIAN        -508 /*-- PROBLEM DEFINITION ERROR */
#define KTR_RC_BAD_HESSIAN         -509 /*-- PROBLEM DEFINITION ERROR */
#define KTR_RC_BAD_CON_INDEX       -510 /*-- PROBLEM DEFINITION ERROR */
#define KTR_RC_BAD_JAC_INDEX       -511 /*-- PROBLEM DEFINITION ERROR */
#define KTR_RC_BAD_HESS_INDEX      -512 /*-- PROBLEM DEFINITION ERROR */
#define KTR_RC_BAD_CON_BOUNDS      -513 /*-- PROBLEM DEFINITION ERROR */
#define KTR_RC_BAD_VAR_BOUNDS      -514 /*-- PROBLEM DEFINITION ERROR */
#define KTR_RC_ILLEGAL_CALL        -515 /*-- KNITRO CALL IS OUT OF SEQUENCE */
#define KTR_RC_BAD_KCPTR           -516 /*-- KNITRO PASSED A BAD KC POINTER */
#define KTR_RC_NULL_POINTER        -517 /*-- KNITRO PASSED A NULL ARGUMENT */
#define KTR_RC_BAD_INIT_VALUE      -518 /*-- APPLICATION INITIAL POINT IS BAD */
#define KTR_RC_NEWPOINT_HALT       -519 /*-- APPLICATION TOLD KNITRO TO HALT */
#define KTR_RC_BAD_LICENSE         -520 /*-- LICENSE CHECK FAILED */
#define KTR_RC_BAD_PARAMINPUT      -521 /*-- INVALID USER OPTION DETECTED */
#define KTR_RC_LINEAR_SOLVER_ERR   -522 /*-- ERROR IN LINEAR SOLVER */
#define KTR_RC_DERIV_CHECK_FAILED  -523 /*-- DERIVATIVE CHECK FAILED */
#define KTR_RC_INTERNAL_ERROR      -600 /*-- CONTACT info@ziena.com */
```

Termination values in the range -505 to -600 imply some input error or other non-standard failure. If `outlev`>0, details of this error will be printed to standard output or the file `knitro.log` depending on the value of `outmode`.

Return codes used by KNITRO for reverse communication.

```
#define KTR_RC_EVALFC          1
#define KTR_RC_EVALGA          2
#define KTR_RC_EVALH           3
#define KTR_RC_NEWPOINT        6
#define KTR_RC_EVALHV          7
#define KTR_RC_EVALH_NO_F      8
#define KTR_RC_EVALHV_NO_F     9
```

3.3.3 KNITRO user options

KNITRO has a great number and variety of user option settings and although it tries to choose the best settings by default, often significant performance improvements can be realized by choosing some non-default option settings.

Note: User parameters cannot be set after beginning the optimization process; i.e., for users of the KNITRO callable library, after making the first call to `KTR_solve()` or `KTR_mip_solve()`. In addition, the `gradopt` and `hessopt` options must be set before calling *KTR_init_problem()* or *KTR_mip_init_problem()* and remain unchanged after being set.

User options are defined in the `knitro.h`. A more detailed description of individual options and their possible values is provided below.

algorithm

KTR_PARAM_ALG

```
#define KTR_PARAM_ALGORITHM            1003
#define KTR_PARAM_ALG                  1003
#  define KTR_ALG_AUTOMATIC               0
#  define KTR_ALG_AUTO                    0
#  define KTR_ALG_BAR_DIRECT              1
#  define KTR_ALG_BAR_CG                  2
#  define KTR_ALG_ACT_CG                  3
#  define KTR_ALG_ACT_SQP                 4
#  define KTR_ALG_MULTI                   5
```

Indicates which algorithm to use to solve the problem

- 0 (auto) let KNITRO automatically choose an algorithm, based on the problem characteristics.
- 1 (direct) use the Interior/Direct algorithm.
- 2 (cg) use the Interior/CG algorithm.
- 3 (active) use the Active Set algorithm.
- 4 (sqp) use the SQP algorithm.
- 5 (multi) run all algorithms, perhaps in parallel (see *Algorithms*).

Default value: 0

bar_directinterval

KTR_PARAM_BAR_DIRECTINTERVAL

```
#define KTR_PARAM_BAR_DIRECTINTERVAL   1058
```

Controls the maximum number of consecutive conjugate gradient (CG) steps before KNITRO will try to enforce that a step is taken using direct linear algebra.

This option is only valid for the Interior/Direct algorithm and may be useful on problems where KNITRO appears to be taking lots of conjugate gradient steps. Setting `bar_directinterval` to 0 will try to enforce that only direct steps are taken which may produce better results on some problems.

Default value: 10

bar_feasible

KTR_PARAM_BAR_FEASIBLE

```
#define KTR_PARAM_BAR_FEASIBLE         1006
#  define KTR_BAR_FEASIBLE_NO            0
#  define KTR_BAR_FEASIBLE_STAY          1
#  define KTR_BAR_FEASIBLE_GET           2
#  define KTR_BAR_FEASIBLE_GET_STAY      3
```

Specifies whether special emphasis is placed on getting and staying feasible in the interior-point algorithms.

- 0 (no) No special emphasis on feasibility.
- 1 (stay) Iterates must satisfy inequality constraints once they become sufficiently feasible.
- 2 (get) Special emphasis is placed on getting feasible before trying to optimize.
- 3 (get_stay) Implement both options 1 and 2 above.

Default value: 0

Note: This option can only be used with the Interior/Direct and Interior/CG algorithms.

If `bar_feasible` = *stay* or `bar_feasible` = *get_stay*, this will activate the feasible version of KNITRO. The feasible version of KNITRO will force iterates to strictly satisfy inequalities, but does not require satisfaction of equality constraints at intermediate iterates. This option and the honorbnds option may be useful in applications where functions are undefined outside the region defined by inequalities. The initial point must satisfy inequalities to a sufficient degree; if not, KNITRO may generate infeasible iterates and does not switch to the feasible version until a sufficiently feasible point is found. Sufficient satisfaction occurs at a point x if it is true for all inequalities that

$$cl + tol \leq c(x) \leq cu - tol$$

The constant *tol* is determined by the option `bar_feasmodetol`.

If `bar_feasible` = *get* or `bar_feasible` = *get_stay*, KNITRO will place special emphasis on first trying to get feasible before trying to optimize.

bar_feasmodetol

KTR_PARAM_BAR_FEASMODETOL

```
#define KTR_PARAM_BAR_FEASMODETOL      1021
```

Specifies the tolerance in equation that determines whether KNITRO will force subsequent iterates to remain feasible.

The tolerance applies to all inequality constraints in the problem. This option only has an effect if option bar_feasible = *stay* or bar_feasible = *get_stay*.

Default value: 1.0e-4

bar_initmu

KTR_PARAM_BAR_INITMU

```
#define KTR_PARAM_BAR_INITMU          1025
```

Specifies the initial value for the barrier parameter μ used with the barrier algorithms.

This option has no effect on the Active Set algorithm.

Default value: 1.0e-1

bar_initpt

KTR_PARAM_BAR_INITPT

```
#define KTR_PARAM_BAR_INITPT          1009
#  define KTR_BAR_INITPT_AUTO            0
#  define KTR_BAR_INITPT_STRAT1          1
#  define KTR_BAR_INITPT_STRAT2          2
#  define KTR_BAR_INITPT_STRAT3          3
```

Indicates initial point strategy for x, slacks and multipliers when using a barrier algorithm. Note, this option only alters the initial x values if the user does not specify an initial x.

This option has no effect on the Active Set algorithm.

- 0 (auto) Let KNITRO automatically choose the strategy.
- 1 (strat1) Initialization strategy 1.
- 2 (strat2) Initialization strategy 2.
- 3 (strat3) Initialization strategy 3.

Default value: 0

bar_maxbacktrack

KTR_PARAM_BAR_MAXBACKTRACK

```
#define KTR_PARAM_BAR_MAXBACKTRACK    1044
```

Indicates the maximum allowable number of backtracks during the linesearch of the Interior/Direct algorithm before reverting to a CG step.

Increasing this value will make the Interior/Direct algorithm less likely to take CG steps. If the Interior/Direct algorithm is taking a large number of CG steps (as indicated by a positive value for "CGits" in the output), this may improve performance. This option has no effect on the Active Set algorithm.

Default value: 3

bar_maxcrossit

KTR_PARAM_BAR_MAXCROSSIT

```
#define KTR_PARAM_BAR_MAXCROSSIT       1039
```

Specifies the maximum number of crossover iterations before termination.

If the value is positive and the algorithm in operation is Interior/Direct or Interior/CG, then KNITRO will crossover to the Active Set algorithm near the solution. The Active Set algorithm will then perform at most bar_maxcrossit iterations to get a more exact solution. If the value is 0, no Active Set crossover occurs and the interior-point solution is the final result.

If Active Set crossover is unable to improve the approximate interior-point solution, then KNITRO will restore the interior-point solution. In some cases (especially on large-scale problems or difficult degenerate problems) the cost of the crossover procedure may be significant – for this reason, crossover is disabled by default. Enabling crossover generally provides a more accurate solution than Interior/Direct or Interior/CG.

Default value: 0

bar_maxrefactor

KTR_PARAM_BAR_MAXREFACTOR

```
#define KTR_PARAM_BAR_MAXREFACTOR      1043
```

Indicates the maximum number of refactorizations of the KKT system per iteration of the Interior/Direct algorithm before reverting to a CG step. If this value is set to -1, it will use a dynamic strategy.

These refactorizations are performed if negative curvature is detected in the model. Rather than reverting to a CG step, the Hessian matrix is modified in an attempt to make the subproblem convex and then the KKT system is refactorized. Increasing this value will make the Interior/Direct algorithm less likely to take CG steps. If the Interior/Direct algorithm is taking a large number of CG steps (as indicated by a positive value for "CGits" in the output), this may improve performance. This option has no effect on the Active Set algorithm.

Default value: -1

bar_murule

KTR_PARAM_BAR_MURULE

```
#define KTR_PARAM_BAR_MURULE           1004
#  define KTR_BAR_MURULE_AUTOMATIC        0
#  define KTR_BAR_MURULE_AUTO             0
#  define KTR_BAR_MURULE_MONOTONE         1
#  define KTR_BAR_MURULE_ADAPTIVE         2
#  define KTR_BAR_MURULE_PROBING          3
#  define KTR_BAR_MURULE_DAMPMPC          4
#  define KTR_BAR_MURULE_FULLMPC          5
#  define KTR_BAR_MURULE_QUALITY          6
```

Indicates which strategy to use for modifying the barrier parameter mu in the barrier algorithms.

Not all strategies are available for both barrier algorithms, as described below. This option has no effect on the Active Set algorithm.

- 0 (auto) Let KNITRO automatically choose the strategy.
- 1 (monotone) Monotonically decrease the barrier parameter. Available for both barrier algorithms.
- 2 (adaptive) Use an adaptive rule based on the complementarity gap to determine the value of the barrier parameter. Available for both barrier algorithms.
- 3 (probing) Use a probing (affine-scaling) step to dynamically determine the barrier parameter. Available only for the Interior/Direct algorithm.

- 4 (dampmpc) Use a Mehrotra predictor-corrector type rule to determine the barrier parameter, with safeguards on the corrector step. Available only for the Interior/Direct algorithm.
- 5 (fullmpc) Use a Mehrotra predictor-corrector type rule to determine the barrier parameter, without safeguards on the corrector step. Available only for the Interior/Direct algorithm.
- 6 (quality) Minimize a quality function at each iteration to determine the barrier parameter. Available only for the Interior/Direct algorithm.

Default value: 0

bar_penaltycons

KTR_PARAM_BAR_PENCONS

```
#define KTR_PARAM_BAR_PENCONS          1050
#  define KTR_BAR_PENCONS_AUTO            0
#  define KTR_BAR_PENCONS_NONE            1
#  define KTR_BAR_PENCONS_ALL             2
```

Indicates whether a penalty approach is applied to the constraints.

Using a penalty approach may be helpful when the problem has degenerate or difficult constraints. It may also help to more quickly identify infeasible problems, or achieve feasibility in problems with difficult constraints.

This option has no effect on the Active Set algorithm.

- 0 (auto) Let KNITRO automatically choose the strategy.
- 1 (none) No constraints are penalized.
- 2 (all) A penalty approach is applied to all general constraints.

Default value: 0

bar_penaltyrule

KTR_PARAM_BAR_PENRULE

```
#define KTR_PARAM_BAR_PENRULE          1049
#  define KTR_BAR_PENRULE_AUTO            0
#  define KTR_BAR_PENRULE_SINGLE          1
#  define KTR_BAR_PENRULE_FLEX            2
```

Indicates which penalty parameter strategy to use for determining whether or not to accept a trial iterate. This option has no effect on the Active Set algorithm.

- 0 (auto) Let KNITRO automatically choose the strategy.
- 1 (single) Use a single penalty parameter in the merit function to weight feasibility versus optimality.
- 2 (flex) Use a more tolerant and flexible step acceptance procedure based on a range of penalty parameter values.

Default value: 0

bar_refinement

KTR_PARAM_BAR_REFINEMENT

```
#define KTR_PARAM_BAR_REFINEMENT       1079
#  define KTR_BAR_REFINEMENT_NO           0
#  define KTR_BAR_REFINEMENT_YES          1
```

Specifies whether to try to refine the barrier solution for better precision. If enabled, once the optimality conditions are satisfied, KNITRO will apply an additional refinement/postsolve phase to try to obtain more precision in the barrier solution. The effect is similar to the effect of enabling bar_maxcrossit, but it is usually much more efficient since it does not involve switching to the Active Set algorithm.

Default value: 0

bar_relaxcons

KTR_PARAM_BAR_RELAXCONS

```
#define KTR_PARAM_BAR_RELAXCONS        1077
#  define KTR_BAR_RELAXCONS_NONE          0
#  define KTR_BAR_RELAXCONS_EQS           1
#  define KTR_BAR_RELAXCONS_INEQS         2
#  define KTR_BAR_RELAXCONS_ALL           3
```

Indicates whether a relaxation approach is applied to the constraints.

Using a relaxation approach may be helpful when the problem has degenerate or difficult constraints.

This option has no effect on the Active Set algorithm.

- 0 (none) No constraints are relaxed.
- 1 (eqs) A relaxation approach is applied to general equality constraints.
- 2 (ineqs) A relaxation approach is applied to general inequality constraints.
- 3 (all) A relaxation approach is applied to all general constraints.

Default value: 2

bar_switchrule

KTR_PARAM_BAR_SWITCHRULE

```
#define KTR_PARAM_BAR_SWITCHRULE       1061
#  define KTR_BAR_SWITCHRULE_AUTO         0
#  define KTR_BAR_SWITCHRULE_NEVER        1
#  define KTR_BAR_SWITCHRULE_LEVEL1       2
#  define KTR_BAR_SWITCHRULE_LEVEL2       3
```

Indicates whether or not the barrier algorithms will allow switching from an optimality phase to a pure feasibility phase. This option has no effect on the Active Set algorithm.

- 0 (auto) Let KNITRO determine the switching procedure.
- 1 (never) Never switch to feasibility phase.
- 2 (level1) Allow switches to feasibility phase.
- 3 (level2) Use a more aggressive switching rule.

Default value: 0

blasoption

KTR_PARAM_BLASOPTION

```
#define KTR_PARAM_BLASOPTION           1042
#  define KTR_BLASOPTION_KNITRO           0
#  define KTR_BLASOPTION_INTEL            1
#  define KTR_BLASOPTION_DYNAMIC          2
```

Specifies the BLAS/LAPACK function library to use for basic vector and matrix computations.

- 0 (KNITRO) Use KNITRO built-in functions.
- 1 (intel) Use Intel Math Kernel Library (MKL) functions on available platforms.
- 2 (dynamic) Use the dynamic library specified with option `blasoptionlib`.

Default value: 1

Note: BLAS and LAPACK functions from Intel Math Kernel Library (MKL) are provided with the KNITRO distribution. The MKL is available for Windows (32-bit and 64-bit), Linux (32-bit and 64-bit), and Mac OS X; it is not available for Solaris. Beginning with KNITRO 8.1, the multi-threaded version of the MKL BLAS is included with KNITRO. The number of threads to use for the MKL BLAS are specified with `par_blasnumthreads`. The MKL is not included with the free student edition of KNITRO. On platforms, where the intel MKL is not available, the KNITRO built-in functions are used by default.

BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra PACKage) functions are used throughout KNITRO for fundamental vector and matrix calculations. The CPU time spent in these operations can be measured by setting option `debug` = 1 and examining the output file `kdbg_profile*.txt`. Some optimization problems are observed to spend very little CPU time in BLAS/LAPACK operations, while others spend more than 50%. Be aware that the different function implementations can return slightly different answers due to roundoff errors in double precision arithmetic. Thus, changing the value of `blasoption` sometimes alters the iterates generated by KNITRO, or even the final solution point.

The KNITRO option uses built-in BLAS/LAPACK functions based on standard netlib routines (www.netlib.org). The intel option uses MKL functions written especially for x86 and x86_64 processor architectures. On a machine running an Intel processor (e.g., Pentium 4), testing indicates that the MKL functions can significantly reduce the CPU time in BLAS/LAPACK operations. The dynamic option allows users to load any library that implements the functions declared in the file `include/blas_lapack.h`. Specify the library name with option `blasoptionlib`.

Some Intel MKL libraries may be provided in the KNITRO lib directory and may need to be loaded at runtime by KNITRO. If so, the operating system's load path must be configured to find this directory or the MKL will fail to load.

---

blasoptionlib

KTR_PARAM_BLASOPTIONLIB

```
#define KTR_PARAM_BLASOPTIONLIB        1045
```

Specifies a dynamic library name that contains object code for BLAS/LAPACK functions.

The library must implement all the functions declared in the file `include/blas_lapack.h`. The source file `blasAcmlExample.c` in `examples/C` provides a wrapper for the AMD Core Math Library (ACML), suitable for machines with an AMD processor. Instructions are given in the file for creating a BLAS/LAPACK dynamic library from the ACML. The operating system's load path must be configured to find the dynamic library.

Note: This option has no effect unless `blasoption` = 2.

---

cplexlibname

KTR_PARAM_CPLEXLIB

```
#define KTR_PARAM_CPLEXLIB              1048
```

See option `lpsolver`.

debug

KTR_PARAM_DEBUG

```
#define KTR_PARAM_DEBUG                 1031
#  define KTR_DEBUG_NONE                  0
#  define KTR_DEBUG_PROBLEM               1
#  define KTR_DEBUG_EXECUTION             2
```

Controls the level of debugging output.

Debugging output can slow execution of KNITRO and should not be used in a production setting. All debugging output is suppressed if option `outlev` = 0.

- 0 (none) No debugging output.
- **1 (problem) Print algorithm information to kdbg*.log** output files.
- 2 (execution) Print program execution information.

Default value: 0

delta

KTR_PARAM_DELTA

```
#define KTR_PARAM_DELTA                 1020
```

Specifies the initial trust region radius scaling factor used to determine the initial trust region size.

Default value: 1.0e0

derivcheck

KTR_PARAM_DERIVCHECK

```
#define KTR_PARAM_DERIVCHECK            1080
#  define KTR_DERIVCHECK_NONE             0
#  define KTR_DERIVCHECK_FIRST            1
```

Determine whether or not to perform a derivative check on the model.

- 0 (none) Do not perform a derivative check.
- 1 (first) Check first derivatives.

Default value: 0

derivcheck_tol

KTR_PARAM_DERIVCHECK_TOL

```
#define KTR_PARAM_DERIVCHECK_TOL        1082
```

Specifies the relative tolerance used for detecting derivative errors, when the KNITRO derivative checker is enabled.

Default value: 1.0e-6

derivcheck_type

KTR_PARAM_DERIVCHECK_TYPE

```
#define KTR_PARAM_DERIVCHECK_TYPE      1081
#  define KTR_DERIVCHECK_FORWARD          1
#  define KTR_DERIVCHECK_CENTRAL          2
```

Specifies whether to use forward or central finite differencing for the derivative checker when it is enabled.

- 1 (forward) Use forward finite differencing for the derivative checker.
- 2 (central) Use central finite differencing for the derivative checker.

Default value: 1

feastol

KTR_PARAM_FEASTOL

```
#define KTR_PARAM_FEASTOL              1022
```

Specifies the final relative stopping tolerance for the feasibility error.

Smaller values of feastol result in a higher degree of accuracy in the solution with respect to feasibility.

Default value: 1.0e-6

feastol_abs

KTR_PARAM_FEASTOLABS

```
#define KTR_PARAM_FEASTOLABS           1023
```

Specifies the final absolute stopping tolerance for the feasibility error. Smaller values of `feastol_abs` result in a higher degree of accuracy in the solution with respect to feasibility.

Default value: 1.0e-3

gradopt

KTR_PARAM_GRADOPT

```
#define KTR_PARAM_GRADOPT              1007
#  define KTR_GRADOPT_EXACT               1
#  define KTR_GRADOPT_FORWARD             2
#  define KTR_GRADOPT_CENTRAL             3
```

Specifies how to compute the gradients of the objective and constraint functions.

- 1 (exact) User provides a routine for computing the exact gradients.
- 2 (forward) KNITRO computes gradients by forward finite differences.
- 3 (central) KNITRO computes gradients by central finite differences.

Default value: 1

Note: It is highly recommended to provide exact gradients if at all possible as this greatly impacts the performance of the code.

hessian_no_f

KTR_PARAM_HESSIAN_NO_F

```
#define KTR_PARAM_HESSIAN_NO_F          1062
#  define KTR_HESSIAN_NO_F_FORBID          0
#  define KTR_HESSIAN_NO_F_ALLOW           1
```

Determines whether or not to allow KNITRO to request Hessian (or Hessian-vector product) evaluations without the objective component included. If `hessian_no_f=0`, KNITRO will only ask the user for the standard Hessian and will internally approximate the Hessian without the objective component when it is needed. When `hessian_no_f=1`, KNITRO will provide a flag to the user EVALH_NO_F (or EVALHV_NO_F) when it wants an evaluation of the Hessian (or Hessian-vector product) without the objective component. Using `hessian_no_f=1` (and providing the appropriate Hessian) may improve KNITRO performance on some problems.

This option only has an effect when `hessopt=1` (i.e. user-provided exact Hessians), or `hessopt=5` (i.e. user-provided exact Hessians-vector products).

- 0 (forbid) KNITRO will not ask for Hessian evaluations without the objective component.
- 1 (allow) KNITRO may ask for Hessian evaluations without the objective component.

Default value: 0

hessopt

KTR_PARAM_HESSOPT

```
#define KTR_PARAM_HESSOPT               1008
#  define KTR_HESSOPT_EXACT                1
#  define KTR_HESSOPT_BFGS                 2
#  define KTR_HESSOPT_SR1                  3
#  define KTR_HESSOPT_FINITE_DIFF          4
#  define KTR_HESSOPT_PRODUCT              5
#  define KTR_HESSOPT_LBFGS                6
```

Specifies how to compute the (approximate) Hessian of the Lagrangian.

- 1 (exact) User provides a routine for computing the exact Hessian.
- 2 (bfgs) KNITRO computes a (dense) quasi-Newton BFGS Hessian.
- 3 (sr1) KNITRO computes a (dense) quasi-Newton SR1 Hessian.
- 4 (finite_diff) KNITRO computes Hessian-vector products using finite-differences.
- 5 (product) User provides a routine to compute the Hessian-vector products.
- 6 (lbfgs) KNITRO computes a limited-memory quasi-Newton BFGS Hessian (its size is determined by the option lmsize).

Default value: 1

Note: Options `hessopt` = 4 and `hessopt` = 5 are not available with the Interior/Direct or SQP algorithms.

KNITRO usually performs best when the user provides exact Hessians (hessopt = 1) or exact Hessian-vector products (hessopt = 5). If neither can be provided but exact gradients are available (i.e., gradopt = 1), then hessopt = 4 may be a good option. This option is comparable in terms of robustness to the exact Hessian option and typically not much slower in terms of time, provided that gradient evaluations are not a dominant cost. However, this option is only available for some algorithms. If exact gradients cannot be provided, then one of the quasi-Newton options is preferred. Options hessopt = 2 and hessopt = 3 are only recommended for small problems (say, $n < 1000$) since they require working with a dense Hessian approximation. Note that with these last two options, the Hessian pattern will be ignored since KNITRO computes a dense approximation. Option hessopt = 6 should be used for large problems.

---

honorbnds

KTR_PARAM_HONORBNDS

```
#define KTR_PARAM_HONORBNDS            1002
#  define KTR_HONORBNDS_NO               0
#  define KTR_HONORBNDS_ALWAYS           1
#  define KTR_HONORBNDS_INITPT           2
```

Indicates whether or not to enforce satisfaction of simple variable bounds throughout the optimization. This option and the bar_feasible option may be useful in applications where functions are undefined outside the region defined by inequalities.

- 0 (no) KNITRO does not require that the bounds on the variables be satisfied at intermediate iterates.
- 1 (always) KNITRO enforces that the initial point and all subsequent solution estimates satisfy the bounds on the variables.
- 2 (initpt) KNITRO enforces that the initial point satisfies the bounds on the variables.

Default value: 2

infeastol

KTR_PARAM_INFEASTOL

```
#define KTR_PARAM_INFEASTOL            1056
```

Specifies the (relative) tolerance used for declaring infeasibility of a model.

Smaller values of infeastol make it more difficult to satisfy the conditions KNITRO uses for detecting infeasible models. If you believe KNITRO incorrectly declares a model to be infeasible, then you should try a smaller value for infeastol.

Default value: 1.0e-8

linsolver

KTR_PARAM_LINSOLVER

```
#define KTR_PARAM_LINSOLVER            1057
#  define KTR_LINSOLVER_AUTO             0
#  define KTR_LINSOLVER_INTERNAL         1
#  define KTR_LINSOLVER_HYBRID           2
#  define KTR_LINSOLVER_DENSEQR          3
#  define KTR_LINSOLVER_MA27             4
#  define KTR_LINSOLVER_MA57             5
#  define KTR_LINSOLVER_MKLPARDISO       6
```

Indicates which linear solver to use to solve linear systems arising in KNITRO algorithms.

- 0 (auto) Let KNITRO automatically choose the linear solver.
- 1 (internal) Not currently used; reserved for future use. Same as auto for now.
- 2 (hybrid) Use a hybrid approach where the solver chosen depends on the particular linear system which needs to be solved.
- 3 (qr) Use a dense QR method. This approach uses LAPACK QR routines. Since it uses a dense method, it is only efficient for small problems. It may often be the most efficient method for small problems with dense Jacobians or Hessian matrices.
- 4 (ma27) Use the HSL MA27 sparse symmetric indefinite solver.
- 5 (ma57) Use the HSL MA57 sparse symmetric indefinite solver.
- 6 (mklpardiso) Use the Intel MKL PARDISO sparse symmetric indefinite solver.

Default value: 0

Note: The QR linear solver, the HSL MA57 linear solver and the Intel MKL PARDISO solver all make frequent use of Basic Linear Algebra Subroutines (BLAS) for internal linear algebra operations. If using option `linsolver` = *qr*, `linsolver` = *ma57* or `linsolver` = *mklpardiso* it is highly recommended to use optimized BLAS for your particular machine. This can result in dramatic speedup. This BLAS library is optimized for Intel processors and can be selected by setting blasoption=intel. Please read the notes under the blasoption user option in this section for more details about the BLAS options in KNITRO and how to make sure that the Intel MKL BLAS or other user-specified BLAS can be used by KNITRO.

---

linsolver_ooc

KTR_PARAM_LINSOLVER_OOC

```
#define KTR_PARAM_LINSOLVER_OOC        1076
#  define KTR_LINSOLVER_OOC_NO            0
#  define KTR_LINSOLVER_OOC_MAYBE         1
#  define KTR_LINSOLVER_OOC_YES           2
```

Indicates whether to use Intel MKL PARDISO out-of-core solve of linear systems when `linsolver` = *mklpardiso*.

This option is only active when `linsolver` = *mklpardiso*.

- 0 (no) Do not use Intel MKL PARDISO out-of-core option.
- 1 (maybe) Maybe solve out-of-core depending on how much space is needed.
- 2 (yes) Solve linear systems out-of-core when using Intel MKL PARDISO.

Default value: 0

Note: See the Intel MKL PARDISO documentation for more details on how this option works.

---

lmsize

KTR_PARAM_LMSIZE

```
#define KTR_PARAM_LMSIZE               1038
```

Specifies the number of limited memory pairs stored when approximating the Hessian using the limited-memory quasi-Newton BFGS option. The value must be between 1 and 100 and is only used with `hessopt` = 6.

Larger values may give a more accurate, but more expensive, Hessian approximation. Smaller values may give a less accurate, but faster, Hessian approximation. When using the limited memory BFGS approach it is recommended to experiment with different values of this parameter.

Default value: 10

lpsolver

KTR_PARAM_LPSOLVER

```
#define KTR_PARAM_LPSOLVER              1012
#  define KTR_LP_INTERNAL                 1
#  define KTR_LP_CPLEX                    2
#  define KTR_LP_XPRESS                   3
```

Indicates which linear programming simplex solver the KNITRO Active Set algorithm uses when solving internal LP subproblems.

This option has no effect on the Interior/Direct and Interior/CG algorithms.

- 1 (internal) KNITRO uses its default LP solver.
- 2 (cplex) KNITRO uses IBM ILOG-CPLEX(R), provided the user has a valid CPLEX license. The CPLEX library is loaded dynamically after KTR_solve() is called.
- 3 (xpress) KNITRO uses the FICO Xpress(R) solver, provided the user has a valid Xpress license. The Xpress library is loaded dynamically after KTR_solve() is called.

Default value: 1

If `lpsolver` = *cplex* then the CPLEX shared object library or DLL must reside in the operating system's load path. If this option is selected, KNITRO will automatically look for (in order): CPLEX 12.6, CPLEX 12.5, CPLEX 12.4, CPLEX 12.3, CPLEX 12.2, CPLEX 12.1, CPLEX 12.0, CPLEX 11.2, CPLEX 11.1, CPLEX 11.0, CPLEX 10.2, CPLEX 10.1, CPLEX 10.0, CPLEX 9.1, CPLEX 9.0, or CPLEX 8.0.

To override the automatic search and load a particular CPLEX library, set its name with the character type user option `cplexlibname`. Either supply the full path name in this option, or make sure the library resides in a directory that is listed in the operating system's load path. For example, to specifically load the Windows CPLEX library `cplex123.dll`, make sure the directory containing the library is part of the PATH environment variable, and call the following (also be sure to check the return status of this call):

```
KTR_set_char_param_by_name (kc, "cplexlibname", "cplex90.dll");
```

If `lpsolver` = *xpress* then the Xpress shared object library or DLL must reside in the operating system's load path. If this option is selected, KNITRO will automatically look for the standard Xpress dll/shared library name.

To override the automatic search and load a particular Xpress library, set its name with the character type user option `xpresslibname`. Either supply the full path name in this option, or make sure the library resides in a directory that is listed in the operating system's load path.

ma_maxtime_cpu

KTR_PARAM_MA_MAXTIMECPU

```
#define KTR_PARAM_MA_MAXTIMECPU         1064
```

Specifies, in seconds, the maximum allowable CPU time before termination for the multi-algorithm ("MA") procedure (`alg=5`).

Default value: 1.0e8

ma_maxtime_real

KTR_PARAM_MA_MAXTIMEREAL

```
#define KTR_PARAM_MA_MAXTIMEREAL       1065
```

Specifies, in seconds, the maximum allowable real time before termination for the multi-algorithm ("MA") procedure (`alg=5`).

Default value: 1.0e8

Note: When using the multi-algorithm procedure, the options `maxtime_cpu` and `maxtime_real` control time limits for the individual algorithms, while `ma_maxtime_cpu` and `ma_maxtime_real` impose time limits for the overall procedure.

---

ma_outsub

KTR_PARAM_MA_OUTSUB

```
#define KTR_PARAM_MA_OUTSUB            1067
#  define KTR_MA_OUTSUB_NONE             0
#  define KTR_MA_OUTSUB_YES              1
```

Enable writing algorithm output to files for the multi-algorithm (`alg=5`) procedure.

- 0 Do not write detailed algorithm output to files.
- 1 Write detailed algorithm output to files named `knitro_ma_*.log`.

Default value: 0

ma_terminate

KTR_PARAM_MA_TERMINATE

```
#define KTR_PARAM_MA_TERMINATE         1063
#  define KTR_MA_TERMINATE_ALL           0
#  define KTR_MA_TERMINATE_OPTIMAL       1
#  define KTR_MA_TERMINATE_FEASIBLE      2
#  define KTR_MA_TERMINATE_ANY           3
```

Define the termination condition for the multi-algorithm (`alg=5`) procedure.

- 0 Terminate after all algorithms have completed.
- 1 Terminate at first locally optimal solution.
- 2 Terminate at first feasible solution estimate.
- 3 Terminate at first solution estimate of any type.

Default value: 1

maxcgit

KTR_PARAM_MAXCGIT

```
#define KTR_PARAM_MAXCGIT              1013
```

Determines the maximum allowable number of inner conjugate gradient (CG) iterations per KNITRO minor iteration.

- 0 Let KNITRO automatically choose a value based on the problem size.
- *n* At most *n>0* CG iterations may be performed during one minor iteration of KNITRO.

Default value: 0

maxit

KTR_PARAM_MAXIT

```
#define KTR_PARAM_MAXIT                1014
```

Specifies the maximum number of iterations before termination.

- 0 Let KNITRO automatically choose a value based on the problem type. Currently KNITRO sets this value to 10000 for LPs/NLPs and 3000 for MIP problems.
- *n* At most *n>0* iterations may be performed before terminating.

Default value: 0

maxtime_cpu

KTR_PARAM_MAXTIMECPU

```
#define KTR_PARAM_MAXTIMECPU           1024
```

Specifies, in seconds, the maximum allowable CPU time before termination.

Default value: 1.0e8

maxtime_real

KTR_PARAM_MAXTIMEREAL

```
#define KTR_PARAM_MAXTIMEREAL          1040
```

Specifies, in seconds, the maximum allowable real time before termination.

Default value: 1.0e8

mip_branchrule

KTR_PARAM_MIP_BRANCHRULE

```
#define KTR_PARAM_MIP_BRANCHRULE       2002
#  define KTR_MIP_BRANCH_AUTO            0
#  define KTR_MIP_BRANCH_MOSTFRAC        1
#  define KTR_MIP_BRANCH_PSEUDOCOST      2
#  define KTR_MIP_BRANCH_STRONG          3
```

Specifies which branching rule to use for MIP branch and bound procedure.

- 0 (auto) Let KNITRO automatically choose the branching rule.
- 1 (most_frac) Use most fractional (most infeasible) branching.

- 2 (pseudcost) Use pseudo-cost branching.
- 3 (strong) Use strong branching (see options mip_strong_candlim, mip_strong_level and mip_strong_maxit for further control of strong branching procedure).

Default value: 0

mip_debug

KTR_PARAM_MIP_DEBUG

```
#define KTR_PARAM_MIP_DEBUG            2013
#  define KTR_MIP_DEBUG_NONE             0
#  define KTR_MIP_DEBUG_ALL              1
```

Specifies debugging level for MIP solution.

- 0 (none) No MIP debugging output created.
- 1 (all) Write MIP debugging output to the file kdbg_mip.log.

Default value: 0

mip_gub_branch

KTR_PARAM_MIP_GUB_BRANCH

```
#define KTR_PARAM_MIP_GUB_BRANCH       2015  /*-- BRANCH ON GENERALIZED BOUNDS */
#  define KTR_MIP_GUB_BRANCH_NO          0
#  define KTR_MIP_GUB_BRANCH_YES         1
```

Specifies whether or not to branch on generalized upper bounds (GUBs).

- 0 (no) Do not branch on GUBs.
- 1 (yes) Allow branching on GUBs.

Default value: 0

mip_heuristic

KTR_PARAM_MIP_HEURISTIC

```
#define KTR_PARAM_MIP_HEURISTIC        2022
#  define KTR_MIP_HEURISTIC_AUTO         0
#  define KTR_MIP_HEURISTIC_NONE         1
#  define KTR_MIP_HEURISTIC_FEASPUMP     2
#  define KTR_MIP_HEURISTIC_MPEC         3
```

Specifies which MIP heuristic search approach to apply to try to find an initial integer feasible point.

If a heuristic search procedure is enabled, it will run for at most mip_heuristic_maxit iterations, before starting the branch and bound procedure.

- 0 (auto) Let KNITRO choose the heuristic to apply (if any).
- 1 (none) No heuristic search applied.
- 2 (feaspump) Apply feasibility pump heuristic.
- 3 (mpec) Apply heuristic based on MPEC formulation.

Default value: 0

mip_heuristic_maxit

KTR_PARAM_MIP_HEURISTIC_MAXIT

```
#define KTR_PARAM_MIP_HEUR_MAXIT       2023
```

Specifies the maximum number of iterations to allow for MIP heuristic, if one is enabled.

Default value: 100

KTR_PARAM_MIP_HEUR_MAXTIMECPU

```
#define KTR_PARAM_MIP_HEUR_MAXTIMECPU  2024
```

KTR_PARAM_MIP_HEUR_MAXTIMEREAL

```
#define KTR_PARAM_MIP_HEUR_MAXTIMEREAL 2025
```

mip_implications

KTR_PARAM_MIP_IMPLICATNS

```
#define KTR_PARAM_MIP_IMPLICATNS       2014  /*-- USE LOGICAL IMPLICATIONS */
#  define KTR_MIP_IMPLICATNS_NO           0
#  define KTR_MIP_IMPLICATNS_YES          1
```

Specifies whether or not to add constraints to the MIP derived from logical implications.

- 0 (no) Do not add constraints from logical implications.
- 1 (yes) KNITRO adds constraints from logical implications.

Default value: 1

mip_integer_tol

KTR_PARAM_MIP_INTEGERTOL

```
#define KTR_PARAM_MIP_INTEGERTOL       2009
```

This value specifies the threshold for deciding whether or not a variable is determined to be an integer.

Default value: 1.0e-8

mip_integral_gap_abs

KTR_PARAM_MIP_INTGAPABS

```
#define KTR_PARAM_MIP_INTGAPABS        2004
```

The absolute integrality gap stop tolerance for MIP.

Default value: 1.0e-6

mip_integral_gap_rel

KTR_PARAM_MIP_INTGAPREL

```
#define KTR_PARAM_MIP_INTGAPREL        2005
```

The relative integrality gap stop tolerance for MIP.

Default value: 1.0e-6

mip_knapsack

KTR_PARAM_MIP_KNAPSACK

```
#define KTR_PARAM_MIP_KNAPSACK         2016  /*-- KNAPSACK CUTS */
#  define KTR_MIP_KNAPSACK_NO             0  /*--   NONE */
#  define KTR_MIP_KNAPSACK_INEQ           1  /*--   ONLY FOR INEQUALITIES */
#  define KTR_MIP_KNAPSACK_INEQ_EQ        2  /*--   FOR INEQS AND EQS */
```

Specifies rules for adding MIP knapsack cuts.

- 0 (none) Do not add knapsack cuts.
- 1 (ineqs) Add cuts derived from inequalities only.
- 2 (ineqs_eqs) Add cuts derived from both inequalities and equalities.

Default value: 1

mip_lpalg

KTR_PARAM_MIP_LPALG

```
#define KTR_PARAM_MIP_LPALG            2019
#  define KTR_MIP_LPALG_AUTO              0
#  define KTR_MIP_LPALG_BAR_DIRECT        1
#  define KTR_MIP_LPALG_BAR_CG            2
#  define KTR_MIP_LPALG_ACT_CG            3
```

Specifies which algorithm to use for any linear programming (LP) subproblem solves that may occur in the MIP branch and bound procedure.

LP subproblems may arise if the problem is a mixed integer linear program (MILP), or if using `mip_method` = *HQG*. (Nonlinear programming subproblems use the algorithm specified by the `algorithm` option.)

- 0 (auto) Let KNITRO automatically choose an algorithm, based on the problem characteristics.
- 1 (direct) Use the Interior/Direct (barrier) algorithm.
- 2 (cg) Use the Interior/CG (barrier) algorithm.
- 3 (active) Use the Active Set (simplex) algorithm.

Default value: 0

mip_maxnodes

KTR_PARAM_MIP_MAXNODES

```
#define KTR_PARAM_MIP_MAXNODES         2021
```

Specifies the maximum number of nodes explored (0 means no limit).

Default value: 100000

mip_maxsolves

KTR_PARAM_MIP_MAXSOLVES

```
#define KTR_PARAM_MIP_MAXSOLVES        2008
```

Specifies the maximum number of subproblem solves allowed (0 means no limit).

Default value: 200000

mip_maxtime_cpu

KTR_PARAM_MIP_MAXTIMECPU

```
#define KTR_PARAM_MIP_MAXTIMECPU       2006
```

Specifies the maximum allowable CPU time in seconds for the complete MIP solution.

Use `maxtime_cpu` to additionally limit time spent per subproblem solve.

Default value: 1.0e8

mip_maxtime_real

KTR_PARAM_MIP_MAXTIMEREAL

```
#define KTR_PARAM_MIP_MAXTIMEREAL      2007
```

Specifies the maximum allowable real time in seconds for the complete MIP solution.

Use `maxtime_real` to additionally limit time spent per subproblem solve.

Default value: 1.0e8

mip_method

KTR_PARAM_MIP_METHOD

```
#define KTR_PARAM_MIP_METHOD           2001
#  define KTR_MIP_METHOD_AUTO             0
#  define KTR_MIP_METHOD_BB               1
#  define KTR_MIP_METHOD_HQG              2
```

Specifies which MIP method to use.

- 0 (auto) Let KNITRO automatically choose the method.
- 1 (BB) Use the standard branch and bound method.
- 2 (HQG) Use the hybrid Quesada-Grossman method (for convex, nonlinear problems only).

Default value: 0

mip_outinterval

KTR_PARAM_MIP_OUTINTERVAL

```
#define KTR_PARAM_MIP_OUTINTERVAL      2011
```

Specifies node printing interval for `mip_outlevel` when `mip_outlevel` > 0.

- 1 Print output every node.
- 2 Print output every 2nd node.

•N Print output every Nth node.

Default value: 10

mip_outlevel

KTR_PARAM_MIP_OUTLEVEL

```
#define KTR_PARAM_MIP_OUTLEVEL          2010
#  define KTR_MIP_OUTLEVEL_NONE            0
#  define KTR_MIP_OUTLEVEL_ITERS           1
#  define KTR_MIP_OUTLEVEL_ITERSTIME       2
```

Specifies how much MIP information to print.

•0 (none) Do not print any MIP node information.

•1 (iters) Print one line of output for every node.

•2 (iterstime) Also print accumulated time for every node.

Default value: 1

mip_outsub

KTR_PARAM_MIP_OUTSUB

```
#define KTR_PARAM_MIP_OUTSUB            2012
#  define KTR_MIP_OUTSUB_NONE              0
#  define KTR_MIP_OUTSUB_YES               1
#  define KTR_MIP_OUTSUB_YESPROB           2
```

Specifies MIP subproblem solve debug output control. This output is only produced if `mip_debug` = 1 and appears in the file `kdbg_mip.log`.

•0 Do not print any debug output from subproblem solves.

•1 Subproblem debug output enabled, controlled by option outlev.

•2 Subproblem debug output enabled and print problem characteristics.

Default value: 0

mip_pseudoinit

KTR_PARAM_MIP_PSEUDOINIT

```
#define KTR_PARAM_MIP_PSEUDOINIT        2026
#  define KTR_MIP_PSEUDOINIT_AUTO          0
#  define KTR_MIP_PSEUDOINIT_AVE           1
#  define KTR_MIP_PSEUDOINIT_STRONG        2
```

Specifies the method used to initialize pseudo-costs corresponding to variables that have not yet been branched on in the MIP method.

•0 Let KNITRO automatically choose the method.

•1 Initialize using the average value of computed pseudo-costs.

•2 Initialize using strong branching.

Default value: 0

mip_rootalg

KTR_PARAM_MIP_ROOTALG

```
#define KTR_PARAM_MIP_ROOTALG          2018
#  define KTR_MIP_ROOTALG_AUTO           0
#  define KTR_MIP_ROOTALG_BAR_DIRECT     1
#  define KTR_MIP_ROOTALG_BAR_CG         2
#  define KTR_MIP_ROOTALG_ACT_CG         3
```

Specifies which algorithm to use for the root node solve in MIP (same options as `algorithm` user option).

Default value: 0

mip_rounding

KTR_PARAM_MIP_ROUNDING

```
#define KTR_PARAM_MIP_ROUNDING         2017
#  define KTR_MIP_ROUND_AUTO             0
#  define KTR_MIP_ROUND_NONE             1  /*-- DO NOT ATTEMPT ROUNDING */
#  define KTR_MIP_ROUND_HEURISTIC        2  /*-- USE FAST HEURISTIC */
#  define KTR_MIP_ROUND_NLP_SOME         3  /*-- SOLVE NLP IF LIKELY TO WORK */
#  define KTR_MIP_ROUND_NLP_ALWAYS       4  /*-- SOLVE NLP ALWAYS */
```

Specifies the MIP rounding rule to apply.

- 0 (auto) Let KNITRO choose the rounding rule.
- 1 (none) Do not round if a node is infeasible.
- 2 (heur_only) Round using a fast heuristic only.
- 3 (nlp_sometimes) Round and solve a subproblem if likely to succeed.
- 4 (nlp_always) Always round and solve a subproblem.

Default value: 0

mip_selectrule

KTR_PARAM_MIP_SELECTRULE

```
#define KTR_PARAM_MIP_SELECTRULE       2003
#  define KTR_MIP_SEL_AUTO             0
#  define KTR_MIP_SEL_DEPTHFIRST       1
#  define KTR_MIP_SEL_BESTBOUND        2
#  define KTR_MIP_SEL_COMBO_1          3
```

Specifies the MIP select rule for choosing the next node in the branch and bound tree.

- 0 (auto) Let KNITRO choose the node selection rule.
- 1 (depth_first) Search the tree using a depth first procedure.
- 2 (best_bound) Select the node with the best relaxation bound.
- 3 (combo_1) Use depth first unless pruned, then best bound.

Default value: 0

mip_strong_candlim

KTR_PARAM_MIP_STRONG_CANDLIM

```
#define KTR_PARAM_MIP_STRONG_CANDLIM   2028
```

Specifies the maximum number of candidates to explore for MIP strong branching.

Default value: 10

mip_strong_level

KTR_PARAM_MIP_STRONG_LEVEL

```
#define KTR_PARAM_MIP_STRONG_LEVEL     2029
```

Specifies the maximum number of tree levels on which to perform MIP strong branching.

Default value: 10

mip_strong_maxit

KTR_PARAM_MIP_STRONG_MAXIT

```
#define KTR_PARAM_MIP_STRONG_MAXIT     2027
```

Specifies the maximum number of iterations to allow for MIP strong branching solves.

Default value: 1000

mip_terminate

KTR_PARAM_MIP_TERMINATE

```
#define KTR_PARAM_MIP_TERMINATE        2020
#  define KTR_MIP_TERMINATE_OPTIMAL       0
#  define KTR_MIP_TERMINATE_FEASIBLE      1
```

Specifies conditions for terminating the MIP algorithm.

- 0 (optimal) Terminate at optimum.
- 1 (feasible) Terminate at first integer feasible point.

Default value: 0

ms_deterministic

KTR_PARAM_MSDETERMINISTIC

```
#define KTR_PARAM_MSDETERMINISTIC      1078
#  define KTR_MSDETERMINISTIC_NO          0
#  define KTR_MSDETERMINISTIC_YES         1
```

Indicates whether KNITRO multi-start procedure will be deterministic (when `ms_terminate` = 0).

- 0 (no) multithreaded multi-start is non-deterministic.
- 1 (yes) multithreaded multi-start is deterministic (when `ms_terminate` = 0).

Default value: 1

ms_enable

KTR_PARAM_MULTISTART

```
#define KTR_PARAM_MULTISTART           1033
#  define KTR_MULTISTART_NO               0
#  define KTR_MULTISTART_YES              1
```

Indicates whether KNITRO will solve from multiple start points to find a better local minimum.

- 0 (no) KNITRO solves from a single initial point.
- 1 (yes) KNITRO solves using multiple start points.

Default value: 0

ms_maxbndrange

KTR_PARAM_MSMAXBNDRANGE

```
#define KTR_PARAM_MSMAXBNDRANGE        1035
```

Specifies the maximum range that an unbounded variable can take when determining new start points.

If a variable is unbounded in one or both directions, then new start point values are restricted by the option. If x_i is such a variable, then all initial values satisfy

$$\max\{b_i^L, x_i^0 - \texttt{ms\_maxbndrange}/2\} \leq x_i \leq \min\{b_i^U, x_i^0 + \texttt{ms\_maxbndrange}/2\},$$

where x_i^0 is the initial value of x_i provided by the user, and b_i^L and b_i^U are the variable bounds (possibly infinite) on x_i. This option has no effect unless `ms_enable` = *yes*.

Default value: 1000.0

ms_maxsolves

KTR_PARAM_MSMAXSOLVES

```
#define KTR_PARAM_MSMAXSOLVES          1034
```

Specifies how many start points to try in multi-start. This option has no effect unless `ms_enable` = *yes*.

- 0 Let KNITRO automatically choose a value based on the problem size. The value is *min(200, 10 N)*, where *N* is the number of variables in the problem.
- *n* Try *n>0* start points.

Default value: 0

ms_maxtime_cpu

KTR_PARAM_MSMAXTIMECPU

```
#define KTR_PARAM_MSMAXTIMECPU         1036
```

Specifies, in seconds, the maximum allowable CPU time before termination.

The limit applies to the operation of KNITRO since multi-start began; in contrast, the value of `maxtime_cpu` limits how long KNITRO iterates from a single start point. Therefore, `ms_maxtime_cpu` should be greater than `maxtime_cpu`. This option has no effect unless `ms_enable` = *yes*.

Default value: 1.0e8

ms_maxtime_real

KTR_PARAM_MSMAXTIMEREAL

```
#define KTR_PARAM_MSMAXTIMEREAL        1037
```

Specifies, in seconds, the maximum allowable real time before termination.

The limit applies to the operation of KNITRO since multi-start began; in contrast, the value of maxtime_real limits how long KNITRO iterates from a single start point. Therefore, ms_maxtime_real should be greater than maxtime_real. This option has no effect unless ms_enable = *yes*.

Default value: 1.0e8

ms_num_to_save

KTR_PARAM_MSNUMTOSAVE

```
#define KTR_PARAM_MSNUMTOSAVE          1051
```

Specifies the number of distinct feasible points to save in a file named KNITRO_mspoints.log.

Each point results from a KNITRO solve from a different starting point, and must satisfy the absolute and relative feasibility tolerances. The file stores points in order from best objective to worst. Points are distinct if they differ in objective value or some component by the value of ms_savetol using a relative tolerance test. This option has no effect unless ms_enable = *yes*.

Default value: 0

ms_outsub

KTR_PARAM_MS_OUTSUB

```
#define KTR_PARAM_MS_OUTSUB            1068
#  define KTR_MS_OUTSUB_NONE             0
#  define KTR_MS_OUTSUB_YES              1
```

Enable writing algorithm output to files for the parallel multistart procedure.

- 0 Do not write detailed algorithm output to files.
- 1 Write detailed algorithm output to files named knitro_ms_*.log.

Default value: 0

ms_savetol

KTR_PARAM_MSSAVETOL

```
#define KTR_PARAM_MSSAVETOL            1052
```

Specifies the tolerance for deciding if two feasible points are distinct.

Points are distinct if they differ in objective value or some component by the value of ms_savetol using a relative tolerance test. A large value can cause the saved feasible points in the file KNITRO_mspoints.log to cluster around more widely separated points. This option has no effect unless ms_enable = *yes*. and ms_num_to_save is positive.

Default value: 1.0e-6

ms_seed

KTR_PARAM_MSSEED

```
#define KTR_PARAM_MSSEED              1066
```

Seed value used to generate random initial points in multi-start; should be a non-negative integer.

Default value: 0

ms_startptrange

KTR_PARAM_MSSTARTPTRANGE

```
#define KTR_PARAM_MSSTARTPTRANGE      1055
```

Specifies the maximum range that each variable can take when determining new start points.

If a variable has upper and lower bounds and the difference between them is less than ms_startptrange, then new start point values for the variable can be any number between its upper and lower bounds.

If the variable is unbounded in one or both directions, or the difference between bounds is greater than the minimum of ms_startptrange and ms_maxbndrange, then new start point values are restricted by the option. If x_i is such a variable, then all initial values satisfy

$$\max\{b_i^L, x_i^0 - \tau\} \leq x_i \leq \min\{b_i^U, x_i^0 + \tau\},$$
$$\tau = \min\{\texttt{ms\_startptrange}/2, \texttt{ms\_maxbndrange}/2\}$$

where x_i^0 is the initial value of x_i provided by the user, and b_i^L and b_i^U are the variable bounds (possibly infinite) on x_i. This option has no effect unless *ms_enable=yes*.

Default value: 1.0e20

ms_terminate

KTR_PARAM_MSTERMINATE

```
#define KTR_PARAM_MSTERMINATE         1054
#  define KTR_MSTERMINATE_MAXSOLVES      0
#  define KTR_MSTERMINATE_OPTIMAL        1
#  define KTR_MSTERMINATE_FEASIBLE       2
#  define KTR_MSTERMINATE_ANY            3
```

Specifies the condition for terminating multi-start.

This option has no effect unless ms_enable = *yes*.

- 0 Terminate after ms_maxsolves.
- 1 Terminate after the first local optimal solution is found or ms_maxsolves, whichever comes first.
- 2 Terminate after the first feasible solution estimate is found or ms_maxsolves, whichever comes first.
- 3 Terminate after the first solution estimate of any type is found or ms_maxsolves, whichever comes first.

Default value: 0

newpoint

KTR_PARAM_NEWPOINT

```
#define KTR_PARAM_NEWPOINT              1001
#  define KTR_NEWPOINT_NONE               0
#  define KTR_NEWPOINT_SAVEONE            1
#  define KTR_NEWPOINT_SAVEALL            2
#  define KTR_NEWPOINT_USER               3
```

Specifies additional action to take after every iteration in a solve of a continuous problem.

An iteration of KNITRO results in a new point that is closer to a solution. The new point includes values of x and Lagrange multipliers *lambda*. The "newpoint" feature in KNITRO is currently only available for continuous problems (solved via `KTR_solve()`).

- 0 (none) KNITRO takes no additional action.
- 1 (saveone) KNITRO writes x and *lambda* to the file `KNITRO_newpoint.log`. Previous contents of the file are overwritten.
- 2 (saveall) KNITRO appends x and *lambda* to the file `KNITRO_newpoint.log`. Warning: this option can generate a very large file. All iterates, including the start point, crossover points, and the final solution are saved. Each iterate also prints the objective value at the new point, except the initial start point.
- 3 (user) If using callback mode and a user callback function is defined with `KTR_set_newpt_callback()`, then KNITRO will invoke the callback function after every iteration. If using reverse communications mode, then KNITRO will return to the driver level after every iteration with `KTR_solve()` returning the integer value defined by `KTR_RC_NEWPOINT` (6).

Default value: 0

objrange

KTR_PARAM_OBJRANGE

```
#define KTR_PARAM_OBJRANGE              1026
```

Specifies the extreme limits of the objective function for purposes of determining unboundedness.

If the magnitude of the objective function becomes greater than `objrange` for a feasible iterate, then the problem is determined to be unbounded and KNITRO proceeds no further.

Default value: 1.0e20

opttol

KTR_PARAM_OPTTOL

```
#define KTR_PARAM_OPTTOL                1027
```

Specifies the final relative stopping tolerance for the KKT (optimality) error.

Smaller values of `opttol` result in a higher degree of accuracy in the solution with respect to optimality.

Default value: 1.0e-6

opttol_abs

KTR_PARAM_OPTTOLABS

```
#define KTR_PARAM_OPTTOLABS             1028
```

Specifies the final absolute stopping tolerance for the KKT (optimality) error.

Smaller values of opttol_abs result in a higher degree of accuracy in the solution with respect to optimality.

Default value: 1.0e-3

outappend

KTR_PARAM_OUTAPPEND

```
#define KTR_PARAM_OUTAPPEND             1046
#  define KTR_OUTAPPEND_NO                0
#  define KTR_OUTAPPEND_YES               1
```

Specifies whether output should be started in a new file, or appended to existing files.

The option affects KNITRO.log and files produced when debug = 1. It does not affect KNITRO_newpoint.log, which is controlled by option newpoint.

- 0 (no) Erase any existing files when opening for output.
- 1 (yes) Append output to any existing files.

Default value: 0

Note: The option should not be changed after calling KTR_init_problem().

---

outdir

KTR_PARAM_OUTDIR

```
#define KTR_PARAM_OUTDIR                1047
```

Specifies a single directory as the location to write all output files.

The option should be a full pathname to the directory, and the directory must already exist.

Note: The option should not be changed after calling KTR_init_problem() or KTR_mip_init_problem().

---

outlev

KTR_PARAM_OUTLEV

```
#define KTR_PARAM_OUTLEV                1015
#  define KTR_OUTLEV_NONE                 0
#  define KTR_OUTLEV_SUMMARY              1
#  define KTR_OUTLEV_ITER_10              2
#  define KTR_OUTLEV_ITER                 3
#  define KTR_OUTLEV_ITER_VERBOSE         4
#  define KTR_OUTLEV_ITER_X               5
#  define KTR_OUTLEV_ALL                  6
```

Controls the level of output produced by KNITRO.

- 0 (none) Printing of all output is suppressed.
- 1 (summary) Print only summary information.
- 2 (iter_10) Print basic information every 10 iterations.
- 3 (iter) Print basic information at each iteration.

- 4 (iter_verbose) Print basic information and the function count at each iteration.
- 5 (iter_x) Print all the above, and the values of the solution vector *x*.
- 6 (all) Print all the above, and the values of the constraints *c* at *x* and the Lagrange multipliers *lambda*.

Default value: 2

outmode

KTR_PARAM_OUTMODE

```
#define KTR_PARAM_OUTMODE              1016
#  define KTR_OUTMODE_SCREEN             0
#  define KTR_OUTMODE_FILE               1
#  define KTR_OUTMODE_BOTH               2
```

Specifies where to direct the output from KNITRO.

- 0 (screen) Output is directed to standard out (e.g., screen).
- 1 (file) Output is sent to a file named `knitro.log`.
- 2 (both) Output is directed to both the screen and file `knitro.log`.

Default value: 0

par_blasnumthreads

KTR_PARAM_PAR_BLASNUMTHREADS

```
#define KTR_PARAM_PAR_BLASNUMTHREADS   3003
```

Specify the number of threads to use for BLAS operations when `blasoption` = 1 (see *Parallelism*).

Default value: 1

par_concurrent_evals

KTR_PARAM_PAR_CONCURRENT_EVALS

```
#define KTR_PARAM_PAR_CONCURRENT_EVALS 3002
#  define KTR_PAR_CONCURRENT_EVALS_NO     0
#  define KTR_PAR_CONCURRENT_EVALS_YES    1
```

Determines whether or not the user provided callback functions used for function and derivative evaluations can take place concurrently in parallel (for possibly different values of "*x*"). If it is not safe to have concurrent evaluations, then setting `par_concurrent_evals=0`, will put these evaluations in a critical region so that only one evaluation can take place at a time. If `par_concurrent_evals=1` then concurrent evaluations are allowed when KNITRO is run in parallel, and it is the responsibility of the user to ensure that these evaluations are stable. See *Parallelism*.

- 0 (no) Do not allow concurrent callback evaluations.
- 1 (yes) Allow concurrent callback evaluations.

Default value: 1

par_lsnumthreads

KTR_PARAM_PAR_LSNUMTHREADS

```
#define KTR_PARAM_PAR_LSNUMTHREADS      3004
```

Specify the number of threads to use for linear system solve operations when `linsolver` = 6 (see *Parallelism*).

Default value: 1

par_numthreads

KTR_PARAM_PAR_NUMTHREADS

```
#define KTR_PARAM_PAR_NUMTHREADS        3001
```

Specify the number of threads to use for parallel (excluding BLAS) computing features (see *Parallelism*).

Default value: 1

pivot

KTR_PARAM_PIVOT

```
#define KTR_PARAM_PIVOT                 1029
```

Specifies the initial pivot threshold used in factorization routines.

The value should be in the range [0, ..., 0.5] with higher values resulting in more pivoting (more stable factorizations). Values less than 0 will be set to 0 and values larger than 0.5 will be set to 0.5. If `pivot` is non-positive, initially no pivoting will be performed. Smaller values may improve the speed of the code but higher values are recommended for more stability (for example, if the problem appears to be very ill-conditioned).

Default value: 1.0e-8

presolve

KTR_PARAM_PRESOLVE

```
#define KTR_PARAM_PRESOLVE              1059
#  define KTR_PRESOLVE_NONE               0
#  define KTR_PRESOLVE_BASIC              1
```

Determine whether or not to use the KNITRO presolver to try to simplify the model by removing variables or constraints.

- 0 (none) Do not use KNITRO presolver.
- 1 (basic) Use the KNITRO basic presolver.

Default value: 1

presolve_tol

KTR_PARAM_PRESOLVE_TOL

```
#define KTR_PARAM_PRESOLVE_TOL          1060
```

Determines the tolerance used by the KNITRO presolver to remove variables and constraints from the model. If you believe the KNITRO presolver is incorrectly modifying the model, use a smaller value for this tolerance (or turn the presolver off).

Default value: 1.0e-6

scale

KTR_PARAM_SCALE

```
#define KTR_PARAM_SCALE                 1017
#  define KTR_SCALE_NEVER                 0
#  define KTR_SCALE_ALLOW                 1
```

Performs a scaling of the objective and constraint functions based on their values at the initial point.

If scaling is performed, all internal computations, including the stopping tests, are based on the scaled values.

- 0 (no) No scaling is performed.
- 1 (yes) KNITRO is allowed to scale the objective function and constraints.

Default value: 1

soc

KTR_PARAM_SOC

```
#define KTR_PARAM_SOC                   1019
#  define KTR_SOC_NO                      0
#  define KTR_SOC_MAYBE                   1
#  define KTR_SOC_YES                     2
```

Specifies whether or not to try second order corrections (SOC).

A second order correction may be beneficial for problems with highly nonlinear constraints.

- 0 (no) No second order correction steps are attempted.
- 1 (maybe) Second order correction steps may be attempted on some iterations.
- 2 (yes) Second order correction steps are always attempted if the original step is rejected and there are nonlinear constraints.

Default value: 1

tuner

KTR_PARAM_TUNER

```
#define KTR_PARAM_TUNER                 1070
    #  define KTR_TUNER_OFF                    0
    #  define KTR_TUNER_ON                     1
```

Indicates whether to invoke the KNITRO-Tuner (see *The KNITRO-Tuner*).

- 0 (off) Do not invoke the KNITRO-Tuner.
- 1 (on) Invoke the KNITRO-Tuner.

Default value: 0

tuner_optionsfile

KTR_PARAM_TUNER_OPTIONSFILE

```
#define KTR_PARAM_TUNER_OPTIONSFILE    1071
```

Can be used to specify the location of a Tuner options file (see *The KNITRO-Tuner*).

Default value: NULL

tuner_maxtime_cpu

KTR_PARAM_TUNER_MAXTIMECPU

```
#define KTR_PARAM_TUNER_MAXTIMECPU     1072
```

Specifies, in seconds, the maximum allowable CPU time before terminating the KNITRO-Tuner.

The limit applies to the operation of KNITRO since the KNITRO-Tuner began. In contrast, the value of `maxtime_cpu` places a time limit on each individual KNITRO-Tuner solve for a particular option setting. Therefore, `tuner_maxtime_cpu` should be greater than `maxtime_cpu`. This option has no effect unless `tuner` = *on*.

Default value: 1.0e8

tuner_maxtime_real

KTR_PARAM_TUNER_MAXTIMEREAL

```
#define KTR_PARAM_TUNER_MAXTIMEREAL    1073
```

Specifies, in seconds, the maximum allowable real time before terminating the KNITRO-Tuner.

The limit applies to the operation of KNITRO since the KNITRO-Tuner began. In contrast, the value of `maxtime_real` places a time limit on each individual KNITRO-Tuner solve for a particular option setting. Therefore, `tuner_maxtime_real` should be greater than `maxtime_real`. This option has no effect unless `tuner` = *on*.

Default value: 1.0e8

tuner_outsub

KTR_PARAM_TUNER_OUTSUB

```
#define KTR_PARAM_TUNER_OUTSUB         1074
#  define KTR_TUNER_OUTSUB_NONE           0
#  define KTR_TUNER_OUTSUB_SUMMARY        1
#  define KTR_TUNER_OUTSUB_ALL            2
```

Enable writing additional Tuner subproblem solve output to files for the KNITRO-Tuner procedure (`tuner=1`).

- 0 Do not write detailed solve output to files.
- 1 Write summary solve output to a file named `knitro_tuner_summary.log`.
- 2 Write detailed individual solve output to files named `knitro_tuner_*.log`.

Default value: 0

tuner_terminate

KTR_PARAM_TUNER_TERMINATE

```
#define KTR_PARAM_TUNER_TERMINATE      1075
#  define KTR_TUNER_TERMINATE_ALL         0
#  define KTR_TUNER_TERMINATE_OPTIMAL     1
#  define KTR_TUNER_TERMINATE_FEASIBLE    2
#  define KTR_TUNER_TERMINATE_ANY         3
```

Define the termination condition for the KNITRO-Tuner procedure (`tuner=1`).

- 0 Terminate after all solves have completed.
- 1 Terminate at first locally optimal solution.
- 2 Terminate at first feasible solution estimate.
- 3 Terminate at first solution estimate of any type.

Default value: 0

xpresslibname

KTR_PARAM_XPRESSLIB

```
#define KTR_PARAM_XPRESSLIB            1069
```

See option lpsolver.

xtol

KTR_PARAM_XTOL

```
#define KTR_PARAM_XTOL                 1030
```

The optimization process will terminate if the relative change in all components of the solution point estimate is less than xtol. If using the Interior/Direct or Interior/CG algorithm and the barrier parameter is still large, KNITRO will first try decreasing the barrier parameter before terminating.

Default value: 1.0e-15

3.4 List of output files

- knitro.log:

  This is the standard output from KNITRO. The file is created if outmode = *file* or outmode = *both*.

- knitro_mspoints.log:

  This file contains a set of feasible points found by multi-start, each distinct, in order of best to worst. The file is created if ms_enable = *yes* and ms_num_to_save is greater than zero.

- knitro_newpoint.log:

  This file contains a set of iterates generated by KNITRO. It is created if newpoint equals *saveone* or *saveall*.

- kdbg_barrierIP.log ; kdbg_directIP.log ; kdbg_normalIP.log ; kdbg_profileIP.log ; kdbg_stepIP.log ; kdbg_summIP.log ; kdbg_tangIP.log:

  These files contain detailed debug information. The files are created if debug = *problem* and either barrier method (Interior/Direct or Interior/CG) executes. The kdbg_directIP.log file is created only for the Interior/Direct method.

- kdbg_actsetAS.log ; kdbg_eqpAS.log ; kdbg_lpAS.log ; kdbg_profileAS.log ; kdbg_stepAS.log ; kdbg_summAS.log:

  These files contain detailed debug information. The files are created if debug = *problem* and the Active Set method executes.

- kdbg_mip.log:

  This file contains detailed debug information. The file is created if mip_debug = *all* and one of the MIP methods executes.

- `knitro_ma_*.log`:

  This file contains detailed algorithm output for each algorithm run in the multi-algorithm procedure (`alg=5`) when `ma_outsub=1`. The "*" in the filename represents the algorithm number.

- `knitro_ms_*.log`:

  This file contains detailed algorithm output for each subproblem solve in the parallel multi-start procedure when `ms_outsub=1`. The "*" in the filename represents the multi-start subproblem solve number.

- `knitro_tuner_summary.log`, `knitro_tuner_*.log`:

  These files contain detailed algorithm output for each subproblem solve in the KNITRO-Tuner procedure when `tuner_outsub=2`. The "*" in the filename represents the Tuner subproblem solve number. If `tuner_outsub=1` then only the summary file is generated.