
Artelys Knitro Documentation

Release 11.0.0

Artelys

Apr 26, 2018

CONTENTS

1	Introduction	3
2	User guide	15
3	Reference manual	121

This documentation is divided into three parts. The *Introduction* provides an overview of the Artelys 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.

INTRODUCTION

This chapter contains an overview of what Artelys 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

Artelys 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

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

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;
- quadratically constrained quadratic programs (QCQPs);
- second order cone programs (SOCPs)
- 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;
- derivative free (DFO) or black-box optimization.

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++, C#, Fortran, Java, Python, R;
- modeling language interfaces: [AMPL](#) ©, [AIMMS](#) ©, [GAMS](#) ©, [MATLAB](#) ©, [MPL](#) ©, [Microsoft Excel Premium Solver](#) ©;
- thread-safe libraries for easy embedding into application software;
- a specialized API for bound-constrained nonlinear least-squares problems.

1.2 Getting Knitro

Knitro is developed, marketed and supported by Artelys. We have offices in Chicago, London, Los Angeles, Montréal and Paris. Support is provided in English or French.

Free, time-limited trial versions of Knitro can be downloaded from:

<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 11.0 is supported on the platforms described in the table below.

PLAT-FORM	OPERATING SYSTEM	PROCESSOR
Windows 32-bit	Windows Server 2008, Vista, Windows 7, Windows 8, Windows 8.1, Windows 10	AMD Duron/Intel Pentium3 or later x86 CPU
Windows 64-bit	Windows Server 2008, Vista, Windows 7, Windows 8, Windows 8.1, Windows 10	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 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 Artelys to compile Knitro in a different environment. From the 11.0 release, Knitro is compiled with the Intel compilers on all platforms. Contact us if your application requires special compilation of Knitro.

```
> Windows (32-bit x86)
> > C/C++      > Intel C/C++ compiler 17.0.4
> > Java:      > 1.6
> > R:         > R 3.0 (R interface)
> Windows (64-bit x86_64)
> > C/C++:     > Intel C/C++ compiler 17.0.4
> > Java:      > 1.6
> > R:         > R 3.0 (R interface)
> Linux (64-bit x86_64)
> > C/C++:     > Intel C/C++ compiler 17.0.4
> > Java:      > 1.6
> > R:         > R 3.0 (R interface)
> Mac OS X (64-bit x86_64)
> > C/C++:     > Intel C/C++ compiler 17.0.4
> > Java:      > 1.6
> > R:         > R 3.0 (R interface)
```

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\Artelys
```

Unpacking will create a folder named `knitro-11.0.0-z`. 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.
- `Knitro_11_0_ReleaseNotes`: a file containing release notes.
- `get_machine_ID`: an executable that identifies the machine ID, required for obtaining a Artelys 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++, 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 Artelys license file. If you purchased a floating network license, then refer to the Artelys 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 Artelys 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\Artelys\
```

(create the folder above if it does not exist). The license file name may be changed, but must begin with the characters `artelys_lic`. If this does not work, try creating a new environment variable called `ARTELYS_LICENSE` and set it to the folder holding your license file(s). See information on setting environment variables below and refer to the Artelys 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 and later

- 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 `ARTELYS_LICENSE` environment variable to activate your license, then follow the instructions above, but in the last step create a new environment variable called `ARTELYS_LICENSE` and give it the value of the folder containing your Artelys 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\Artelys\Knitro |release|\knitromatlab');
> savepath();
```

1.3.2 Unix (Linux, Mac OS X)

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-|release|-platformname.tar.gz
> tar -xvf knitro-|release|-platformname.tar
```

Unpacking will create a directory named `knitro-11.0.0-z`. 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.
- `Knitro_11_0_ReleaseNotes`: A file containing release notes.
- `get_machine_ID`: An executable that identifies the machine ID, required for obtaining a Artelys 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 Artelys license file. If you purchased a floating network license, then refer to the Artelys 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 Artelys 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 `artelys_lic` (use lower-case letters). If this does not work, try creating a new environment variable called `ARTELYS_LICENSE` and set it to the folder holding your license file(s). See information on setting environment variables below and refer to the Artelys 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 `ARTELYS_LICENSE` environment variable to activate your license, then follow the instructions above to create a new environment variable called `ARTELYS_LICENSE` and give it the value of the directory containing your Artelys license file (specify the whole path to this directory). For more installation options and general troubleshooting, read the Artelys 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-|release|/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 `ARTELYS_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 Artelys License Manager User's Manual for more details on how to set the `ARTELYS_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 Artelys 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.

- Make sure the Knitro/MATLAB interface files `knitromatlab_mex.mex*`, `knitromatlab.m`, `knitromatlab_mip.p`, 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.
- On Mac OS X, if Knitro/MATLAB is not finding the license file (or a library), try starting MATLAB from the Terminal by typing "matlab" from a Terminal window. Sometimes environment variables are not inherited properly by a MATLAB session on Mac OS X when the session is started by double-clicking on the MATLAB icon.

- 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.
- If you encounter the error message *cannot load any more object with static TLS* this is a MATLAB bug (bug# 961964) on Linux. You may try one of the following workaround, if the issue remains, you may contact Mathworks directly.
 1. Insert `ones (10) *ones (10) ;` in the file `startup.m`.
 2. Preload any library before starting MATLAB using the environment variable `LD_PRELOAD` (ex: `export LD_PRELOAD=/usr/lib64/libgomp.so.1`).
 3. Run MATLAB without the GUI by running the following command from a terminal `matlab -nojvm`.

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 11.0.0 is installed at its default location):

```
set PATH=%PATH%;C:\Program Files\Artelys\knitro-|release|-z\lib
```

- On Mac OS X, type (assuming Knitro 11.0.0 is installed at /tmp):

```
export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:/tmp/knitro-|release|-z/lib
```

- If you run a Unix bash shell, then type (assuming Knitro 11.0.0 is installed at /tmp):

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/tmp/knitro-|release|-z/lib
```

- If you run a Unix csh or tcsh shell, then type (assuming Knitro 11.0.0 is installed at /tmp):

```
setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/tmp/knitro-|release|-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>::deallocate(void*, unsigned int)'  
../lib/libknitro.a(.text+0x28837): In function `ktr_xeb4':  
: undefined reference to `std::__default_alloc_template<true, 0>::deallocate(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_xl150':  
: 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_xl150':  
: undefined reference to `std::__default_alloc_template<true, 0>::deallocate(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 Artelys 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 11.0 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 `knitro1100.lib`) is recommended and should be compatible across multiple versions of the Microsoft Visual C++ (MSVC) compiler.

1.5 Release notes

Note: Knitro 11.0 will be the last major Knitro release to provide a version for the Windows 32-bit platform. Impacted users are advised to migrate to 64-bit Windows. Not all interfaces and features are available for the 32-bit Windows version of Knitro.

What's new in release 11.0 ?

- Knitro 11.0 introduces a completely new C callable library API. The new API allows the user to build up the model in pieces and to provide Knitro with a lot of structural information about the model (e.g. linear, structure, quadratic structure, etc). The API also supports multiple callback objects for nonlinear evaluations. This allows, for instance, the ability to provide some first derivatives using callbacks while having Knitro approximate other first derivatives with finite-differencing. The old API will continue to be supported in the near term, but may be deprecated in a future release. Please see [Callable library API reference](#) for an overview of the new API.
- Knitro 11.0 introduces a new solver for models with conic constraints. This includes Second Order Cone Programs (SOCPs) as well as any more general, nonlinear optimization models with second order cone constraints. To enable the conic solver you must choose the Interior/Direct algorithm and set the new user option `bar_conic_enable=1`. Conic constraints can be specified in the new API by specifying L2 norm structures along with linear structures to create constraints of the form $\|Ax+b\| \leq c'x+d$. In addition, Knitro may automatically detect some quadratic constraints to be conic constraints.
- Knitro 11.0 offers improved performance on quadratically constrained quadratic programs (QCQPs) and more general convex models. If a model is known to be convex, it is recommended that a user set the new user option `convex=1` (this option is disabled by default for nonlinear models). Setting this option will cause Knitro to apply some specializations and tunings that often work better on convex models compared to the default user option settings.
- Knitro 11.0 offers two new parallel linear solvers:
 - MA97, a deterministic parallel linear solver (`linsolver=7`)
 - MA86, a non-deterministic parallel linear solver (`linsolver=8`)

For very large models where the linear solves are the dominant cost, we recommend trying these new parallel linear solvers with multiple threads. MA86 will often be faster than MA97, but at the cost of generating non-deterministic behavior. These new linear solvers are currently available on Linux only.

- Knitro 11.0 extends the existing incomplete Cholesky preconditioner option `cg_precond` introduced in Knitro 10.3 to handle models with equality constraints. The original version only worked on models without equality constraints. Use of the preconditioner may offer large speedups on ill-conditioned models that take a lot of CG iterations, particularly when using the Interior/CG algorithm.
- Knitro 11.0 offers several new user options for tuning the LP subproblems that arise in the active-set algorithm (`algorithm=3`), and occasionally also in the SQP algorithm (`algorithm=4`). Tuning these options can sometimes lead to significant improvements.
 - `act_parametric`: specifies whether to solve the LP subproblems using a parametric approach
 - `act_lpalg`: algorithm to use in LP subproblems
 - `act_lppresolve`: controls the presolve in LP subproblems
 - `act_lppenalty`: controls constraint penalization in LP subproblems

See [Knitro user options](#) for more details on these options.

- Knitro 11.0 includes modifications for the C++ API. These modifications improve efficiency and numerical accuracy. Upgrading existing codes in order to use the updated API is recommended and requires some slight modifications, mostly in order to use arrays instead of vectors.
- Knitro 11.0 offers the following new user options, in addition to those already mentioned above:

- *bndrange*: any variable or constraint bound larger in magnitude than *bndrange* will be treated as an infinite bound (new API only)
- *eval_fcga*: set this option to tell Knitro you are providing one callback routine to evaluate functions and gradients together (new API only)
- *mip_selectdir*: specifies the MIP node selection direction rule
- *out_hints*: new output option to enable or disable hints about option settings printed at the end of the optimization
- *outname*: used to specify the name of the Knitro output file (default name `knitro.log`) when output is printed to a file (*outmode*=1 or 2).

See *Knitro user options* for more details on these options.

- Knitro 11.0 offers several enhancements to the KnitroR “R” interface:
 - It is now open-source under LGPL license and will be available on CRAN.
 - Knitro’s multi-threading options can now be used via KnitroR.
 - The *newpoint* option is now fully functional for users to print information after each Knitro iteration.
- The high-volume license feature in Knitro, which allows checking out a license once and re-using it for many solves, now works with floating licenses. Please see the Artelys License Manager User’s Manual for more information.
- Knitro 11.0 offers improvements to the derivative checker (*derivcheck*), in order to better detect errors in user-provided sparsity structure for the Jacobian or Hessian matrices.

Bug Fixes in release 11.0

- Fixed bug that could sometimes lead to non-deterministic behavior when running parallel multistart, multi-alg or tuner.
- Fixed issues using MKLPARDISO linear solver with multi-threading (i.e. *par_lsnumthreads*>1).
- Fixed bug that could cause false claim of optimality when using finite differences to compute gradients, and an evaluation error is encountered during the finite-difference gradient computation.
- Fixed issue with *KTR_get_int_param_by_name()* returning the wrong value for some user options.

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, a specific interface for Microsoft Excel is available):

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

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, R, and C++, the latter with both the callable library and with the object-oriented interface.

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 `KN_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.

A detailed list of Knitro options and settings available through AMPL is provided in [Knitro / AMPL reference](#).

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 demonstrate 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.

```

1 alg=2
2 bar_maxcrossit=2
3 outlev=1
4
5 =====
6           Commercial License
7           Artelys Knitro 11.0.0
8 =====
9
10 Knitro presolve eliminated 0 variables and 0 constraints.
11
12 algorithm:                2
13 bar_maxcrossit:           2
14 datacheck:                 0
15 hessian_no_f:             1
16 outlev:                    1
17 par_concurrent_evals:     0
18 The problem is identified as a QCQP.
19 Knitro changing bar_initpt from AUTO to 3.
20 Knitro changing bar_murule from AUTO to 1.
21 Knitro changing bar_penaltycons from AUTO to 1.
22 Knitro changing bar_penaltyrule from AUTO to 2.
23 Knitro changing bar_switchrule from AUTO to 2.
24 Knitro changing linesearch from AUTO to 1.
25 Knitro changing linsolver from AUTO to 4.
26
27 Problem Characteristics ( Presolved)
28 -----
29 Objective goal: Minimize
30 Objective type: quadratic
31 Number of variables:      3 (          3)
32   bounded below only:    3 (          3)
33   bounded above only:    0 (          0)
34   bounded below and above: 0 (          0)
35   fixed:                  0 (          0)
36   free:                   0 (          0)
37 Number of constraints:    2 (          2)
38   linear equalities:      1 (          1)
39   quadratic equalities:   0 (          0)
40   gen. nonlinear equalities: 0 (          0)
41   linear one-sided inequalities: 0 (          0)
42   quadratic one-sided inequalities: 1 (          1)
43   gen. nonlinear one-sided inequalities: 0 (          0)
44   linear two-sided inequalities: 0 (          0)
45   quadratic two-sided inequalities: 0 (          0)
46   gen. nonlinear two-sided inequalities: 0 (          0)
47 Number of nonzeros in Jacobian: 6 (          6)
48 Number of nonzeros in Hessian: 5 (          5)
49
50 EXIT: Locally optimal solution found.
51
52 Final Statistics

```

```

53 -----
54 Final objective value           = 9.360000000000000e+02
55 Final feasibility error (abs / rel) = 0.00e+00 / 0.00e+00
56 Final optimality error (abs / rel) = 0.00e+00 / 0.00e+00
57 # of iterations                 = 7
58 # of CG iterations              = 8
59 # of function evaluations       = 0
60 # of gradient evaluations       = 0
61 # of Hessian evaluations        = 0
62 Total program time (secs)       = 0.00115 ( 0.001 CPU time)
63 Time spent in evaluations (secs) = 0.00000
64
65 =====
66
67 Knitro 11.0.0: Locally optimal or satisfactory solution.
68 objective 936; feasibility error 0
69 7 iterations; 0 function evaluations
70 ampl:

```

The output from Knitro tells us that the algorithm terminated successfully (“Exit: Locally optimal solution found.” on line 44), that the objective value at the optimum found is about 936.0 (line 48) and that it took Knitro about 1 millisecond to solve the problem (line 56). 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. See [Knitro / MATLAB reference](#) for a more extensive description of *knitromatlab* interface.

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 constraint ("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 License
      Artelys Knitro 11.0.0
=====

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_initpt from AUTO to 3.
Knitro changing bar_murule from AUTO to 4.
Knitro changing bar_penaltycons from AUTO to 1.
Knitro changing bar_penaltyrule from AUTO to 2.
Knitro changing bar_switchrule from AUTO to 2.
Knitro changing linesearch from AUTO to 1.
Knitro changing linsolver from AUTO to 2.

Problem Characteristics ( Presolved)
-----
Objective goal: Minimize
Objective type: general
Number of variables:          3 (          3)
  bounded below only:         0 (          0)
  bounded above only:         0 (          0)
  bounded below and above:    3 (          3)
  fixed:                       0 (          0)
  free:                         0 (          0)
Number of constraints:        2 (          2)
  linear equalities:           1 (          1)
  quadratic equalities:        0 (          0)
  gen. nonlinear equalities:    0 (          0)
  linear one-sided inequalities: 0 (          0)
  quadratic one-sided inequalities: 0 (          0)
  gen. nonlinear one-sided inequalities: 1 (          1)
  linear two-sided inequalities: 0 (          0)
  quadratic two-sided inequalities: 0 (          0)
  gen. nonlinear two-sided inequalities: 0 (          0)
Number of nonzeros in Jacobian: 6 (          6)
Number of nonzeros in Hessian: 6 (          6)

EXIT: Locally optimal solution found.

Final Statistics
-----
Final objective value          = 9.360000000000049e+02
Final feasibility error (abs / rel) = 7.11e-15 / 5.47e-16
Final optimality error (abs / rel) = 1.21e-07 / 7.56e-09
```

```

# of iterations           =           7
# of CG iterations       =           0
# of function evaluations =          32
# of gradient evaluations =           0
Total program time (secs) =      0.01931 (      0.019 CPU time)
Time spent in evaluations (secs) =      0.01341
=====

```

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, there were 0 function evaluations. This is because the model only has linear and quadratic structure and AMPL is able to provide all this structural information directly to Knitro so that Knitro does not callback to AMPL for any evaluations. If there were more general nonlinear structure in the model, then Knitro would callback to AMPL to get these evaluations as well as the evaluations of their derivatives. The MATLAB interface does not currently provide quadratic structure to Knitro, so 32 function evaluations are needed 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 used to build a model in pieces while providing special structures to Knitro (e.g. linear structures, quadratic structures), while providing callbacks to handle general, nonlinear structures. A typical sequence of function calls looks as follows:

- `KN_new()`: create a new Knitro solver context pointer, allocating resources.
- `KN_add_vars()/KN_add_cons()/KN_set_*bnds()`: add basic problem information to Knitro.
- `KN_add_*_linear_struct()/KN_add_*_quadratic_struct()`: add special problem structures.
- `KN_add_eval_callback()`: add callback for nonlinear evaluations if needed.
- `KN_set_cb_*()`: set properties for nonlinear evaluation callbacks.
- `KN_set_*_param()`: set user options/parameters.
- `KN_solve()`: solve the problem.
- `KN_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 callable library equivalent is the following (see `exampleQCQP.c` provided with the distribution in `examples/C/`).

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "knitro.h"
4
5  /* main */
6  int main (int argc, char *argv[]) {
7      int i, nStatus, error;
8
9      /** Declare variables. */
10     KN_context *kc;
11     int n, m;
12     double x[3];
13     double xLoBnds[3] = {0, 0, 0};
14     double xInitVals[3] = {2.0, 2.0, 2.0};
15     /** Used to define linear constraint. */
16     int lconIndexVars[3] = { 0, 1, 2};
17     double lconCoefs[3] = {8.0, 14.0, 7.0};
18     /** Used to specify quadratic constraint. */
19     int qconIndexVars1[3] = { 0, 1, 2};
20     int qconIndexVars2[3] = { 0, 1, 2};
21     double qconCoefs[3] = {1.0, 1.0, 1.0};
22     /** Used to specify quadratic objective terms. */
23     int qobjIndexVars1[5] = { 0, 1, 2, 0, 0};
24     int qobjIndexVars2[5] = { 0, 1, 2, 1, 2};
25     double qobjCoefs[5] = {-1.0, -2.0, -1.0, -1.0, -1.0};
26     /** Solution information */
27     double objSol;
28     double feasError, optError;
29
30     /** Create a new Knitro solver instance. */
31     error = KN_new(&kc);
32     if (error) exit(-1);
33     if (kc == NULL)
34     {
35         printf ("Failed to find a valid license.\n");
36         return( -1 );
37     }
38
39     /** Illustrate how to override default options by reading from
40      * the knitro.opt file. */
41     error = KN_load_param_file (kc, "knitro.opt");
42     if (error) exit(-1);
43
44     /** Initialize Knitro with the problem definition. */
45
46     /** Add the variables and set their bounds and initial values.
47      * Note: unset bounds assumed to be infinite. */
48     n = 3;
49     error = KN_add_vars(kc, n, NULL);
50     if (error) exit(-1);
51     error = KN_set_var_lobnds_all(kc, xLoBnds);
52     if (error) exit(-1);
53     error = KN_set_var_primal_init_values_all(kc, xInitVals);
54     if (error) exit(-1);
55
56     /** Add the constraints and set their bounds. */
57     m = 2;
58     error = KN_add_cons(kc, m, NULL);

```

```

59  if (error) exit(-1);
60  error = KN_set_con_eqbnd(kc, 0, 56.0);
61  if (error) exit(-1);
62  error = KN_set_con_lobnd(kc, 1, 25.0);
63  if (error) exit(-1);
64
65  /** Add coefficients for linear constraint. */
66  error = KN_add_con_linear_struct_one (kc, 3, 0, lconIndexVars,
67                                       lconCoefs);
68  if (error) exit(-1);
69
70  /** Add coefficients for quadratic constraint */
71  error = KN_add_con_quadratic_struct_one (kc, 3, 1, qconIndexVars1,
72                                         qconIndexVars2, qconCoefs);
73  if (error) exit(-1);
74
75  /** Set minimize or maximize (if not set, assumed minimize) */
76  error = KN_set_obj_goal(kc, KN_OBJGOAL_MINIMIZE);
77  if (error) exit(-1);
78
79  /** Add constant value to the objective. */
80  error= KN_add_obj_constant(kc, 1000.0);
81  if (error) exit(-1);
82
83  /** Set quadratic objective structure. */
84  error = KN_add_obj_quadratic_struct (kc, 5, qobjIndexVars1,
85                                       qobjIndexVars2, qobjCoefs);
86  if (error) exit(-1);
87
88  /** Solve the problem.
89   * 
90   * Return status codes are defined in "knitro.h" and described
91   * in the Knitro manual. */
92  nStatus = KN_solve (kc);
93
94  printf ("\n\n");
95  printf ("Knitro converged with final status = %d\n",
96         nStatus);
97
98  /** An example of obtaining solution information. */
99  error = KN_get_solution(kc, &nStatus, &objSol, x, NULL);
100 if (!error) {
101     printf (" optimal objective value = %e\n", objSol);
102     printf (" optimal primal values x = (%e, %e, %e)\n", x[0], x[1], x[2]);
103 }
104 error = KN_get_abs_feas_error (kc, &feasError);
105 if (!error)
106     printf (" feasibility violation = %e\n", feasError);
107 error = KN_get_abs_opt_error (kc, &optError);
108 if (!error)
109     printf (" KKT optimality violation = %e\n", optError);
110
111 /** Delete the Knitro solver instance. */
112 KN_free (&kc);
113
114 return( 0 );
115 }

```

Note that the AMPL equivalent is much shorter and simpler (only a few lines of code). In both the AMPL example and this example, the quadratic structure is passed directly to Knitro, so no callback evaluations are needed. However, when there is more general nonlinear structure AMPL will often be more efficient since it is able to provide Knitro the exact derivatives of all nonlinear functions automatically as needed. 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. 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 License
3           Artelys Knitro 11.0.0
4 =====
5
6 Knitro presolve eliminated 0 variables and 0 constraints.
7
8 The problem is identified as a QCQP.
9 Knitro changing algorithm from AUTO to 1.
10 Knitro changing bar_initpt from AUTO to 3.
11 Knitro changing bar_murule from AUTO to 4.
12 Knitro changing bar_penaltycons from AUTO to 1.
13 Knitro changing bar_penaltyrule from AUTO to 2.
14 Knitro changing bar_switchrule from AUTO to 2.
15 Knitro changing linesearch from AUTO to 1.
16 Knitro changing linsolver from AUTO to 2.
17
18 Problem Characteristics                ( Presolved)
19 -----
20 Objective goal: Minimize
21 Objective type: quadratic
22 Number of variables:                   3 (          3)
23     bounded below only:                 3 (          3)
24     bounded above only:                 0 (          0)
25     bounded below and above:            0 (          0)
26     fixed:                              0 (          0)
27     free:                               0 (          0)
28 Number of constraints:                  2 (          2)
29     linear equalities:                   1 (          1)
30     quadratic equalities:                0 (          0)
31     gen. nonlinear equalities:           0 (          0)
32     linear one-sided inequalities:        0 (          0)
33     quadratic one-sided inequalities:     1 (          1)
34     gen. nonlinear one-sided inequalities: 0 (          0)
35     linear two-sided inequalities:        0 (          0)
36     quadratic two-sided inequalities:     0 (          0)
37     gen. nonlinear two-sided inequalities: 0 (          0)
38 Number of nonzeros in Jacobian:          6 (          6)
39 Number of nonzeros in Hessian:          5 (          5)
40
41      Iter      Objective      FeasError      OptError      ||Step||      CGits
42 -----
43          0      9.760000e+02      1.300e+01
44          9      9.360000e+02      0.000e+00      1.515e-09      5.910e-05      0
45
46 EXIT: Locally optimal solution found.
47

```

```

48 Final Statistics
49 -----
50 Final objective value           = 9.36000000015579e+02
51 Final feasibility error (abs / rel) = 0.00e+00 / 0.00e+00
52 Final optimality error (abs / rel) = 1.51e-09 / 9.47e-11
53 # of iterations                 = 9
54 # of CG iterations              = 2
55 # of function evaluations       = 0
56 # of gradient evaluations       = 0
57 # of Hessian evaluations        = 0
58 Total program time (secs)       = 0.00207 ( 0.001 CPU time)
59 Time spent in evaluations (secs) = 0.00000
60
61 =====
62
63
64 Knitro converged with final status = 0
65   optimal objective value = 9.360000e+02
66   optimal primal values x = (1.514577e-09, 1.484137e-14, 8.000000e+00)
67   feasibility violation   = 0.000000e+00
68   KKT optimality violation = 1.514577e-09

```

Again, the solution value is the same (about 936.0), and the details of the log are similar (we used a different algorithm in the AMPL example).

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 API reference*) also contains a comprehensive documentation of the Knitro callable library API.

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.

2.1.4 Getting started with the object-oriented interface

The Knitro object-oriented interface provides an object-oriented wrapper around the Knitro callable library. The interface is available in C++, C#, and Java. This document focuses on the C++ version of the interface. The interfaces are the same in functionality, differing only in language syntax and data types used in functions (e.g., `std::vector<>` in C++, `IList<>` in C#, and `List<>` in Java). Examples for each of the languages are available in the Knitro examples folders.

The C++ object-oriented interface is a header-only interface. Complete source code for the interface is included with Knitro for informational purposes. Usage requires including the `knitro.h` header within the `include` directory of Knitro and linking against the appropriate Knitro library file within the `lib` directory of Knitro.

The object-oriented API is used to solve an optimization problem through a sequence of function calls:

- `KTRIPProblem*` instance: create an instance of the problem to be solved by Knitro. The class is user-defined, inherits from the `KTRIPProblem` class, and defines the problem characteristics.
- `KTRSolver solver(instance)`: load the problem definition into the Knitro solver and check out a Knitro license.
- `solver.solve()`: solve the problem, with output stored in the `solver` object.

The example below shows how to define a problem and class and use these function calls.

First example

The following defines the same toy problem solved using AMPL, MATLAB, and the callable library.

```

1  #include "KTRSolver.h"
2  #include "KTRProblem.h"
3  #include <iostream>
4
5  class ProblemExample : public knitro::KTRProblem {
6  private:
7      // objective properties
8      void setObjectiveProperties() {
9          setObjType(knitro::KTREnums::ObjectiveType::ObjGeneral);
10         setObjGoal(knitro::KTREnums::ObjectiveGoal::Minimize);
11     }
12
13     // variable bounds. All variables 0 <= x.
14     void setVariableProperties() {
15         setVarLoBnds(0.0);
16     }
17
18     // constraint properties
19     void setConstraintProperties() {
20         // set constraint types
21         setConTypes(0, knitro::KTREnums::ConstraintType::ConGeneral);
22         setConTypes(1, knitro::KTREnums::ConstraintType::ConGeneral);
23
24         // set constraint lower bounds to zero for all variables
25         setConLoBnds(0.0);
26
27         // set constraint upper bounds
28         setConUpBnds(0, 0.0);
29         setConUpBnds(1, KTR_INFBOUND);
30     }
31
32 public:
33     // constructor: pass number of variables and constraints to base class.
34     // 3 variables, 2 constraints.
35     ProblemExample() : KTRProblem(3, 2) {
36         // set problem properties in constructor
37         setObjectiveProperties();
38         setVariableProperties();
39         setConstraintProperties();
40     }
41
42     // Objective and constraint evaluation function
43     // overrides KTRIPProblem class
44     double evaluateFC(

```

```

45     const std::vector<double>& x,
46     std::vector<double>& c,
47     std::vector<double>& objGrad,
48     std::vector<double>& jac) {
49
50     // constraints
51     c[0] = 8.0e0*x[0] + 14.0e0*x[1] + 7.0e0*x[2] - 56.0e0;
52     c[1] = x[0] * x[0] + x[1] * x[1] + x[2] * x[2] - 25.0e0;
53
54     // return objective function value
55     return 1000 - x[0] * x[0] - 2.0e0*x[1] * x[1] - x[2] * x[2]
56           - x[0] * x[1] - x[0] * x[2];
57     }
58 };
59
60 int main(int argc, char *argv[]) {
61     // Create a problem instance.
62     ProblemExample* problem = new ProblemExample();
63
64     // Create a solver - optional arguments: use numerical derivative evaluation.
65     knitro::KTRSolver solver(problem, KTR_GRADOPT_FORWARD, KTR_HESSOPT_BFGS);
66
67     int solveStatus = solver.solve();
68
69     if (solveStatus != 0) {
70         std::cout << std::endl;
71         std::cout << "Knitro failed to solve the problem, final status = ";
72         std::cout << solveStatus << std::endl;
73     }
74     else {
75         std::cout << std::endl << "Knitro successful, objective is = ";
76         std::cout << solver.getObjValue() << std::endl;
77     }
78
79     return 0;
80 }

```

This is similar to the callable library example. The problem definition is contained in a class definition and is simpler. Variable and constraint properties can be defined more compactly; memory for the problem characteristics does not need to be allocated by the user; and the Jacobian sparsity pattern is automatically defined as a full matrix because no Jacobian non-zero size is provided.

The Knitro solver functions are called from the `KTRSolver` class. Like the callable library, the object-oriented interface does not provide automatic derivatives. Derivatives can be computed manually and defined in the `KTRProblem` class. This is covered in the chapter on *Derivatives*.

The above example can be compiled and linked against Knitro, and produces the same output as the callable library output. As in the callable library, this requires more problem definition than AMPL, such as defining the variable and constraint types. The object-oriented interface provide ease-of-use over the callable library with similar functionality and performance.

Further information

Another chapter of this documentation is dedicated to the object-oriented interface (*Object-oriented interface reference*). The reference manual chapter on the callable library (*Callable library API reference*) provides information on the callable library underlying the object-oriented interface. This section provides a comprehensive documentation of the Knitro callable library functions, which are accessible through methods of the `KTRSolver` class.

Finally, the source code for the object-oriented interface is provided as a reference. The `.h` header files for the C++ interface document the interface functionality.

Additional examples

More examples using the object-oriented interface are provided in the `examples/C++`, `examples/CSharp` and `examples/Java` directories of the Knitro distribution.

2.1.5 Getting started with R

The Knitro interface for R, called *KnitroR*, is provided with your Knitro distribution. In order to install it, you need R 3.0 or later and run the following command in the R prompt:

```
install.packages('KnitroR', repos= NULL)
```

Do not forget to define an environment variable `KNITRODIR` pointing to your local installation directory of Knitro.

To test whether your installation is correct, type in the expression:

```
knitro(objective=function(x) x[1]*x[2], x0=c(1,1))
```

at the R command prompt. If your installation was successful, *KnitroR* returns the following message:

```
$statusMessage
[1] "Optimal solution found !"
$x
[1] 0 0
$lambda
[1] 0 0
$objective
[1] 0
$constraints
numeric(0)
$iter
[1] 1
$objEval
[1] 9
$gradEval
[1] 0
```

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 *KnitroR* interface

The Knitro solver can be called via the following optimization function:

```
sol <- knitro(nvar=..., ncon=..., x0=...,
             objective=..., gradient=..., constraints=...,
             jacobian=..., jacIndexCons=..., jacIndexVars=...,
             hessianLag=..., hessIndexRows=..., hessIndexCols=...,
             xL=..., xU=..., cL=..., cU=...,
             options=...)
```

but the simplest function call reduces to:

```
sol <- knitro(objective=..., x0=...)
sol <- knitro(objective=..., xL=...)
sol <- knitro(objective=..., xU=...)
sol <- knitro(nvar=..., objective=...)
```

First R example

The following introductory examples shows how to solve the Rosenbrock banana function.

```
library('KnitroR')

# Rosenbrock Banana function
eval_f <- function(x) {
  return( 100 * (x[2] - x[1] * x[1])^2 + (1 - x[1])^2 )
}

eval_grad_f <- function(x) {
  grad_f <- rep(0, length(x))

  grad_f[1] <- 2*x[1]-2+400*x[1]^3-400*x[1]*x[2]
  grad_f[2] <- 200*(x[2]-x[1]^2)

  return( grad_f )
}

# initial values
x0 <- c( -1.2, 1 )

sol <- knitro(x0 = x0, objective = eval_f)
```

We can save this code in a file 'example.R' and run it from the R command prompt via the following command:

```
source('example.R')
```

KnitroR returns the following output:

```
=====
      Commercial License
      Artelys Knitro 10.0.1
=====

Knitro performing finite-difference gradient computation with 1 thread.
Knitro presolve eliminated 0 variables and 0 constraints.

gradopt:          2
hessopt:          2
outlev:           1
par_concurrent_evals: 0
The problem is identified as unconstrained.
Knitro changing algorithm from AUTO to 1.
Knitro changing bar_initpt from AUTO to 3.
Knitro changing bar_murule from AUTO to 4.
Knitro changing bar_penaltycons from AUTO to 1.
Knitro changing bar_penaltyrule from AUTO to 2.
Knitro changing bar_switchrule from AUTO to 1.
Knitro changing linsolver from AUTO to 2.
```

```

Knitro performing finite-difference gradient computation with 1 thread.

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

EXIT: Locally optimal solution found.

Final Statistics
-----
Final objective value                               = 2.00430825877435e-011
Final feasibility error (abs / rel) = 0.00e+000 / 0.00e+000
Final optimality error (abs / rel) = 1.66e-007 / 1.66e-007
# of iterations                                     = 36
# of CG iterations                                  = 5
# of function evaluations                           = 134
# of gradient evaluations                           = 0
Total program time (secs)                           = 0.069 ( 0.000 CPU time)
Time spent in evaluations (secs)                     = 0.001

=====

```

Further information

More functions are available and R interface can be used to solve MINLP and least squares problems as well. Another chapter of this documentation is dedicated to the R interface (*Knitro / R reference*) and provides exhaustive references.

Any Knitro option can also be provided to the R interface. A comprehensive documentation of Knitro options is available in the section *Knitro user options*.

Additional examples

More examples using the R interface are provided in the `examples/R` directory 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, `KN_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 `act_` apply only to the active-set/SQP 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 KN_set_int_param (KN_context *kc, int param_id, int value)
```

for setting integer valued parameters, or:

```
int KN_set_double_param (KN_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 = KN_set_int_param (kc, KN_PARAM_ALG, KN_ALG_BAR_CG);
status = KN_set_double_param (kc, KN_PARAM_OPTTOL, 1.0e-8);
```

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

2.2.4 Setting KNITRO options with the object-oriented interface

User options are set with a single overloaded KTRSolver function that has the form:

```
void solver.setParam(param_identifier, param_value);
```

The argument `param_identifier` is either a string with the parameter's name or an integer with the parameter's identifier number (enumerated object with all options have been defined to simplify this step). The argument `param_value` is an integer, double, or string, depending on the type of parameter that is set.

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

```
solver.setParam(KTR_PARAM_ALG, KTR_ALG_BAR_CG);
solver.setParam(KTR_PARAM_OPTTOL, 1.0e-8);
```

Refer to the Callable Library Reference Manual (*Changing and reading solver parameters*) for a comprehensive list of parameters. The object-oriented interface uses the same parameters as the callable library.

2.2.5 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:

```
int KN_load_param_file (KN_context *kc, char const *filename)
```

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

```
status = KN_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 KN_save_param_file (KN_context *kc, char const *filename)
```

For example:

```
status = KN_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 `KN_load_param_file()`, such as `examples/C/exampleNLP1.c`.

In the object oriented interface, the equivalent functions for loading and saving parameter files are the following:

```
void KTRSolver::loadParamFile(std::string filename);
void KTRSolver::saveParamFile(std::string filename);
```

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.

Note: When using finite-difference gradients, the sparsity pattern for the constraint Jacobian should still be provided if possible to improve the efficiency of the finite-difference computation and decrease the memory requirements.

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 (e.g., $n < 1000$). 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 (i.e., the problem is not convex) 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 (e.g., $n < 1000$). 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`.

Note: When using a Hessian approximation option (i.e. `hessopt > 1`), you do not need to provide any sparsity pattern for the Hessian matrix.

As with exact first derivatives, exact second derivatives often provide a substantial benefit to Knitro and it is advised to provide them whenever possible. If the exact second derivative (i.e. the Hessian) matrix is provided by the user, it can (and should) be checked against a finite-difference approximation for errors using the Knitro derivative checker. See (*Checking derivatives*) below.

2.3.3 Jacobian and Hessian derivative matrices

The Jacobian matrix of the constraints is defined as

$$J(x) = [\nabla c_0(x) \quad \dots \quad \nabla c_{m-1}(x)]$$

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:

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

Rewriting in the Knitro standard notation, we have

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

Note: For demonstration purposes we show how to compute the Jacobian and Hessian corresponding to all constraints and components of this model. However, with the new API introduced in Knitro 11.0 the quadratic constraint c_1 and the linear constraint c_2 should be loaded separately and should not be included in the sparse Jacobian or Hessian structures provided through the nonlinear callbacks. In addition, the linear term in the objective could also be provided separately if desired.

Note: Please see the examples provided in `examples/C/` which demonstrate how to provide the derivatives (e.g. objective gradient, constraint Jacobian, and Hessian) for nonlinear terms via callbacks for different types of models, while loading linear and quadratic structures separately.

Computing the Sparse Jacobian Matrix

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

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

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 `KN_set_cb_grad()` 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 `KN_set_cb_hess()` by the values of `hessIndexVars1` and `hessIndexVars2`.

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#brhkgvh-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;
```

In the object-oriented interface, these values are set in the user-defined problem class by implementing:

```
std::vector<int> KTRIPProblem::getJacIndexCons();
std::vector<int> KTRIPProblem::getJacIndexVars();
```

to return vectors with the constraint and variable indices in either column-wise or row-wise order. If using the `KTRIPProblem` class, setting the values with:

```
KTRIPProblem::setJacIndexCons(int id, int val);
KTRIPProblem::setJacIndexVars(int id, int val);
```

will store the values to be returned by the appropriate `get` functions.

Note: Using `KTRIPProblem` class, by default the Jacobian is assumed to be dense and stored row-wise.

Note: Even if the application does not evaluate derivatives (i.e. finite-difference first derivatives are used), 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.

Note: When using finite-difference first derivatives (*gradopt* > 1), if the sparsity pattern is unknown, then the application should specify a fully dense pattern (i.e., assume all partial derivatives are nonzero). This can easily and automatically be done by setting *nnzJ* to either `KN_DENSE_ROWMAJOR` or `KN_DENSE_ROWMAJOR` in the callable library function `KN_set_cb_grad()` (and setting *jacIndexCons* and *jacIndexVars* to be `NULL`).

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 *hessIndexVars1* and *hessIndexVars2* 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*, *hessIndexVars1* and *hessIndexVars2* are as follows:

```
hess[0] = -lambda[0]*cos(x[0]) + 2*lambda[1];
hessIndexVars1[0] = 0;
hessIndexVars2[0] = 0;

hess[1] = 2*lambda[1];
hessIndexVars1[1] = 1;
hessIndexVars2[1] = 1;

hess[2] = sigma*3*x[2]*x[2];
hessIndexVars1[2] = 1;
hessIndexVars2[2] = 2;

hess[3] = sigma*6*x[1]*x[2];
hessIndexVars1[3] = 2;
hessIndexVars2[3] = 2;
```

In the object-oriented interface, the Hessian matrix column indices are set in the user-defined problem class by implementing:

```
std::vector<int> KTRProblem::getHessIndexRows();
std::vector<int> KTRProblem::getHessIndexCols();
```

and having them return vectors with the Hessian row and column indices, respectively. If using the `KTRProblem` class, setting the values with:

```
KTRProblem::setHessIndexRows(int id, int val);
KTRProblem::setHessIndexCols(int id, int val);
```

will store the values to be returned by the appropriate `get` functions.

Note: In Knitro, the array *objGrad* corresponding to $\nabla f(x)$, can be provided in dense or sparse form. The arrays *jac*, *jacIndexCons*, and *jacIndexVars* store information concerning *only the nonzero* (and typically nonlinear) 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 either row-wise or 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 nargin == 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 nargin==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 License
      Artelys Knitro 11.0.0
=====

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_initpt from AUTO to 3.
Knitro changing bar_murule from AUTO to 4.
Knitro changing bar_penaltycons from AUTO to 1.
Knitro changing bar_penaltyrule from AUTO to 2.
```

```

Knitro changing bar_switchrule from AUTO to 2.
Knitro changing linsolver from AUTO to 2.

Problem Characteristics ( Presolved)
-----
Objective goal: Minimize
Objective type: general
Number of variables: 3 ( 3)
    bounded below only: 0 ( 0)
    bounded above only: 0 ( 0)
    bounded below and above: 3 ( 3)
    fixed: 0 ( 0)
    free: 0 ( 0)
Number of constraints: 2 ( 2)
    linear equalities: 1 ( 1)
    quadratic equalities: 0 ( 0)
    gen. nonlinear equalities: 0 ( 0)
    linear one-sided inequalities: 0 ( 0)
    quadratic one-sided inequalities: 0 ( 0)
    gen. nonlinear one-sided inequalities: 1 ( 1)
    linear two-sided inequalities: 0 ( 0)
    quadratic two-sided inequalities: 0 ( 0)
    gen. nonlinear two-sided inequalities: 0 ( 0)
Number of nonzeros in Jacobian: 6 ( 6)
Number of nonzeros in Hessian: 6 ( 6)

EXIT: Locally optimal solution found.

Final Statistics
-----
Final objective value = 9.36000000000049e+02
Final feasibility error (abs / rel) = 0.00e+00 / 0.00e+00
Final optimality error (abs / rel) = 4.78e-08 / 2.99e-09
# of iterations = 7
# of CG iterations = 0
# of function evaluations = 8
# of gradient evaluations = 8
Total program time (secs) = 0.01651 ( 0.015 CPU time)
Time spent in evaluations (secs) = 0.00472
=====

```

The number of function evaluation was reduced to 8, 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 show how to provide derivatives through the callable library API. Here we look at the C example examples/C/exampleNLP2.c, which solves the model:

```

max    x0*x1*x2*x3          (obj)
s.t.   x0^3 + x1^2 = 1      (c0)
       x0^2*x3 - x2 = 0    (c1)

```

$$x^3 - x = 0 \quad (c2)$$

Note that this problem has linear terms, quadratic terms and general nonlinear terms. We will show how to provide both first and second derivatives for the nonlinear structure through callbacks while separately loading the linear and quadratic structure.

```

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

/** Callback for nonlinear function evaluations */
int callbackEvalFC (KN_context_ptr      kc,
                  CB_context_ptr      cb,
                  KN_eval_request_ptr  const evalRequest,
                  KN_eval_result_ptr  const evalResult,
                  void                 * const userParams)
{
    const double *x;
    double *obj;
    double *c;

    if (evalRequest->type != KN_RC_EVALFC)
    {
        printf ("*** callbackEvalFC incorrectly called with eval type %d\n",
                evalRequest->type);
        return( -1 );
    }
    x = evalRequest->x;
    obj = evalResult->obj;
    c = evalResult->c;

    /** Evaluate nonlinear term in objective */
    *obj = x[0]*x[1]*x[2]*x[3];

    /** Evaluate nonlinear terms in constraints */
    c[0] = x[0]*x[0]*x[0];
    c[1] = x[0]*x[0]*x[3];

    return( 0 );
}

/** Callback for nonlinear gradient/Jacobian evaluations */
int callbackEvalGA (KN_context_ptr      kc,
                  CB_context_ptr      cb,
                  KN_eval_request_ptr  const evalRequest,
                  KN_eval_result_ptr  const evalResult,
                  void                 * const userParams)
{
    const double *x;
    double *objGrad;
    double *jac;

    if (evalRequest->type != KN_RC_EVALGA)
    {
        printf ("*** callbackEvalGA incorrectly called with eval type %d\n",
                evalRequest->type);
        return( -1 );
    }

```

```

    }
    x = evalRequest->x;
    objGrad = evalResult->objGrad;
    jac = evalResult->jac;

    /** Evaluate nonlinear term in objective gradient */
    objGrad[0] = x[1]*x[2]*x[3];
    objGrad[1] = x[0]*x[2]*x[3];
    objGrad[2] = x[0]*x[1]*x[3];
    objGrad[3] = x[0]*x[1]*x[2];

    /** Evaluate nonlinear terms in constraint gradients (Jacobian) */
    jac[0] = 3.0*x[0]*x[0]; /* derivative of x0^3 term wrt x0 */
    jac[1] = 2.0*x[0]*x[3]; /* derivative of x0^2*x3 term wrt x0 */
    jac[2] = x[0]*x[0]; /* derivative of x0^2*x3 terms wrt x3 */

    return( 0 );
}

/** Callback for nonlinear Hessian evaluation */
int callbackEvalH (KN_context_ptr      kc,
                  CB_context_ptr      cb,
                  KN_eval_request_ptr  const evalRequest,
                  KN_eval_result_ptr  const evalResult,
                  void                 * const userParams)
{
    const double *x;
    const double *lambda;
    double sigma;
    double *hess;

    if ( evalRequest->type != KN_RC_EVALH
        && evalRequest->type != KN_RC_EVALH_NO_F )
    {
        printf ("*** callbackEvalHess incorrectly called with eval type %d\n",
                evalRequest->type);
        return( -1 );
    }

    x = evalRequest->x;
    lambda = evalRequest->lambda;
    /** Scale objective component of the Hessian by sigma */
    sigma = *(evalRequest->sigma);
    hess = evalResult->hess;

    /** Evaluate nonlinear term in the Hessian of the Lagrangian */
    hess[0] = lambda[0]*6.0*x[0] + lambda[1]*2.0*x[3];
    hess[1] = sigma*x[2]*x[3];
    hess[2] = sigma*x[1]*x[3];
    hess[3] = sigma*x[1]*x[2] + lambda[1]*2.0*x[0];
    hess[4] = sigma*x[0]*x[3];
    hess[5] = sigma*x[0]*x[2];
    hess[6] = sigma*x[0]*x[1];

    return( 0 );
}

/** main function */

```

```

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

    /** Declare variables. */
    KN_context *kc;
    int n, m;
    double cEqBnds[3] = {1.0, 0.0, 0.0};
    /** Used to define linear constraint structure */
    int lconIndexCons[2];
    int lconIndexVars[2];
    double lconCoefs[2];
    /** Used to define quadratic constraint structure */
    int qconIndexCons[2];
    int qconIndexVars1[2];
    int qconIndexVars2[2];
    double qconCoefs[2];
    /** Pointer to structure holding information for nonlinear
     * evaluation callback for terms:
     * x0*x1*x2*x3 in the objective
     * x0^3 in first constraint
     * x0^2*x3 in second constraint */
    CB_context *cb;
    int cIndices[2];
    /** Used to define Jacobian structure for nonlinear terms
     * evaluated in the callback. */
    int cbjacIndexCons[3];
    int cbjacIndexVars[3];
    double cbjacCoefs[3];
    /** Used to define Hessian structure for nonlinear terms
     * evaluated in the callback. */
    int cbhessIndexVars1[7];
    int cbhessIndexVars2[7];
    double cbhessCoefs[7];
    /** For solution information */
    double x[4];
    double objSol;
    double feasError, optError;

    /** Create a new Knitro solver instance. */
    error = KN_new(&kc);
    if (error) exit(-1);
    if (kc == NULL)
    {
        printf ("Failed to find a valid license.\n");
        return( -1 );
    }

    /** Initialize Knitro with the problem definition. */

    /** Add the variables and specify initial values for them.
     * Note: any unset lower bounds are assumed to be
     * unbounded below and any unset upper bounds are
     * assumed to be unbounded above. */
    n = 4;
    error = KN_add_vars(kc, n, NULL);
    if (error) exit(-1);
    for (i=0; i<n; i++) {

```

```

        error = KN_set_var_primal_init_value(kc, i, 0.8);
        if (error) exit(-1);
    }

    /** Add the constraints and set the rhs and coefficients */
    m = 3;
    error = KN_add_cons(kc, m, NULL);
    if (error) exit(-1);
    error = KN_set_con_eqbnds_all(kc, cEqBnds);
    if (error) exit(-1);

    /** Coefficients for 2 linear terms */
    lconIndexCons[0] = 1; lconIndexVars[0] = 2; lconCoefs[0] = -1.0;
    lconIndexCons[1] = 2; lconIndexVars[1] = 1; lconCoefs[1] = -1.0;
    error = KN_add_con_linear_struct (kc, 2,
                                     lconIndexCons, lconIndexVars,
                                     lconCoefs);

    if (error) exit(-1);

    /** Coefficients for 2 quadratic terms */

    /* x1^2 term in c0 */
    qconIndexCons[0] = 0; qconIndexVars1[0] = 1; qconIndexVars2[0] = 1;
    qconCoefs[0] = 1.0;

    /* x3^2 term in c2 */
    qconIndexCons[1] = 2; qconIndexVars1[1] = 3; qconIndexVars2[1] = 3;
    qconCoefs[1] = 1.0;

    error = KN_add_con_quadratic_struct (kc, 2, qconIndexCons,
                                         qconIndexVars1, qconIndexVars2,
                                         qconCoefs);

    if (error) exit(-1);

    /** Add callback to evaluate nonlinear (non-quadratic) terms in the model:
     *   x0*x1*x2*x3   in the objective
     *   x0^3          in first constraint c0
     *   x0^2*x3      in second constraint c1 */
    cIndices[0] = 0; cIndices[1] = 1;
    error = KN_add_eval_callback (kc, KNTRUE, 2, cIndices, callbackEvalFC, &cb);
    if (error) exit(-1);

    /** Set obj. gradient and nonlinear jac provided through callbacks.
     *   Mark objective gradient as dense, and provide non-zero sparsity
     *   structure for constraint Jacobian terms. */
    cbjacIndexCons[0] = 0; cbjacIndexVars[0] = 0;
    cbjacIndexCons[1] = 1; cbjacIndexVars[1] = 0;
    cbjacIndexCons[2] = 1; cbjacIndexVars[2] = 3;
    error = KN_set_cb_grad(kc, cb, KN_DENSE, NULL, 3, cbjacIndexCons,
                          cbjacIndexVars, callbackEvalGA);

    if (error) exit(-1);

    /** Set nonlinear Hessian provided through callbacks. Since the
     *   Hessian is symmetric, only the upper triangle is provided.
     *   The upper triangular Hessian for nonlinear callback structure is:
     *
     *   lambda0*6*x0 + lambda1*2*x3      x2*x3      x1*x3      x1*x2 + lambda1*2*x0
     *           0                        0          x0*x3      x0*x2
    */

```

```

*
*
*      0      x0*x1
*      0
* (7 nonzero elements)
*/
cbhessIndexVars1[0] = 0; /* (row0,col0) element */
cbhessIndexVars2[0] = 0;
cbhessIndexVars1[1] = 0; /* (row0,col1) element */
cbhessIndexVars2[1] = 1;
cbhessIndexVars1[2] = 0; /* (row0,col2) element */
cbhessIndexVars2[2] = 2;
cbhessIndexVars1[3] = 0; /* (row0,col3) element */
cbhessIndexVars2[3] = 3;
cbhessIndexVars1[4] = 1; /* (row1,col2) element */
cbhessIndexVars2[4] = 2;
cbhessIndexVars1[5] = 1; /* (row1,col3) element */
cbhessIndexVars2[5] = 3;
cbhessIndexVars1[6] = 2; /* (row2,col3) element */
cbhessIndexVars2[6] = 3;
error = KN_set_cb_hess(kc, cb, 7, cbhessIndexVars1, cbhessIndexVars2,
↳callbackEvalH);
if (error) exit(-1);

/** Set minimize or maximize (if not set, assumed minimize) */
error = KN_set_obj_goal(kc, KN_OBJGOAL_MAXIMIZE);
if (error) exit(-1);

/** Demonstrate setting a "newpt" callback. the callback function
 * "callbackNewPoint" passed here is invoked after Knitro computes
 * a new estimate of the solution. */
error = KN_set_newpt_callback(kc, callbackNewPoint, NULL);
if (error) exit(-1);

/** Set option to print output after every iteration. */
error = KN_set_int_param (kc, KN_PARAM_OUTLEV, KN_OUTLEV_ITER);
if (error) exit(-1);

/** Solve the problem.
 *
 * Return status codes are defined in "knitro.h" and described
 * in the Knitro manual. */
nStatus = KN_solve (kc);

printf ("\n\n");
printf ("Knitro converged with final status = %d\n",
nStatus);

/** An example of obtaining solution information. */
error = KN_get_solution(kc, &nStatus, &objSol, x, NULL);
if (!error) {
    printf (" optimal objective value = %e\n", objSol);
    printf (" optimal primal values x = (%e, %e, %e, %e)\n", x[0], x[1], x[2],
↳x[3]);
}
error = KN_get_abs_feas_error (kc, &feasError);
if (!error)
    printf (" feasibility violation = %e\n", feasError);
error = KN_get_abs_opt_error (kc, &optError);
if (!error)

```

```

        printf (" KKT optimality violation = %e\n", optError);

    /** Delete the Knitro solver instance. */
    KN_free (&kc);

    return( 0 );
}

```

Running this code produces the following output.

```

=====
      Commercial License
      Artelys Knitro 11.0.0
=====

Knitro presolve eliminated 0 variables and 0 constraints.

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

Problem Characteristics                ( Presolved)
-----
Objective goal: Maximize
Objective type: general
Number of variables:                   4 (          4)
  bounded below only:                   0 (          0)
  bounded above only:                   0 (          0)
  bounded below and above:               0 (          0)
  fixed:                                 0 (          0)
  free:                                  4 (          4)
Number of constraints:                  3 (          3)
  linear equalities:                     0 (          0)
  quadratic equalities:                   1 (          1)
  gen. nonlinear equalities:              2 (          2)
  linear one-sided inequalities:          0 (          0)
  quadratic one-sided inequalities:        0 (          0)
  gen. nonlinear one-sided inequalities:   0 (          0)
  linear two-sided inequalities:          0 (          0)
  quadratic two-sided inequalities:        0 (          0)
  gen. nonlinear two-sided inequalities:   0 (          0)
Number of nonzeros in Jacobian:          7 (          7)
Number of nonzeros in Hessian:           9 (          9)

EXIT: Locally optimal solution found.

Final Statistics
-----
Final objective value                    = 2.50000000082290e-01
Final feasibility error (abs / rel) = 1.86e-10 / 1.86e-10
Final optimality error (abs / rel) = 2.66e-09 / 2.66e-09
# of iterations                          = 3

```

```

# of CG iterations           =           0
# of function evaluations    =           4
# of gradient evaluations    =           4
# of Hessian evaluations     =           3
Total program time (secs)    =           0.00183 (    0.001 CPU time)
Time spent in evaluations (secs) =           0.00001

=====

Knitro converged with final status = 0
  optimal objective value = 2.500000e-01
  optimal primal values x = (7.937005e-01, 7.071068e-01, 5.297315e-01, 8.408964e-01)
  feasibility violation   = 1.863212e-10
  KKT optimality violation = 2.660655e-09

```

Providing both first and second derivatives allows KNitro to solve this model in only 4 function evaluations.

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 Object-oriented C++ example

Let us now modify our C++ example from *Getting started with the object-oriented interface*, so as to provide first derivatives.

```

#include "KTRSolver.h"
#include "KTRProblem.h"
#include <iostream>

class ProblemExample : public KNITRO::KTRProblem {
  // objective properties
  void setObjectiveProperties() {
    setObjType(KTR_OBJTYPE_GENERAL);
    setObjGoal(KTR_OBJGOAL_MINIMIZE);
  }

  // constraint properties
  void setConstraintProperties()
  {
    // set constraint types
    setConTypes(0, KNITRO::KTREnums::ConstraintType::ConLinear);
    setConTypes(1, KNITRO::KTREnums::ConstraintType::ConQuadratic);

    // set constraint lower bounds
    setConLoBnds(0.0);

    // set constraint upper bounds
    setConUpBnds(0, 0.0);
    setConUpBnds(1, KTR_INFBOUND);
  }

  // Variable bounds. All variables 0 <= x.
  void setVariableProperties() {
    setVarLoBnds(0.0);
  }
};

```

```

    }

public:
    // constructor: pass number of variables and constraints to base class
    ProblemQCQP() : KTRProblem(3, 2) {
        // set problem properties
        setObjectiveProperties();
        setVariableProperties();
        setConstraintProperties();
    }

    // Objective and constraint evaluation function
    // overrides KTRProblem class
    double evaluateFC(const std::vector<double>& x,
        std::vector<double>& c,
        std::vector<double>& objGrad,
        std::vector<double>& jac) {

        // constraints
        c[0] = 8.0e0*x[0] + 14.0e0*x[1] + 7.0e0*x[2] - 56.0e0;
        c[1] = x[0] * x[0] + x[1] * x[1] + x[2] * x[2] - 25.0e0;

        // return objective function value
        return 1000 - x[0] * x[0] - 2.0e0*x[1] * x[1] - x[2] * x[2]
            - x[0] * x[1] - x[0] * x[2];
    }

    // Gradient and Jacobian evaluation function
    // overrides KTRProblem class
    int evaluateGA(const std::vector<double>& x,
        std::vector<double>& objGrad,
        std::vector<double>& jac) override {

        objGrad[0] = -2.0e0*x[0] - x[1] - x[2];
        objGrad[1] = -4.0e0*x[1] - x[0];
        objGrad[2] = -2.0e0*x[2] - x[0];

        // gradient of the first constraint, c[0].
        jac[0] = 8.0e0;
        jac[1] = 14.0e0;
        jac[2] = 7.0e0;

        // gradient of the second constraint, c[1]. */
        jac[3] = 2.0e0*x[0];
        jac[4] = 2.0e0*x[1];
        jac[5] = 2.0e0*x[2];
        return 0;
    }
};

int main(int argc, char *argv[]) {
    // Create a problem instance.
    ProblemExample problem = ProblemExample();

    // Create a solver - optional arguments:
    // exact first derivatives
    // BFGS approximate second derivatives
    KNITRO::KTRSolver solver(&instance, KTR_GRADOPT_EXACT, KTR_HESSOPT_BFGS);

```

```

int solveStatus = solver.solve();

if (solveStatus != 0) {
    std::cout << std::endl;
    std::cout << "KNITRO failed to solve the problem, final status = ";
    std::cout << solveStatus << std::endl;
}
else {
    std::cout << std::endl << "KNITRO successful, objective is = ";
    std::cout << solver.getObj() << std::endl;
}

return 0;
}

```

Two changes were made to the previous example. This adds `evaluateGA()` function to the problem class, defining the derivatives, and the `KTRSolver` constructor is passed `KTR_GRADOPT_EXACT` instead of `KTR_GRADOPT_FORWARD`, since the exact gradient function is now defined. Running this example produces the same output as the callable library example.

2.3.8 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) and second derivatives (i.e. the Hessian matrix).

2.3.9 Derivative Check Options

Option	Meaning
<code>derivcheck</code>	Specifies whether or not to enable the derivative checker, and which derivatives to check (first, second or both)
<code>derivcheck_terminate</code>	Whether to terminate after the derivative check or continue to the optimization if successful
<code>derivcheck_tol</code>	Specifies the relative tolerance used for identifying potential errors in the derivatives
<code>derivcheck_type</code>	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` (to check the first derivatives) and/or `hessopt=1` (to check the second derivatives/Hessian). You must also supply callback routines that compute the objective and constraint functions and analytic first derivatives (to check the first derivatives), and/or analytic second derivatives/Hessian (to check the second derivatives). By default, the derivative checker is turned off. To check first derivatives only, simply set `derivcheck = 1`; to check second derivatives/Hessian only set `derivcheck = 2`; and to check both first and second derivatives set `derivcheck = 3`. 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. Setting `derivcheck_terminate` will determine whether Knitro always stops after the derivative check is completed, or continues with the optimization (when the derivative check is successful).

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 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, with value `derivcheck = 1`, and the relative differences between all the user-supplied first derivatives and finite-difference first 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 'x' 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 'x' 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.

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 `KN_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.

The option `par_mnumthreads` can be used to set the number of threads used by the multistart procedure. For instance, if `par_numthreads = 16` and `par_mnumthreads = 8`, Knitro will run 8 solves in parallel and each solve will be allocated 2 threads.

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
<i>ms_deterministic</i>	Control whether to use deterministic multistart
<i>ms_enable</i>	Enable multistart
<i>ms_maxbndrange</i>	Maximum unbounded variable range for multistart
<i>ms_maxsolves</i>	Maximum Knitro solves for multistart
<i>ms_maxtime_cpu</i>	Maximum CPU time for multistart, in seconds
<i>ms_maxtime_real</i>	Maximum real time for multistart, in seconds
<i>ms_num_to_save</i>	Feasible points to save from multistart
<i>ms_outsub</i>	Can write each solve to a file (parallel only)
<i>ms_savetol</i>	Tol for feasible points being equal
<i>ms_seed</i>	Initial seed for generating random start points
<i>ms_startptrange</i>	Maximum variable range for multistart
<i>ms_terminate</i>	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 KN_set_ms_process_callback (KN_context_ptr      kc,
                                          KN_user_callback * const fnPtr,
                                          void                * const userParams);
int KNITRO_API KN_set_ms_initpt_callback (KN_context_ptr      kc,
                                          KN_ms_initpt_callback * const fnPtr,
                                          void                * const userParams);
```

The first callback can be used to perform some user task after each multistart solve and is set by calling `KN_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 `KN_set_ms_initpt_callback()`.

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

In the object-oriented interface, the following functions are used to set the callbacks:

```
void KTRProblem::setMSProcessCallback (KTRMSProcessCallback* MSProcessCallback);
void KTRProblem::setMSInitPtCallback (KTRMSInitPtCallback* MSInitPtCallback);
```

See the *Object-oriented interface reference* section for details on setting these callback functions in the object-oriented interface.

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:

```
1  ampl: reset;
2  ampl: option solver knitroampl;
3  ampl: option knitro_options "ms_enable=1 ms_num_to_save=5 ms_savetol=0.01";
4  ampl: model testproblem.mod;
5  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                 = 415
# of CG iterations               = 90
# of function evaluations       = 545
# of gradient evaluations       = 475
# of Hessian evaluations        = 422
```

```
Total program time (secs)          =          0.02660 (          0.027 CPU time)
```

```
=====
Knitro 11.0.0: Locally optimal or satisfactory solution.
objective 935.9999997; feasibility error 1.44e-07
415 iterations; 545 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 11.0.0 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 similar 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 `KN_solve()`:

```
// multistart
if (KN_set_int_param_by_name (kc, "ms_enable", 1) != 0)
  exit( -1 );
if (KN_set_int_param_by_name (kc, "ms_num_to_save", 5) != 0)
  exit( -1 );
if (KN_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.

2.4.9 Object-oriented example

The object-oriented example can be modified to enable multistart by adding the following lines before the call to `solver.solve()`:

```
// multistart
solver.setParam("ms_enable", 1);
solver.setParam("ms_num_to_save", 5);
solver.setParam("ms_savetol", 0.01);
```

Again, running this example we get a Knitro log that looks similar to the trace obtained 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 three algorithms for mixed-integer nonlinear programming (MINLP). The first is a nonlinear branch and bound method, the second implements the hybrid Quesada-Grossman method for convex MINLP, and the third implements a mixed-integer Sequential Quadratic Programming (MISQP) method that is able to handle non-relaxable integer variables.

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 Overview

The table below presents a brief overview of the main features included in the three MINLP algorithms. For a more detailed description, check [Algorithms/Methods](#).

Features	Branch-and-Bound	Quesada-Grossmann	Mixed Integer Sequential Quadratic Programming
Non-convex MINLP	++	+	++
Convex MNILP		++	
Expensive evaluations			++
Warm-start	+	+	++
MIP heuristics	Rounding / Feasibility pump / MPEC	Rounding / Feasibility pump / MPEC	-
MIP cutting planes	Knapsack	Knapsack	-
LP solver	-	IP/Direct or IP/CG or SLQP	-

- Non-convex MINLP: performance on non-convex MINLP
- Convex MINLP: performance on convex MINLP
- Expensive evaluations: performance on problems with expensive function evaluations
- Warm-start: ability to warm-start
- MIP heuristics: heuristic search approach available to find an initial integer feasible point
- MIP cutting planes: cutting plane methods available
- LP solver: solver available for the resolution of the linear subproblems

2.5.2 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, from [Getting started with AMPL](#) 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 License
      Artelys Knitro 11.0.0
=====

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
  quadratic equalities:              0
  gen. nonlinear equalities:         0
  linear one-sided inequalities:     0
  quadratic one-sided inequalities:  1
  gen. nonlinear one-sided inequalities: 0
  linear two-sided inequalities:     0
  quadratic two-sided inequalities:  0
  gen. nonlinear two-sided inequalities: 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                = 9.360000000000000e+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 11.0.0: Locally optimal or satisfactory 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.

2.5.3 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. Uncertain, convex, and nonconvex are set with 0, 1, and 2, respectively. Passing an empty array, [], is equivalent to passing zero.

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. Uncertain, convex, and nonconvex inequality constraints are set with 0, 1, and 2, respectively. Passing an empty array, [], is equivalent to passing an array of all zeros.

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);
```

2.5.4 C example

As with AMPL, defining a MIP problem only requires declaring integer variables via the API function *KN_set_var_types()* (by default, variables are assumed to be continuous).

In order to turn our C toy example into a MINLP problem, it thus suffices to add

```

/* in the declarations */
int i;

/* mark all variables as integer */
for (i=0; i<n; i++) {
    error = KN_set_var_type (kc, i, KN_VARTYPE_INTEGER);
}

```

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.

2.5.5 Object-oriented C++ example

A MIP problem is defined and solved via the object-oriented interface by adding additional problem information in the problem class.

In the following, we will define how to turn the toy example into a MINLP problem. The `ProblemExample` class has to be extended with new definitions.

In the function `setObjectiveProperties()`, the function `KTRProblem::setObjFnType(int fnType)` is used to define the objective function type:

```
setObjFnType (KNITRO::KTREnums::FunctionType::Convex);
```

In the function `setConstraintProperties()`, the constraint function types are defined with the function `KTRProblem::setConFnTypes(int id,int fnType)`:

```
setConFnTypes(0, KNITRO::KTREnums::FunctionType::Convex);
setConFnTypes(1, KNITRO::KTREnums::FunctionType::Nonconvex);
```

In the function `setVariableProperties()`, the variable types are defined with the function `KTRProblem::setVarFnTypes(int fnTypes)`:

```
setVarFnTypes (KNITRO::KTREnums::VariableType::Integer);
```

Without specifying a variable index, the function sets variable types for all variables to integer.

This example uses the `KTRProblem` class to simplify implementing `KTRIPProblem`. If using `KTRIPProblem` only, the functions `KTRIPProblem::getObjFnType`, `KTRIPProblem::getConFnType`, and `KTRIPProblem::getVarFnType` should be implemented to return the appropriate values.

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.

2.5.6 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
<code>mip_branchrule</code>	MIP branching rule
<code>mip_debug</code>	MIP debugging level (0=none, 1=all)
Continued on next page	

Table 2.1 – continued from previous page

Name	Meaning
<i>mip_gub_branch</i>	Branch on GUBs (0=no, 1=yes)
<i>mip_heuristic</i>	MIP heuristic search
<i>mip_heuristic_maxit</i>	MIP heuristic iteration limit
<i>mip_heuristic_terminate</i>	MIP heuristic termination condition
<i>mip_implications</i>	Add logical implications (0=no, 1=yes)
<i>mip_integer_tol</i>	Threshold for deciding integrality
<i>mip_integral_gap_abs</i>	Absolute integrality gap stop tolerance
<i>mip_integral_gap_rel</i>	Relative integrality gap stop tolerance
<i>mip_intvar_strategy</i>	Treatment of integer variables
<i>mip_knapsack</i>	Add knapsack cuts (0=no, 1=ineqs, 2=ineqs+eqs)
<i>mip_lpalg</i>	LP subproblem algorithm
<i>mip_maxnodes</i>	Maximum nodes explored
<i>mip_maxsolves</i>	Maximum subproblem solves
<i>mip_maxtime_cpu</i>	Maximum CPU time in seconds for MIP
<i>mip_maxtime_real</i>	Maximum real in seconds time for MIP
<i>mip_method</i>	MIP method (0=auto, 1=BB, 2=HQG, 3=MISQP)
<i>mip_nodealg</i>	Standard node relaxation algorithm
<i>mip_outinterval</i>	MIP output interval
<i>mip_outlevel</i>	MIP output level
<i>mip_outsub</i>	Enable MIP subproblem output
<i>mip_pseudoinit</i>	Pseudo-cost initialization
<i>mip_relaxable</i>	Are integer variables relaxable?
<i>mip_rootalg</i>	Root node relaxation algorithm
<i>mip_rounding</i>	MIP rounding rule
<i>mip_selectdir</i>	MIP node selection direction
<i>mip_selectrule</i>	MIP node selection rule
<i>mip_strong_candlim</i>	Strong branching candidate limit
<i>mip_strong_level</i>	Strong branching tree level limit
<i>mip_strong_maxit</i>	Strong branching iteration limit
<i>mip_terminate</i>	Termination condition for MIP

2.5.7 Algorithms/Methods

The default MINLP method in Knitro is a standard implementation of branch-and-bound for nonlinear optimization. This method involves solving a relaxed, continuous nonlinear optimization subproblem at every node of the branch-and-bounds tree. This method is generally the preferred method. It is primarily designed for convex models, and in this case the integrality gap measure can be trusted. It can also be applied to non-convex models, and often works well on these models. However it may sometimes get stuck at integer feasible points that are not globally optimal solutions when the model is nonconvex. In addition, the integrality gap measure may not be accurate since this measure is based on the assumption that the nonlinear optimization subproblems are always solved to global optimality (which may not be the case when the model is nonconvex).

The hybrid Quesada-Grossman (HQG) method in Knitro is a variant of branch-and-bound for MINLP. It maintains one branch-and-bound tree but solves linear programming (LP) subproblems at most of the nodes, while only occasionally solving nonlinear optimization subproblems at integer feasible nodes. The solutions of the LP subproblems are used to generate outer approximations/cuts, which are continually added to the master problem. This method should generally only be applied to convex models since the outer approximations are only valid when the model is convex. This method will typically take many more nodes to solve compared with the standard branch-and-bound method, but the node subproblems are often easier to solve since most of them are LPs.

The third method (MISQP) is a largely heuristic method that attempts to extend the SQP method for continuous,

nonlinear optimization to the case where there are integer variables. This method is primarily designed for small problems (e.g. less than 100 variables) where function evaluations may involve expensive black-box simulations and derivatives may not be available. In contrast to the other MINLP algorithms in Knitro, this method is able to handle models where the integer variables cannot be relaxed. This means that the simulations or function evaluations can only occur when integer variables are at integer values (e.g. the integer variables may have no meaning at non-integral values). This method will typically converge in far fewer function evaluations compared with the other MINLP methods in Knitro and is primarily intended for small problems where these evaluations are the dominant cost. This method can be applied to either convex or non-convex models, but may converge to non-global integer, feasible points. However, since this algorithm runs similarly to the continuous SQP algorithm, you can apply the parallel multi-start feature (see Section *Multistart*) to the MISQP method to increase the chances of finding the global solution.

2.5.8 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 using the AMPL suffixes feature (see *AMPL suffixes defined for Knitro*) 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 `KN_set_set_mip_branching_priorities()` functions shown below.

```
int KNITRO_API KN_set_mip_branching_priorities
(
    KN_context_ptr kc,
    const KNINT nV,
    const KNINT * const indexVars,
    const int * const xPriorities);
int KNITRO_API KN_set_mip_branching_priorities_all
(
    KN_context_ptr kc,
    const int * const xPriorities);
int KNITRO_API KN_set_mip_branching_priority
(
    KN_context_ptr kc,
    const KNINT indexVar,
    const int xPriority);
```

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 `KN_set_var_types()` to mark integer variables.

Branching priorities in the object-oriented interface

Branching priorities for integer variables can be specified through the object-oriented interface using the function shown below.

```
void KTRSolver::mipSetBranchingPriorities(const std::vector<int>& xPriorities);
```

The `std::vector<int> xPriorities` has length “ n ”, where n is the number of variables. Values for continuous variables are ignored. This method must be called after calling the `KTRSolver` constructor and before calling `KTRSolver::solve()`.

2.5.9 Special Treatment of Integer Variables

You can specify special treatment of integer variables using the `mip_intvar_strategy` user option in Knitro. In particular, you can use this option to specify that all integer variables are relaxed, or that all binary variables should be converted to complementarity constraints (see Section [Complementarity constraints](#) for a description of complementarity constraints).

In addition you can specify special treatments of individual integer variables through the callable library interface function `KN_set_mip_intvar_strategies()`

```
int KNITRO_API KN_set_mip_intvar_strategies
(
    KN_context_ptr   kc,
    const KNINT      nV,
    const KNINT *    const   indexVars,
    const int *      const   xStrategies);
int KNITRO_API KN_set_mip_intvar_strategies_all
(
    KN_context_ptr   kc,
    const int *      const   xStrategies);
int KNITRO_API KN_set_mip_intvar_strategy
(
    KN_context_ptr   kc,
    const KNINT      indexVar,
    const int        xStrategy);
```

Here `indexVars` specifies the index of the integer variable you want to apply the special treatment to, and `xStrategies` specifies how you want to handle that particular integer variable (e.g., no special treatment, relax, or convert to a complementarity constraint).

Special strategies for integer variables can be specified in the AMPL interface using the `intvarstrategy` AMPL suffix, and in the MATLAB interface using the `extendedFeatures.xIntStrategy` structure.

2.5.10 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 KN_set_mip_node_callback (KN_context_ptr      kc,
                                         KN_user_callback * const fnPtr,
                                         void *             * const userParams);
```

See the [Callable library API reference](#) section in the Reference Manual for details on setting this callback function and the prototype for this callback function.

2.5.11 Determining convexity/concavity

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 API allows for the user to specify whether or not the problem functions (objective and constraints) are convex or concave via the functions: `KN_set_obj_property()` and `KN_set_con_properties()`.

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

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

and concave if

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

By default functions are assumed to be nonconvex (i.e. neither convex, nor concave). The objective function and any constraint functions that satisfy the conditions above should be marked as convex or concave. All linear functions are convex. Any nonlinear equality constraint is nonconvex. Knitro will use function convexity/concavity information to determine whether the optimization problem as a whole is convex or not. If the problem is determined to be convex Knitro may be able to apply specializations to improve performance. If you know your model is convex, you can also directly tell Knitro this by setting the `convex` option.

The MIP solvers in Knitro are designed for convex problems. For nonconvex problems, 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.

2.5.12 Additional examples

Examples for solving MINLP problems using the MATLAB, C, C++, Java, C#, Python and R 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 types 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.

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

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:

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

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 *KN_set_compcons()* which has the following prototype:

```
int KNITRO_API KN_set_compcons (      KN_context_ptr  kc,
                                     const KNINT      nCC,
                                     const int         * const ccTypes,
                                     const KNINT      * const indexComps1,
                                     const KNINT      * const indexComps2);
```

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

- *nCC*, the number of complementarity constraints to be added to the problem (i.e., the number of pairs of variables that are complementary to each other).
- *ccTypes*, array of length *nCC* specifying the type for each complementarity constraint. Currently this MUST be set to `KN_CCTYPE_VARVAR` since Knitro currently only supports complementarity constraints between two (non-negative) variables. However, this parameter will be used in the future to support more general types of complementarities (such as complementarities between a variable and a constraint).
- *indexComps1* and *indexComps2*, two arrays of length *nCC* specifying the variable indices for the first and second sets of variables in the pairs of complementary variables.

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

Note: `KN_set_compcons()` can only be called once to load all complementarity constraints in the problem at one time.

2.6.5 Complementarity constraints with the object-oriented interface

Complementarity constraints can be specified in the object-oriented interface by defining the constraints in a class inheriting from `KTRIPProblem`.

The `KTRIPProblem` should implement the functions:

```
std::vector<int> complementarityIndexList1();
std::vector<int> complementarityIndexList2();
```

to return the lists of complementary variables. Parameter `indexList1` and `indexList2`, of the same length, specifying the variable indices for the first and second sets of variables in the pairs of complementary variables.

When using the `KTRProblem` class, the values can be passed to the function:

```
KTRIPProblem::setComplementarity(const std::vector<int>& indexList1,
                                const std::vector<int>& indexList2)
```

to set the values returned by the `complementarityIndexList` functions.

Note: Variables which are specified as complementary through `KTRIPProblem::setComplementarity()` functions should have a lower bound of 0. This can be set using `KTRProblem::setVarLoBnds()`.

2.6.6 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 License
      Artelys Knitro 11.0.0
=====

No start point provided -- Knitro computing one.

Knitro presolve eliminated 0 variables and 0 constraints.

datacheck:          0
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_initpt from AUTO to 3.
Knitro changing bar_murule from AUTO to 4.
Knitro changing bar_penaltycons from AUTO to 1.
Knitro changing bar_penaltyrule from AUTO to 2.
Knitro changing bar_switchrule from AUTO to 2.
Knitro changing linesearch from AUTO to 1.
Knitro changing linsolver from AUTO to 2.

Problem Characteristics ( Presolved)
-----
Objective goal: Minimize
Objective type: quadratic
Number of variables:          11 (          11)
    bounded below only:      11 (          11)
    bounded above only:       0 (           0)
    bounded below and above:  0 (           0)
    fixed:                    0 (           0)
    free:                     0 (           0)
Number of constraints:        7 (           7)
    linear equalities:        7 (           7)
    quadratic equalities:     0 (           0)
    gen. nonlinear equalities: 0 (           0)
    linear one-sided inequalities: 0 (           0)
    quadratic one-sided inequalities: 0 (           0)
    gen. nonlinear one-sided inequalities: 0 (           0)
    linear two-sided inequalities: 0 (           0)
    quadratic two-sided inequalities: 0 (           0)
    gen. nonlinear two-sided inequalities: 0 (           0)
Number of complementarities:  3 (           3)
Number of nonzeros in Jacobian: 20 (          20)

```

```

Number of nonzeros in Hessian:                2 (          2)

  Iter      Objective      FeasError      OptError      ||Step||      CGits
-----
    0      2.811162e+01      1.548e+00
   10      1.700000e+01      6.178e-10      4.001e-07      3.202e-05      0

EXIT: Locally optimal solution found.

Final Statistics
-----
Final objective value                = 1.70000000199027e+01
Final feasibility error (abs / rel)   = 6.18e-10 / 3.99e-10
Final optimality error (abs / rel)   = 4.00e-07 / 5.00e-08
# of iterations                      = 10
# of CG iterations                   = 1
# of function evaluations             = 0
# of gradient evaluations            = 0
# of Hessian evaluations             = 0
Total program time (secs)            = 0.00268 ( 0.002 CPU time)
Time spent in evaluations (secs)     = 0.00000

=====

Knitro 11.0.0: Locally optimal or satisfactory solution.
objective 17.000000019902657; feasibility error 6.18e-10
10 iterations; 0 function evaluations

```

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

2.6.7 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', '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);

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);

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

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 License
      Artelys Knitro 11.0.0
=====

Knitro presolve eliminated 0 variables and 0 constraints.

algorithm:          1
feastol:            1e-08
honorbnds:          1
maxit:              1000
opttol:             1e-08
outlev:             4
par_concurrent_evals: 0
The problem is identified as an MPEC.
Knitro changing bar_initpt from AUTO to 3.
Knitro changing bar_murule from AUTO to 4.
Knitro changing bar_penaltycons from AUTO to 1.

```

Knitro changing bar_penaltyrule from AUTO to 2.
 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                ( Presolved)
-----
Objective goal:  Minimize
Objective type:  general
Number of variables:                8 (          8)
    bounded below only:              8 (          8)
    bounded above only:              0 (          0)
    bounded below and above:         0 (          0)
    fixed:                            0 (          0)
    free:                             0 (          0)
Number of constraints:              4 (          4)
    linear equalities:                4 (          4)
    quadratic equalities:             0 (          0)
    gen. nonlinear equalities:        0 (          0)
    linear one-sided inequalities:    0 (          0)
    quadratic one-sided inequalities: 0 (          0)
    gen. nonlinear one-sided inequalities: 0 (          0)
    linear two-sided inequalities:    0 (          0)
    quadratic two-sided inequalities: 0 (          0)
    gen. nonlinear two-sided inequalities: 0 (          0)
Number of complementarities:        3 (          3)
Number of nonzeros in Jacobian:     14 (         14)
Number of nonzeros in Hessian:      2 (          2)
    
```

Iter	fCount	Objective	FeasError	OptError	Step	CGits
0	2	2.496050e+01	4.030e+00			
1	3	2.847389e+01	1.748e+00	2.160e+00	1.990e+00	1
2	4	4.226663e+01	3.832e-01	4.643e+00	1.442e+00	0
3	5	4.667799e+01	1.126e-02	3.638e+00	5.993e-01	0
4	6	4.213217e+01	4.179e-03	1.258e+01	1.185e+00	0
5	7	4.074018e+01	3.072e-03	1.265e+01	1.580e-01	1
6	8	3.810894e+01	1.133e-04	1.259e+01	3.113e-01	0
7	9	1.701407e+01	1.682e-04	1.542e+00	4.771e+00	0
8	10	1.699966e+01	1.522e-04	6.416e-02	2.385e-02	0
9	11	1.700003e+01	1.799e-06	3.532e-05	2.154e-04	0
10	12	1.700000e+01	6.354e-11	1.530e-09	5.298e-05	0

EXIT: Locally optimal solution found.

Final Statistics

```

-----
Final objective value                = 1.70000000010379e+01
Final feasibility error (abs / rel) = 6.35e-11 / 1.58e-11
Final optimality error (abs / rel) = 1.53e-09 / 1.91e-10
# of iterations                      = 10
# of CG iterations                   = 2
# of function evaluations            = 12
# of gradient evaluations            = 12
# of Hessian evaluations             = 10
Total program time (secs)           = 0.00827 ( 0.019 CPU time)
Time spent in evaluations (secs)    = 0.00464
    
```

2.6.8 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 *KN_set_compcons()* function is called to register the list.

```

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

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

    /** Declare variables. */
    KN_context *kc;
    int n, m;
    double xLoBnds[8] = {0, 0, 0, 0, 0, 0, 0, 0};
    double xInitVals[8] = {0, 0, 0, 0, 0, 0, 0, 0};
    double cEqBnds[4] = {2, 3, -4, -7};
    /** Used to define linear constraints. */
    int lconIndexCons[14];
    int lconIndexVars[14];
    double lconCoefs[14];
    /** Used to specify linear objective terms. */
    int lobjIndexVars[2];
    double lobjCoefs[2];
    /** Used to specify quadratic objective terms. */
    int qobjIndexVars1[2];
    int qobjIndexVars2[2];
    double qobjCoefs[2];
    /** Used to specify complementarity constraints. */
    int ccTypes[3] = {KN_CCTYPE_VARVAR, KN_CCTYPE_VARVAR, KN_CCTYPE_VARVAR};
    int indexComps1[3] = {2, 3, 4};
    int indexComps2[3] = {5, 6, 7};
    /** Solution information */
    double x[8];
    double objSol;
    double feasError, optError;

    /** Create a new Knitro solver instance. */
    error = KN_new(&kc);
    if (error) exit(-1);
    if (kc == NULL)
    {
        printf ("Failed to find a valid license.\n");
        return( -1 );
    }

    /** Initialize Knitro with the problem definition. */

    /** Add the variables and set their bounds and initial values.
     * Note: unset bounds assumed to be infinite. */
    n = 8;
    error = KN_add_vars(kc, n, NULL);

```

```

    if (error) exit(-1);
    error = KN_set_var_lobnds_all(kc, xLoBnds);
    if (error) exit(-1);
    error = KN_set_var_primal_init_values_all(kc, xInitVals);
    if (error) exit(-1);

    /** Add the constraints and set their bounds. */
    m = 4;
    error = KN_add_cons(kc, m, NULL);
    if (error) exit(-1);
    error = KN_set_con_eqbnds_all(kc, cEqBnds);
    if (error) exit(-1);

    /** Add coefficients for all linear constraints at once. */

    /** c0 */
    lconIndexCons[0]=0; lconIndexVars[0]=0; lconCoefs[0]=-1.5;
    lconIndexCons[1]=0; lconIndexVars[1]=1; lconCoefs[1]=2.0;
    lconIndexCons[2]=0; lconIndexVars[2]=2; lconCoefs[2]=1.0;
    lconIndexCons[3]=0; lconIndexVars[3]=3; lconCoefs[3]=-0.5;
    lconIndexCons[4]=0; lconIndexVars[4]=4; lconCoefs[4]=1.0;

    /** c1 */
    lconIndexCons[5]=1; lconIndexVars[5]=0; lconCoefs[5]=3.0;
    lconIndexCons[6]=1; lconIndexVars[6]=1; lconCoefs[6]=-1.0;
    lconIndexCons[7]=1; lconIndexVars[7]=5; lconCoefs[7]=-1.0;

    /** c2 */
    lconIndexCons[8]=2; lconIndexVars[8]=0; lconCoefs[8]=-1.0;
    lconIndexCons[9]=2; lconIndexVars[9]=1; lconCoefs[9]=0.5;
    lconIndexCons[10]=2; lconIndexVars[10]=6; lconCoefs[10]=-1.0;

    /** c3 */
    lconIndexCons[11]=3; lconIndexVars[11]=0; lconCoefs[11]=-1.0;
    lconIndexCons[12]=3; lconIndexVars[12]=1; lconCoefs[12]=-1.0;
    lconIndexCons[13]=3; lconIndexVars[13]=7; lconCoefs[13]=-1.0;

    error = KN_add_con_linear_struct (kc, 14, lconIndexCons, lconIndexVars,
                                     lconCoefs);
    if (error) exit(-1);

    /** Note that the objective (x0 - 5)^2 + (2 x1 + 1)^2 when
     * expanded becomes:
     *      x0^2 + 4 x1^2 - 10 x0 + 4 x1 + 26 */

    /** Add quadratic coefficients for the objective */
    qobjIndexVars1[0]=0; qobjIndexVars2[0]=0; qobjCoefs[0]=1.0;
    qobjIndexVars1[1]=1; qobjIndexVars2[1]=1; qobjCoefs[1]=4.0;
    error = KN_add_obj_quadratic_struct (kc, 2, qobjIndexVars1,
                                         qobjIndexVars2, qobjCoefs);
    if (error) exit(-1);

    /** Add linear coefficients for the objective */
    lobjIndexVars[0]=0; lobjCoefs[0]=-10.0;
    lobjIndexVars[1]=1; lobjCoefs[1]=4.0;
    error = KN_add_obj_linear_struct (kc, 2,
                                      lobjIndexVars, lobjCoefs);
    if (error) exit(-1);

```

```

/** Add constant to the objective */
error = KN_add_obj_constant (kc, 26.0);
if (error) exit(-1);

/** Set minimize or maximize (if not set, assumed minimize) */
error = KN_set_obj_goal(kc, KN_OBJGOAL_MINIMIZE);
if (error) exit(-1);

/** Now add the complementarity constraints */
error = KN_set_compcons (kc, 3, ccTypes, indexComps1, indexComps2);
if (error) exit(-1);

/** Solve the problem.
 *
 * Return status codes are defined in "knitro.h" and described
 * in the Knitro manual. */
nStatus = KN_solve (kc);

/** Delete the Knitro solver instance. */
KN_free (&kc);

return( 0 );
}

```

Running this code produces an output similar to what we obtained with AMPL.

2.7 Nonlinear Least-Squares

Knitro provides a specialized API for nonlinear least-squares models of the following form:

$$\min_p \frac{1}{2} \|F(p)\|_2^2$$

s.t. $p_L \leq p \leq p_U$,

where p is a parameter to be optimized and F is a differentiable function, which is called a residual. This type of problem appears very often in statistics, data-mining and machine learning. Using the nonlinear least-squares API, you are able to model a nonlinear least-squares problem in standard form above and use the Gauss-Newton Hessian option.

The Gauss-Newton Hessian provides a positive semi-definite Hessian approximation $J(p)'J(p)$ (where $J(p)$ is the Jacobian matrix of the residual functions $F(p)$) at every iteration and has good local convergence properties in practice. The Gauss-Newton Hessian option `KN_HESSOPT_GAUSS_NEWTON`, is the default Hessian option when using the nonlinear least-squares API. The quasi-Newton Hessian options are also available through the least-squares API, however, the user-supplied exact Hessian can only be specified using the standard API.

Any of the Knitro algorithms can be used through the least-squares API. Knitro will behave like a Gauss-Newton method by using the linesearch methods `algorithm = KN_ALG_BAR_DIRECT` or `KN_ALG_ACT_SQP`, and will be very similar to the classical Levenberg-Marquardt method when using the trust-region methods `algorithm = KN_ALG_BAR_CG` or `KN_ALG_ACT_CG`.

Residuals are added to a least-squares model using the `KN_add_rsds()`. The coefficients and sparsity structure for linear residuals (or linear terms inside nonlinear residuals) can be provided to Knitro through the API function `KN_add_rsd_linear_struct()`. Constants can be added to residuals through `KN_add_rsd_constants()`. The nonlinear residuals and Jacobian are provided to Knitro using the callback

functions `KN_add_lsq_eval_callback()` and `KN_set_cb_rsd_jac()` 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. If a callback function to evaluate the residual Jacobian is not provided, Knitro will approximate it using finite-differences. Please see *Callable library API reference* for more details on these API functions.

```

/** Set the callback function to evaluate the residuals "res" of a
 * nonlinear least-squares problem.
 * Do not modify "jac" in this function.
 */
int KNITRO_API KTR_lsq_set_res_callback(KTR_context_ptr kc,
                                       KTR_lsq_callback * const fnPtr);

/** Add an evaluation callback for a least-squares models. Similar to KN_add_eval_
↳callback()
 * but for least-squares models.
 *
 * nR          - number of residuals evaluated in the callback
 * indexRsds   - (length nR) index of residuals evaluated in the callback
 * rsdCallback - a pointer to a function that evaluates any residual parts
 *              (specified by nR and indexRsds) involved in this callback
 * cb          - (output) the callback structure that gets created by
 *              calling this function; all the memory for this structure is
 *              handled by Knitro
 *
 * After a callback is created by "KN_add_lsq_eval_callback()", the user can then
 * specify residual Jacobian information and structure through "KN_set_cb_rsd_jac()".
 * If not set, Knitro will approximate the residual Jacobian. However, it is highly
 * recommended to provide a callback routine to specify the residual Jacobian if at_
↳all
 * possible as this will greatly improve the performance of Knitro. Even if a_
↳callback
 * for the residual Jacobian is not provided, it is still helpful to provide the_
↳sparse
 * Jacobian structure for the residuals through "KN_set_cb_rsd_jac()" to improve the
 * efficiency of the finite-difference Jacobian approximation. Other optional
 * information can also be set via "KN_set_cb_*( )" functions as detailed below.
 *
 * Returns 0 if OK, nonzero if error.
 */
int KNITRO_API KN_add_lsq_eval_callback (      KN_context_ptr      kc,
                                              const KNINT        nR,
                                              const KNINT        * const indexRsds,
                                              KN_eval_callback * const rsdCallback,
                                              CB_context_ptr     * const cb);

/** This API function is used to set the residual Jacobian structure and also
 * (optionally) a callback function to evaluate the residual Jacobian provided
 * through this callback.
 *
 * cb          - a callback structure created from a previous call to
 *              KN_add_lsq_eval_callback()
 * nnzJ        - number of nonzeros in the sparse residual Jacobian
 *              computed through this callback; set to KN_DENSE_ROWMAJOR to
 *              provide the full Jacobian in row major order (i.e. ordered
 *              by rows/residuals), or KN_DENSE_COLMAJOR to provide the full
 *              Jacobian in column major order (i.e. ordered by columns/

```

```

*          variables)
*   jacIndexRsds  - (length nnzJ) residual index (row) of each nonzero;
*                  set to NULL if nnzJ=KN_DENSE_ROWMAJOR/KN_DENSE_COLMAJOR or
↪nnzJ=0
*   jacIndexVars  - (length nnzJ) variable index (column) of each nonzero;
*                  set to NULL if nnzJ=KN_DENSE_ROWMAJOR/KN_DENSE_COLMAJOR or
↪nnzJ=0
*   rsdJacCallback - a pointer to a function that evaluates any residual Jacobian
*                  parts involved in this callback; set to NULL if using a
↪finite-
*                  difference Jacobian approximation (specified via KN_set_cb
↪gradopt())
*
*   The user should generally always try to define the sparsity structure
*   for the Jacobian ("nnzJ", "jacIndexRsds", "jacIndexVars"). Even when
*   using a finite-difference approximation to compute the Jacobian, knowing the
*   sparse structure of the Jacobian can allow Knitro to compute this
*   finite-difference approximation faster. However, if the user is unable to
*   provide this sparsity structure, then one can set "nnzJ" to KN_DENSE_ROWMAJOR or
*   KN_DENSE_COLMAJOR and set "jacIndexRsds" and "jacIndexVars" to NULL.
*/
int KNITRO_API KN_set_cb_rsd_jac (      KN_context_ptr      kc,
                                       CB_context_ptr      cb,
                                       const KNLONG         nnzJ, /* or KN_
↪DENSE_* */
                                       const KNINT          * jacIndexRsds,
                                       const KNINT          * jacIndexVars,
                                       KN_eval_callback *    rsdJacCallback); /
↪* nullable *
    
```

There is currently no callback for the exact Hessian in the least-squares API. If you wish to provide a callback for the user-supplied exact Hessian, you must use the standard API.

After solving, the residuals and residual Jacobian can be retrieved through the API functions `KN_get_rsd_values()` and `KN_get_rsd_jacobian_values()`. See *Callable library API reference* for more details.

2.7.1 C example

The following C example illustrates how to use the Knitro least squares interface.

```

/* A simple nonlinear least-squares problem with 6 residual functions:
*
*   min   ( x0*1.309*x1 - 2.138 )^2 + ( x0*1.471*x1 - 3.421 )^2
*         + ( x0*1.49*x1 - 3.597 )^2 + ( x0*1.565*x1 - 4.34 )^2
*         + ( x0*1.611*x1 - 4.882 )^2 + ( x0*1.68*x1-5.66 )^2
*/

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

#include "knitro.h"

int callbackEvalR (KN_context_ptr      kc,
                  CB_context_ptr      cb,
                  KN_eval_request_ptr  const evalRequest,
    
```

```

        KN_eval_result_ptr  const  evalResult,
        void                * const  userParams)
{
    const double *x;
    double *rsd;

    if (evalRequest->type != KN_RC_EVALR)
    {
        printf ("*** callbackEvalR incorrectly called with eval type %d\n",
                evalRequest->type);
        return( -1 );
    }
    x = evalRequest->x;
    rsd = evalResult->rsd;

    /** Evaluate nonlinear residual components */
    rsd[0] = x[0] * pow(1.309, x[1]);
    rsd[1] = x[0] * pow(1.471, x[1]);
    rsd[2] = x[0] * pow(1.49, x[1]);
    rsd[3] = x[0] * pow(1.565, x[1]);
    rsd[4] = x[0] * pow(1.611, x[1]);
    rsd[5] = x[0] * pow(1.68, x[1]);

    return( 0 );
}

int callbackEvalRJ (KN_context_ptr          kc,
                   CB_context_ptr         cb,
                   KN_eval_request_ptr const  evalRequest,
                   KN_eval_result_ptr const  evalResult,
                   void                * const  userParams)
{
    const double *x;
    double *rsdJac;

    if (evalRequest->type != KN_RC_EVALRJ)
    {
        printf ("*** callbackEvalRJ incorrectly called with eval type %d\n",
                evalRequest->type);
        return( -1 );
    }
    x = evalRequest->x;
    rsdJac = evalResult->rsdJac;

    /** Evaluate non-zero residual Jacobian elements (row major order). */
    rsdJac[0] = pow(1.309, x[1]);
    rsdJac[1] = x[0] * log(1.309) * pow(1.309, x[1]);
    rsdJac[2] = pow(1.471, x[1]);
    rsdJac[3] = x[0] * log(1.471) * pow(1.471, x[1]);
    rsdJac[4] = pow(1.49, x[1]);
    rsdJac[5] = x[0] * log(1.49) * pow(1.49, x[1]);
    rsdJac[6] = pow(1.565, x[1]);
    rsdJac[7] = x[0] * log(1.565) * pow(1.565, x[1]);
    rsdJac[8] = pow(1.611, x[1]);
    rsdJac[9] = x[0] * log(1.611) * pow(1.611, x[1]);
    rsdJac[10] = pow(1.68, x[1]);
    rsdJac[11] = x[0] * log(1.68) * pow(1.68, x[1]);
}

```

```

    return( 0 );
}

int main (int argc, char *argv[])
{
    /** Declare variables. */
    KN_context *kc;
    int i, error;
    int n, m;
    /** Used to set constants for residuals */
    double constants[6] = {-2.138, -3.421, -3.597, -4.34, -4.882, -5.66};
    /** Pointer to structure holding information for evaluation
     * callbacks. */
    CB_context *cb;
    /** Solution information. */
    int nRC, nStatus;
    double x[2];
    double obj;

    /** Create a new Knitro solver instance. */
    error = KN_new(&kc);
    if (error) exit(-1);
    if (kc == NULL)
    {
        printf ("Failed to find a valid license.\n");
        return( -1 );
    }

    /** Add the variables/parameters.
     * Note: Any unset lower bounds are assumed to be
     * unbounded below and any unset upper bounds are
     * assumed to be unbounded above. */
    n = 2; /** # of variables/parameters */
    error = KN_add_vars(kc, n, NULL);
    if (error) exit(-1);

    /** Add the residuals. */
    m = 6; /** # of residuals */
    error = KN_add_rsds(kc, m, NULL);
    if (error) exit(-1);

    /** Set the array of constants in the residuals */
    error = KN_add_rsd_constants_all(kc, constants);
    if (error) exit(-1);

    /** Add a callback function "callbackEvalR" to evaluate the nonlinear
     * residual components. Note that the constant terms are added
     * separately above, and will not be included in the callback. */
    error = KN_add_lsq_eval_callback_all (kc, callbackEvalR, &cb);
    if (error) exit(-1);

    /** Also add a callback function "callbackEvalRJ" to evaluate the
     * Jacobian of the residuals. If not provided, Knitro will approximate
     * the residual Jacobian using finite-differencing. However, we recommend
     * providing callbacks to evaluate the exact Jacobian whenever
     * possible as this can drastically improve the performance of Knitro.
     * We specify the residual Jacobian in "dense" row major form for simplicity.
     * However for models with many sparse residuals, it is important to specify

```

```

    * the non-zero sparsity structure of the residual Jacobian for efficiency
    * (this is true even when using finite-difference gradients). */
error = KN_set_cb_rsd_jac (kc, cb, KN_DENSE_ROWMAJOR, NULL, NULL, callbackEvalRJ);
if (error) exit(-1);

/** Solve the problem.
 *
 * Return status codes are defined in "knitro.h" and described
 * in the Knitro manual.
 */
nRC = KN_solve (kc);

/** Delete the knitro solver instance. */
KN_free (&kc);

return( 0 );
}

```

2.8 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.

2.8.1 Overview

The table below presents a brief overview of the main features included in the four NLP algorithms.

Features	Interior-Point/Direct	Interior-Point/Conjugate-Gradient	Sequential Linear Quadratic Programming	Sequential Quadratic Programming
Large scale	++ (sparse)	++ (sparse or dense)	+	
Expensive evaluations			+	++
Warm-start	+	+	++	++
Least square problems	++	++	+	+
Globalization technique	Line-search/Trust-region	Trust-region	Trust-region	Line-search/Trust-region
Linear solver	Lapack QR, MA27/57/86/97, MKL PARDISO	Lapack QR, MA27/57/86/97, MKL PARDISO	Lapack QR, MA27/57/86/97, MKL PARDISO	Lapack QR, MA27/57/86/97, MKL PARDISO
LP solver	-	-	Clp (incl.) or Xpress/Cplex (not incl.)	Clp (incl.) or Xpress/Cplex (not incl.)
QP solver	-	-	-	IP/Direct or IP/CG or SLQP

- Large scale: ability to solve large scale problems
- Expensive evaluations: performance on problems with expensive function evaluations
- Warm-start: ability to warm-start
- Least square problems: performance on least square problems
- Globalization technique: method used to improve the likelihood of convergence from any initial point. This is not related to finding a global optima of the optimized function.
- Linear solver: solvers available for the resolution of internal linear systems
- LP solver: solvers available for the resolution of linear subproblems
- QP solver: solvers available for the resolution of quadratic subproblems

2.8.2 Algorithms description

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. An incomplete Cholesky preconditioner can be computed and applied during the conjugate gradient iterations for problems with equality and inequality constraints. This generally results in improved performances in terms of number of conjugate gradient iterations and CPU time.

- 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. A third method (MISQP) extends the SQP method for continuous, nonlinear optimization to the case where there are integer variables.

2.8.3 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.

2.8.4 Multiple algorithms

Setting user option `algorithm=5` (KN_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 solutions 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`.

2.8.5 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.

2.9 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 certain points.

2.9.1 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.

2.9.2 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 to 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.

2.9.3 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`.

In addition, the API function `KN_set_var_honorbnds()` can be used to set this option individually for each variable (as opposed to the global `honorbnds` option which applies to all variables). The settings through this API function will override the setting through the global `honorbnds` user option.

2.10 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, the object-oriented interfaces, and through the callable library. Parallel features are also available through the MATLAB interface, but some may be less efficient in this environment.

Knitro offers the following parallel features:

2.10.1 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.

2.10.2 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.

The option `par_msnumthreads` can be used to set the number of threads used by the multistart procedure. For instance, if `par_numthreads = 16` and `par_msnumthreads = 8`, Knitro will run 8 solves in parallel and each solve will be allocated 2 threads.

2.10.3 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.

2.10.4 Parallel Tuning

The Knitro-Tuner can help you identify some non-default options settings that may improve performance on a particular model or set of models. When `par_numthreads` is set to use multiple threads, Knitro will test the Tuner options in parallel.

2.10.5 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.

2.10.6 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_lnumthreads > 1` (in combination with `linsolver = 6`). It is also possible to use `par_lnumthreads > 1` with the linear solvers MA86 and MA97.

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_lnumthreads = 1` and vice versa.

2.10.7 Parallel Options

Option	Meaning
<code>par_numthreads</code>	Specifies the max number of threads to use for all parallel features. You can just set this and let Knitro decide how to distribute the threads.
<code>par_concurrent</code>	Whether or not to allow concurrent evaluations
<code>par_blasnumthreads</code>	Specifies the number of threads to use for parallel BLAS (when <code>blasoption = 1</code>)
<code>par_lnumthreads</code>	Specifies the number of threads to use for parallel linear system solves (when <code>linsolver = 6</code>)
<code>par_mnumthreads</code>	Specifies the number of threads to use for the parallel multi-start procedure.

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 determined by OpenMP. If `par_numthreads < 0`, Knitro will run in sequential mode.

Generally, if you are unsure of how best to apply parallel threads in Knitro you should just set the general option `par_numthreads` to the maximum number of threads you want Knitro to use, and leave `par_blasnumthreads` and `par_lnumthreads` at their default values. Then Knitro will try to allocate work to these different threads in the most sensible way. Typically, if you are performing a single solve, the threads will get applied to the BLAS operations. If, for example, you are using multi-start then the multi-start solves are run in parallel but BLAS is sequential (typically applying 2 layers of parallelism is not good).

The options `par_blasnumthreads` and `par_lnumthreads` allow the expert user more fine-grained control over parallelism of these specific features.

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_lnumthreads` is used to determine the number of threads Knitro can use for parallel linear system solves. This option is only active when using the Intel MKL PARDISO linear solver (`linsolver = 6`), the HSL MA97 linear solver (`linsolver = 7`) and the HSL MA86 linear solver (`linsolver = 8`). The domain specific `par_lnumthreads`, will override the general thread setting specified by `par_numthreads` for linear system solve operations.

The user option `par_mnumthreads` is used to determine the number of threads to use for the multi-start procedure. See *Multistart* for more details.

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.

2.10.8 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):

```

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

```

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

```

1  =====
2          Commercial License
3          Artelys Knitro 11.0.0
4  =====
5
6  Knitro presolve eliminated 0 variables and 0 constraints.
7
8  algorithm:          5
9  datacheck:         0
10 hessian_no_f:      1
11 ma_terminate:      0
12 par_concurrent_evals: 0
13 par_numthreads:    4
14
15 Problem Characteristics ( Presolved)
16 -----
17 Objective goal: Minimize
18 Objective type: quadratic
19 Number of variables: 3 ( 3)
20   bounded below only: 3 ( 3)
21   bounded above only: 0 ( 0)
22   bounded below and above: 0 ( 0)
23   fixed: 0 ( 0)
24   free: 0 ( 0)
25 Number of constraints: 2 ( 2)
26   linear equalities: 1 ( 1)
27   quadratic equalities: 0 ( 0)
28   gen. nonlinear equalities: 0 ( 0)
29   linear one-sided inequalities: 0 ( 0)
30   quadratic one-sided inequalities: 1 ( 1)
31   gen. nonlinear one-sided inequalities: 0 ( 0)

```

```

32     linear two-sided inequalities:                0 (          0)
33     quadratic two-sided inequalities:            0 (          0)
34     gen. nonlinear two-sided inequalities:       0 (          0)
35 Number of nonzeros in Jacobian:                 6 (          6)
36 Number of nonzeros in Hessian:                 5 (          5)
37
38 Knitro running multiple algorithms in parallel with 4 threads.
39
40     Alg      Status   Objective      FeasError      OptError      Real Time
41 -----
42         2         0   9.360000e+02   0.000e+00   1.945e-07     0.002
43         1         0   9.360000e+02   6.738e-08   6.614e-08     0.002
44         4         0   9.360000e+02   0.000e+00   2.387e-12     0.005
45         3         0   9.360000e+02   0.000e+00   0.000e+00     0.010
46 Multiple algorithms stopping, all solves have completed.
47
48 EXIT: Locally optimal solution found.
49
50 Final Statistics
51 -----
52 Final objective value                        = 9.35999997829394e+02
53 Final feasibility error (abs / rel) = 6.74e-08 / 5.18e-09
54 Final optimality error (abs / rel) = 6.61e-08 / 4.13e-09
55 # of iterations                             = 16
56 # of CG iterations                           = 12
57 # of function evaluations                    = 28
58 # of gradient evaluations                    = 24
59 # of Hessian evaluations                     = 16
60 Total program time (secs)                    = 0.01169 ( 0.023 CPU time)
61
62 =====
63
64 Knitro 11.0.0: Locally optimal or satisfactory solution.
65 objective 935.9999978293937; feasibility error 6.74e-08
66 16 iterations; 28 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.

2.10.9 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 `KN_solve()`:

```

// parallelism
if (KN_set_int_param_by_name (kc, "algorithm", KN_ALG_MULTI) != 0)
exit( -1 );
if (KN_set_int_param_by_name (kc, "ma_terminate", 0) != 0)
exit( -1 );
if (KN_set_int_param_by_name (kc, "par_numthreads", 4) != 0)
exit( -1 );

```

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

2.11 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 describes how to use the Knitro-Tuner.

2.11.1 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.

2.11.2 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 `KN_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 "linsolver_pivottol" 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).

algorithm
bar_directinterval 0 1 10
bar_murule
linsolver_pivottol 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 `linsolver_pivottol` option.

2.11.3 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					
Option Name	Value	#Runs	Percent Optimal	Average #FuncEvals	Average 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
linsolver_pivottol	1.00e-08	38	100.00	9.1	0.001
linsolver_pivottol	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. A corresponding file called `knitro_tuner_summary.csv` is also created, which allows easily reading these results into a spreadsheet. 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.

2.11.4 Tuner Options

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

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

Note that setting *par_numthreads* to use multiple threads allows the tuner to be run in parallel.

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

2.11.5 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'";
```

2.11.6 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
```

2.11.7 C example

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

```
/*----- TURN ON THE KNITRO-TUNER */
if (KN_set_int_param (kc, KN_PARAM_TUNER, KN_TUNER_ON) != 0)
    exit( -1 );

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

2.11.8 Object-oriented C++ example

In the object-oriented interface, a Tuner options file can be loaded through the *KTRSolver::loadTunerFile()* method.

```
// Turn on the KNITRO options file.
solver.setParam(KTR_PARAM_TUNER, KTR_TUNER_ON);

// Load tuner options file "tuner-explore.opt".
solver.loadTunerFile("tuner-explore.opt");
```

2.12 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.

2.12.1 Continuous problems

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

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

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

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

$$c_i^L \leq c_i(x) \leq c_i^U, \quad i = 0, \dots, m-1 \quad (3)$$

$$b_j^L \leq x_j \leq b_j^U, \quad j = 0, \dots, n-1 \quad (3b)$$

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

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

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

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

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.,

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

while the optimality error (*OptErr*) is defined as the maximum violation of the first three conditions (1)-(2b), with a small modification to conditions (2) and (2b). In these complementarity conditions, we really only need that either the multiplier or the corresponding constraint is 0, so we change the terms on the left side of these conditions to:

$$\min(\lambda_i^c g_i^c(x), \lambda_i^c, g_i^c(x)), \quad i = 0, \dots, m-1,$$

$$\min(\lambda_j^b g_j^x(x), \lambda_j^b, g_j^x(x)), \quad j = 0, \dots, n-1,$$

where

$$g_i^c(x) = \min[(c_i(x) - c_i^L), (c_i^U - c_i(x))],$$

$$g_j^x(x) = \min[(x_j - b_j^L), (b_j^U - x_j)],$$

to protect against numerical problems that may occur when the Lagrange multipliers become very large. 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

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

where x^0 represents the initial point.

For unconstrained problems, the scaling factor τ_2 is not effective since $\|\nabla f(x^k)\|_\infty \rightarrow 0$ 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:

$$\begin{aligned}\text{FeasErr} &\leq \min(\tau_1 * \text{feastol}, \text{feastol_abs}) && \text{(stop1)} \\ \text{OptErr} &\leq \min(\tau_2 * \text{opttol}, \text{opttol_abs}) && \text{(stop2)}\end{aligned}$$

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*.

Scaling

Note that the optimality conditions (stop2) apply to the problem being solved internally by Knitro. If the user option *scale* is enabled to perform some scaling of the problem, then the problem objective and constraint functions as well as the variables 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*.

Note that scaling the variables or constraints in the problem via the *scale* user option and scaling/modifying the stopping tolerances are two different things. You should use *scale* to try to make the variables/constraints in your model all have roughly the same magnitude (e.g. close to 1) so that the Knitro algorithms work better. Separately you should use the Knitro stopping tolerances to specify how much accuracy you require in the solution.

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 functions `KN_set_con_feastols()`, `KN_set_var_feastols()`, and `KN_set_compccon_feastols()`, which allows you to define custom tolerances for the general constraints, the variable bounds and any complementarity constraints. Please see section [Callable library API reference](#) for more details on these API functions. 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.

2.12.2 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

$$f^*(x_I) - f^*(x_R) \leq \text{mip_integral_gap_abs},$$

or

$$f^*(x_I) - f^*(x_R) \leq \text{mip_integral_gap_rel} * \max(1, |f^*(x_I)|),$$

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).

2.13 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.

2.13.1 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/exampleNLP1.c` with full output.

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 Artelys 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 License
      Artelys Knitro 11.0.0
=====

Knitro presolve eliminated 0 variables and 0 constraints.

outlev:                6
Knitro changing algorithm from AUTO to 1.
Knitro changing bar_initpt from AUTO to 3.
Knitro changing bar_murule from AUTO to 4.
Knitro changing bar_penaltycons from AUTO to 1.
Knitro changing bar_penaltyrule from AUTO to 2.
Knitro changing bar_switchrule from AUTO to 2.
Knitro changing linesearch from AUTO to 1.
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. If the Knitro presolver is enabled, then information about the presolved form of the problem is printed as well.

```

Problem Characteristics ( Presolved)
-----
Objective goal: Minimize
Objective type: general
Number of variables:      2 (      2)
  bounded below only:    0 (      0)
  bounded above only:    1 (      1)
  bounded below and above: 0 (      0)

```

fixed:	0	(0)
free:	1	(1)
Number of constraints:	2	(2)
linear equalities:	0	(0)
quadratic equalities:	0	(0)
gen. nonlinear equalities:	0	(0)
linear one-sided inequalities:	0	(0)
quadratic one-sided inequalities:	2	(2)
gen. nonlinear one-sided inequalities:	0	(0)
linear two-sided inequalities:	0	(0)
quadratic two-sided inequalities:	0	(0)
gen. nonlinear two-sided inequalities:	0	(0)
Number of nonzeros in Jacobian:	4	(4)
Number of nonzeros in Hessian:	3	(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	7	9.090000e+02	3.000e+00			
1	8	7.996681e+02	2.859e+00	2.186e+01	7.226e-02	0
2	9	1.859212e+01	9.066e-01	3.943e+01	2.199e+00	0
3	17	3.280079e+02	8.816e-01	6.903e+00	1.356e+00	8
4	18	1.445972e+01	4.912e-01	6.736e-01	1.173e+00	2
5	19	3.562070e+01	3.874e-01	3.874e-01	1.929e-01	0
6	20	1.153310e+02	2.196e-01	5.652e-01	4.098e-01	0
7	21	2.363651e+02	7.226e-02	7.226e-02	4.156e-01	0
8	22	3.018949e+02	5.268e-03	1.827e-02	1.861e-01	0
9	23	3.064952e+02	6.791e-06	1.513e-04	1.267e-02	0
10	24	3.065000e+02	9.480e-11	9.480e-11	1.358e-05	0

The meaning of each column is described below.

- **Iter:** iteration number.
- **fCount:** the cumulative number of (nonlinear) function evaluations. (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           = 3.06499999937285e+02
Final feasibility error (abs / rel) = 9.48e-11 / 3.16e-11
Final optimality error (abs / rel) = 9.48e-11 / 6.50e-12
# of iterations                 = 10
# of CG iterations              = 10
# of function evaluations       = 24
# of gradient evaluations       = 15
# of Hessian evaluations        = 10
Total program time (secs)       = 0.00258 ( 0.002 CPU time)
Time spent in evaluations (secs) = 0.00002
```

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 ] = 9.9999999905e-01,      lambda[ 0 ] = -6.9999999925e+02
c[ 1 ] = 4.4999999927e+00,      lambda[ 1 ] = -8.09211653221e-10

Solution Vector
-----
x[ 0 ] = 4.9999999998e-01,      lambda[ 2 ] = 1.75099999969e+03
x[ 1 ] = 1.9999999982e+00,      lambda[ 3 ] = 0.0000000000e+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`.

2.13.2 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/callbackMINLP1.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 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 License
      Artelys Knitro 11.0.0
=====

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 2.
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
Objective type:  general
Number of variables:          6
      bounded below only:      0

```

```

bounded above only:                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
  quadratic equalities:             0
  gen. nonlinear equalities:        0
  linear one-sided inequalities:     4
  quadratic one-sided inequalities:  0
  gen. nonlinear one-sided inequalities: 2
  linear two-sided inequalities:     0
  quadratic two-sided inequalities:  0
  gen. nonlinear two-sided inequalities: 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 searching for integer feasible point using heuristic
* iter =    1: Iinf =    0, FeasError =  2.068e-26, Obj =  1.000e+01
Knitro found integer feasible point in 1 heuristic iteration
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	7.592844e-01		7.592844e-01	1.000000e+01
2	1	1	5.171320e+00		7.592844e-01	1.000000e+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 `KN_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        = 8
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.

2.13.3 Getting information programmatically in callable library

Important solution information from Knitro can be retrieved through special API function calls.

Information related to the final statistics can be retrieved through the following function calls. The precise meaning of each function is described in the reference manual (*Callable library API reference*).

```
int KNITRO_API KN_get_number_FC_evals (const KN_context_ptr kc,
                                       int * const numFCevals);
int KNITRO_API KN_get_number_GA_evals (const KN_context_ptr kc,
                                       int * const numGAevals);
int KNITRO_API KN_get_number_H_evals (const KN_context_ptr kc,
                                       int * const numHevals);
int KNITRO_API KN_get_number_HV_evals (const KN_context_ptr kc,
                                       int * const numHVevals);
int KNITRO_API KN_get_solution (const KN_context_ptr kc,
                               int * const status,
                               double * const obj,
                               double * const x,
                               double * const lambda);
int KNITRO_API KN_get_obj_value (const KN_context_ptr kc,
                                 double * const obj);
int KNITRO_API KN_get_con_values (const KN_context_ptr kc,
                                 const KNINT nC,
                                 const KNINT * const indexCons,
                                 double * const c);
int KNITRO_API KN_get_rsd_values (const KN_context_ptr kc,
                                 const KNINT nR,
                                 const KNINT * const indexRsds,
                                 double * const r);
```

Continuous problems

```
int KNITRO_API KN_get_number_iters (const KN_context_ptr kc,
                                    int * const numIters);
int KNITRO_API KN_get_number_cg_iters (const KN_context_ptr kc,
                                       int * const numCGiters);
int KNITRO_API KN_get_abs_feas_error (const KN_context_ptr kc,
                                      double * const absFeasError);
int KNITRO_API KN_get_rel_feas_error (const KN_context_ptr kc,
                                      double * const relFeasError);
int KNITRO_API KN_get_abs_opt_error (const KN_context_ptr kc,
                                     double * const absOptError);
int KNITRO_API KN_get_rel_opt_error (const KN_context_ptr kc,
                                     double * const relOptError);
int KNITRO_API KN_get_objgrad_nnz (const KN_context_ptr kc,
                                   KNINT * const nnz);
int KNITRO_API KN_get_objgrad_values (const KN_context_ptr kc,
                                      KNINT * const indexVars,
                                      double * const objGrad);
int KNITRO_API KN_get_objgrad_values_all (const KN_context_ptr kc,
                                          double * const objGrad);
int KNITRO_API KN_get_jacobian_nnz (const KN_context_ptr kc,
                                    KNLONG * const nnz);
int KNITRO_API KN_get_jacobian_values (const KN_context_ptr kc,
                                       KNINT * const indexCons,
                                       KNINT * const indexVars,
                                       double * const jac);
int KNITRO_API KN_get_rsd_jacobian_nnz (const KN_context_ptr kc,
```

```

                                KNLONG * const  nnz);
int  KNITRO_API KN_get_rsd_jacobian_values (const KN_context_ptr  kc,
                                KNINT * const  indexRsds,
                                KNINT * const  indexVars,
                                double * const  rsdJac);
int  KNITRO_API KN_get_hessian_nnz      (const KN_context_ptr  kc,
                                KNLONG * const  nnz);
int  KNITRO_API KN_get_hessian_values (const KN_context_ptr  kc,
                                KNINT * const  indexVars1,
                                KNINT * const  indexVars2,
                                double * const  hess);

```

Discrete or mixed integer problems

```

int  KNITRO_API KN_get_mip_number_nodes (const KN_context_ptr  kc,
                                int * const  numNodes);
int  KNITRO_API KN_get_mip_number_solves (const KN_context_ptr  kc,
                                int * const  numSolves);
int  KNITRO_API KN_get_mip_abs_gap (const KN_context_ptr  kc,
                                double * const  absGap);
int  KNITRO_API KN_get_mip_rel_gap (const KN_context_ptr  kc,
                                double * const  relGap);
int  KNITRO_API KN_get_mip_incumbent_obj (const KN_context_ptr  kc,
                                double * const  incumbentObj);
int  KNITRO_API KN_get_mip_relaxation_bnd (const KN_context_ptr  kc,
                                double * const  relaxBound);
int  KNITRO_API KN_get_mip_lastnode_obj (const KN_context_ptr  kc,
                                double * const  lastNodeObj);
int  KNITRO_API KN_get_mip_incumbent_x (const KN_context_ptr  kc,
                                double * const  x);

```

2.13.4 Getting information programmatically in the object-oriented interface

Solution information can be retrieved after a call to `KTRSolver::solve()`. After `solve()` is called, `KTRSolver::getObj()` returns the objective function value, and `KTRSolver::getXValues()` and `KTRSolver::getLambdaValues()` return the final primal and dual variable values, respectively. The solution status code is returned by `KTRSolver::solve()` method.

In addition, information related to the final statistics can be retrieved through the following `KTRSolver` methods. The precise meaning of each function is described in the reference manual (*Callable library API reference*).

```

int  KTRSolver::getNumberFCEvals();
int  KTRSolver::getNumberGAEvals();
int  KTRSolver::getNumberHEvals();
int  KTRSolver::getNumberHVEvals();
std::vector<double> KTRSolver::getConstraintValues();

```

Continuous problems

```

int  KTRSolver::getNumberIters();
int  KTRSolver::getNumberCGIters();
double KTRSolver::getAbsFeasError();
double KTRSolver::getRelFeasError();
double KTRSolver::getAbsOptError();
double KTRSolver::getRelOptError();
std::vector<double> KTRSolver::getObjgradValues();

```

```
std::vector<double> KTRSolver::getJacobianValues();
std::vector<double> KTRSolver::getHessianValues();
```

Discrete or mixed integer problems

```
int KTRSolver::getMipNumNodes();
int KTRSolver::getMipNumSolves();
double KTRSolver::getMipAbsGap();
double KTRSolver::getMipRelGap();
double KTRSolver::getMipIncumbentObj();
std::vector<double> KTRSolver::getMipIncumbentX();
double KTRSolver::getMipRelaxationBnd();
double KTRSolver::getMipLastnodeObj();
```

2.13.5 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 functions `KN_set_obj_name()`, `KN_set_var_names()`, `KN_set_con_names()`, and `KN_set_compcon_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;
```

2.13.6 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.

2.13.7 AMPL solution information through suffixes

Some Knitro solution information can be retrieved and displayed through AMPL using AMPL suffixes defined for Knitro (see *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).

2.13.8 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 11.0.0: 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  (general)
  5.600000e+01  <=  c[  1]  <=  5.600000e+01  c1  (linear)
-----
Knitro 11.0.0: Locally optimal or satisfactory solution.
objective 935.9999978; feasibility error 6.74e-08
5 iterations; 7 function evaluations
```

2.14 Callbacks

Knitro needs to evaluate the objective function and constraints (function values and ideally, their derivatives) at various points along the optimization process. If these functions are linear or quadratic then Knitro has specialized API routines for loading their linear and quadratic structures. However, in order to pass this information to Knitro for more general nonlinear functions, 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 for nonlinear functions (as well as for other specialized user non-evaluation callbacks

described below or in *Callable library API reference*).

If your callback requires additional parameters beyond what is passed through the arguments, 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 are defined in *knitro.h*. The prototype used for evaluation callbacks is:

```

/** Function prototype for evaluation callbacks. */
typedef int KN_eval_callback (KN_context_ptr          kc,
                             CB_context_ptr         cb,
                             KN_eval_request_ptr     const evalRequest,
                             KN_eval_result_ptr      const evalResult,
                             void                   * const userParams);

```

where the structures used to define the arguments *evalRequest* and *evalResult* are given by:

```

/** Structure used to pass back evaluation information for evaluation callbacks.
 *
 * type:          - indicates the type of evaluation requested
 * threadID:     - the thread ID associated with this evaluation request;
 *                useful for multi-threaded, concurrent evaluations
 * x:            - values of unknown (primal) variables used for all evaluations
 * lambda:       - values of unknown dual variables/Lagrange multipliers
 *                used for the evaluation of the Hessian
 * sigma:        - scalar multiplier for the objective component of the Hessian
 * vec:          - vector array value for Hessian-vector products (only used
 *                when user option hessopt=KN_HESSOPT_PRODUCT)
 */
typedef struct KN_eval_request {
    int         type;
    int         threadID;
    const double * x;
    const double * lambda;
    const double * sigma;
    const double * vec;
} KN_eval_request, *KN_eval_request_ptr;

/** Structure used to return results information for evaluation callbacks.
 * The arrays (and their indices and sizes) returned in this structure are
 * local to the specific callback structure used for the evaluation.
 *
 * obj:          - objective function evaluated at "x" for EVALFC or
 *                EVALFCGA request (funcCallback)
 * c:            - (length nC) constraint values evaluated at "x" for
 *                EVALFC or EVALFCGA request (funcCallback)
 * objGrad:      - (length nV) objective gradient evaluated at "x" for
 *                EVALGA request (gradCallback) or EVALFCGA request_
 * ↪ (funcCallback)
 * jac:          - (length nnzJ) constraint Jacobian evaluated at "x" for
 *                EVALGA request (gradCallback) or EVALFCGA request_
 * ↪ (funcCallback)
 * hess:         - (length nnzH) Hessian evaluated at "x", "lambda", "sigma"
 *                for EVALH or EVALH_NO_F request (hessCallback)
 * hessVec:      - (length n=number variables in the model) Hessian-vector
 *                product evaluated at "x", "lambda", "sigma"
 *                for EVALHV or EVALHV_NO_F request (hessCallback)
 * rsd:          - (length nR) residual values evaluated at "x" for EVALR

```

```

*          request (rsdCallback)
*   rsdJac:      - (length nnzJ) residual Jacobian evaluated at "x" for
*                 EVALRJ request (rsdJacCallback)
*/
typedef struct KN_eval_result {
  double * obj;
  double * c;
  double * objGrad;
  double * jac;
  double * hess;
  double * hessVec;
  double * rsd;
  double * rsdJac;
} KN_eval_result, *KN_eval_result_ptr;

```

The callback functions for evaluating the functions, gradients and Hessian 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. See the *Derivatives* section for details on how to compute the Jacobian and Hessian matrices in a form suitable for Knitro.

Minimally, and as a first step, you must call `KN_add_eval_callback()` to establish a callback to evaluate the nonlinear functions in your model.

```

/** This is the routine for adding a callback for (nonlinear) evaluations
* of objective and constraint functions. This routine can be called
* multiple times to add more than one callback structure (e.g. to create
* different callback structures to handle different blocks of constraints).
* This routine specifies the minimal information needed for a callback, and
* creates the callback structure "cb", which can then be passed to other
* callback functions to set additional information for that callback.
*
*   evalObj      - boolean indicating whether or not any part of the objective
*                 function is evaluated in the callback
*   nC           - number of constraints evaluated in the callback
*   indexCons    - (length nC) index of constraints evaluated in the callback
*                 (set to NULL if nC=0)
*   funcCallback - a pointer to a function that evaluates the objective parts
*                 (if evalObj=KNTRUE) and any constraint parts (specified by
*                 nC and indexCons) involved in this callback; when
*                 eval_fcga=KN_EVAL_FCGA_YES, this callback should also evaluate
*                 the relevant first derivatives/gradients
*   cb           - (output) the callback structure that gets created by
*                 calling this function; all the memory for this structure is
*                 handled by Knitro
*
* After a callback is created by "KN_add_eval_callback()", the user can then specify
* gradient information and structure through "KN_set_cb_grad()" and Hessian
* information and structure through "KN_set_cb_hess()". If not set, Knitro will
* approximate these. However, it is highly recommended to provide a callback_
↳ routine
* to specify the gradients if at all possible as this will greatly improve the
* performance of Knitro. Even if a gradient callback is not provided, it is still
* helpful to provide the sparse Jacobian structure through "KN_set_cb_grad()" to
* improve the efficiency of the finite-difference gradient approximations.
* Other optional information can also be set via "KN_set_cb_*( )" functions as
* detailed below.
*
* Returns 0 if OK, nonzero if error.

```

```

*/
int KNITRO_API KN_add_eval_callback (      KN_context_ptr      kc,
                                          const KNBOOL        evalObj,
                                          const KNINT         nC,
                                          const KNINT         * const indexCons,  /
↳ * nullable if nC=0 */
                                          KN_eval_callback * const funcCallback,
                                          CB_context_ptr * const cb);

```

This callback returns a callback structure `cb` that can then be passed to other callback evaluation routines to specify particular, optional, properties for that callback. For example, it can be passed to `KN_set_cb_grad()` to specify a callback function for gradients (i.e. first derivatives) or `KN_set_cb_hess()` to specify a callback for the Hessian. If these are not set, Knitro will approximate derivatives internally. However, we highly recommend providing callbacks to evaluate derivatives whenever possible as this can dramatically improve the performance of Knitro.

Although the easiest approach is to create one callback structure to use for all evaluations, it is possible to call `KN_add_eval_callback()` multiple times to create different `cb` callback structures for different groups of non-linear functions. This would allow, for instance, providing exact derivatives for some functions via callbacks, while having Knitro approximate other derivatives using finite-differencing.

For least-squares problems use `KN_add_lsq_eval_callback()` and `KN_set_cb_rsd_jac()`. See *Nonlinear Least-Squares* for more details.

Other evaluation callback API functions include `KN_set_cb_user_params()`, `KN_set_cb_gradopt()`, and `KN_set_cb_relstepsizes()`. More may be added in the future. These are described in detail in *Callable library API reference*.

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 KNITRO_API KN_set_puts_callback (KN_context_ptr kc,
                                     KN_puts * const fnPtr,
                                     void * const userParams);

```

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

```

typedef int KN_puts (const char * const str,
                    void * const userParams);

```

In addition to the callbacks defined above, Knitro makes additional callbacks available to the user for features such as multi-start and MINLP, including `KN_set_newpt_callback()`, `KN_set_mip_node_callback()`, and `KN_set_ms_initpt_callback()`.

The prototype used for many of the other non-evaluation user callbacks (e.g. the *newpoint* callback) is:

```

/** Type declaration for several non-evaluation user callbacks defined
 * below.
 */
typedef int KN_user_callback (      KN_context_ptr kc,
                                   const double * const x,
                                   const double * const lambda,
                                   void * const userParams);

```

Please see a complete list and description of Knitro callback functions in the *Callable library API reference* section in the Reference Manual. In addition, we recommend closely reviewing the examples provided in `examples/C` which provide examples of how to use most of these callback functions.

For information on setting callbacks in the object-oriented interface, see the *Object-oriented interface reference*.

2.14.1 Example

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

$$\begin{aligned} \min & 100 - (x_2 - x_1^2)^2 + (1 - x_1)^2 \\ & 1 \leq x_1 x_2, 0 \leq x_1 + x_2^2, x_1 \leq 0.5. \end{aligned}$$

This problem is coded as `examples/C/exampleNLP1.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 (`KN_new()` might return NULL if the Artelys license check fails):

```
#include "knitro.h"

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

KN_context      *kc;

/*... Declare other local variables ...*/
double xLoBnds[2] = {-KN_INFINITY, -KN_INFINITY};
double xUpBnds[2] = {0.5, KN_INFINITY};
double xInitVals[2] = {-2.0, 1.0};
double cLoBnds[2] = {1.0, 0.0};

error = KN_new(&kc);
if (error) exit(-1);
if (kc == NULL)
{
    printf ("Failed to find a valid license.\n");
    return( -1 );
}
```

The next task is to load the problem definition into the solver using various API functions for adding variables and constraints, bounds, linear and quadratic structures, etc. The code below captures the basic problem definition passed to Knitro:

```
/** Initialize Knitro with the problem definition. */

/** Add the variables and set their bounds.
 * Note: any unset lower bounds are assumed to be
 * unbounded below and any unset upper bounds are
 * assumed to be unbounded above. */
n = 2;
error = KN_add_vars(kc, n, NULL);
if (error) exit(-1);
error = KN_set_var_lobnds_all(kc, xLoBnds); /* not necessary since infinite */
if (error) exit(-1);
error = KN_set_var_upbnds_all(kc, xUpBnds);
if (error) exit(-1);
/** Define an initial point. If not set, Knitro will generate one. */
error = KN_set_var_primal_init_values_all(kc, xInitVals);
if (error) exit(-1);
```

```

/** Add the constraints and set their lower bounds */
m = 2;
error = KN_add_cons(kc, m, NULL);
if (error) exit(-1);
error = KN_set_con_lobnds_all(kc, cLoBnds);
if (error) exit(-1);

/** Both constraints are quadratic so we can directly load all the
 * structure for these constraints. */

/** First load quadratic structure x0*x1 for the first constraint */
indexVar1 = 0; indexVar2 = 1; coef = 1.0;
error = KN_add_con_quadratic_struct_one (kc, 1, 0,
                                         &indexVar1, &indexVar2, &coef);
if (error) exit(-1);

/** Load structure for the second constraint. below we add the linear
 * structure and the quadratic structure separately, though it
 * is possible to add both together in one call to
 * "KN_add_con_quadratic_struct_one()" since this api function also
 * supports adding linear terms. */

/** Add linear term x0 in the second constraint */
indexVar1 = 0; coef = 1.0;
error = KN_add_con_linear_struct_one (kc, 1, 1,
                                       &indexVar1, &coef);
if (error) exit(-1);

/** Add quadratic term x1^2 in the second constraint */
indexVar1 = 1; indexVar2 = 1; coef = 1.0;
KN_add_con_quadratic_struct_one (kc, 1, 1,
                                  &indexVar1, &indexVar2, &coef);

```

After loading the basic problem information, we add the callbacks for evaluating the nonlinear objective function (and its derivatives) as shown below.

```

/** Add a callback function "callbackEvalF" to evaluate the nonlinear
 * (non-quadratic) objective. Note that the linear and
 * quadratic terms in the objective could be loaded separately
 * via "KN_add_obj_linear_struct()" / "KN_add_obj_quadratic_struct()".
 * However, for simplicity, we evaluate the whole objective
 * function through the callback. */
error = KN_add_eval_callback (kc, KNTRUE, 0, NULL, callbackEvalF, &cb);
if (error) exit(-1);

/** Also add a callback function "callbackEvalG" to evaluate the
 * objective gradient. If not provided, Knitro will approximate
 * the gradient using finite-differencing. However, we recommend
 * providing callbacks to evaluate the exact gradients whenever
 * possible as this can drastically improve the performance of Knitro.
 * We specify the objective gradient in "dense" form for simplicity.
 * However for models with many constraints, it is important to specify
 * the non-zero sparsity structure of the constraint gradients
 * (i.e. Jacobian matrix) for efficiency (this is true even when using
 * finite-difference gradients). */
error = KN_set_cb_grad (kc, cb, KN_DENSE, NULL, 0, NULL, NULL, callbackEvalG);
if (error) exit(-1);

```

```

/** Add a callback function "callbackEvalH" to evaluate the Hessian
 * (i.e. second derivative matrix) of the objective. If not specified,
 * Knitro will approximate the Hessian. However, providing a callback
 * for the exact Hessian (as well as the non-zero sparsity structure)
 * can greatly improve Knitro performance and is recommended if possible.
 * Since the Hessian is symmetric, only the upper triangle is provided.
 * Again for simplicity, we specify it in dense (row major) form. */
error = KN_set_cb_hess (kc, cb, KN_DENSE_ROWMAJOR, NULL, NULL, callbackEvalH);
if (error) exit(-1);

```

These evaluation callback functions use the `KN_eval_callback()` prototype. In examples/C/exampleNLP1.c these are named `callbackEvalF`, `callbackEvalG`, and `callbackEvalH`.

```

/*-----*/
/*      FUNCTION callbackEvalF      */
/*-----*/
/** The signature of this function matches KN_eval_callback in knitro.h.
 * Only "obj" is set in the KN_eval_result structure.
 */
int callbackEvalF (KN_context_ptr      kc,
                  CB_context_ptr      cb,
                  KN_eval_request_ptr const evalRequest,
                  KN_eval_result_ptr const evalResult,
                  void                * const userParams)
{
    const double *x;
    double *obj;
    double dTmp;

    if (evalRequest->type != KN_RC_EVALFC)
    {
        printf ("*** callbackEvalFC incorrectly called with eval type %d\n",
              evalRequest->type);
        return( -1 );
    }
    x = evalRequest->x;
    obj = evalResult->obj;

    /** Evaluate nonlinear objective */
    dTmp = x[1] - x[0]*x[0];
    *obj = 100.0 * (dTmp*dTmp) + ((1.0 - x[0])*(1.0 - x[0]));

    return( 0 );
}

/*-----*/
/*      FUNCTION callbackEvalG      */
/*-----*/
/** The signature of this function matches KN_eval_callback in knitro.h.
 * Only "objGrad" is set in the KN_eval_result structure.
 */
int callbackEvalG (KN_context_ptr      kc,
                  CB_context_ptr      cb,
                  KN_eval_request_ptr const evalRequest,
                  KN_eval_result_ptr const evalResult,
                  void                * const userParams)
{

```

```

const double *x;
double *objGrad;
double dTmp;

if (evalRequest->type != KN_RC_EVALGA)
{
    printf ("*** callbackEvalGA incorrectly called with eval type %d\n",
            evalRequest->type);
    return( -1 );
}
x = evalRequest->x;
objGrad = evalResult->objGrad;

/** Evaluate gradient of nonlinear objective */
dTmp = x[1] - x[0]*x[0];
objGrad[0] = (-400.0 * dTmp * x[0]) - (2.0 * (1.0 - x[0]));
objGrad[1] = 200.0 * dTmp;

return( 0 );
}

/*-----*/
/*      FUNCTION callbackEvalH      */
/*-----*/
/** The signature of this function matches KN_eval_callback in knitro.h.
 * Only "hess" and "hessVec" are set in the KN_eval_result structure.
 */
int callbackEvalH (KN_context_ptr          kc,
                  CB_context_ptr         cb,
                  KN_eval_request_ptr     const evalRequest,
                  KN_eval_result_ptr     const evalResult,
                  void                    * const userParams)
{
    const double *x;
    double sigma;
    double *hess;

    if (    evalRequest->type != KN_RC_EVALH
        && evalRequest->type != KN_RC_EVALH_NO_F)
    {
        printf ("*** callbackEvalHess incorrectly called with eval type %d\n",
                evalRequest->type);
        return( -1 );
    }

    x = evalRequest->x;
    /** Scale objective component of hessian by sigma */
    sigma = *(evalRequest->sigma);
    hess = evalResult->hess;

    /** Evaluate the hessian of the nonlinear objective.
     * Note: Since the Hessian is symmetric, we only provide the
     *       nonzero elements in the upper triangle (plus diagonal).
     *       These are provided in row major ordering as specified
     *       by the setting KN_DENSE_ROWMAJOR in "KN_set_cb_hess()".
     * Note: The Hessian terms for the quadratic constraints
     *       will be added internally by Knitro to form
     *       the full Hessian of the Lagrangian. */

```

```

hess[0] = sigma * ( (-400.0 * x[1]) + (1200.0 * x[0]*x[0]) + 2.0); // (0,0)
hess[1] = sigma * (-400.0 * x[0]); // (0,1)
hess[2] = sigma * 200.0; // (1,1)

return( 0 );
}

```

Back in the main program `KN_solve()` is invoked to find the solution:

```

/** Solve the problem.
 *
 * Return status codes are defined in "knitro.h" and described
 * in the Knitro manual.
 */
nStatus = KN_solve (kc);

/** Delete the Knitro solver instance. */
KN_free (&kc);

```

2.15 Other programmatic interfaces

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

2.15.1 Knitro in a C++ application

Note: C++ driver-based interface has been superseded by C++ object-oriented interface, see *Object-oriented interface reference*. The C++ driver and examples are not available anymore since Knitro 10.0.

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.

2.15.2 Knitro in a C# application

Calling Knitro from a C# application is similar to using the object-oriented interface in C++. The primary difference between the C++ and C# version of the object-oriented interface is in the syntax and function name capitalization. The C# function names are capitalized, and the functions use `IList<>` (implemented as `List<>`) for function arguments and return values.

The C# object-oriented interface requires .NET Version 4.0 and up. The interface uses `P/Invoke` to call the C Knitro callable library and convert data and function signatures between C# and C, and uses `knitro.h` and the `knitro.dll` dynamic library.

Examples of problem definitions and Knitro callbacks in C# can be found in the example folders distributed with Knitro, and the source code for the interface is provided for informational purposes.

2.15.3 Knitro in a Java application

Calling Knitro from a Java application is similar to using the object-oriented interface in C++. The primary difference between the C++ and Java version of the object-oriented interface is in the syntax. The Java function names are capitalized, and the functions use *List<>* (implemented as *ArrayList<>*) for function arguments and return values.

The Java object-oriented interface requires Java 1.6 and up. The interface uses JNA (Java Native Access) to call the C Knitro callable library and convert data and function signatures between Java and C, and uses `knitro.h` and the `knitro.dll` dynamic library (`libknitro.so` on Linux; `libknitro.dylib` on Mac OS X).

Examples of problem definitions and Knitro callbacks in Java can be found in the example folders distributed with Knitro, and the source code for the interface is provided for informational purposes.

2.15.4 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 old pre-Knitro 11.0 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.

2.15.5 Knitro in a Python application

Knitro provides a Python interface 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. However, please note that Python interpreters are usually not thread safe so callbacks cannot be evaluated in parallel. Thus `par_concurrent_evals` is always initialized to 0/’no’ in the Python interface, but may still be set to 1/’yes’ by the user.

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 the output value is not requested.

The optimization problem must be defined in terms of Python lists and constants that follows the pre-Knitro 11.0 C API, and the Python version of `KTR_init_problem()` / `KTR_mip_init_problem()` is then 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.

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 Python lists. Although the Python language makes it unnecessary, Python objects may be passed to the callback function through the `userParams` argument.

The sample programs can be run directly from the command line after installing `knitro.py` (or making sure that `knitro.py` 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 supports Python versions 2.7 and 3.6.

2.16 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.

2.16.1 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. Knitro will automatically detect LPs and apply these specializations, provided the linear structure for the model is added using the API functions for adding linear structure (and not more general callbacks). See [Callable library API reference](#) for more detail on API functions for adding linear structure.

2.16.2 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. Knitro will automatically detect QPs and apply these specializations, provided the linear and quadratic structure for the model is added using the API functions for adding linear and quadratic structure (and not more general callbacks). See [Callable library API reference](#) for more detail on API functions for adding linear and quadratic structure.

Typically, these specializations will only help on convex QPs.

2.16.3 Systems of nonlinear equations

Knitro is effective at solving systems of nonlinear equations.

There are two ways to try to solve a square system of nonlinear equations using Knitro. In the first way, you can use the least-squares API (see [Nonlinear Least-Squares](#)) and just specify the nonlinear equations as the residual functions. Knitro will then formulate your model as an unconstrained optimization problem where the objective function to be minimized is the sum of squares of the nonlinear equations and apply the Gauss-Newton Hessian.

In the second way, you can use the standard Knitro API and specify the nonlinear equations as equality constraints and specify the objective function as zero (i.e., $f(x)=0$).

The first approach is the recommended approach, however, you should experiment with both formulations to see which one works better.

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.

2.16.4 Least squares problems

Knitro offers a specialized API for solving least-squares problems,

$$\min f(x) = 0.5 * (r_1(x)^2 + r_2(x)^2 + \dots + r_q(x)^2)$$

with or without bounds on the variables (see *Nonlinear Least-Squares*). By default, this specialized interface will apply the Gauss-Newton Hessian

$$J(x)^T J(x)$$

where $J(x)$ is the Jacobian matrix of the residual functions $r_j(x)$ at x . Knitro will behave like a Gauss-Newton method by using the linesearch methods `algorithm = KN_ALG_BAR_DIRECT` or `KN_ALG_ACT_SQP`, and will be very similar to the classical Levenberg-Marquardt method when using the trust-region methods `algorithm = KN_ALG_BAR_CG` or `KN_ALG_ACT_CG`. The Gauss-Newton and Levenberg-Marquardt approaches consist of using this approximate value for the Hessian and ignoring the remaining term. Using the specialized least-squares interface will generally be the most effective way to solve least-squares models with Knitro, as it only requires first derivatives of the residual functions, $r_j(x)$, and yet can converge rapidly in most cases.

However, in some cases, if the value of the objective function at the solution is not close to zero (the large residual case), and/or the user can provide the full, exact Hessian matrix, then it may be more efficient to use the standard API and solve the least-squares model as any other optimization problem. Any of the Knitro options can be used.

See *Nonlinear Least Squares* for an implementation in *knitromatlab*.

2.16.5 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.

2.16.6 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*.

2.16.7 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.

2.17 Tips and tricks

This last chapter contains some rules of the thumb to improve efficiency, solve memory issues and other frequent problems.

2.17.1 Automatic Differentiation (AD)

Why is Automatic Differentiation important?

Nonlinear optimization software relies on accurate and efficient derivative computations for faster solutions and improved robustness.

Knitro in particular has the ability to utilize second derivative (Hessian matrix) information for faster convergence. Computing partial derivatives and coding them manually in a programming language can be time consuming and error prone (Knitro does provide a function to check first derivatives against finite differences).

Automatic Differentiation (AD) is a modern technique which automatically and efficiently computes the exact derivatives so that the user is freed from dealing with this issue.

Most modeling languages provide automatic differentiation.

2.17.2 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 `cg_maxit` 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 `mip_nodealg` option which will determine the Knitro algorithm to use to solve the nonlinear, continuous subproblems generated during the branch and bound process.

2.17.3 Setting bounds efficiently

Why is Knitro not honoring my bound constraints?

By default Knitro does not enforce that simple bounds on the variables (x) are satisfied throughout the optimization process. Rather, satisfaction of these bounds is only enforced at the solution.

In some applications, however, the user may want to enforce that the initial point and all intermediate iterates satisfy the bounds $b^L \leq x \leq b^U$. This can be enforced by setting `KN_PARAM_HONORBND`s to 1.

Please note, the honor bounds option pertains only to the simple bounds defined with vectors b^U and b^L for x , not to the general equality and inequality constraints defined with the vectors c^U , c^L , and c .

Do I need to specify a constraint with c^U , c^L , and c if I already specified it with the bounds parameters b^U and b^L ?

No, if you have specified a constraint with the bounds parameters then you should not specify it with the general constraints.

For example, $x_0 \leq 2$ is best modeled by setting $b_0^L = -\text{KN_INFINITY}$ and $b_0^U = 2$.

Duplicate specification of a constraint can make the problem more difficult to solve.

Do I need to initialize all of the bounds parameters? What if a variable is unbounded?

You only need to initialize finite bounds in your model using the API functions `KN_set_var_lobnds()`, `KN_set_var_upbnds()`, and `KN_set_var_fxbnds()`. Any variable bounds that are not explicitly set are infinite (i.e. unbounded).

You can also explicitly mark infinite bounds using the API functions above by using Knitro's value for infinity, `KN_INFINITY` to denote unbounded.

Note that any finite variable bound larger than `bndrange()` in magnitude will be treated as infinite by Knitro. To treat it as a real finite bound, you must either increase the value of `bndrange()` to be larger than the largest finite bound, or rescale the problem to make the finite bounds smaller in magnitude.

See `include/knitro.h` for the definition of `KN_INFINITY`.

Do I need to initialize all of the constraint parameters c^U and c^L ? What if a constraint is unbounded?

You only need to initialize finite bounds in your model using the API functions `KN_set_con_lobnds()`, `KN_set_con_upbnds()`, and `KN_set_con_eqbnds()`. Any constraint bounds that are not explicitly set are infinite (i.e. unbounded).

You can also explicitly mark infinite bounds using the API functions above by using Knitro's value for infinity, `KN_INFINITY` to denote unbounded.

Note that any finite constraint bound larger than `bndrange()` in magnitude will be treated as infinite by Knitro. To treat it as a real finite bound, you must either increase the value of `bndrange()` to be larger than the largest finite bound, or rescale the problem to make the finite bounds smaller in magnitude.

See `include/knitro.h` for the definition of `KN_INFINITY`.

2.17.4 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 `product_findiff`, `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 `linsolver_pivottol` user option value may reduce the amount of memory needed for the linear solver.

2.17.5 Reproducibility issues across platform/computer

If you notice different results across platforms/computers for the exact same Knitro run (same model, same initial conditions, same options), it is probably due to one of the following reasons:

- Knitro library is built with different compilers on different OS which can cause small numerical differences that propagate.
- The Intel MKL library has specializations to optimize performance for particular hardware/CPUs/environments which can also cause numerical differences.

To avoid the second issue you may set `blasoption = 0`.

2.18 Bibliography

For a detailed description of the algorithm implemented in *Interior/CG* see Byrd et al., 1999¹ and for the global convergence theory see Byrd et al., 2000². The method implemented in *Interior/Direct* is described in Waltz et al., 2006³. The *Active Set* algorithm is described in Byrd et al., 2004⁴ and the global convergence theory for this algorithm is in Byrd et al., 2006a⁵. A summary of the algorithms and techniques implemented in the Knitro software product is given in Byrd et al., 2006b⁶.

The implementation of the CG preconditioner makes use of the *icfs* software, which is described in details in Lin and Moré, 1999¹³.

For mixed-integer nonlinear optimization, the hybrid Quesada-Grossman (HQG) method in Knitro is based on the algorithm described in⁷. The MISQP algorithm in Knitro is Artelys' own implementation of the MISQP algorithm described in⁸ but differs in some details.

We also recommend the following papers: Byrd et al., 2003⁹, Fourer et al., 2003¹⁰, Hock and Schittkowski, 1981¹¹, and Nocedal and Wright, 1999¹².

To solve linear systems arising at every iteration of the algorithm, Knitro may utilize routines *MA27* or *MA57*¹⁴, a component package of the Harwell Subroutine Library (HSL). HSL, a collection of Fortran codes for large-scale scientific computation. See <http://www.hsl.rl.ac.uk/>

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.artelys.com/tools/clp/>

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.

¹³ C.-J. Lin and J. J. Moré, "Incomplete Cholesky factorizations with limited memory", *SIAM J. Sci. Comput.*, 21(1):24–45, 1999.

⁷ I. Quesada, and I. E. Grossmann, "An LP/NLP based branch and bound algorithm for convex MINLP optimization problems", *Computers and Chemical Engineering*, 16(10-11):937–947, 1992.

⁸ O. Exler, and K. Schittkowski, "A trust-region SQP algorithm for mixed-integer nonlinear programming", *Optimization Letters*, Vol. 1:269–280, 2007.

⁹ 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.

¹⁰ R. Fourer, D. M. Gay, and B. W. Kernighan, "AMPL: A Modeling Language for Mathematical Programming", 2nd Ed., Brooks/Cole – Thomson Learning, 2003.

¹¹ Hock, W. and Schittkowski, K. "Test Examples for Nonlinear Programming Codes", volume 187 of *Lecture Notes in Economics and Mathematical Systems*, Springer-Verlag, 1981.

¹² J. Nocedal and S. J. Wright, "Numerical Optimization", *Springer Series in Operations Research*, Springer, 1999.

¹⁴ Harwell Subroutine Library, "A catalogue of subroutines (HSL 2002)", AEA Technology, Harwell, Oxfordshire, England, 2002.

Lastly, Knitro may make use of the Intel(R) Math Kernel Library (<https://software.intel.com/en-us/intel-mkl>) for some linear algebra computations.

REFERENCE MANUAL

The reference manual describes in details the different available interfaces, the callable library API (including the return codes), the Knitro user options and the output files that can be generated by Knitro.

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.

```
act_lpalg          LP algorithm used in Active or SQP subproblems
act_lpdumpmps      Dump LPs to MPS files in Active or SQP algorithm
act_lpfeastol      Feasibility tolerance for LP subproblems
act_lppenalty      Controls constraint penalization in LP subproblems
act_lppresolve     Controls LP presolve in Active or SQP subproblems
act_lpsolver       LP solver used by Active or SQP algorithm
act_parametric     Use parametric LP in Active or SQP algorithm
act_qpalg          QP subproblem alg used by Active or SQP algorithm
alg                Algorithm (0=auto, 1=direct, 2=cg, 3=active, 4=sqp, 5=multi)
algorithm          Synonym for alg
bar_conic_enable   Special handling of conic constraints
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_initpi_mpec    Initial value for barrier MPEC penalty parameter
bar_initpt         Barrier initial point strategy for slacks/multipliers
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_slackboundpush Amount by which slacks are pushed inside bounds
bar_switchobj      Objective for barrier switching alg
bar_switchrule     Rule for barrier switching alg
bar_watchdog       Enable watchdog heuristic for barrier algs?
blasoption         Which BLAS/LAPACK library to use
blasoptionlib      Name of dynamic BLAS/LAPACK library
bndrange          Constraint/variable bound range
```

cg_maxit	Maximum number of conjugate gradient iterations
cg_pmem	Memory for incomplete Cholesky
cg_precond	Preconditioning method
cg_stoptol	Stopping tolerance for CG subproblems
convex	Declare the problem as convex
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_terminate	Derivative checker type (1=error, 2=always)
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
fstopval	Stop based on obj. function value
ftol	Stop based on small change in obj. function
ftol_iters	Stop based on small change in obj. function
gradopt	Gradient computation method
hessopt	Hessian computation method
honorbnds	Enforce satisfaction of the bounds
infeastol	Infeasibility stopping tolerance
initpenalty	Initial merit function penalty value
linesearch	Which linesearch method to use
linesearch_maxtrials	Maximum number of linesearch trial points
linsolver	Which linear solver to use
linsolver_ooc	Use out-of-core option?
linsolver_pivottol	Initial pivot tolerance
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'
maxfevals	Maximum number of function evaluations
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_heuristic_terminate	MIP heuristic termination
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_intvar_strategy	Treatment of integer variables
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, 3=MISQP)
mip_nodealg	Standard node relaxation algorithm
mip_outinterval	MIP output interval

mip_outlevel	MIP output level
mip_outsub	Enable MIP subproblem output
mip_pseudoinit	Pseudo-cost initialization
mip_relaxable	Are integer variables relaxable?
mip_rootalg	Root node relaxation algorithm
mip_rounding	MIP rounding rule
mip_selectdir	MIP node selection direction
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
out_csvinfo	Create knitro_solve.csv info file
out_csvname	Name for csv info file
out_hints	Print hints for parameter settings
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)
outname	Name for output file
par_blasnumthreads	Number of parallel threads for BLAS
par_lsnumthreads	Number of parallel threads for linear solver
par_mnumthreads	Number of parallel threads for multistart
par_numthreads	Number of parallel threads

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
restarts	Maximum number of restarts allowed
restarts_maxit	Maximum number of iterations before restarting
scale	Automatic scaling option
soc	Second order correction options
threads	Number of parallel threads
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
xtol_iters	Stop based on small change in variables

These options are detailed below.

3.1.1 Knitro options in AMPL

- **act_lpalg**: LP subproblem algorithm in Active Set or SQP algorithm (default 0). See [act_lpalg](#).

Value	Description
0	default LP algorithm
1	primal simplex algorithm
2	dual simplex algorithm
3	barrier/interior-point algorithm

- **act_lpdumpmps**: Used for internal debugging.
- **act_lpfeastol**: feasibility tolerance for LP subproblems (default 1.0e-8). See [act_lpfeastol](#).
- **act_lppenalty**: Use penalty formulation for LP subproblem in Active Set or SQP algorithm (default 1). See [act_lppenalty](#).

Value	Description
1	penalize all constraints
2	penalize only nonlinear constraints
3	dynamically choose which constraints to penalize

- **act_lppresolve**: Control presolve for LP subproblems in Active Set or SQP algorithm (default 0). See [act_lppresolve](#).

Value	Description
0	presolve turned off for LP subproblems
1	presolve turned on for LP subproblems

- **act_lpsolver**: LP solver used in Active Set or SQP algorithm (default 1). See [act_lpsolver](#).
- **act_parametric**: Use parametric LP subproblems in Active Set or SQP algorithm (default 1). See [act_parametric](#).

Value	Description
0	do not use a parametric solve
1	use a parametric solve sometimes
2	always try a parametric solve

- **act_qpalg**: QP subproblem algorithm in Active Set or SQP algorithm (default 0). See [act_qpalg](#).

Value	Description
0	let Knitro decide the QP algorithm
1	Interior/Direct (barrier) algorithm
2	Interior/CG (barrier) algorithm
3	Active Set algorithm

- **alg** or **algorithm**: 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
4	Sequential Quadratic Programming (SQP) algorithm
5	Run multiple algorithms

- **bar_conic_enable**: enable special treatment for conic constraints (default 0). See [bar_conic_enable](#).

Value	Description
0	do not apply any special treatment for conic constraints
1	apply special treatments for any Second Order Cone (SOC) constraints

- **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_initpi_mpec**: initial value for barrier MPEC penalty parameter. Knitro uses an internal formula to initialize the MPEC penalty parameter if a non-positive value is specified (default 0.0). See [bar_initpi_mpec](#).
- **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 (<i>x</i> unaffected if initialized by user)
2	initialization strategy 2 (<i>x</i> unaffected if initialized by user)
3	initialization strategy 3 (<i>x</i> unaffected if initialized by user)

- **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_slackboundpush**: minimum amount by which initial slack variables are pushed inside the bounds (default 1.0e-1). See *bar_slackboundpush*.
- **bar_switchobj**: the objective function to use when Knitro switches to feasibility phase in the barrier algorithms (default 1). See *bar_switchobj*.

Value	Description
0	no (or zero) objective function
1	proximal point objective scaled by a scalar value
2	proximal point objective using a diagonal scaling

- **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

- **bar_watchdog**: specify whether to enable watchdog heuristic for barrier algorithms (default 0). See [bar_watchdog](#).

Value	Description
0	do not apply watchdog heuristic
1	enable watchdog heuristic

- **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](#).
- **bndrange**: max limit for finite bounds (default 1e20). See [bndrange](#).
- **cg_maxit**: maximum allowable conjugate gradient (CG) iterations (default 0). See [cg_maxit](#).

Value	Description
0	let Knitro set the number based on the problem size
n	maximum of $n > 0$ CG iterations per minor iteration

- **cg_pmem**: amount of memory n used for the incomplete Cholesky preconditioner where n enforces the maximum number of nonzero elements per column in the matrix (default 10). See [cg_pmem](#).
- **cg_precond**: whether or not to apply an incomplete Cholesky preconditioner for the CG subproblems in the barrier algorithms (default 0). See [cg_precond](#).

Value	Description
0	no preconditioner used
1	apply incomplete Cholesky preconditioner

- **cg_stoptol**: relative stopping tolerance for CG subproblems (default 1.0e-2). See [cg_stoptol](#).
- **convex**: declare the problem as convex (default 0). See [convex](#).

Value	Description
0	Knitro will try to automatically determine convexity
1	Knitro will treat the problem as convex

- **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*.
- **fstopval**: stop based on user-defined function value (default KN_INFINITY). See *fstopval*.
- **ftol**: stop based on small (feasible) changes in the objective function (default 1.0e-15). See *ftol*.
- **ftol_iters**: stop based on small (feasible) changes in the objective function (default 5). See *ftol_iters*.
- **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*.
- **linesearch**: linesearch strategy for algorithms using a linesearch (default 0). See *linesearch*.

Value	Description
0	let Knitro automatically choose the strategy
1	use a simple backtracking linesearch
2	use a cubic interpolation linesearch

- **linesearch_maxtrials**: maximum number of linesearch trial evaluations (default 3). See *linesearch_maxtrials*.
- **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)

- **linsolver_pivottol**: initial pivot threshold for matrix factorizations (default 1.0e-8). See [linsolver_pivottol](#).
- **lmsize**: number of limited-memory pairs stored in LBFGS approach (default 10). See [lmsize](#).
- **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 <code>knitro_ma_*.log</code>

- **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
3	terminate after first completed optimization (any termination status)

- **maxfevals**: maximum number of function evaluations before terminating (default unlimited). See [maxfevals](#).
- **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_heuristic_terminate**: heuristic termination condition (default 1). See *mip_heuristic_terminate*.

Value	Description
1	terminate at first feasible point or iteration limit
2	always run the heuristic to the iteration limit

- **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_intvar_strategy**: treatment of integer variables (default 0). See *mip_intvar_strategy*.

Value	Description
0	no special treatment
1	relax all integer variables
2	convert all binary variables to complementarity constraints

- **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
3	mixed-integer SQP method

- **mip_nodealg**: standard node relaxation algorithm (default 0). See *mip_nodealg*.

Value	Description
0	let Knitro decide the node algorithm
1	Interior/Direct (barrier) algorithm
2	Interior/CG (barrier) algorithm
3	Active Set algorithm
4	SQP algorithm
5	Run multiple algorithms

- **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_relaxable**: are integer variables relaxable? (default 1). See *mip_relaxable*.

Value	Description
0	integer variables are not relaxable
1	all integer variables are relaxable

- **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
5	Run multiple algorithms

- **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_selectdir**: MIP node selection direction (default 0). See *mip_selectdir*.

Value	Description
0	choose down (i.e. \leq) node first
1	choose up (i.e. \geq) node first

- **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 <i>ms_terminate</i> = 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 <code>knitro_ms_*.log</code>

- **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_startprange**: maximum range to vary all x when generating start points (default 1.0e20). See [ms_startprange](#).
- **ms_terminate**: termination condition for multi-start (default 0). See [ms_terminate](#).

Value	Description
0	terminate after <code>ms_maxsolves</code>
1	terminate at first local optimum (if before <code>ms_maxsolves</code>)
2	terminate at first feasible solution (if before <code>ms_maxsolves</code>)
3	terminate after first completed optimization (any termination status)

- **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 <code>knitro_newpoint.log</code>
2	append all new points to file <code>knitro_newpoint.log</code>

- **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](#).
- **out_csvinfo**: create `knitro_solve.csv` information file (default 0). See [out_csvinfo](#).

Value	Description
0	do not create solve information file
1	create solve information file

- **out_csvname**: custom name for csv information file (default `knitro_solve.csv`). See [out_csvname](#).
- **out_hints**: print diagnostic hints at the end of the solve (default 1). See [out_hints](#).

Value	Description
0	do not print hints
1	print hints

- **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 (sensitivities)

- **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 <code>knitro.log</code>
2	print to both the screen and file <code>knitro.log</code>

- **outname**: custom name for Knitro log file (default `knitro.log`). See *outname*.
- **par_blasnumthreads**: specify the number of threads to use for BLAS (default 1). See *par_blasnumthreads*.

Value	Description
1	for any non-positive value
<i>n</i>	use $n > 0$ threads

- **par_lnumthreads**: specify the number of threads to use for linear system solves (default 1). See *par_lnumthreads*.

Value	Description
1	for any non-positive value
<i>n</i>	use $n > 0$ threads

- **par_mnumthreads**: specify the number of threads to use for multistart (default 0). See *par_mnumthreads*.

Value	Description
0	let Knitro choose the number of threads
<i>n</i>	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 <code>\$OMP_NUM_THREADS</code>
<i>n</i>	use $n > 0$ threads

- **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*.
- **restarts**: enable automatic restarts (default 0). See *restarts*.

Value	Description
0	do not enable automatic restarts
n	maximum of $n > 0$ restarts allowed

- **restarts_maxit**: maximum number of iterations before enforcing a restart (default 0). See *restarts_maxit*.
- **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 <code>knitro_tuner_summary.log</code>
2	write detailed algorithm output to files named <code>knitro_tuner_*.log</code>

- **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
3	terminate after first completed optimization (any termination status)

- **xtol**: stepsize termination tolerance (default 1.0e-15). See *xtol*.
- **xtol_iters**: stop based on small changes in the solution estimate. See *xtol_iters*.

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 or satisfactory solution
```

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 or satisfactory solution.
100	Current feasible solution estimate cannot be improved. Nearly optimal.
101	Relative change in feasible solution estimate < xtol.
102	Current feasible solution estimate cannot be improved.
103	Relative change in feasible objective < ftol for ftol_iters.
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.
402	Function evaluation 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.
412	Function evaluation 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	Terminated after derivative check.
506	Input or other API error.
507	Internal Knitro error.
508	Unknown termination.
509	Illegal objno value.

For more information on return codes, see [Return codes](#).

3.1.3 AMPL suffixes defined for Knitro

To represent values associated with a model component, AMPL employs various qualifiers or suffixes appended to component names. A suffix consists of a period or “dot” (.) followed by a short identifier (ex: `x1.lb` returns the current lower bound of the variable `x1`).

A lot of built-in suffixes are available in AMPL, you may find the list at <http://www.ampl.com/NEW/suffbuiltin.html>.

To allow more solver-specific results of optimization, AMPL permits solver drivers to define new suffixes and to associate solution result information with them. Below is the list of the suffixes defined specifically for Knitro.

Suffix Name	Description	Model component
honorbnd	Specify variables that must always satisfy bounds; see <i>honorbnds</i> (input)	variable
intvarstrategy	Treatment of integer variables; see <i>mip_intvar_strategy</i> (input)	variable
cfeastol	Specify individual constraint feasibility tolerances (input)	constraint
xfeastol	Specify individual variable bound feasibility tolerances (input)	variable
xscalefactor	Specify custom variable scaling factors (input)	variable
xscalecenter	Specify custom variable scaling centers (input)	variable
cscalefactor	Specify custom constraint scaling factors (input)	constraint
objscalefactor	Specify custom objective scaling factor (input)	objective
relaxbnd	Retrieve the best relaxation bound for MIP (output)	objective
incumbent	Retrieve the incumbent solution for MIP (output)	objective
priority	Specify branch priorities for MIP (input)	variable
numiters	Retrieve the number of iterations (output)	objective
numfcevals	Retrieve the number of function evaluations (output)	objective
opterror	Retrieve the final optimality error (output)	objective, variable, constraint
feaserror	Retrieve the final feasibility error (output)	objective, variable, constraint

Below is an example on how to use the specific Knitro suffixes in AMPL:

```

1 var x{j in 1..3} >= 0;
2
3 minimize obj: 1000 - x[1]^2 - 2*x[2]^2 - x[3]^2 - x[1]*x[2] - x[1]*x[3];
4
5 s.t. c1: 8*x[1] + 14*x[2] + 7*x[3] - 56 = 0;
6
7 s.t. c2: x[1]^2 + x[2]^2 + x[3]^2 -25 >= 0;
8
9 suffix xfeastol IN, >=0, <=1e6;
10 suffix cfeastol IN, >=0, <=1e6;
11 suffix objscalefactor IN, >=1e-6, <=1e6;
12
13 let x[1].xfeastol := 1e-1;
14 let c1.cfeastol := 1e-2;
15 let obj.objscalefactor := 2;
16
17 solve;
18
19 display x[1].feaserror;
20 display c1.opterror;
21 display obj.numfcevals;
22 display obj.feaserror;
23 display obj.opterror;

```

Below is the corresponding output:

```

Final Statistics
-----
Final objective value           = 9.51000000020162e+002
Final feasibility error (abs / rel) = 7.11e-015 / 4.55e-016
Final optimality error (abs / rel) = 3.84e-009 / 1.37e-010
# of iterations                 = 9

```

```
# of CG iterations          =          2
# of function evaluations  =          12
# of gradient evaluations  =          11
# of Hessian evaluations   =           9
Total program time (secs)  =          0.035 (    0.000 CPU time)
Time spent in evaluations (secs) =          0.000

=====

Locally optimal or satisfactory solution.
objective 951; feasibility error 7.11e-15
9 iterations; 12 function evaluations

suffix feaserror OUT;
suffix opterror OUT;
suffix numfcevals OUT;
suffix numiters OUT;
x[1].feaserror = 0

c1.opterror = 0

obj.numfcevals = 12

obj.feaserror = 7.10543e-15

obj.opterror = 3.84018e-09
```

3.1.4 Nonlinear Least Squares

In some cases it may be more efficient to use the specialized Knitro API for nonlinear least-squares (see *Nonlinear Least Squares*), which internally applies the Gauss-Newton Hessian, to solve a least-squares model formulated in AMPL. In particular this may be useful if the exact Hessian computed by AMPL is expensive. You can apply this specialized interface through AMPL by following these steps:

- Set the objective function to 0
- Specify each residual function as an equality constraint
- Turn the AMPL presolver off by setting

```
option presolve 0;
```

- Tell Knitro to apply the least-squares interface and disable presolve by setting

```
option knitro_options "leastquares=1 presolve=0";
```

Below is an example of how to solve nonlinear least-squares problems in AMPL:

```
1 #####
2 #####          LSQ in AMPL with Knitro          #####
3 #####
4 ##### This example illustrates how to optimize least #####
5 ##### squares problems in AMPL by formulating it using #####
6 ##### AMPL syntax and also using Knitro least squares #####
7 ##### dedicated API. #####
8 #####
9 #####
```

```

10 # Reset AMPL
11 reset;
12
13 # Reset initial guesses between consecutive runs
14 option reset_initial_guesses 1;
15
16 # Reinitialize random seed for generating same values over runs
17 option randseed 1;
18
19 ### The first part of the example will demonstrate how to formulate a
20 ### least squares problem in AMPL using usual AMPL syntax.
21 ### Also, we will illustrate an AMPL trick to improve performances.
22
23 # We use a large number to demonstrate the AMPL expansion trick
24 param M := 1000000;
25
26 # Create random values for the "estimates"
27 param alpha{1..M};
28 let{i in 1..M} alpha[i] := Uniform01();
29
30 # Variable: minimize the sum of squares of the distance between var_alpha
31 # and the "estimates"
32 var var_alpha;
33
34 ### 1. Straightforward least squares formulation with no expansion ###
35
36 ## Straightforward least square problem.
37 ## The objective is expressed directly, without expanding the square terms.
38 minimize obj_no_expand:
39     0.5 * sum{i in 1..M} (alpha[i]-var_alpha)^2;
40
41 # Optimize non-expanded problem
42 solve obj_no_expand;
43
44
45 ### 2. Least squares with square terms expansion ###
46
47 ## Same problem but this time the objective is expanded.
48 ## Notice that, using this trick, the runtime decreases significantly.
49 minimize obj_expanded:
50     0.5 * (
51         M * var_alpha^2 -
52         2 * var_alpha * ( sum{i in 1..M} alpha[i] ) +
53         sum{i in 1..M} alpha[i]^2
54     );
55
56 # Optimize expanded problem
57 solve obj_expanded;
58
59 # Check objective value
60 display obj_expanded - obj_no_expand;
61
62
63 ### 3. Least squares using Knitro LSQ API ###
64
65 # Set Ampl and Knitro options
66 option presolve 0; # disable AMPL presolve, this is mandatory!
67 option knitro_options "least squares=1 presolve=0"; # Enable Knitro LSQ

```

```

68
69 ## Same problem but this time based on Knitro's least-squares API.
70 # Objective must be constant
71 minimize obj_lsqr: 0;
72
73 # Each residual is a constraint: residual = 0
74 # s.t. res{i in 1..M}:(alpha[i]-var_alpha)^2 = 0;
75 s.t. res{i in 1..M}:
76     alpha[i] - var_alpha = 0;
77
78 # Optimize problem using on Knitro LSQ API
79 solve obj_lsqr;

```

Below is the corresponding (filtered) output:

```

[...]

  Iter      Objective      FeasError      OptError      ||Step||      CGits
-----
    0  1.665516e+005  0.000e+000
    1  4.168899e+004  0.000e+000  2.754e-013  4.997e-001      0

EXIT: Locally optimal solution found.

Final Statistics
-----
Final objective value           = 4.16889869527361e+004
Final feasibility error (abs / rel) = 0.00e+000 / 0.00e+000
Final optimality error (abs / rel) = 2.75e-013 / 3.30e-014
# of iterations                 = 1
# of CG iterations              = 0
# of function evaluations       = 4
# of gradient evaluations       = 3
# of Hessian evaluations        = 1
Total program time (secs)       = 0.452 ( 0.453 CPU time)
Time spent in evaluations (secs) = 0.274

=====

[...]

  Iter      Objective      FeasError      OptError      ||Step||      CGits
-----
    0  1.665516e+005  0.000e+000
    1  4.168899e+004  0.000e+000  0.000e+000  4.997e-001      0

EXIT: Locally optimal solution found.

Final Statistics
-----
Final objective value           = 4.16889869527314e+004
Final feasibility error (abs / rel) = 0.00e+000 / 0.00e+000
Final optimality error (abs / rel) = 0.00e+000 / 0.00e+000
# of iterations                 = 1
# of CG iterations              = 0
# of function evaluations       = 4
# of gradient evaluations       = 3
# of Hessian evaluations        = 1

```

```

Total program time (secs)      =      0.002 (      0.000 CPU time)
Time spent in evaluations (secs) =      0.000

=====

[...]

  Iter      Objective      FeasError      OptError      ||Step||      CGits
-----
      0      1.665516e+005      0.000e+000
      1      4.168899e+004      0.000e+000      1.376e-009      4.997e-001      0

EXIT: Locally optimal solution found.

Final Statistics
-----
Final objective value          = 4.16889869527361e+004
Final feasibility error (abs / rel) = 0.00e+000 / 0.00e+000
Final optimality error (abs / rel) = 1.38e-009 / 3.30e-014
# of iterations                = 1
# of CG iterations             = 0
# of residual evaluations      = 4
# of Jacobian evaluations      = 2
Total program time (secs)      = 0.301 (      0.500 CPU time)
Time spent in evaluations (secs) = 0.163

```

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 *ktlink* interface. *knitrolink* has the same input and output arguments as *ktlink*, but it is equivalent to using *knitromatlab* with an empty array for the value of *extendedFeatures*. If you are using *knitrolink*, please switch to *knitromatlab* as the *knitrolink* function will be deprecated in a future release.

3.2.3 Input Arguments

Input Argument	Description
<i>fun</i>	The function to be minimized. <i>fun</i> 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.
<i>x0</i>	The initial point vector.
<i>A</i>	Linear inequality constraint coefficient matrix.
<i>b</i>	Linear inequality constraint upper bound vector.
<i>Aeq</i>	Linear equality constraint coefficient matrix.
<i>beq</i>	Linear equality constraint right-hand side vector.
<i>lb</i>	Variable lower bound vector.
<i>ub</i>	Variable upper bound vector.
<i>nonlcon</i>	The function that computes the nonlinear inequality and equality constraints. <i>nonlcon</i> 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.

<p><i>extendedFeatures</i></p>	<p>The structure used to define other, extended modeling features of Knitro.</p> <p>Currently it is used for complementarity constraints, the output function, parallel finite differencing, initial lambda values, custom finite-difference step sizes, custom feasibility tolerances, custom scalings, custom treatments of integer variables, specification of linear variables, custom setting of honor bounds, and some Hessian and Jacobian information.</p> <p>The two complementarity constraint fields are <i>extendedFeatures.ccIndexList1</i> and <i>extendedFeatures.ccIndexList2</i> which contain the variable index lists for variables complementary to each other. The same index may not appear more than once in the lists.</p> <p>The <i>lambdaInitial</i> field allows the user to specify the initial lambda values in a structure with a different field for each constraint type. The fields are <i>ineqlin</i> for linear inequality constraints, <i>eqlin</i> for linear equality constraints, <i>ineqnonlin</i> for nonlinear inequality constraints, <i>eqnonlin</i> for nonlinear equality constraints, <i>upper</i> for upper bounds of variables, and <i>lower</i> 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.</p> <p>The <i>FinDiffRelStep</i> field can be used to specify custom step size values for finite-differencing.</p> <p>The <i>LinearVars</i> field can be used to specify which variables in the model only appear linearly in the objective and constraints. This information can then be used to perform additional presolve operations. See KTR_set_linearvars() for more details.</p> <p>The <i>HonorBnds</i> field can be used to specify which variables in the model must satisfy their bounds throughout the optimization. See KTR_set_honorbnds() for more details.</p> <p>The fields <i>AFeasTols</i>, <i>AeqFeasTols</i>, <i>cFeasTols</i>, <i>ceqFeasTols</i>, and <i>xFeasTols</i> can be used to specify custom feasibility tolerances for problem constraints and variables.</p> <p>The fields <i>AScaleFactors</i>, <i>AeqScaleFactors</i>, <i>cScaleFactors</i>, <i>ceqScaleFactors</i>, <i>xScaleFactors</i>, <i>xScaleCenters</i>, and <i>objScaleFactor</i> can be used to specify custom scalings for problem constraints, variables, and the objective. See KTR_set_var_scalings(), KTR_set_con_scalings() and KTR_set_obj_scaling() for details on how these scalings should be defined.</p> <p>The field <i>xIntStrategy</i> can be used to specify customized treatments for integer variables. See KTR_mip_set_intvar_strategy() for more details.</p> <p>The other six available fields are <i>JacobPattern</i>, <i>HessPattern</i>, <i>HessFcn</i>, <i>HessMult</i>, <i>OutputFcn</i>, and <i>UseParallel</i>. They have the same properties as the options set by <i>optimset</i>.</p>
--------------------------------	--

<i>options</i>	The options structure set with <i>optimset</i> .
<i>knitroOpts</i>	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
<i>x</i>	The optimal solution vector.
<i>fval</i>	The optimal solution objective value.
<i>exitflag</i>	Integer identifying the reason for termination of the algorithm.
<i>output</i>	Structure containing information about the optimization.
<i>lambda</i>	Structure containing the Lagrange multipliers at the solution with a different field for each constraint type.
<i>grad</i>	Gradient vector at <i>x</i> .
<i>hessian</i>	Hessian matrix at <i>x</i> . For notes on when this option is available, see <i>KTR_get_hessian_values()</i> .

3.2.5 Output Structure Fields

Output Argument	Field	Description
<i>output</i>	<i>iterations</i>	Number of iterations.
<i>output</i>	<i>funcCount</i>	Number of function evaluations.
<i>output</i>	<i>constrviolation</i>	Maximum of constraint violations.
<i>output</i>	<i>firstorderopt</i>	Measure of first-order optimality.
<i>lambda</i>	<i>lower</i>	Lower bounds
<i>lambda</i>	<i>upper</i>	Upper bounds
<i>lambda</i>	<i>ineqlin</i>	Linear inequalities
<i>lambda</i>	<i>eqlin</i>	Linear equalities
<i>lambda</i>	<i>ineqnonlin</i>	Nonlinear inequalities
<i>lambda</i>	<i>eqnonlin</i>	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 Option	Knitro	Description
<i>Algorithm</i>	<i>algorithm</i>		The optimization algorithm: <i>'interior-point'</i> , <i>'active-set'</i> , or <i>'sqp'</i> . Default: <i>'interior-point'</i>
<i>AlwaysHonorConstraints</i>	<i>honorbnds</i>		Bounds are satisfied at every iteration if set to the default <i>'bounds'</i> . They are not necessarily satisfied if set to <i>'none'</i> .
<i>DerivativeCheck</i>	<i>derivcheck</i>		<p>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 <i>'off'</i> (default) or <i>'on'</i>.</p> <p>The default relative tolerance is 1e-6, but can be changed with the <i>derivcheck_tol</i> option in the Knitro options file. The finite difference method is set by <i>FinDiffType</i>, and is set to <i>'forward'</i> by default.</p>
<i>Display</i>	<i>outlev</i>		<p>Level of display</p> <ul style="list-style-type: none"> <i>'off'</i> or <i>'none'</i> displays no output. <i>'iter'</i> displays information for each iteration, and gives the default exit message. <i>'iter-detailed'</i> displays information for each iteration, and gives the technical exit message. <i>'notify'</i> displays output only if the function does not converge, and gives the default exit message. <i>'notify-detailed'</i> displays output only if the function does not converge, and gives the technical exit message. <i>'final'</i> (default) displays just the final output, and gives the default exit message. <i>'final-detailed'</i> displays just the final output, and gives the technical exit message.
<i>FinDiffType</i>	<i>gradopt,</i> <i>derivcheck_type</i>		<p>The finite difference type is either <i>'forward'</i> (default) or <i>'central'</i>. <i>'central'</i> takes twice as many function evaluations and may violate bounds during evaluation, but is usually more accurate.</p> <p>When exact derivatives are used and <i>DerivativeCheck</i> is used, this option sets the finite difference type to <i>'forward'</i> (default) or <i>'central'</i> to compare with the exact derivatives.</p>

Option	Equivalent Option	Knitro	Description
<i>GradConstr</i>	<i>gradopt</i>		<p>Gradient for nonlinear constraint functions.</p> <ul style="list-style-type: none"> • ‘<i>off</i>’ (default) sets the algorithm to use finite differences to estimate the gradients of nonlinear constraints. • ‘<i>on</i>’ sets the algorithm to expect exact gradients of the nonlinear constraints in the third and fourth constraint function outputs, as described for <i>nonlcon</i> in the Input Arguments section. <p>To use sparse gradients, the sparsity pattern must be set with the <i>JacobPattern</i> option.</p>
<i>GradObj</i>	<i>gradopt</i>		<p>Gradient for nonlinear objective function.</p> <ul style="list-style-type: none"> • ‘<i>off</i>’ (default) sets the algorithm to use finite differences to estimate the gradients of the objective function. • ‘<i>on</i>’ sets the algorithm to expect exact gradients of the objective in the second objective function outputs, as described for <i>fval</i> in the Input Arguments section.
<i>HessFcn</i>			Function handle to the user-supplied Hessian. Default: []
<i>Hessian</i>	<i>hessopt</i>		Sets the Hessian option for Knitro. Default: ‘ <i>bfgs</i> ’
<i>HessMult</i>			Handle to a user-supplied function that returns a Hessian-times-vector product. Default: []
<i>HessPattern</i>			Sparsity pattern of Hessian. Default: ‘ <i>sparse(ones(n))</i> ’
<i>InitBarrierParam</i>	<i>bar_initmu</i>		Initial barrier value. Default: <i>0.1</i>
<i>InitTrustRegionRadius</i>	<i>delta</i>		Initial radius of the trust region. Default: ‘ <i>sqrt(n)</i> ’
<i>JacobPattern</i>			Sparsity pattern of the Jacobian of the nonlinear constraint matrix. It can be used for finite-differencing or for user-supplied gradients. Default: ‘ <i>sparse(ones(Jrows,Jcols))</i> ’
<i>MaxFunEvals</i>	<i>maxfevals</i>		Maximum number of function evaluations allowed. Default: <i>-1</i> (no limit)
<i>MaxIter</i>	<i>maxit</i>		Maximum number of iterations allowed. Default: <i>10000</i>
<i>MaxProjCGIter</i>	<i>cg_maxit</i>		Tolerance for the number of projected conjugate gradient iterations. Default: ‘ <i>2*(n-numberOfEqualities)</i> ’
<i>ObjectiveLimit</i>	<i>objrange</i>		Specifies the extreme limits of the objective function for purposes of determining unboundedness. If the magnitude of the objective function becomes greater than <i>objrange</i> for a feasible iterate, then the problem is determined to be unbounded and Knitro proceeds no further. Default: <i>1.0e20</i>
<i>OutputFcn</i>	<i>newpoint</i> <i>KTR_set_newpt_callback</i>		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 Option	Knitro	Description
<i>ScaleProblem</i>	<i>scale</i>		The default value of <i>'obj-and-con'</i> allows Knitro to scale the objective and constraint functions based on their values at the initial point. Setting the option to <i>'none'</i> disables scaling. If scaling is performed, all internal computations, including the stopping tests, are based on the scaled values.
<i>SubproblemAlgorithm</i>	<i>algorithm</i>		Determines how the iteration step is calculated. The default option is <i>'ldl-factorization'</i> , (in Knitro this is a symmetric, indefinite factorization) which is usually faster than the alternative, <i>'cg'</i> (conjugate gradient). Conjugate gradient may be faster for large problems with dense Hessians.
<i>TolCon</i>	<i>feastol</i>		Termination tolerance on the constraint violation. Default: <i>1.0e-6</i>
<i>TolFun</i>	<i>opttol</i>		Termination tolerance on the function value. Default: <i>1.0e-6</i>
<i>TolX</i>	<i>xtol</i>		Termination tolerance on <i>x</i> . Default: <i>1.0e-6</i>
<i>UseParallel</i>			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 <i>"par_numthreads"</i> does not have an effect on parallel finite differences in MATLAB. The MATLAB command <i>"parpool(n)"</i> will set the number of workers to the minimum of <i>n</i> and the maximum number allowed, which can be set in the cluster profile. If the parallel pool is not started before <i>knitromatlab</i> 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: <i>false</i>

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
<i>lambda</i>	Structure containing the Lagrange multipliers at the solution with a different field for each constraint type.
<i>fval</i>	The objective value at x .
<i>c</i>	The nonlinear inequality constraint values at x .
<i>ceq</i>	The nonlinear equality constraint values at x .
<i>gradient</i>	The gradient vector at x .
<i>cineqjac</i>	The nonlinear inequality constraint Jacobian matrix.
<i>ceqjac</i>	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, knitroOpts)
```

but the simplest function call reduces to:

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

3.2.14 Input Arguments for `knitromatlab_lsqnonlin`

Input Argument	Description
<i>fun</i>	The function whose sum of squares is minimized. <i>fun</i> accepts a vector <i>x</i> and returns a vector <i>F</i> , the values of the functions evaluated at <i>x</i> . If exact gradients are used, an additional matrix should be returned with the Jacobian for the function at <i>x</i> . Unlike the user-supplied Jacobian for <i>knitromatlab</i> , the entries <i>J(i,j)</i> of the Jacobian for <i>knitromatlab_lsqnonlin</i> represent the partial derivative of the function component <i>i</i> with respect to variable <i>j</i> .
<i>x0</i>	The initial point vector.
<i>lb</i>	Variable lower bound vector.
<i>ub</i>	Variable upper bound vector.
<i>extendedFeatures</i>	The structure used to define other, extended modeling features of Knitro. It is similar to the <code>extendedFeatures</code> input to <i>knitromatlab</i> , but currently it is only used for the <i>JacobPattern</i> (rows are the function components and columns are the variables) and <i>OutputFcn</i> fields.
<i>options</i>	The options structure set with <i>optimset</i> . See details below.
<i>knitroOpts</i>	The text file with Knitro options.

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*.

3.2.15 Nonlinear System of Equations

There is another special function, *knitromatlab_fsolve*, for using Knitro to solve nonlinear systems of equations. It behaves similarly to the *fsolve* function in the MATLAB Optimization Toolbox. Note that the *extendedFeatures* structure is not an input argument of *fsolve*, but is the argument before *options* in *knitromatlab_fsolve*. If the structure is not used, an empty matrix, [], should be used in its place.

The most elaborate form is:

```
[x, fval, exitflag, output, jacobian] = ...
    knitromatlab_fsolve(fun, x0, extendedFeatures, options, knitroOpts)
```

but the simplest function call reduces to:

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

3.2.16 Input Arguments for *knitromatlab_fsolve*

Input Argument	Description
<i>fun</i>	The function whose components are to be solved to equal zero. <i>fun</i> accepts a vector <i>x</i> and returns a vector <i>F</i> , the values of the functions evaluated at <i>x</i> . If exact gradients are used, an additional matrix should be returned with the Jacobian for the function at <i>x</i> . Unlike the user-supplied Jacobian for <i>knitromatlab</i> , the entries <i>J(i,j)</i> of the Jacobian for <i>knitromatlab_fsolve</i> represent the partial derivative of the function component <i>i</i> with respect to variable <i>j</i> .
<i>x0</i>	The initial point vector.
<i>extendedFeatures</i>	The structure used to define other, extended modeling features of Knitro. It is similar to the <i>extendedFeatures</i> input to <i>knitromatlab</i> , but currently it is only used for the <i>JacobPattern</i> (rows are the function components and columns are the variables) and <i>OutputFcn</i> fields.
<i>options</i>	The options structure set with <i>optimset</i> . See details below.
<i>knitroOpts</i>	The text file with Knitro options.

The options available in the *options* structure are the same as those available in *knitromatlab*, except the differences noted here.

- *GradObj*, *HessFcn*, *HessMult*, and *HessPattern* are not available. *Hessian* is available, except when using the ‘*levenberg-marquardt*’ algorithm option, but may only be set to ‘*bfgs*’ (default), ‘*lbfgs*’, or ‘*fin-diff-grads*’.
- *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.
- In addition to the four standard Knitro algorithm options, *Algorithm* may also be set to ‘*trust-region-dogleg*’ (default), or ‘*levenberg-marquardt*’. Using ‘*trust-region-dogleg*’ is equivalent to using ‘*interior-point*’ with ‘*SubproblemMethod*’ set to ‘*cg*’ (Knitro algorithm 2). Using ‘*levenberg-marquardt*’ is equivalent to using *knitromatlab_lsqrnonlin*.

3.2.17 Note on exit flags

The returned exit flags will correspond with Knitro's return code, rather than matching *fmincon*'s exit flags.

3.2.18 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 feasible solution estimate cannot be improved. Nearly optimal.
-101	Relative change in feasible solution estimate < <code>xtol</code> .
-102	Current feasible solution estimate cannot be improved.
-103	Relative change in feasible objective < <code>ftol</code> for <code>ftol_iters</code> .
-200	Convergence to an infeasible point. Problem may be locally infeasible.
-201	Relative change in infeasible solution estimate < <code>xtol</code> .
-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.
-402	Function evaluation 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.
-412	Function evaluation 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 to -522	Input or other API error.
-523	Derivative check failed.
-524	Derivative check finished.
-600	Internal Knitro error.

For more information on return codes, see [Return codes](#).

3.3 Knitro / R reference

Usage of *KnitroR* is described here.

3.3.1 What is *KnitroR*?

KnitroR is the interface used to call Knitro from the R environment (requires R 3.0 or later).

KnitroR offers several routines to call Knitro from R:

- *knitro*, which is an interface for the standard Knitro nonlinear optimizer
- *knitrolsq*, which solves bound constrained nonlinear least-squares problems in vectorial format, that is

$$\min_p 0.5 \cdot \|F(X, p) - Y\|_2^2$$

$$p_L \leq p \leq p_U.$$

This type of fitting problems is ubiquitous in statistics and data analytics. This function is based on the internal Knitro implementation of nonlinear least-squares.

- *knitromip*, which is tailored to solve mixed-integer nonlinear programs
- An important point when passing sparse matrices (jacobian of the nonlinear constraints or hessian of Lagrangian) to Knitro is that indices of the sparse matrix vectors must start from 0.

3.3.2 Syntax

The most elaborate form for general nonlinear programs is

```
sol <- knitro(nvar=..., ncon=..., x0=...,
             objective=..., gradient=..., constraints=...,
             jacobian=..., jacIndexCons=..., jacIndexVars=...,
             hessianLag=..., hessIndexRows=..., hessIndexCols=...,
             xL=..., xU=..., cL=..., cU=...,
             options=...)
```

but the simplest function calls reduce to:

```
sol <- knitro(objective=..., x0=...)
sol <- knitro(objective=..., xL=...)
sol <- knitro(objective=..., xU=...)
sol <- knitro(nvar=..., objective=...)
```

You must provide an objective function and the number of variables or a vector that is used to compute the number of variables. All other parameters are optional. For instance any of the following other forms may be used (note that this list is not exhaustive):

```
sol <- knitro(nvar=..., objective=...)
sol <- knitro(nvar=..., ncon=..., objective=..., constraints=...)
sol <- knitro(x0=..., objective=..., constraints=...)
sol <- knitro(x0=..., objective=..., gradient=..., constraints=...,
             jacobian=..., jacIndexCons=..., jacIndexVars=...)
sol <- knitro(x0=..., objective=..., gradient=..., constraints=...,
             jacobian=..., hessianLag=...)
```

```

sol <- knitro(x0=..., objective=..., gradient=..., constraints=...,
             jacobian=..., jacBitMap=...)
sol <- knitro(x0=..., objective=..., gradient=..., constraints=...,
             jacobian=..., jacIndexCons=..., jacIndexVars=...,
             hessianLag=..., hessIndexRows=..., hessIndexCols=...)
sol <- knitro(x0=..., objective=..., gradient=..., constraints=...,
             jacobian=..., jacBitMap=...,
             hessianLag=..., hessBitMap=...)
sol <- knitro(x0=..., objective=..., xL=...)
sol <- knitro(x0=..., objective=..., xL=..., xU=...)
sol <- knitro(x0=..., objective=..., constraints=..., cL=...)
sol <- knitro(x0=..., objective=..., constraints=..., cU=...)
sol <- knitro(x0=..., objective=..., constraints=..., cL=..., cU=...)
sol <- knitro(x0=..., objective=..., xL=..., xU=..., constraints=...)
sol <- knitro(x0=..., objective=..., gradient=..., constraints=...,
             jacobian=..., jacIndexCons=..., jacIndexVars=...,
             hessianLag=..., hessIndexRows=..., hessIndexCols=..., options=...)
sol <- knitro(nvar=..., objective=..., options=...)

```

When the functions *jacobian* or *hessianLag* are provided but the corresponding sparsity vectors *jacIndexCons*, *jacIndexVars*, *jacBitMap*, *hessIndexRows*, *hessIndexCols* or *hessBitMap* are not, *KnitroR* computes dense sparsity patterns internally.

An API for MPECs is also available via the R function *Knitro*. It takes the following form :

```

sol <- knitro(nvar=..., ncon=..., x0=..., objective=..., gradient=..., constraints=...
↪,
             jacobian=..., jacIndexCons=..., jacIndexVars=...,
             hessianLag=..., hessIndexRows=..., hessIndexCols=...,
             xL=..., xU=..., cL=..., cU=...,
             numCompConstraints=..., ccIdxList1=..., ccIdxList2=...,
             options=...)

```

The API for bound constrained nonlinear least-squares problems has the following form :

```

sol <- knitrolsq(dimp=..., par0=..., dataFrameX=..., dataFrameY=...,
               residual=..., jacobian=...,
               parL=..., parU=...,
               xScaleFactors=..., xScaleCenters=...,
               objScaleFactor=...,
               jacIndexRows=..., jacIndexCols=...,
               options=..., optionsFile=...)

```

The API for mixed-integer NLPs is

```

sol <- knitromip(nvar=..., ncon=..., x0=...,
                objective=..., gradient=..., constraints=...,
                jacobian=..., jacIndexCons=..., jacIndexVars=...,
                hessianLag=..., hessIndexRows=..., hessIndexCols=...,
                xL=..., xU=..., cL=..., cU=...,
                xType=..., cFnType=..., objfntype=...,
                options=...)

```

The following other forms may be used:

```

sol <- knitromip(nvar=..., ncon=..., x0=...,
                objective=..., gradient=..., constraints=...,
                jacobian=..., jacBitMap=...,

```

```
        hessianLag=..., hessBitMap=...,
        xL=..., xU=..., cL=..., cU=...,
        xType=..., cFnType=..., objfnType=...,
        options=...)
sol <- knitromip(nvar=..., ncon=..., x0=...,
               objective=..., gradient=..., constraints=...,
               jacobian=..., hessianLag=...,
               xL=..., xU=..., cL=..., cU=...,
               xType=..., cFnType=..., objfnType=...,
               options=...)
```

3.3.3 Input Arguments of *knitro*

Below is a description of the input arguments of the *knitro* function of *KnitroR*.

Input Argument	Description
<i>objective</i>	Objective function to be minimized or maximized. <i>objective</i> accepts a vector x and returns a scalar f , the objective function evaluated at x .
<i>gradient</i>	Gradient of the objective function. <i>gradient</i> accepts a vector x and returns a vector g (using the R function $c(\dots)$)
<i>constraints</i>	Nonlinear constraints function. <i>constraints</i> accepts a vector x and returns a vector c (using the R function $c(\dots)$)
<i>jacobian</i>	Jacobian of nonlinear constraints function, as a sparse matrix. <i>jacobian</i> accepts a vector x and returns a vector containing the nonzero elements of the jacobian. Via R function $c(\dots)$
<i>jacIndexCons</i>	Vector containing the row indices of the nonzero elements of the jacobian Via R function $c(\dots)$
<i>jacIndexVars</i>	Vector containing the column indices of the nonzero elements of the jacobian Via R function $c(\dots)$
<i>jacBitMap</i>	Vector containing 1 or 0 depending on whether jacobian element is non-zero or not. Vector elements are supposed to be given in row major. Via R function $c(\dots)$
<i>hessianLag</i>	Hessian of Lagrangian, as a sparse matrix. Vector of nonzero elements, via R function $c(\dots)$
<i>hessIndexRows</i>	Row indices of nonzero elements of hessian of Lagrangian. Via R function $c(\dots)$
<i>hessIndexCols</i>	Column indices of nonzero elements of hessian of Lagrangian. Via R function $c(\dots)$
<i>hessBitMap</i>	Vector containing 1 or 0 depending on whether hessian element is non-zero or not. Vector elements are supposed to be given in row major. Via R function $c(\dots)$
<i>printPrimal</i>	User-defined function to print primal iterate after every Knitro iteration. Activated by setting option <code>newpoint=3</code> .
<i>printDual</i>	User-defined function to print dual iterate after every Knitro iteration. Activated by setting option <code>newpoint=3</code> .
<i>printObjective</i>	User-defined function to print objective after every Knitro iteration. Activated by setting option <code>newpoint=3</code> .
<i>printGradient</i>	User-defined function to display gradient after every Knitro iteration. Activated by setting option <code>newpoint=3</code> .
<i>printConstraints</i>	User-defined function to display constraints after every Knitro iteration. Activated by setting option <code>newpoint=3</code> .
<i>printJacobian</i>	User-defined function to display jacobian after every Knitro iteration. Activated by setting option <code>newpoint=3</code> .

<i>xL</i>	Vector of lower bounds on variables.
<i>xU</i>	Vector of upper bounds on variables.
<i>cL</i>	Vector of lower bounds on constraints.
<i>cU</i>	Vector of upper bounds on constraints.
<i>xScaleFactors</i>	Vector of scaling factors on variables.
<i>xScaleCenters</i>	Vector of scaling centers on variables.
<i>cScaleFactors</i>	Vector of scaling factors on nonlinear constraints.
<i>ccScaleFactors</i>	Vector of scaling factors on complementarity constraints.
<i>objScaleFactor</i>	Scaling factor on objective function.
<i>constraint-Types</i>	Constraint types (general, linear or quadratic).
<i>options</i>	Options stored as an R list. They can also be set in a file named <code>ktrOptions.opt</code> via the field <code>optionsFile</code> .
<i>optionsFile</i>	Options file. The user can specify all basic KNITRO options via a file with extension <code>.opt</code> .

3.3.4 Input Arguments for MPECS in *knitro*

Below is a description of the input arguments, which are specific to MPECS in the *knitro* function of *KnitroR*.

Input Argument	Description
<i>numCompConstraints</i>	Number of complementarity constraints among nonlinear constraints.
<i>ccIdxList1</i>	List of indices of first variables in complementarity constraints. Via R function <code>c(...)</code>
<i>ccIdxList2</i>	List of indices of second variables in complementarity constraints. Via R function <code>c(...)</code>

3.3.5 Input Arguments for *knitromip*

Below is a description of the input arguments, which are specific to MINLPs in the *knitromip* function of *KnitroR*.

Input Argument	Description
<i>xType</i>	R vector indicating variable types: 0 (continuous), 1 (integer) or 2 (binary).
<i>cFnType</i>	R vector indicating constraint types: 0 (convex), 1 (nonconvex) or 2 (uncertain)
<i>objFn-Type</i>	Type of objective function: 0 (convex), 1 (nonconvex) or 2 (uncertain)
<i>options</i>	List of options which are specific to MINLPs. <code>xPriorities</code> : branching priorities for integer variables <code>mipBranchRule</code> : branching rule in Branch-and-Bound algorithm <code>mipHeuristic</code> : heuristic to find initial integer feasible solution <code>mipMethod</code> : 0 (automatic choice), 1 (Branch-and-Bound), 2 (Quesada-Grossmann) <code>mipLPalg</code> : algorithm for LP subproblem.

3.3.6 Input Arguments for *knitrolsq*

Below is a descriptions of the API of the R function *knitrolsq* in *KnitroR*. The function *knitrolsq* is based on Knitro's internal implementation of bound-constrained nonlinear least-squares.

Input Argument	Description
<i>dimp</i>	Dimension of fitting parameter.
<i>par0</i>	Initial guess on fitting parameter.
<i>dataFrameX</i>	Data frame containing samples x_i .
<i>dataFrameY</i>	Data frame containing samples y_i .
<i>residual</i>	Vector of nonlinear least-squares residuals (R function returning $c(\dots)$).
<i>jacobian</i>	Jacobian of nonlinear least-squares residuals <i>residual</i> (R function returning $c(\dots)$).
<i>parL</i>	Vector of lower bounds on fitting parameter (R vector).
<i>parU</i>	Vector of upper bounds on fitting parameter (R vector).
<i>xScaleCenters</i>	Vector of scaling centers on fitting parameter. (R vector).
<i>xScaleFactors</i>	Vector of scaling factors on fitting parameter. (R vector).
<i>objScaleFactor</i>	Scaling factor on nonlienar least-squares objective function.

3.3.7 Output Arguments of *knitro* and *knitromip*

Output Argument	Description
<i>statusMessage</i>	Knitro's status message.
<i>x</i>	Locally optimal primal solution.
<i>lambda</i>	Locally optimal dual solution.
<i>objective</i>	Objective value at the optimal solution x .
<i>constraints</i>	Constraints value at the optimal solution x .
<i>objEval</i>	Number of objective evaluations.
<i>gradEval</i>	Number of gradient evaluations.

3.3.8 Output Arguments of *knitrolsq*

Output Argument	Description
<i>statusMessage</i>	Knitro's status message.
<i>paramFit</i>	Locally optimal solution for fitting parameter.
<i>objective</i>	Least-squares objective value at the optimal solution <i>paramFit</i> .
<i>iter</i>	Number of iterations.
<i>objEval</i>	Number of objective evaluations.
<i>gradEval</i>	Number of gradient evaluations.

3.4 Object-oriented interface reference

3.4.1 What is the object-oriented interface?

The object-oriented interface provides the functionality of the callable library with an easy-to-use set of classes. The interface is available in C++, C#, and Java. The problem definition is contained in a class definition and is simpler: variable and constraint properties can be defined more compactly; memory for the problem characteristics does not need to be allocated; and Knitro API functions are simplified with some of the arguments handled internally within the classes.

This guide focuses on the C++ version of the interface.

Simple examples of the object-oriented interface can be found in *User guide*. More complex examples can be found in the examples folder.

3.4.2 Getting started with the Java object-oriented interface

Java interfaces are distributed as a JAR with additional packages containing Javadoc, sources and dependencies.

Java interfaces require using Java 6 or higher and Java Native Access library, which is also provided.

Import JAR files and dependencies within your project in order to enable using Knitro with Java interfaces.

Examples can be compiled and run from your favorite IDE or using the provided makefile.

3.4.3 Getting started with the C# object-oriented interface

C# interface requires .net 4.0 or higher.

A project is distributed as a Microsoft Visual Studio solution with all the sources and examples.

In order to run, the project only needs to have `KNITRODIR` environment variable set to the Knitro directory.

To modify the example run by Visual Studio, right click on “KTRC” project then select “Properties” and modify “Startup object” (in Application tab).

3.4.4 Getting started with the C++ object-oriented interface

The C++ object oriented interface is distributed as a header library which is straightforward to include within your projects.

Examples are provided within a CMake project. Please refer to the README file in `examples/C++` directory for more details.

The following sections and subsequent references to object-oriented interfaces within the documentation, use C++ interface methods and classes.

3.4.5 Defining a problem

This section describes how to define a problem in the object-oriented interface by implementing the abstract `KTRProblem` class. The `KTRProblem` class inherits from the `KTRIPProblem` class and defines several functions that make implementing the problem easier. Users should consult `KTRIPProblem.h` for more information on how to implement the `KTRIPProblem` class if the `KTRProblem` is not used.

Minimal required implementation

In order to define an optimization problem, a problem class that inherits from `KTRProblem` must be defined by the user. A class should at least:

- pass the number of variables to the `KTRProblem` constructor.
- pass the number of constraints to the `KTRProblem` constructor.
- set variable upper and lower bounds with `KTRProblem::setVarLoBnds()` and `KTRProblem::setVarUpBnds()`
- set constraint upper and lower bounds with `KTRProblem::setConLoBnds()` and `KTRProblem::setConUpBnds()`
- set constraint types with `KTRProblem::setConTypes()`
- set the objective type with `KTRProblem::setObjType()`

- set the objective goal with `KTRProblem::setObjGoal()`

It should also define evaluation functions; at minimum

- define the function `double KTRProblem::evaluateFC()`, evaluating the objective function and constraints, setting the constraint values in the function parameter `std::vector<double> c` and returning the objective function value.

If possible, the user should also

- pass the number of non-zero elements of the Jacobian to the `KTRProblem` constructor
- set the Jacobian sparsity pattern with `KTRProblem::setJacIndexCons()` and `KTRProblem::setJacIndexVars()`.

If these are not known, a dense pattern will automatically be set, but a sparsity pattern can help solver performance significantly regardless of whether exact first derivatives are implemented or not.

The functions `KTRProblem::setXInitial()` and `KTRProblem::setLambdaInitial()` can respectively be used to set values for the initial primal and dual variable values. If these values are not set, Knitro will automatically determine initial values.

Implementing a MIP problem

If the problem has integer or binary variables, the following must also be defined:

- set variable types with `KTRProblem::setVarTypes()`
- set constraint function types with `KTRProblem::setConFnTypes()`
- set the objective function type with `KTRProblem::setObjFnType()`

Implementing derivatives

Like the callable library, the object-oriented interface does not compute derivatives automatically.

To evaluate first derivatives exactly, the problem class should define `double KTRProblem::evaluateGA()`, evaluating the gradient and Jacobian and setting their values in the function parameters `std::vector<double> objGrad` and `std::vector<double> jac`. The function should return 0 to indicate that no error occurred, or `KTR_RC_CALLBACK_ERR` can be returned; this return code will stop the Knitro solver.

To evaluate second derivatives exactly, the problem class should:

- pass the number of non-zero elements of the Hessian to the `KTRProblem` constructor
- define the Hessian sparsity pattern with `KTRProblem::setHessIndexCols()` and `KTRProblem::setHessIndexRows()`.

The problem class should define the function `int KTRProblem::evaluateHess()` to set the value of the Hessian in the parameter `hess`. The function should return 0 to indicate that no error occurred, or `KTR_RC_CALLBACK_ERR` can be returned; this return code will stop the Knitro solver.

To evaluate the Hessian-vector product, the problem class should define the function `int KTRProblem::evaluateHessianVector()` to set the value of the Hessian-vector product in the parameter `vector`. The function should return 0 to indicate that no error occurred, or `KTR_RC_CALLBACK_ERR` can be returned; this return code will stop the Knitro solver.

Complementarity Constraints

Complementarity constraints can be specified in the object-oriented interface by passing the lists of complementary variables to the function:

```
KTRIPProblem::setComplementarity(const std::vector<int>& indexList1,
                                const std::vector<int>& indexList2)
```

3.4.6 Using the KTRSolver class to solve a problem

Once a problem is defined by inheriting from `KTRIPProblem` or `KTRProblem`, the `KTRSolver` class is used to call Knitro to solve the problem. This class is also used to set most Knitro parameters, and access solution information after Knitro has completed solving the problem.

To use the `KTRSolver` class, one of four constructors can be used. Each of the constructors takes at least a pointer to a `KTRIPProblem` object (each `KTRSolver` object is associated with one problem definition. If different problems are to be solved, multiple `KTRSolver` objects are needed).

```
explicit KTRSolver(KTRIPProblem * problem);
```

This constructor should be used when using exact gradient and Hessian evaluation, which must be defined in the `KTRIPProblem` object.

```
KTRSolver(KTRIPProblem * problem, int gradopt, int hessopt);
```

This constructor should be used when specifying a gradient and Hessian evaluation other than the default exact gradient and Hessian evaluations.

For both of these constructors, a pointer to a ZLM object can also be passed as an additional argument, when using a network license of Knitro with the Artelys License Manager. Otherwise, a local Knitro license is used.

Once the solver object is created, Knitro options can be set with `KTRSolver::setParam()`, or by loading a parameters file with `KTRSolver::loadParamFile()`. Finally, the function `KTRSolver::solve()` will call Knitro to solve the problem and will return a solution status code.

`KTRSolver::solve()` can be called multiple times. Between each call to `solve()`, two types of changes can be made:

- `KTRSolver::setParam()` can be used to change problem parameters, except for the gradient and Hessian evaluation types.
- Variable bounds can be changed by calling `KTRIPProblem::setVarLoBnds()` and `KTRIPProblem::setVarUpBnds()` in the problem object that the solver object points to.

3.4.7 Accessing callable library functions from the object-oriented interface

The object-oriented interface provides access to the Knitro callable library functions. The table below shows the correspondence between callable library functions and object-oriented interface functions. Note that in the C# interface, function names are capitalized keeping with C# convention.

The majority of the functions are accessed directly through `KTRSolver` methods, or in the case of the callback setting functions, `KTRIPProblem` methods. There are a few major differences between the callable library functions and the object-oriented interface methods:

- The callable library methods take a `KTR_context_ptr` argument (created from a call to `KTR_new()`), which holds problem information. The object-oriented interface methods do not take this argument, storing the necessary information in the `KTRSolver` object.

- The callable library methods return status codes, with a non-zero status code usually indicating an error. The object-oriented interface methods (with the exception of `KTRSolver::solve()`) do not return status codes. If the methods encounter an error, usually related to an invalid Knitro license or invalid function arguments, a `KTRException` is thrown.
- Several callable library methods, such as `KTR_get_constraint_values()`, modify input parameters. Instead, the object-oriented interface methods return these values as output parameters (rather than returning status codes).
- Function arguments use `std::vector<>` (C++), `IList<>` (Java), or `List<>` (C#) instead of C-style (pointer) arrays. Instead of character arrays, functions use `std::string` (C++), or `String` (Java and C#).

Callable Library Function	Object-Oriented Interface Methods
<code>KTR_new()</code>	Not necessary - problem information stored in <code>KTRSolver</code> object.
<code>KTR_new_puts()</code>	Not necessary - redirect output by inheriting from <code>KTRPuts</code> class.
<code>KTR_free()</code>	Not necessary - problem information stored in <code>KTRSolver</code> object.
<code>KTR_reset_params_to_defaults()</code>	<code>KTRSolver::resetParamsToDefaults()</code>
<code>KTR_load_param_file()</code>	<code>KTRSolver::loadParamFile()</code>
<code>KTR_save_param_file()</code>	<code>KTRSolver::saveParamFile()</code>
<code>KTR_set_int_param_by_name()</code>	<code>KTRSolver::setParam()</code>
<code>KTR_set_char_param_by_name()</code>	<code>KTRSolver::setParam()</code>
<code>KTR_set_double_param_by_name()</code>	<code>KTRSolver::setParam()</code>
<code>KTR_set_int_param()</code>	<code>KTRSolver::setParam()</code>
<code>KTR_set_char_param()</code>	<code>KTRSolver::setParam()</code>
<code>KTR_set_double_param()</code>	<code>KTRSolver::setParam()</code>
<code>KTR_get_int_param_by_name()</code>	<code>KTRSolver::getIntParam()</code>
<code>KTR_get_double_param_by_name()</code>	<code>KTRSolver::getDoubleParam()</code>
<code>KTR_get_int_param()</code>	<code>KTRSolver::getIntParam()</code>
<code>KTR_get_double_param()</code>	<code>KTRSolver::getDoubleParam()</code>
<code>KTR_get_param_name()</code>	<code>KTRSolver::getParamName()</code>
<code>KTR_get_param_doc()</code>	<code>KTRSolver::getParamDoc()</code>
<code>KTR_get_param_type()</code>	<code>KTRSolver::getParamType()</code>
<code>KTR_get_num_param_values()</code>	<code>KTRSolver::getNumParamValues()</code>
<code>KTR_get_param_value_doc()</code>	<code>KTRSolver::getParamValueDoc()</code>
<code>KTR_get_param_id()</code>	<code>KTRSolver::getParamID()</code>
<code>KTR_get_release()</code>	<code>KTRSolver::getRelease()</code>
<code>KTR_load_tuner_file()</code>	<code>KTRSolver::loadTunerFile()</code>
<code>KTR_set_feastols()</code>	<code>KTRSolver::setFeastols()</code>
<code>KTR_set_names()</code>	<code>KTRSolver::setNames()</code>
<code>KTR_set_compcons()</code>	Not necessary - define complementarity constraints in the <code>KTRIPProblem</code> class constructor.
<code>KTR_chgvarbnds()</code>	Not necessary - change variable bounds in a <code>KTRIPProblem</code> object.
<code>KTR_init_problem()</code>	Not necessary - problem initialized in <code>KTRSolver</code> constructor.
<code>KTR_solve()</code>	<code>KTRSolver::solve()</code>
<code>KTR_restart()</code>	<code>KTRSolver::restart()</code>
<code>KTR_mip_init_problem()</code>	Not necessary - problem initialized in <code>KTRSolver</code> constructor.
<code>KTR_mip_set_branching_priorities()</code>	<code>KTRSolver::mipSetBranchingPriorities()</code>

Continued on next page

Table 3.3 – continued from previous page

Callable Library Function	Object-Oriented Interface Methods
KTR_mip_solve()	KTRSolver::solve()
KTR_set_findiff_relstepsizes()	KTRSolver::setFindiffRelstepsizes()
KTR_set_func_callback()	Not necessary - define function evaluation in class that inherits from KTRIPProblem()
KTR_set_grad_callback()	Not necessary - define gradient evaluation in class that inherits from KTRIPProblem()
KTR_set_hess_callback()	Not necessary - define Hessian evaluation in class that inherits from KTRIPProblem()
KTR_set_newpt_callback()	KTRIPProblem::setNewPointCallback()
KTR_set_ms_process_callback()	KTRIPProblem::setMSProcessCallback()
KTR_set_mip_node_callback()	KTRIPProblem::setMipNodeCallback()
KTR_set_ms_initpt_callback()	KTRIPProblem::setMSInitptCallback()
KTR_set_puts_callback()	KTRIPProblem::setPutsCallback()
KTR_get_number_FC_evals()	KTRSolver::getNumberFCEvals()
KTR_get_number_GA_evals()	KTRSolver::getNumberGAEvals()
KTR_get_number_H_evals()	KTRSolver::getNumberHEvals()
KTR_get_number_HV_evals()	KTRSolver::getNumberHVEvals()
KTR_get_number_iters()	KTRSolver::getNumberIters()
KTR_get_number_cg_iters()	KTRSolver::getNumberCGIters()
KTR_get_abs_feas_error()	KTRSolver::getAbsFeasError()
KTR_get_rel_feas_error()	KTRSolver::getRelFeasError()
KTR_get_abs_opt_error()	KTRSolver::getAbsOptError()
KTR_get_rel_opt_error()	KTRSolver::getRelOptError()
KTR_get_solution()	KTRSolver::getSolution()
KTR_get_constraint_values()	KTRSolver::getConstraintValues()
KTR_get_objgrad_values()	KTRSolver::getObjgradValues()
KTR_get_jacobian_values()	KTRSolver::getJacobianValues()
KTR_get_hessian_values()	KTRSolver::getHessianValues()
KTR_get_mip_num_nodes()	KTRSolver::getMipNumNodes()
KTR_get_mip_num_solves()	KTRSolver::getMipNumSolves()
KTR_get_mip_abs_gap()	KTRSolver::getMipAbsGap()
KTR_get_mip_rel_gap()	KTRSolver::getMipRelGap()
KTR_get_mip_incumbent_obj()	KTRSolver::getMipIncumbentObj()
KTR_get_mip_relaxation_bnd()	KTRSolver::getMipRelaxationBnd()
KTR_get_mip_lastnode_obj()	KTRSolver::getMipLastnodeObj()
KTR_get_mip_incumbent_x()	KTRSolver::getMipIncumbentX()
KTR_set_var_scaling()	KTRSolver::setVarScaling()
KTR_set_con_scaling()	KTRSolver::setConScaling()
KTR_set_obj_scaling()	KTRSolver::setObjScaling()
KTR_set_int_var_strategy()	KTRSolver::setIntVarStrategy()
No callable library equivalent	KTRSolver::setIntegralityRelaxed()
double * obj set by KTR_solve()	KTRSolver::getObj()
double * x set by KTR_solve()	KTRSolver::getXValues()
double * lambda set by KTR_solve()	KTRSolver::getLambdaValues()
KTR_check_first_ders()	Deprecated - set derivative check options with KTR-Solver::setParam().

3.4.8 Callbacks in the object-oriented interface

The object-oriented interface supports all of the callbacks that are supported by the callable library: MIP node callbacks; multi-start initial point callbacks; multi-start process callbacks; new point callbacks, and Knitro output redirection callbacks.

Each of these callbacks can be implemented by extending the appropriate callback class, and passing the callback object to a `KTRProblem` object via the function `KTRProblem::set {Callbacktype}` (for some callback type).

The callback functionality is the same as described in the callable library reference section.

Below, we show an example of implementing `KTRNewptCallback`. This type of callback is called by Knitro during the problem iteration whenever Knitro finds a new estimate of the solution point (i.e., after each major iteration). This callback cannot modify any of its arguments, but can provide information about the solve before it is completed. In this example, the callback prints the number of objective function and constraint evaluations.

The following defines the callback, inheriting from `KTRNewptCallback`.

```
#include <iostream>
#include "KTRNewptCallback.h"
#include "KTRSolver.h"

class ExampleNewPointCallback : public knitro::KTRNewptCallback {
public:

    int CallbackFunction(const std::vector<double>& x, const std::vector<double>&_
↳lambda,
                        double obj,
                        const std::vector<double>& c, const std::vector<double>&_
↳objGrad,
                        const std::vector<double>& jac,
                        knitro::KTRSolver * solver)
    {
        int n = x.size();
        std::cout << ">> New point computed by Knitro: (";
        for (int i = 0; i < n - 1; i++) {
            std::cout << x[i] << ", ";
        }

        std::cout << x[n - 1] << std::endl;

        std::cout << "Number FC evals= " << solver->getNumberFCEvals() << std::endl;
        std::cout << "Current feasError= " << solver->getAbsFeasError() << std::endl;

        return 0;
    }
};
```

To use this callback, it should be passed to the `KTRProblem` object, before passing it to the `KTRSolver` constructor. This is shown below. The problem solved is the same example problem solved in previous sections, but the callback defined above is independent of the problem solved.

```
#include "KTRSolver.h"
#include "ProblemQCQP.h"
#include "ExampleNewPointCallback.h"
#include "ExampleHelpers.h"

int main() {
    // Create a problem instance.
```

```

ProblemQCQP instance;

ExampleNewPointCallback callback;

instance.setNewPointCallback(&callback);

// Create a solver
knitro::KTRSolver solver(&instance, KTR_GRADOPT_FORWARD, KTR_HESSOPT_BFGS);
solver.useNewptCallback();

int solveStatus = solver.solve();

printSolutionResults(solver, solveStatus);

return 0;
}

```

Calling this function gives the following output, showing additional information each time Knitro finds a new estimate of the solution value:

```

=====
      Commercial License
      Artelys Knitro 10.1.0
=====

Knitro performing finite-difference gradient computation with 1 thread.
Knitro presolve eliminated 0 variables and 0 constraints.

gradopt:          2
hessopt:          2
newpoint:         3
The problem is identified as a QCQP.
Knitro changing algorithm from AUTO to 1.
Knitro changing bar_initpt from AUTO to 3.
Knitro changing bar_murule from AUTO to 4.
Knitro changing bar_penaltycons from AUTO to 1.
Knitro changing bar_penaltyrule from AUTO to 2.
Knitro changing bar_switchrule from AUTO to 2.
Knitro changing linsolver from AUTO to 2.
Knitro performing finite-difference gradient computation with 1 thread.

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

```

```

  Iter      Objective      FeasError      OptError      ||Step||      CGits
-----
      0      9.760000e+02      1.300e+01
>> New point computed by Knitro: (3.8794, 0.01, 3.69211
Number FC evals= 12
Current feasError= 1.01993
>> New point computed by Knitro: (3.86709, 5e-05, 3.71871
Number FC evals= 16
Current feasError= 0.968364
>> New point computed by Knitro: (3.21473, 2.5e-07, 4.34569
Number FC evals= 20
Current feasError= 0.137681
>> New point computed by Knitro: (0.0160737, 7.84537e-08, 7.99343
Number FC evals= 24
Current feasError= 0.0826197
>> New point computed by Knitro: (8.03683e-05, 7.8344e-08, 8.01171
Number FC evals= 28
Current feasError= 0.08261
>> New point computed by Knitro: (4.01842e-07, 7.25596e-08, 8.01118
Number FC evals= 32
Current feasError= 0.078234
>> New point computed by Knitro: (2.00921e-09, 4.9857e-10, 8.00003
Number FC evals= 36
Current feasError= 0.000202022
>> New point computed by Knitro: (9.71398e-10, 3.11247e-10, 8
Number FC evals= 40
Current feasError= 6.53699e-13
>> New point computed by Knitro: (1.65208e-11, 5.31022e-12, 8
Number FC evals= 44
Current feasError= 7.10543e-15
      9      9.360000e+02      7.105e-15      1.374e-07      1.976e-09      0

EXIT: Locally optimal solution found.

Final Statistics
-----
Final objective value          = 9.360000000000340e+02
Final feasibility error (abs / rel) = 7.11e-15 / 5.47e-16
Final optimality error (abs / rel) = 1.37e-07 / 8.59e-09
# of iterations                = 9
# of CG iterations             = 0
# of function evaluations      = 44
# of gradient evaluations      = 0
Total program time (secs)      = 0.010 ( 0.016 CPU time)
Time spent in evaluations (secs) = 0.010

=====

Knitro successful, feasibility violation = 7.10543e-15
KKT optimality violation = 1.37375e-07

```

Below we give an example of how the `KTRSolver::setIntVarStrategy()` callback can be used to reformulate an MINLP.

```

#include "KTRSolver.h"
#include "ProblemMINLP.h"
#include "ExampleHelpers.h"

```

```
/**
 * An example of loading and solving a MINLP problem.
 * Sets MIP parameters using parameter string names to choose the solution algorithm.
 */
int main() {
    // Create a problem instance.
    ProblemMINLP instance;

    // Create a solver
    knitro::KTRSolver solver(&instance);

    solver.setParam("mip_method", KTR_MIP_METHOD_BB);
    solver.setParam("algorithm", KTR_ALG_ACT_CG);
    solver.setIntVarStrategy(4, KTR_MIP_INTVAR_STRATEGY_RELAX );
    solver.setIntVarStrategy(5, KTR_MIP_INTVAR_STRATEGY_RELAX );

    int solveStatus = solver.solve();

    printSolutionResults(solver, solveStatus);

    return 0;
}
```

These examples and examples of other callbacks can be found in the examples directory.

3.4.9 Changing variable bounds in the object-oriented interface

In both the object-oriented interface and the callable library, a problem can be solved multiple times, but the only problem characteristics that can be changed between solves are the variable bounds. The object-oriented interface differs from the callable library in how variable bounds are changed. In the object-oriented interface, variable bounds are set in a `KTRIPProblem` object. When `KTRSolver::solve()` is called to solve the problem, the variable bounds in the `KTRIPProblem` object are passed to the solver. the following example shows changing variable bounds between calls to `KTRSolver::solve`.

```
// Create a problem instance.
ProblemQCQP instance;

// Create a solver
knitro::KTRSolver solver(&instance);

solver.solve();

// changing upper bounds makes previous optimal solution infeasible
instance.setVarUpBnds(7.0);

solver.solve();
```

In this example, the problem characteristics (including variable bounds) are initialized in the constructor of `ProblemQCQP`. After the first call of `KTRSolver::solve()`, the variable bounds are changed in the problem instance with the `KTRIPProblem::setVarUpBnds()` function. This function sets all variable bounds to 7. When `solver.solve()` is called for a second time, the solve function calls `KTRIPProblem::getVarLoBnds()` and `KTRIPProblem::getVarUpBnds()` and updates them. Note that although other `KTRIPProblem` functions can be called after the `KTRIPProblem` object is passed to the `KTRSolver` constructor, all changes except variable bounds are ignored. In addition to variable bounds, Knitro parameters except for the gradient and Hessian evaluation type can be changed between calls to `solve`.

3.4.10 Using the Artelys License Manager with the object-oriented interface

The object-oriented interface can be used with either a standalone Knitro license or a network Knitro license using the Artelys License Manager (ALM) and a license server.

In order to use the ALM with the object-oriented interface, the license server needs to be installed and the user machine (from which Knitro is run) should be configured to find the license server. For information on installing and configuring the ALM, see the Artelys License Manager User's Manual.

To use the ALM with the object-oriented interface, a `ZLM` object needs to be created and passed to a `KTRSolver` object constructor. When the `ZLM` object is created, a network license will be checked out for use and unavailable for other users until the `ZLM` object is destroyed.

An example usage is shown below. This example is identical to (and produces the same output as) the other examples of the object-oriented interface, except for the instantiation of the `ZLM` object and the `KTRSolver` constructor that takes a pointer to the `ZLM` object.

```
knitro::ZLM zlm;

// Create a problem instance.
ProblemQCQP instance = ProblemQCQP();

// Create a solver
knitro::KTRSolver solver(&zlm, &instance);

int solveStatus = solver.solve();

printSolutionResults(solver, solveStatus);
```

3.5 Callable library API 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. In addition, the examples provided with the Knitro distribution highlight most of the key features of the API and are a good starting point.

3.5.1 Introduction and Philosophy

Knitro 11.0 introduces a completely new callable library API. For information on the old Knitro API prior to this release, please see *Knitro 10.x and Earlier Callable Library API*, and the header file `ktr.h`. The old API is still supported for compatibility purposes. However, we recommend using the new API described in this section whenever possible. The old API may be deprecated in the future. In addition, not all new features will be available through the old API. In the new API, functions, types, defines, macros, etc. begin with `KN_`, whereas in the old API they begin with `KTR_`.

The new callable library API for Knitro, introduced with version 11.0, is designed to provide you maximum flexibility and ease-of-use in building a model. In addition, and just as importantly, it is designed to provide Knitro a great amount of structural information about your model, so that Knitro can exploit special structures wherever possible to improve performance. The API is designed so that you can build up a model in pieces based on what is most convenient for you. This means not only allowing you to add constraints one at a time (or in several blocks), but also allowing you to add special structures within constraints separately if desired.

The API is designed so that you can load constant, linear, quadratic, and conic structures as well as complementarity constraints separately. This allows Knitro to mark these structure types internally and provide special care to different structure types. For example, conic structures and complementarities are notoriously difficult if not handled specially.

In addition, the more structural information Knitro has, the more extensive presolve operations Knitro can perform to try to simplify the model internally. For this reason we always recommend making use of the API functions to provide as much fine-grained structural information as possible to Knitro. More general nonlinear structure must be handled through callback evaluation routines.

Structures of the same type can be added in individual pieces, in groups, or all together. Likewise, general nonlinear structures can all be handled by one callback object or broken up into separate callback objects if there are natural groupings that you want to treat differently. For example, you may be able to provide a callback routine to evaluate the exact analytic derivatives for some group of nonlinear constraints, while having Knitro approximate the derivatives for another group of nonlinear constraints using finite-differences. Implementing such a scheme is possible in the new API.

The overhead costs for loading your model by pieces should be trivial in most cases – even for large models. However, if not, it is always possible – and most efficient – to load all the structures of one type together in one API function call.

All functions offered by the Knitro callable library are described in detail below.

3.5.2 Index

Here is a summary of Knitro API functions grouped by functionality.

Creating and destroying solver objects

API function name	Purpose
<i>KN_new()</i>	Create a new Knitro solver object
<i>KN_free()</i>	Free/destroy an existing Knitro solver object

Changing and reading solver parameters

API function name	Purpose
<i>KN_reset_params_to_defaults()</i>	Reset all user options to their default values
<i>KN_load_param_file()</i>	Read user options from a Knitro options file
<i>KN_load_tuner_file()</i>	Read user options and values to explore for Knitro-Tuner
<i>KN_save_param_file()</i>	Write all current user options to a file
<i>KN_set_int_param_by_name()</i>	Set integer valued option using its string name
<i>KN_set_char_param_by_name()</i>	Set character valued option using its string name
<i>KN_set_double_param_by_name()</i>	Set double valued option using its string name
<i>KN_set_param_by_name()</i>	Set integer or doubled valued option using its string name
<i>KN_set_int_param()</i>	Set integer valued option using its integer identifier
<i>KN_set_char_param()</i>	Set character valued option using its integer identifier
<i>KN_set_double_param()</i>	Set double valued option using its integer identifier
<i>KN_get_int_param_by_name()</i>	Get integer valued option using its string name
<i>KN_get_double_param_by_name()</i>	Get double valued option using its string name
<i>KN_get_int_param()</i>	Get integer valued option using its integer identifier
<i>KN_get_double_param()</i>	Get double valued option using its integer identifier
<i>KN_get_param_name()</i>	Get string name associated with user option
<i>KN_get_param_doc()</i>	Get documentation string associated with user option
<i>KN_get_param_type()</i>	Get type (KN_PARAM_TYPE_*) associated with user option
<i>KN_get_num_param_values()</i>	Get number of possible values associated with integer valued user option
<i>KN_get_param_value_doc()</i>	Get documentation string associated with user option value
<i>KN_get_param_id()</i>	Get integer identifier associated with user option

Basic problem construction

API function name	Purpose
<code>KN_add_vars()</code>	Add new variables to a model
<code>KN_add_cons()</code>	Add new constraints to a model
<code>KN_add_rsds()</code>	Add new residuals to a least-squares model
<code>KN_set_var_lobnds()</code>	Set lower bounds on variables
<code>KN_set_var_upbnds()</code>	Set upper bounds on variables
<code>KN_set_var_fxbnds()</code>	Set fixed bounds on variables
<code>KN_set_var_types()</code>	Set variable types (e.g., continuous, integer, etc)
<code>KN_set_var_properties()</code>	Set variable properties (e.g., linear)
<code>KN_set_con_lobnds()</code>	Set lower bounds on constraints
<code>KN_set_con_upbnds()</code>	Set upper bounds on constraints
<code>KN_set_con_eqbnds()</code>	Set equality bounds on constraints
<code>KN_set_con_properties()</code>	Set constraint properties (e.g., convex)
<code>KN_set_obj_property()</code>	Set objective properties (e.g., convex)
<code>KN_set_obj_goal()</code>	Specify minimize or maximize
<code>KN_set_var_primal_init_values()</code>	Set initial values for primal variables
<code>KN_set_var_dual_init_values()</code>	Set initial values for dual variables
<code>KN_set_con_dual_init_values()</code>	Set initial values for constraint multipliers

Adding constant structure

API function name	Purpose
<code>KN_add_obj_constant()</code>	Add a constant to the objective
<code>KN_add_con_constants()</code>	Add constants to the constraints
<code>KN_add_rsd_constants()</code>	Add constants to the residuals for least-squares models

Adding linear structure

API function name	Purpose
<code>KN_add_obj_linear_struct()</code>	Add linear structure to the objective
<code>KN_add_con_linear_struct()</code>	Add linear structure to the constraints
<code>KN_add_rsd_linear_struct()</code>	Add linear structure to the residuals for least-squares models

Adding quadratic structure

API function name	Purpose
<code>KN_add_obj_quadratic_struct()</code>	Add quadratic structure to the objective
<code>KN_add_con_quadratic_struct()</code>	Add quadratic structure to the constraints

Adding conic structure

API function name	Purpose
<code>KN_add_con_L2norm()</code>	Add L2norm structure used to define conic constraints

Adding complementarity constraints

API function name	Purpose
<code>KN_set_compcons()</code>	Set all complementarity constraints for a model

Adding evaluation callbacks

API function name	Purpose
<i>KN_add_eval_callback()</i>	Add a callback for nonlinear evaluations
<i>KN_add_eval_callback_all()</i>	Add a callback for nonlinear evaluations of all functions
<i>KN_add_eval_callback_one()</i>	Add a callback for nonlinear evaluations of one function
<i>KN_add_lsq_eval_callback()</i>	Add a callback for nonlinear least-squares evaluations
<i>KN_add_lsq_eval_callback_all()</i>	Add a callback for nonlinear least-squares evaluations of all functions
<i>KN_add_lsq_eval_callback_one()</i>	Add a callback for nonlinear least-squares evaluations of one function
<i>KN_set_cb_grad()</i>	Set callback for objective gradient and constraint Jacobian
<i>KN_set_cb_hess()</i>	Set callback for Hessian of the Lagrangian matrix
<i>KN_set_cb_rsd_jac()</i>	Set callback for least-squares residual Jacobian
<i>KN_set_cb_user_params()</i>	Set a user parameters structure for evaluation callback
<i>KN_set_cb_gradopt()</i>	Specify how to evaluate gradients in evaluation callback
<i>KN_set_cb_relstepsizes()</i>	Specify finite-difference relative stepsizes
<i>KN_get_cb_number_cons()</i>	Get the number of constraints evaluated through the callback
<i>KN_get_cb_number_rsds()</i>	Get the number of residuals evaluated through the callback
<i>KN_get_cb_objgrad_nnz()</i>	Get the number of non-zero objective gradient elements evaluated through the callback
<i>KN_get_cb_jacobian_nnz()</i>	Get the number of non-zero Jacobian elements evaluated through the callback
<i>KN_get_cb_rsd_jacobian_nnz()</i>	Get the number of non-zero residual Jacobian elements evaluated through the callback
<i>KN_get_cb_hessian_nnz()</i>	Get the number of non-zero Hessian elements evaluated through the callback

Other user callbacks

API function name	Purpose
<i>KN_set_newpt_callback()</i>	Callback to perform some user-defined task after new solution estimate
<i>KN_set_mip_node_callback()</i>	Callback to perform some user-defined task after MIP node solve
<i>KN_set_ms_process_callback()</i>	Callback to perform some user-defined task after multi-start solve
<i>KN_set_ms_initpt_callback()</i>	Callback to specify custom initial points for multi-start
<i>KN_set_puts_callback()</i>	Callback to specify custom handling of output

Other algorithmic/modeling features

API function name	Purpose
<i>KN_set_var_feastols()</i>	Set custom feasibility tolerances for variables
<i>KN_set_con_feastols()</i>	Set custom feasibility tolerances for constraints
<i>KN_set_compcon_feastols()</i>	Set custom feasibility tolerances for complementarity constraints
<i>KN_set_var_scalings()</i>	Set custom scalings for variables
<i>KN_set_con_scalings()</i>	Set custom scalings for constraints
<i>KN_set_compcon_scalings()</i>	Set custom scalings for complementarity constraints
<i>KN_set_obj_scaling()</i>	Set a custom scaling for the objective
<i>KN_set_var_names()</i>	Set names for variables
<i>KN_set_con_names()</i>	Set names for constraints
<i>KN_set_compcon_names()</i>	Set names for complementarity constraints
<i>KN_set_obj_name()</i>	Set a name for the objective
<i>KN_set_var_honorbnds()</i>	Enforce variables satisfy bounds throughout optimization
<i>KN_set_mip_branching_priorities()</i>	Set branching priorities for integer variables
<i>KN_set_mip_intvar_strategies()</i>	Set strategies for handling for integer variables

Solving

API function name	Purpose
<code>KN_solve()</code>	Call Knitro to solve/optimize the current model

Reading model/solution properties

API function name	Purpose
<code>KN_get_release()</code>	Get Knitro release number
<code>KN_get_number_vars()</code>	Get the number of variables in the model
<code>KN_get_number_cons()</code>	Get the number of constraints in the model
<code>KN_get_number_rsd()</code>	Get the number of residuals in the model
<code>KN_get_number_FC_evals()</code>	Get the number of function evaluations during the solve
<code>KN_get_number_GA_evals()</code>	Get the number of gradient evaluations during the solve
<code>KN_get_number_H_evals()</code>	Get the number of Hessian evaluations during the solve
<code>KN_get_number_HV_evals()</code>	Get the number of Hessian-vector product evaluations during the solve
<code>KN_get_solution()</code>	Get the solution status, objective and variables
<code>KN_get_obj_value()</code>	Get the value of the objective function
<code>KN_get_obj_type()</code>	Get the objective function type (e.g. linear, quadratic, general, etc.)
<code>KN_get_con_values()</code>	Get the value of the constraint functions
<code>KN_get_con_types()</code>	Get the constraint function types (e.g. linear, quadratic, general, etc.)
<code>KN_get_rsd_values()</code>	Get the value of the residual functions
<code>KN_get_number_iters()</code>	Get the number of iterations (continuous models only)
<code>KN_get_number_cg_iters()</code>	Get the number of conjugate gradient iterations (continuous models only)
<code>KN_get_abs_feas_error()</code>	Get the absolute feasibility error (continuous models only)
<code>KN_get_rel_feas_error()</code>	Get the relative feasibility error (continuous models only)
<code>KN_get_abs_opt_error()</code>	Get the absolute optimality error (continuous models only)
<code>KN_get_rel_opt_error()</code>	Get the relative optimality error (continuous models only)
<code>KN_get_objgrad_values()</code>	Get the objective gradient values (continuous models only)
<code>KN_get_objgrad_values_all()</code>	Get the objective gradient values in dense form (continuous models only)
<code>KN_get_jacobian_values()</code>	Get the constraint Jacobian values (continuous models only)
<code>KN_get_rsd_jacobian_values()</code>	Get the residual Jacobian values (continuous models only)
<code>KN_get_hessian_values()</code>	Get the Hessian values (continuous models only)
<code>KN_get_mip_number_nodes()</code>	Get the number of MIP nodes explored (MIP models only)
<code>KN_get_mip_number_solves()</code>	Get the number of MIP subproblem solves (MIP models only)
<code>KN_get_mip_abs_gap()</code>	Get the absolute integrality gap (MIP models only)
<code>KN_get_mip_rel_gap()</code>	Get the relative integrality gap (MIP models only)
<code>KN_get_mip_incumbent_obj()</code>	Get the objective value of the incumbent solution (MIP models only)
<code>KN_get_mip_relaxation_bnd()</code>	Get the current relaxation bound (MIP models only)
<code>KN_get_mip_lastnode_obj()</code>	Get the objective value from the most recently solved node (MIP models only)
<code>KN_get_mip_incumbent_x()</code>	Get the MIP incumbent solution variables (MIP models only)

`KN_get_release()`

```
int KNITRO_API KN_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. Returns 0 if OK, nonzero if error.

3.5.3 Creating and destroying solver objects

KN_new()

```
int KNITRO_API KN_new (KN_context_ptr * kc);
```

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 floating network license handler, then this call also checks out a license and reserves it until *KN_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 0 if OK, nonzero if error.

KN_free()

```
int KNITRO_API KN_free (KN_context_ptr * kc);
```

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.

3.5.4 Changing and reading solver parameters

With the exception of the *hessopt* user option, all parameters can be changed between successive calls to *KN_solve()*.

Note: The *hessopt* user option cannot be changed after calling *KN_solve()*. You must first call *KN_free()* and then reload the model before changing *hessopt* and solving again.

All methods return 0 if OK, nonzero if there was an error. In most cases, parameter values are not validated until *KN_solve()* is called.

KN_reset_params_to_defaults()

```
int KNITRO_API KN_reset_params_to_defaults (KN_context_ptr kc);
```

Reset all parameters to default values.

KN_load_param_file()

```
int KNITRO_API KN_load_param_file  
(KN_context_ptr kc, const char * const filename);
```

Set all parameters specified in the given file.

KN_load_tuner_file()

```
int KNITRO_API KN_load_tuner_file  
(KN_context_ptr kc, const char * const filename);
```

Similar to *KN_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*).

KN_save_param_file()

```
int KNITRO_API KN_save_param_file  
(KN_context_ptr kc, const char * const filename);
```

Write all current parameter values to a file.

KN_set_int_param_by_name()

```
int KNITRO_API KN_set_int_param_by_name
(KN_context_ptr kc, const char * const name, const int value);
```

Set an integer valued parameter using its string name.

KN_set_char_param_by_name()

```
int KNITRO_API KN_set_char_param_by_name
(KN_context_ptr kc, const char * const name, const char * const value);
```

Set a character valued parameter using its string name.

KN_set_double_param_by_name()

```
int KNITRO_API KN_set_double_param_by_name
(KN_context_ptr kc, const char * const name, const double value);
```

Set a double valued parameter using its string name.

KN_set_param_by_name()

```
int KNITRO_API KN_set_param_by_name
(KN_context_ptr kc, const char * const name, const double value);
```

Set an integer or double valued parameter using its string name.

KN_set_int_param()

```
int KNITRO_API KN_set_int_param
(KN_context_ptr kc, const int param_id, const int value);
```

Set an integer valued parameter using its integer identifier (see *Knitro user options*).

KN_set_char_param()

```
int KNITRO_API KN_set_char_param
(KN_context_ptr kc, const int param_id, const char * const value);
```

Set a character valued parameter using its integer identifier (see *Knitro user options*).

KN_set_double_param()

```
int KNITRO_API KN_set_double_param
(KN_context_ptr kc, const int param_id, const double value);
```

Set a double valued parameter using its integer identifier (see *Knitro user options*).

KN_get_int_param_by_name()

```
int KNITRO_API KN_get_int_param_by_name
(KN_context_ptr kc, const char * const name, int * const value);
```

Get an integer valued parameter using its string name.

KN_get_double_param_by_name()

```
int KNITRO_API KN_get_double_param_by_name
(KN_context_ptr kc, const char * const name, double * const value);
```

Get a double valued parameter using its string name.

KN_get_int_param()

```
int KNITRO_API KN_get_int_param
    (KN_context_ptr kc, const int param_id, int * const value);
```

Get an integer valued parameter using its integer identifier (see *Knitro user options*).

KN_get_double_param()

```
int KNITRO_API KN_get_double_param
    (KN_context_ptr kc, const int param_id, double * const value);
```

Get a double valued parameter using its integer identifier (see *Knitro user options*).

KN_get_param_name()

```
int KNITRO_API KN_get_param_name
    (
        KN_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.

KN_get_param_doc()

```
int KNITRO_API KN_get_param_doc
    (
        KN_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.

KN_get_param_type()

```
int KNITRO_API KN_get_param_type
    (
        KN_context_ptr kc,
        const int param_id,
        int * const param_type);
```

Sets the `int * param_type` to the type of the parameter indexed by integer identifier `param_id` (see *Knitro user options*). Possible values are `KN_PARAMTYPE_INT`, `KN_PARAMTYPE_FLOAT`, `KN_PARAMTYPE_STRING`. Returns an error if `param_id` does not correspond to any parameter.

KN_get_num_param_values()

```
int KNITRO_API KN_get_num_param_values
    (
        KN_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 the 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.

KN_get_param_value_doc()

```
int KNITRO_API KN_get_param_value_doc
(
    KN_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 the 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.

KN_get_param_id()

```
int KNITRO_API KN_get_param_id
(
    KN_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.

3.5.5 Basic problem construction

Problem structure is passed to Knitro using KN API functions. The problem is solved by calling `KN_solve()`. 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.)

The typical calling sequence is:

```
KN_new
KN_add_vars/KN_add_cons/KN_set_*bnds, etc. (problem setup)
KN_add*_linear_struct/KN_add*_quadratic_struct (add special structures)
KN_add_eval_callback (add callback for nonlinear evaluations if needed)
KN_set_cb_* (set properties for nonlinear evaluation callbacks)
KN_set_xxx_param (set any number of parameters/user options)
KN_solve
KN_free
```

As long as no structural changes are made to the model `KN_solve()` can be called in succession to re-solve a model after small changes. For example, user options (with the exception of `hessopt`), variable bounds, and constraint bounds can be changed between calls to `KN_solve()`, without having to first call `KN_free()` and reload the model from scratch. More extensive additions or changes to the model (such as adding linear structure, quadratic structure, callbacks, etc) require freeing the existing Knitro solver object and rebuilding the model from scratch.

KN_add_vars()

```
int KNITRO_API KN_add_vars (
    KN_context_ptr kc,
    const KNINT nV,
    KNINT * const indexVars);
```

```
int KNITRO_API KN_add_var (      KN_context_ptr  kc,
                                KNINT * const   indexVar);
```

Add variables to the model. The parameter `indexVars` may be set to `NULL`. Otherwise, on return it holds the global indices associated with the variables that were added (indices are typically allocated sequentially). Parameter `indexVars` can then be passed into other API routines that operate on the set of variables added through a particular call to `KN_add_vars()`. Returns 0 if OK, nonzero if error.

KN_add_cons()

```
int KNITRO_API KN_add_cons (      KN_context_ptr  kc,
                                const KNINT      nC,
                                KNINT * const   indexCons);
int KNITRO_API KN_add_con  (      KN_context_ptr  kc,
                                KNINT * const   indexCon);
```

Add constraints to the model. The parameter `indexCons` may be set to `NULL`. Otherwise, on return it holds the global indices associated with the constraints that were added (indices are typically allocated sequentially). Parameter `indexCons` can then be passed into other API routines that operate on the set of constraints added through a particular call to `KN_add_cons()`. Returns 0 if OK, nonzero if error.

KN_add_rsds()

```
int KNITRO_API KN_add_rsds (      KN_context_ptr  kc,
                                const KNINT      nR,
                                KNINT * const   indexRsds);
int KNITRO_API KN_add_rsd  (      KN_context_ptr  kc,
                                KNINT * const   indexRsd);
```

Add residuals for least squares optimization. The parameter `indexRsds` may be set to `NULL`. Otherwise, on return it holds the global indices associated with the residuals that were added (indices are typically allocated sequentially). Parameter `indexRsds` can then be passed into other API routines that operate on the set of residuals added through a particular call to `KN_add_rsds()`. Note that the current Knitro API does not support adding both constraints and residuals. Returns 0 if OK, nonzero if error.

KN_set_var_lobnds()

```
int KNITRO_API KN_set_var_lobnds (      KN_context_ptr  kc,
                                       const KNINT      nV,
                                       const KNINT * const indexVars,
                                       const double * const xLoBnds);
int KNITRO_API KN_set_var_lobnds_all (      KN_context_ptr  kc,
                                       const double * const xLoBnds);
int KNITRO_API KN_set_var_lobnd  (      KN_context_ptr  kc,
                                       const KNINT      indexVar,
                                       const double      xLoBnd);
```

Set lower bounds on variables – either in groups, all at once, or individually. If not set, variables are assumed to be unbounded (e.g. lower bounds are assumed to be `-KN_INFINITY`). Returns 0 if OK, nonzero if error.

KN_set_var_upbnds()

```
int KNITRO_API KN_set_var_upbnds (      KN_context_ptr  kc,
                                       const KNINT      nV,
                                       const KNINT * const indexVars,
                                       const double * const xUpBnds);
int KNITRO_API KN_set_var_upbnds_all (      KN_context_ptr  kc,
                                       const double * const xUpBnds);
```

```
int KNITRO_API KN_set_var_upbnd (      KN_context_ptr  kc,
                                     const KNINT      indexVar,
                                     const double      xUpBnd);
```

Set upper bounds on variables – either in groups, all at once, or individually. If not set, variables are assumed to be unbounded (e.g. upper bounds are assumed to be `KN_INFINITY`). Returns 0 if OK, nonzero if error.

`KN_set_var_fxbnds()`

```
int KNITRO_API KN_set_var_fxbnds (      KN_context_ptr  kc,
                                     const KNINT      nV,
                                     const KNINT * const indexVars,
                                     const double * const xFxBnds);
int KNITRO_API KN_set_var_fxbnds_all (  KN_context_ptr  kc,
                                     const double * const xFxBnds);
int KNITRO_API KN_set_var_fxbnd (      KN_context_ptr  kc,
                                     const KNINT      indexVar,
                                     const double      xFxBnd);
```

Set fixed bounds on variables – either in groups, all at once, or individually. Adding a fixed bound creates a fixed variable and is equivalent to adding identical lower and upper bounds on the same variable. If the Knitro presolver is enabled, fixed variables will typically be presolved out of the model. If not set, variables are assumed to be unbounded (e.g. lower bounds are assumed to be `-KN_INFINITY` and upper bounds are assumed to be `KN_INFINITY`). Returns 0 if OK, nonzero if error.

`KN_set_var_types()`

```
int KNITRO_API KN_set_var_types (      KN_context_ptr  kc,
                                     const KNINT      nV,
                                     const KNINT * const indexVars,
                                     const int * const xTypes);
int KNITRO_API KN_set_var_types_all (  KN_context_ptr  kc,
                                     const int * const xTypes);
int KNITRO_API KN_set_var_type (      KN_context_ptr  kc,
                                     const KNINT      indexVar,
                                     const int        xType);
```

Set variable types (e.g. `KN_VARTYPE_CONTINUOUS`, `KN_VARTYPE_BINARY`, `KN_VARTYPE_INTEGER`). If not set, variables are assumed to be continuous. Returns 0 if OK, nonzero if error.

`KN_set_var_properties()`

```
int KNITRO_API KN_set_var_properties (  KN_context_ptr  kc,
                                     const KNINT      nV,
                                     const KNINT * const indexVars,
                                     const int * const xProperties);
int KNITRO_API KN_set_var_properties_all (KN_context_ptr  kc,
                                     const int * const xProperties);
int KNITRO_API KN_set_var_property (    KN_context_ptr  kc,
                                     const KNINT      indexVar,
                                     const int        xProperty);
```

Specify some properties of the variables. Currently this API routine is only used to mark variables as linear, but other variable properties will be added in the future. Note: use bit-wise specification of the features:

bit	value	meaning
0	1	<code>KN_VAR_LINEAR</code>

default = 0 (variables are assumed to be nonlinear)

If a variable only appears linearly in the model, it can be very helpful to mark this by enabling bit 0. This information can then be used by Knitro to perform more extensive preprocessing. If a variable appears nonlinearly in any constraint or the objective (or if the user does not know) then it should not be marked as linear. Variables are assumed to be nonlinear variables by default. Knitro makes a local copy of all inputs, so the application may free memory after the call. Returns 0 if OK, nonzero if error.

KN_set_con_lobnds()

```
int KNITRO_API KN_set_con_lobnds (      KN_context_ptr  kc,
                                     const KNINT       nC,
                                     const KNINT * const  indexCons,
                                     const double * const  cLoBnds);
int KNITRO_API KN_set_con_lobnds_all (  KN_context_ptr  kc,
                                     const double * const  cLoBnds);
int KNITRO_API KN_set_con_lobnd (      KN_context_ptr  kc,
                                     const KNINT         indexCon,
                                     const double         cLoBnd);
```

Set lower bounds on constraints – either in groups, all at once, or individually. If not set, constraints are assumed to be unbounded (e.g. lower bounds are assumed to be `-KN_INFINITY`). Returns 0 if OK, nonzero if error.

KN_set_con_upbnds()

```
int KNITRO_API KN_set_con_upbnds (      KN_context_ptr  kc,
                                     const KNINT       nC,
                                     const KNINT * const  indexCons,
                                     const double * const  cUpBnds);
int KNITRO_API KN_set_con_upbnds_all (  KN_context_ptr  kc,
                                     const double * const  cUpBnds);
int KNITRO_API KN_set_con_upbnd (      KN_context_ptr  kc,
                                     const KNINT         indexCon,
                                     const double         cUpBnd);
```

Set upper bounds on constraints – either in groups, all at once, or individually. If not set, constraints are assumed to be unbounded (e.g. upper bounds are assumed to be `KN_INFINITY`). Returns 0 if OK, nonzero if error.

KN_set_con_eqbnds()

```
int KNITRO_API KN_set_con_eqbnds (      KN_context_ptr  kc,
                                     const KNINT       nC,
                                     const KNINT * const  indexCons,
                                     const double * const  cEqBnds);
int KNITRO_API KN_set_con_eqbnds_all (  KN_context_ptr  kc,
                                     const double * const  cEqBnds);
int KNITRO_API KN_set_con_eqbnd (      KN_context_ptr  kc,
                                     const KNINT         indexCon,
                                     const double         cEqBnd);
```

Set equality bounds on constraints – either in groups, all at once, or individually. Adding an equality bound creates an equality constraint and is equivalent to adding identical lower and upper bounds on the same constraint. If not set, constraints are assumed to be unbounded (e.g. lower bounds are assumed to be `-KN_INFINITY` and upper bounds are assumed to be `KN_INFINITY`). Returns 0 if OK, nonzero if error.

KN_set_con_properties()

```
int KNITRO_API KN_set_con_properties (  KN_context_ptr  kc,
                                     const KNINT       nC,
                                     const KNINT * const  indexCons,
                                     const int * const   cProperties);
```

```

int KNITRO_API KN_set_con_properties_all (      KN_context_ptr  kc,
                                               const int    * const  cProperties);
int KNITRO_API KN_set_con_property (          KN_context_ptr  kc,
                                             const KNINT      indexCon,
                                             const int      cProperty);
    
```

Specify some properties of the constraint functions. Note: use bit-wise specification of the features:

bit	value	meaning
0	1	KN_CON_CONVEX
1	2	KN_CON_CONCAVE
2	4	KN_CON_CONTINUOUS
3	8	KN_CON_DIFFERENTIABLE
4	16	KN_CON_TWICE_DIFFERENTIABLE
5	32	KN_CON_NOISY
6	64	KN_CON_NONDETERMINISTIC

default = 28 (bits 2-4 enabled: e.g. continuous, differentiable, twice-differentiable)

KN_set_obj_property()

```

int KNITRO_API KN_set_obj_property (      KN_context_ptr  kc,
                                           const int      objProperty);
    
```

Specify some properties of the objective function. Note: use bit-wise specification of the features:

bit	value	meaning
0	1	KN_OBJ_CONVEX
1	2	KN_OBJ_CONCAVE
2	4	KN_OBJ_CONTINUOUS
3	8	KN_OBJ_DIFFERENTIABLE
4	16	KN_OBJ_TWICE_DIFFERENTIABLE
5	32	KN_OBJ_NOISY
6	64	KN_OBJ_NONDETERMINISTIC

default = 28 (bits 2-4 enabled: e.g. continuous, differentiable, twice-differentiable)

KN_set_obj_goal()

```

int KNITRO_API KN_set_obj_goal (      KN_context_ptr  kc,
                                       const int      objGoal);
    
```

Set the objective goal (KN_OBJGOAL_MINIMIZE or KN_OBJGOAL_MAXIMIZE). If not called, minimization assumed by default. Returns 0 if OK, nonzero if error.

KN_set_var_primal_init_values()

```

int KNITRO_API KN_set_var_primal_init_values (      KN_context_ptr  kc,
                                                    const KNINT      nV,
                                                    const KNINT * const  indexVars,
                                                    const double * const  xInitVals);
int KNITRO_API KN_set_var_primal_init_values_all (  KN_context_ptr  kc,
                                                    const double * const  xInitVals);
int KNITRO_API KN_set_var_primal_init_value (      KN_context_ptr  kc,
                                                    const KNINT      indexVar,
                                                    const double      xInitVal);
    
```

Set initial values for primal variables. If not set, variables may be initialized as 0 or initialized by Knitro based on some initialization strategy (perhaps determined by a user option). Returns 0 if OK, nonzero if error.

KN_set_var_dual_init_values()

```
int KNITRO_API KN_set_var_dual_init_values (      KN_context_ptr  kc,
                                                const KNINT      nV,
                                                const KNINT * const indexVars,
                                                const double * const lambdaInitVals);
int KNITRO_API KN_set_var_dual_init_values_all (      KN_context_ptr  kc,
                                                    const double * const λ
↳lambdaInitVals);
int KNITRO_API KN_set_var_dual_init_value (      KN_context_ptr  kc,
                                                const KNINT      indexVar,
                                                const double     lambdaInitVal);
```

Set initial values for dual variables (i.e. the Lagrange multipliers corresponding to the potentially bounded variables). If not set, dual variables may be initialized as 0 or initialized by Knitro based on some initialization strategy (perhaps determined by a user option). Returns 0 if OK, nonzero if error.

KN_set_con_dual_init_values()

```
int KNITRO_API KN_set_con_dual_init_values (      KN_context_ptr  kc,
                                                const KNINT      nC,
                                                const KNINT * const indexCons,
                                                const double * const lambdaInitVals);
int KNITRO_API KN_set_con_dual_init_values_all (      KN_context_ptr  kc,
                                                    const double * const λ
↳lambdaInitVals);
int KNITRO_API KN_set_con_dual_init_value (      KN_context_ptr  kc,
                                                const KNINT      indexCon,
                                                const double     lambdaInitVal);
```

Set initial values for constraint dual variables (i.e. the Lagrange multipliers for the constraints). If not set, constraint dual variables may be initialized as 0 or initialized by Knitro based on some initialization strategy (perhaps determined by a user option). Returns 0 if OK, nonzero if error.

3.5.6 Adding constant structure

KN_add_obj_constant()

```
int KNITRO_API KN_add_obj_constant (      KN_context_ptr  kc,
                                        const double     constant);
```

Add a constant to the objective function.

KN_add_con_constants()

```
int KNITRO_API KN_add_con_constants (      KN_context_ptr  kc,
                                        const KNINT      nC,
                                        const KNINT * const indexCons,      /* size_
↳= nC */
                                        const double * const constants);      /* size_
↳= nC */
int KNITRO_API KN_add_con_constants_all (      KN_context_ptr  kc,
                                        const double * const constants);
int KNITRO_API KN_add_con_constant (      KN_context_ptr  kc,
                                        const KNINT      indexCon,
                                        const double     constant);
```

Add constants to the body of constraint functions. Each component i of arrays `indexCons` and `constants` adds a constant term `constants[i]` to the constraint `c[indexCons[i]]`.

`KN_add_rsd_constants()`

```
int KNITRO_API KN_add_rsd_constants (      KN_context_ptr  kc,
                                         const KNINT      nR,
                                         const KNINT * const indexRsds,      /* size_
↳= nR */
                                         const double * const constants);      /* size_
↳= nR */
int KNITRO_API KN_add_rsd_constants_all (      KN_context_ptr  kc,
                                         const double * const constants);
int KNITRO_API KN_add_rsd_constant (      KN_context_ptr  kc,
                                         const KNINT      indexRsd,
                                         const double      constant);
```

Add constants to the body of residual functions. Each component i of arrays `indexRsds` and `constants` adds a constant term `constants[i]` to the residual `r[indexRsds[i]]`.

3.5.7 Adding linear structure

`KN_add_obj_linear_struct()`

```
int KNITRO_API KN_add_obj_linear_struct (      KN_context_ptr  kc,
                                         const KNINT      nnz,
                                         const KNINT * const indexVars,  /* size_
↳= nnz */
                                         const double * const coefs);      /* size_
↳= nnz */
```

Add linear structure to the objective function. Each component i of arrays `indexVars` and `coefs` adds a linear term `coefs[i]*x[indexVars[i]]` to the objective.

`KN_add_con_linear_struct()`

```
int KNITRO_API KN_add_con_linear_struct (      KN_context_ptr  kc,
                                         const KNLONG     nnz,
                                         const KNINT * const indexCons,  /*_
↳size = nnz */
                                         const KNINT * const indexVars,  /*_
↳size = nnz */
                                         const double * const coefs);      /*_
↳size = nnz */
int KNITRO_API KN_add_con_linear_struct_one (      KN_context_ptr  kc,
                                         const KNLONG     nnz,
                                         const KNINT      indexCon,
                                         const KNINT * const indexVars,  /*_
↳size = nnz */
                                         const double * const coefs);      /*_
↳size = nnz */
```

Add linear structure to the constraint functions. Each component i of arrays `indexCons`, `indexVars` and `coefs` adds a linear term `coefs[i]*x[indexVars[i]]` to constraint `c[indexCons[i]]`.

Use `KN_add_con_linear_struct()` to add linear structure for a group of constraints at once, and `KN_add_con_linear_struct_one()` to add linear structure for just one constraint.

KN_add_rsd_linear_struct()

```

int KNITRO_API KN_add_rsd_linear_struct (      KN_context_ptr  kc,
                                             const KNLONG      nnz,
                                             const KNINT * const indexRsd,    /*_
↳size = nnz */
                                             const KNINT * const indexVars,    /*_
↳size = nnz */
                                             const double * const coefs);        /*_
↳size = nnz */
int KNITRO_API KN_add_rsd_linear_struct_one (  KN_context_ptr  kc,
                                             const KNLONG      nnz,
                                             const KNINT      indexRsd,
                                             const KNINT * const indexVar,    /*_
↳size = nnz */
                                             const double * const coefs);        /*_
↳size = nnz */
    
```

Add linear structure to the residual functions. Each component i of arrays `indexRsd`, `indexVars` and `coefs` adds a linear term `coefs[i]*x[indexVars[i]]` to residual `r[indexRsd[i]]`.

Use `KN_add_rsd_linear_struct()` to add linear structure for a group of residuals at once, and `KN_add_rsd_linear_struct_one()` to add linear structure for just one residual.

3.5.8 Adding quadratic structure

KN_add_obj_quadratic_struct()

```

int KNITRO_API KN_add_obj_quadratic_struct (  KN_context_ptr  kc,
                                             const KNLONG      nnz,
                                             const KNINT * const indexVars1, /*_
↳size = nnz */
                                             const KNINT * const indexVars2, /*_
↳size = nnz */
                                             const double * const coefs);        /*_
↳size = nnz */
    
```

Add quadratic structure to the objective function. Each component i of arrays `indexVars1`, `indexVars2` and `coefs` adds a quadratic term `coefs[i]*x[indexVars1[i]]*x[indexVars2[i]]` to the objective.

Note: if `indexVars2[i]` is < 0 then it adds a linear term `coefs[i]*x[indexVars1[i]]` instead.

KN_add_con_quadratic_struct()

```

int KNITRO_API KN_add_con_quadratic_struct (  KN_context_ptr  kc,
                                             const KNLONG      nnz,
                                             const KNINT * const indexCons, /*_
↳size = nnz */
                                             const KNINT * const indexVars1, /*_
↳size = nnz */
                                             const KNINT * const indexVars2, /*_
↳size = nnz */
                                             const double * const coefs);        /*_
↳size = nnz */
int KNITRO_API KN_add_con_quadratic_struct_one (  KN_context_ptr  kc,
                                             const KNLONG      nnz,
                                             const KNINT      indexCon,
                                             const KNINT * const indexVars1, /*_
↳* size = nnz */
    
```

```

        const KNINT * const indexVars2, /
↪ * size = nnz */
        const double * const coefs); /
↪ * size = nnz */
    
```

Add quadratic structure to the constraint functions. Each component i of arrays `indexCons`, `indexVars1`, `indexVars2` and `coefs` adds a quadratic term `coefs[i]*x[indexVars1[i]]*x[indexVars2[i]]` to the constraint `c[indexCons[i]]`.

Use `KN_add_con_quadratic_struct()` to add quadratic structure for a group of constraints at once, and `KN_add_con_quadratic_struct_one()` to add quadratic structure for just one constraint.

Note: if `indexVars2[i]` is < 0 then it adds a linear term `coefs[i]*x[indexVars1[i]]` instead.

3.5.9 Adding conic structure

`KN_add_con_L2norm()`

```

int KNITRO_API KN_add_con_L2norm (      KN_context_ptr kc,
        const KNINT          indexCon,
        const KNINT          nCoords,
        const KNLONG         nnz,
        const KNINT * const  indexCoords, /* size = nnz_
↪ */
        const KNINT * const  indexVars,  /* size = nnz_
↪ */
        const double * const coefs,      /* size = nnz_
↪ */
        const double * const constants); /* size = _
↪ nCoords or NULL */
    
```

Add L2 norm structure of the form $||Ax + b||_2$ to a constraint.

Parameter	Description
<code>indexCon</code> :	The constraint index that the L2 norm term will be added to.
<code>nCoords</code> :	The number of rows in “A” (or dimension of “b”)
<code>nnz</code> :	The number of sparse non-zero elements in “A”
<code>indexCoords</code> :	The coordinate (row) index for each non-zero element in “A”.
<code>indexVars</code> :	The variable (column) index for each non-zero element in “A”
<code>coefs</code> :	The coefficient value for each non-zero element in “A”
<code>constants</code> :	The array “b” - may be set to NULL to ignore “b”

Note: L2 norm structure can currently only be added to constraints that otherwise only have linear (or constant) structure. In this way they can be used to define conic constraints of the form $||Ax + b|| \leq c'x + d$. The c coefficients should be added through `KN_add_con_linear_struct()` and d can be set as a constraint bound or through `KN_add_con_constants()`.

Note: Models with L2 norm structure are currently only handled by the Interior/Direct (`KN_ALG_BAR_DIRECT`) algorithm in Knitro. Any model with structure defined with `KN_add_L2norm()` will automatically be forced to use this algorithm.

3.5.10 Adding complementarity constraints

`KN_set_compcons()`

```
int KNITRO_API KN_set_comps (      KN_context_ptr  kc,
                                const KNINT      nCC,
                                const int         * ccTypes,
                                const KNINT      * indexComps1,
                                const KNINT      * indexComps2);
```

This function adds complementarity constraints to the problem. 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 only be called once. The array `ccTypes` specifies the type of complementarity:

```
KN_CCTYPE_VARVAR: two (non-negative) variables
KN_CCTYPE_VARCON: a variable and a constraint
KN_CCTYPE_CONCON: two constraints
```

Note: Currently only `KN_CCTYPE_VARVAR` is supported. The other `ccTypes` will be added in future releases. Returns 0 if OK, or a negative value on error.

3.5.11 Defining evaluation callbacks

Applications may define functions for evaluating problem elements at a trial point. The functions must match the prototype defined below, and passed to Knitro with the appropriate `KN_set_cb_*` call. Knitro may request different types of evaluation information, as specified in `evalRequest.type`:

```
KN_RC_EVALFC   - return objective and constraint function values
KN_RC_EVALGA   - return first derivative values in "objGrad" and "jac"
KN_RC_EVALFCGA - return objective and constraint function values
                  AND first derivative "objGrad" and "jac"
KN_RC_EVALH    - return second derivative values in "hessian"
KN_RC_EVALH_NO_F (this version excludes the objective term)
KN_RC_EVALHV   - return a Hessian-vector product in "hessVector"
KN_RC_EVALHV_NO_F (this version excludes the objective term)
KN_RC_EVALR    - return residual function values for least squares
KN_RC_EVALRJ   - return residual Jacobian values for least squares
```

The argument `lambda` is not defined when requesting `EVALFC`, `EVALGA`, `EVALFCGA`, `EVALR` or `EVALRJ`.

Usually, applications for standard optimization models define three callback functions: one for `EVALFC`, one for `EVALGA`, and one for `EVALH` / `EVALHV`. The last function is only used when providing the Hessian (as opposed to using one of the Knitro options to approximate it) and evaluates `H` or `HV` depending on the value of `evalRequest.type`. For least squares models, the application defines the two callback functions for `EVALR` and `EVALRJ` (instead of `EVALFC` and `EVALGA`). Least squares applications do not provide a callback for the Hessian as it is always approximated.

It is possible in most cases to combine `EVALFC` and `EVALGA` into a single callback function. This may be advantageous if the application evaluates functions and their derivatives at the same time. In order to do this, set the user option `eval_fcga=KN_EVAL_FCGA_YES`, and define one callback set in `KN_add_eval_callback()` that evaluates BOTH the functions and gradients (i.e. have it populate `obj`, `c`, `objGrad`, and `jac` in the `evalResult` structure), and do not set a callback in `KN_set_cb_grad()`. Whenever Knitro needs a function + gradient evaluation, it will callback to the function passed to `KN_add_eval_callback()` with an `EVALFCGA` request.

Combining function and gradient evaluations in one callback is not currently allowed if `hesopt=KN_HESSOPT_PRODUCT_FINDIFF`. It is not possible to combine `EVALH` / `EVALHV` because `lambda` may change after the `EVALFC` call. Generally it is most efficient to separate function and gradient callbacks, since a gradient evaluation is not needed at every x value where functions are evaluated.

The `userParams` argument is an arbitrary pointer passed from the Knitro `KN_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.

For simplicity, the following user-defined evaluation callback functions all use the same `KN_eval_callback()` function prototype defined below:

```
funcCallback
gradCallback
hessCallback
rsdCallback      (for least squares)
rsdJacCallback   (for least squares)
```

Callbacks should return 0 if successful, a negative error code if not. Possible unsuccessful (negative) error codes for the `func/grad/hess/rsd/rsdJac` callback functions include:

```
KN_RC_CALLBACK_ERR      (for generic callback errors)
KN_RC_EVAL_ERR          (for evaluation errors, e.g log(-1))
```

In addition, for the “`func`” (as well as the “`newpoint`”, “`ms_process`” and “`mip_node`” user callbacks), the user may set the following return code to force Knitro to terminate based on some user-defined condition.:

```
KN_RC_USER_TERMINATION (to use a callback routine
                        for user specified termination)
```

```
typedef struct KN_eval_request {
    int         type;
    int         threadID;
    const double * x;
    const double * lambda;
    const double * sigma;
    const double * vec;
} KN_eval_request, *KN_eval_request_ptr;
```

Structure used to pass back evaluation information for evaluation callbacks.

Parameter	Description
<code>type:</code>	indicates the type of evaluation requested
<code>threadID:</code>	the thread ID associated with this evaluation request; useful for multi-threaded, concurrent evaluations
<code>x:</code>	values of unknown (primal) variables used for all evaluations
<code>lambda:</code>	values of unknown dual variables/Lagrange multipliers used for the evaluation of the Hessian
<code>sigma:</code>	scalar multiplier for the objective component of the Hessian
<code>vec:</code>	vector array value for Hessian-vector products (only used when user option <code>hessopt=KN_HESSOPT_PRODUCT</code>)

```
typedef struct KN_eval_result {
    double * obj;
    double * c;
    double * objGrad;
    double * jac;
    double * hess;
    double * hessVec;
    double * rsd;
    double * rsdJac;
} KN_eval_result, *KN_eval_result_ptr;
```

Structure used to return results information for evaluation callbacks. The arrays (and their indices and sizes) returned in this structure are local to the specific callback structure used for the evaluation.

Parameter	Description
obj:	objective function evaluated at “x” for EVALFC or EVALFCGA request (funcCallback)
c:	(length nC) constraint values evaluated at “x” for EVALFC or EVALFCGA request (funcCallback)
obj-Grad:	(length nV) objective gradient evaluated at “x” for EVALGA request (gradCallback) or EVALFCGA request (funcCallback)
jac:	(length nnzJ) constraint Jacobian evaluated at “x” for EVALGA request (gradCallback) or EVALFCGA request (funcCallback)
hess:	(length nnzH) Hessian evaluated at “x”, “lambda”, “sigma” for EVALH or EVALH_NO_F request (hessCallback)
hessVec:	(length n=number variables in the model) Hessian-vector product evaluated at “x”, “lambda”, “sigma” for EVALHV or EVALHV_NO_F request (hessCallback)
rsd:	(length nR) residual values evaluated at “x” for EVALR request (rsdCallback)
rsdJac:	(length nnzJ) residual Jacobian evaluated at “x” for EVALRJ request (rsdJacCallback)

```
typedef struct CB_context CB_context, *CB_context_ptr;
```

The callback structure/object. Note the `CB_context_ptr` is allocated and managed by Knitro: the user does not have to free it.

```
typedef int KN_eval_callback (KN_context_ptr      kc,
                             CB_context_ptr      cb,
                             KN_eval_request_ptr const evalRequest,
                             KN_eval_result_ptr  const evalResult,
                             void                * const userParams);
```

Function prototype for evaluation callbacks.

KN_add_eval_callback()

```
int KNITRO_API KN_add_eval_callback (      KN_context_ptr      kc,
                                          const KNBOOL         evalObj,
                                          const KNINT           nC,
                                          const KNINT           * const indexCons,
                                          ↪ * nullable if nC=0 */
                                          KN_eval_callback * const funcCallback,
                                          CB_context_ptr * const cb);
```

This is the routine for adding a callback for (nonlinear) evaluations of objective and constraint functions. This routine can be called multiple times to add more than one callback structure (e.g. to create different callback structures to handle different blocks of constraints). This routine specifies the minimal information needed for a callback, and creates the callback structure `cb`, which can then be passed to other callback functions to set additional information for that callback.

Parameter	Description
evalObj	boolean indicating whether or not any part of the objective function is evaluated in the callback
nC	number of constraints evaluated in the callback
index-Cons	(length nC) index of constraints evaluated in the callback (set to NULL if nC=0)
func-Callback	a pointer to a function that evaluates the objective parts (if evalObj=KNTRUE) and any constraint parts (specified by nC and indexCons) involved in this callback; when eval_fcga=KN_EVAL_FCGA_YES, this callback should also evaluate the relevant first derivatives/gradients
cb	(output) the callback structure that gets created by calling this function; all the memory for this structure is handled by Knitro

After a callback is created by `KN_add_eval_callback()`, the user can then specify gradient information and structure through `KN_set_cb_grad()` and Hessian information and structure through `KN_set_cb_hess()`. If not set, Knitro will approximate these. However, it is highly recommended to provide a callback routine to specify the gradients if at all possible as this will greatly improve the performance of Knitro. Even if a gradient callback is not provided, it is still helpful to provide the sparse Jacobian structure through `KN_set_cb_grad()` to improve the efficiency of the finite-difference gradient approximations. Other optional information can also be set via `KN_set_cb_*()` functions as detailed below.

Returns 0 if OK, nonzero if error.

`KN_add_eval_callback_all()`

```
int KNITRO_API KN_add_eval_callback_all (    KN_context_ptr      kc,
                                             KN_eval_callback * const  _
↳funcCallback,
                                             CB_context_ptr * const  cb)
```

Simplified version of `KN_add_eval_callback()` to create a callback that applies to the objective function and all constraints.

`KN_add_eval_callback_one()`

```
int KNITRO_API KN_add_eval_callback_one (    KN_context_ptr      kc,
                                             const KNINT        index,      /
↳* -1 for obj */
                                             KN_eval_callback * const  _
↳funcCallback,
                                             CB_context_ptr * const  cb)
```

Version of `KN_add_eval_callback()` to create a callback that only applies to a single objective function or constraint. Set `index` to the corresponding constraint index or use -1 for the objective.

`KN_add_lsq_eval_callback()`

```
int KNITRO_API KN_add_lsq_eval_callback (    KN_context_ptr      kc,
                                             const KNINT        nR,
                                             const KNINT        * const  indexRsds,
                                             KN_eval_callback * const  rsdCallback,
                                             CB_context_ptr * const  cb);
```

Add an evaluation callback for a least-squares models. Similar to `KN_add_eval_callback()` above, but for least-squares models.

Parameter	Description
nR	number of residuals evaluated in the callback
indexRsds	(length nR) index of residuals evaluated in the callback
rsdCallback	a pointer to a function that evaluates any residual parts (specified by nR and indexRsds) involved in this callback
cb	(output) the callback structure that gets created by calling this function; all the memory for this structure is handled by Knitro

After a callback is created by `KN_add_lsq_eval_callback()`, the user can then specify residual Jacobian information and structure through `KN_set_cb_rsd_jac()`. If not set, Knitro will approximate the residual Jacobian. However, it is highly recommended to provide a callback routine to specify the residual Jacobian if at all possible as this will greatly improve the performance of Knitro. Even if a callback for the residual Jacobian is not provided, it is still helpful to provide the sparse Jacobian structure for the residuals through `KN_set_cb_rsd_jac()` to improve the efficiency of the finite-difference Jacobian approximation. Other optional information can also be set via `KN_set_cb_*` functions as detailed below. Returns 0 if OK, nonzero if error.

`KN_add_lsq_eval_callback_all()`

```
int KNITRO_API KN_add_lsq_eval_callback_all (
    KN_context_ptr      kc,
    KN_eval_callback * const rsdCallback,
    CB_context_ptr * const cb)
```

Simplified version of `KN_add_lsq_eval_callback()` to create a callback that applies to all residual functions.

`KN_add_lsq_eval_callback_one()`

```
int KNITRO_API KN_add_lsq_eval_callback_one (
    KN_context_ptr      kc,
    const KNINT         indexRsd,
    KN_eval_callback * const rsdCallback,
    CB_context_ptr * const cb);
```

Version of `KN_add_lsq_eval_callback()` to create a callback that only applies to a single residual function. Set `indexRsd` to the corresponding residual index.

`KN_set_cb_grad()`

```
int KNITRO_API KN_set_cb_grad (
    KN_context_ptr      kc,
    CB_context_ptr      cb,
    const KNINT         nV, /* or KN_DENSE_
    */
    const KNINT         objGradIndexVars,
    const KNLONG        nnzJ, /* or KN_DENSE_
    */
    const KNINT         jacIndexCons,
    const KNINT         jacIndexVars,
    KN_eval_callback * const gradCallback); /*
    nullable */
```

This API function is used to set the objective gradient and constraint Jacobian structure and also (optionally) a callback function to evaluate the objective gradient and constraint Jacobian provided through this callback.

Parameter	Description
cb	a callback structure created from a previous call to <code>KN_add_eval_callback()</code>
nV	number of nonzero components in the objective gradient for this callback if providing in sparse form; set to <code>KN_DENSE</code> to provide the full objective gradient
obj-GradIndexVars	(length nV) the nonzero indices of the objective gradient; set to <code>NULL</code> if <code>nV=KN_DENSE</code> or <code>nV=0</code> (i.e. <code>evalObj=KNFALSE</code>)
nnzJ	number of nonzeros in the sparse constraint Jacobian computed through this callback; set to <code>KN_DENSE_ROWMAJOR</code> to provide the full Jacobian in row major order (i.e. ordered by rows/constraints), or <code>KN_DENSE_COLMAJOR</code> to provide the full Jacobian in column major order (i.e. ordered by columns/ variables)
jacIndexCons	(length nnzJ) constraint index (row) of each nonzero; set to <code>NULL</code> if <code>nnzJ=KN_DENSE_ROWMAJOR/KN_DENSE_COLMAJOR</code> or <code>nnzJ=0</code>
jacIndexVars	(length nnzJ) variable index (column) of each nonzero; set to <code>NULL</code> if <code>nnzJ=KN_DENSE_ROWMAJOR/KN_DENSE_COLMAJOR</code> or <code>nnzJ=0</code>
gradCallback	a pointer to a function that evaluates the objective gradient parts and any constraint Jacobian parts involved in this callback; set to <code>NULL</code> if using finite-difference gradient approximations (specified via <code>KN_set_cb_gradopt()</code>), or if gradients and functions are provided together in the <code>funcCallback</code> (i.e. <code>eval_fcga=KN_EVAL_FCGA_YES</code>).

The user should generally always try to define the sparsity structure for the Jacobian (`nnzJ`, `jacIndexCons`, `jacIndexVars`). Even when using finite-difference approximations to compute the gradients, knowing the sparse structure of the Jacobian can allow Knitro to compute these finite-difference approximations faster. However, if the user is unable to provide this sparsity structure, then one can set `nnzJ` to `KN_DENSE_ROWMAJOR` or `KN_DENSE_COLMAJOR` and set `jacIndexCons` and `jacIndexVars` to `NULL`.

`KN_set_cb_hess()`

```
int KNITRO_API KN_set_cb_hess (      KN_context_ptr      kc,
                                   CB_context_ptr      cb,
                                   const KNLONG         nnzH, /* or KN_DENSE_
↳ * */
                                   const KNINT          * hessIndexVars1,
                                   const KNINT          * hessIndexVars2,
                                   KN_eval_callback     * hessCallback);
```

This API function is used to set the structure and a callback function to evaluate the components of the Hessian of the Lagrangian provided through this callback. `KN_set_cb_hess()` should only be used when defining a user-supplied Hessian callback function (via the “`hessopt=KN_HESSOPT_EXACT`” user option). When Knitro is approximating the Hessian, it cannot make use of the Hessian sparsity structure.

Parameter	Description
cb	a callback structure created from a previous call to <code>KN_add_eval_callback()</code>
nnzH	number of nonzeros in the sparse Hessian of the Lagrangian computed through this callback; set to <code>KN_DENSE_ROWMAJOR</code> to provide the full upper triangular Hessian in row major order, or <code>KN_DENSE_COLMAJOR</code> to provide the full upper triangular Hessian in column major order. Note that the Hessian is symmetric, so the lower triangular components are the same as the upper triangular components with row and column indices swapped.
hessIndexVars1	(length nnzH) first variable index of each nonzero; set to <code>NULL</code> if <code>nnzH=KN_DENSE_ROWMAJOR/KN_DENSE_COLMAJOR</code>
hessIndexVars2	(length nnzH) second variable index of each nonzero; set to <code>NULL</code> if <code>nnzH=KN_DENSE_ROWMAJOR/KN_DENSE_COLMAJOR</code>
hessCallback	a pointer to a function that evaluates the components of the Hessian of the Lagrangian provide in this callback

KN_set_cb_rsd_jac()

```
int KNITRO_API KN_set_cb_rsd_jac (      KN_context_ptr    kc,
                                       CB_context_ptr    cb,
                                       const KNLONG       nnzJ, /* or KN_
↳ DENSE_* */
                                       const KNINT        jacIndexRsds,
                                       const KNINT        jacIndexVars,
                                       KN_eval_callback *  const rsdJacCallback); /*
↳ * nullable */
```

This API function is used to set the residual Jacobian structure and also (optionally) a callback function to evaluate the residual Jacobian provided through this callback.

Parameter	Description
cb	a callback structure created from a previous call to KN_add_lsq_eval_callback()
nnzJ	number of nonzeros in the sparse residual Jacobian computed through this callback; set to KN_DENSE_ROWMAJOR to provide the full Jacobian in row major order (i.e. ordered by rows/residuals), or KN_DENSE_COLMAJOR to provide the full Jacobian in column major order (i.e. ordered by columns/ variables)
jacIndexRsds	(length nnzJ) residual index (row) of each nonzero; set to NULL if nnzJ=KN_DENSE_ROWMAJOR/KN_DENSE_COLMAJOR or nnzJ=0
jacIndexVars	(length nnzJ) variable index (column) of each nonzero; set to NULL if nnzJ=KN_DENSE_ROWMAJOR/KN_DENSE_COLMAJOR or nnzJ=0
rsdJac-Callback	a pointer to a function that evaluates any residual Jacobian parts involved in this callback; set to NULL if using a finite- difference Jacobian approximation (specified via KN_set_cb_gradopt())

The user should generally always try to define the sparsity structure for the Jacobian (nnzJ, jacIndexRsds, jacIndexVars). Even when using a finite-difference approximation to compute the Jacobian, knowing the sparse structure of the Jacobian can allow Knitro to compute this finite-difference approximation faster. However, if the user is unable to provide this sparsity structure, then one can set nnzJ to KN_DENSE_ROWMAJOR or KN_DENSE_COLMAJOR and set jacIndexRsds and jacIndexVars to NULL.

KN_set_cb_user_params()

```
int KNITRO_API KN_set_cb_user_params (KN_context_ptr kc,
                                       CB_context_ptr cb,
                                       void * const userParams);
```

Define a userParams structure for an evaluation callback.

KN_set_cb_gradopt()

```
int KNITRO_API KN_set_cb_gradopt (      KN_context_ptr    kc,
                                       CB_context_ptr    cb,
                                       const int         gradopt);
```

Specify which gradient option gradopt will be used to evaluate the first derivatives of the callback functions. If gradopt=:c:macro:KN_GRADOPT_EXACT then a gradient evaluation callback must be set by KN_set_cb_grad() (or KN_set_cb_rsd_jac() for least squares).

KN_set_cb_relstepsizes()

```
int KNITRO_API KN_set_cb_relstepsizes (      KN_context_ptr    kc,
                                       CB_context_ptr    cb,
                                       const KNINT        nV,
                                       const KNINT        * const indexVars,
```

```

                                const double * const xRelStepSizes);
int  KNITRO_API KN_set_cb_relstepsizes_all (    KN_context_ptr kc,
                                                CB_context_ptr cb,
                                                const double * const xRelStepSizes);
int  KNITRO_API KN_set_cb_relstepsize (    KN_context_ptr kc,
                                           CB_context_ptr cb,
                                           const KNINT      indexVar,
                                           const double      xRelStepSize);
    
```

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:

$$\text{delta}[i] = \text{relStepSizes}[i] * \max(\text{abs}(x[i]), 1)$$

The default relative step sizes for each component of “x” are $\sqrt{\text{eps}}$ for forward finite differences, and $\text{eps}^{1/3}$ for central finite differences. Use this function to overwrite the default values. Any zero values will use Knitro default values, while non-zero values will overwrite default values. Knitro makes a local copy of all inputs, so the application may free memory after the call. Returns 0 if OK, nonzero if error.

KN_get_cb_number_cons()

```

int  KNITRO_API KN_get_cb_number_cons (const KN_context_ptr kc,
                                       const CB_context_ptr cb,
                                       KNINT * const      nC);
    
```

Retrieve the number of constraints nC being evaluated through callback cb . Returns 0 if OK, nonzero if error.

KN_get_cb_number_rsds()

```

int  KNITRO_API KN_get_cb_number_rsds (const KN_context_ptr kc,
                                       const CB_context_ptr cb,
                                       KNINT * const      nR);
    
```

Retrieve the number of residuals nR being evaluated through callback cb . Returns 0 if OK, nonzero if error.

KN_get_cb_objgrad_nnz()

```

int  KNITRO_API KN_get_cb_objgrad_nnz (const KN_context_ptr kc,
                                       const CB_context_ptr cb,
                                       KNINT * const      nnz);
    
```

Retrieve the number of non-zero objective gradient elements nnz evaluated through callback cb . Returns 0 if OK, nonzero if error.

KN_get_cb_jacobian_nnz()

```

int  KNITRO_API KN_get_cb_jacobian_nnz (const KN_context_ptr kc,
                                       const CB_context_ptr cb,
                                       KNLONG * const      nnz);
    
```

Retrieve the number of non-zero Jacobian elements nnz evaluated through callback cb . Returns 0 if OK, nonzero if error.

KN_get_cb_rsd_jacobian_nnz()

```

int  KNITRO_API KN_get_cb_rsd_jacobian_nnz (const KN_context_ptr kc,
                                             const CB_context_ptr cb,
                                             KNLONG * const      nnz);
    
```

Retrieve the number of non-zero residual Jacobian elements `nnz` evaluated through callback `cb`. Returns 0 if OK, nonzero if error.

KN_get_cb_hessian_nnz()

```
int KNITRO_API KN_get_cb_hessian_nnz (const KN_context_ptr kc,
                                     const CB_context_ptr cb,
                                     KNLONG * const nnz);
```

Retrieve the number of non-zero Hessian elements `nnz` evaluated through callback `cb`. Returns 0 if OK, nonzero if error.

3.5.12 Other user callbacks

Other user callbacks that aren't involved in evaluations use the `KN_user_callback()` or other function prototypes. These include:

```
KN_set_newpt_callback
KN_set_mip_node_callback
KN_set_ms_process_callback
KN_set_ms_initpt_callback
KN_set_puts_callback
```

Callbacks should return 0 if successful, a negative error code if not. In addition, for the “newpoint”, “ms_process” and “mip_node” callbacks, the user may set the following return code to force Knitro to terminate based on some user-defined condition.:

```
KN_RC_USER_TERMINATION    (to use a callback routine
                           for user specified termination)
```

```
typedef int KN_user_callback (    KN_context_ptr kc,
                                const double * const x,
                                const double * const lambda,
                                void * const userParams);
```

Type declaration for several non-evaluation user callbacks defined below.

KN_set_newpt_callback()

```
int KNITRO_API KN_set_newpt_callback (KN_context_ptr kc,
                                     KN_user_callback * const fnPtr,
                                     void * const userParams);
```

Set the callback function that is invoked after Knitro computes a new estimate of the solution point (i.e., after every iteration). The function should not modify any Knitro arguments. Argument `kc` passed to the callback from inside Knitro 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.). Arguments `x` and `lambda` contain the latest solution estimates. Other values (such as objective, constraint, jacobian, etc.) can be queried using the corresponding `KN_get_XXX_values()` methods. Note: Currently only active for continuous models. Return 0 if successful, a negative error code if not.

KN_set_mip_node_callback()

```
int KNITRO_API KN_set_mip_node_callback (KN_context_ptr kc,
                                         KN_user_callback * const fnPtr,
                                         void * const userParams);
```

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). Argument `kc` passed to the callback from inside Knitro is the context pointer for the last node subproblem solved inside Knitro. The function should not modify any Knitro arguments. Arguments `x` and `lambda` contain the solution from the node solve. Return 0 if successful, a negative error code if not.

KN_set_ms_process_callback()

```
int KNITRO_API KN_set_ms_process_callback (KN_context_ptr      kc,
                                          KN_user_callback * const fnPtr,
                                          void                * const userParams);
```

This callback function is for multistart (MS) problems only. Set the callback function that is invoked after Knitro finishes processing a multistart solve. Argument `kc` passed to the callback from inside Knitro is the context pointer for the last multistart subproblem solved inside Knitro. The function should not modify any Knitro arguments. Arguments `x` and `lambda` contain the solution from the last solve. Return 0 if successful, a negative error code if not.

KN_set_ms_initpt_callback()

```
typedef int KN_ms_initpt_callback (KN_context_ptr kc,
                                   const KNINT    nSolveNumber,
                                   double * const  x,
                                   double * const  lambda,
                                   void * const   userParams);

int KNITRO_API KN_set_ms_initpt_callback (KN_context_ptr      kc,
                                          KN_ms_initpt_callback * const fnPtr,
                                          void                * const userParams);
```

This callback allows applications to specify an initial point before each local solve in the multistart 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 `KN_ms_initpt_callback()` type declaration for this callback.

KN_set_puts_callback()

This callback allows applications to handle/redirect output. 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 `KN_PARAM_OUTMODE`. The `KN_puts()` function takes a `userParams` argument which is a pointer passed directly from `KN_solve()`. The function should return the number of characters that were printed. Use `KN_puts()` type declaration for this callback.

3.5.13 Other algorithmic/modeling features

KN_set_var_feastols()

```
int KNITRO_API KN_set_var_feastols (KN_context_ptr kc,
                                   const KNINT    nV,
                                   const KNINT * const indexVars,
                                   const double * const xFeasTols);
int KNITRO_API KN_set_var_feastols_all (KN_context_ptr kc,
                                       const double * const xFeasTols);
int KNITRO_API KN_set_var_feastol (KN_context_ptr kc,
                                   const KNINT    indexVar,
                                   const double    xFeasTol);
```

Set an array of absolute feasibility tolerances for variable bounds to use for the termination tests. The user options `KN_PARAM_FEASTOL` / `KN_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 variable. Values specified through this function will override the value determined by `KN_PARAM_FEASTOL` / `KN_PARAM_FEASTOLABS`. The tolerances should be positive values. If a non-positive value is specified, that variable will use the standard tolerances based on `KN_PARAM_FEASTOL` / `KN_PARAM_FEASTOLABS`. 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.
```

Returns 0 if OK, nonzero if error.

KN_set_con_feastols()

```
int KNITRO_API KN_set_con_feastols (      KN_context_ptr  kc,
                                         const KNINT      nC,
                                         const KNINT * const indexCons,
                                         const double * const cFeasTols);
int KNITRO_API KN_set_con_feastols_all (  KN_context_ptr  kc,
                                         const double * const cFeasTols);
int KNITRO_API KN_set_con_feastol (      KN_context_ptr  kc,
                                         const KNINT      indexCon,
                                         const double      cFeasTol);
```

Set an array of absolute constraint feasibility tolerances to use for the termination tests. The user options `KN_PARAM_FEASTOL` / `KN_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. Values specified through this function will override the value determined by `KN_PARAM_FEASTOL` / `KN_PARAM_FEASTOLABS`. The tolerances should be positive values. If a non-positive value is specified, that constraint will use the standard tolerances based on `KN_PARAM_FEASTOL` / `KN_PARAM_FEASTOLABS`. 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
```

Returns 0 if OK, nonzero if error.

KN_set_compcon_feastols()

```
int KNITRO_API KN_set_compcon_feastols (  KN_context_ptr  kc,
                                         const KNINT      nCC,
                                         const KNINT * const indexCompCons,
                                         const double * const ccFeasTols);
int KNITRO_API KN_set_compcon_feastols_all (  KN_context_ptr  kc,
                                         const double * const ccFeasTols);
int KNITRO_API KN_set_compcon_feastol (      KN_context_ptr  kc,
                                         const KNINT      indexCompCon,
                                         const double      ccFeasTol);
```

Set an array of absolute feasibility tolerances to use for the complementarity constraint termination tests. The user options `KN_PARAM_FEASTOL` / `KN_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 complementarity constraint. Values specified through this function will override the value determined by `KN_PARAM_FEASTOL` / `KN_PARAM_FEASTOLABS`. The tolerances should be positive values. If a non-positive value is specified, that complementarity constraint will use the standard tolerances based on `KN_PARAM_FEASTOL` / `KN_PARAM_FEASTOLABS`. 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. Returns 0 if OK, nonzero if error.

KN_set_var_scalings()

```
int KNITRO_API KN_set_var_scalings (      KN_context_ptr  kc,
                                         const KNINT      nV,
                                         const KNINT * const indexVars,
                                         const double * const xScaleFactors,
                                         const double * const xScaleCenters);
int KNITRO_API KN_set_var_scalings_all (  KN_context_ptr  kc,
                                         const double * const xScaleFactors,
                                         const double * const xScaleCenters);
int KNITRO_API KN_set_var_scaling (      KN_context_ptr  kc,
                                         const KNINT      indexVar,
                                         const double      xScaleFactor,
                                         const double      xScaleCenter);
```

Set an array of variable scaling and centering values to perform a linear scaling:

```
x[i] = xScaleFactors[i] * xScaled[i] + xScaleCenters[i]
```

for each variable. These scaling factors should try to represent the “typical” values of the x variables so that the scaled variables ($xScaled$) used internally by Knitro are close to one. The values for $xScaleFactors$ should be positive. If a non-positive value is specified, that variable will not be scaled. Returns 0 if OK, nonzero if error.

KN_set_con_scalings()

```
int KNITRO_API KN_set_con_scalings (      KN_context_ptr  kc,
                                         const KNINT      nC,
                                         const KNINT * const indexCons,
                                         const double * const cScaleFactors);
int KNITRO_API KN_set_con_scalings_all (  KN_context_ptr  kc,
                                         const double * const cScaleFactors);
int KNITRO_API KN_set_con_scaling (      KN_context_ptr  kc,
                                         const KNINT      indexCon,
                                         const double      cScaleFactor);
```

Set an array of constraint scaling values to perform a scaling:

```
cScaled[i] = cScaleFactors[i] * c[i]
```

for each constraint. These scaling factors should try to represent the “typical” values of the **inverse** of the constraint values c so that the scaled constraints ($cScaled$) used internally by Knitro are close to one. Scaling factors for standard constraints can be provided with $cScaleFactors$. The values for $cScaleFactors$ should be positive. If a non-positive value is specified, that constraint will use either the standard Knitro scaling (KN_SCALE_USER_INTERNAL), or no scaling (KN_SCALE_USER_NONE). Returns 0 if OK, nonzero if error.

KN_set_compcon_scalings()

```
int KNITRO_API KN_set_compcon_scalings (  KN_context_ptr  kc,
                                         const KNINT      nCC,
                                         const KNINT * const indexCompCons,
                                         const double * const ccScaleFactors);
int KNITRO_API KN_set_compcon_scalings_all (  KN_context_ptr  kc,
                                         const double * const ccScaleFactors);
int KNITRO_API KN_set_compcon_scaling (      KN_context_ptr  kc,
```

```

const KNINT      indexCompCons,
const double     ccScaleFactor);

```

Set an array of complementarity constraint scaling values to perform a scaling:

```
ccScaled[i] = ccScaleFactors[i] * c[i]
```

for each complementarity constraint. These scaling factors should try to represent the “typical” values of the **inverse** of the complementarity constraint values c so that the scaled complementarity constraints ($ccScaled$) used internally by Knitro are close to one. Scaling factors for complementarity constraints can be provided with `ccScaleFactors`. The values for `ccScaleFactors` should be positive. If a non-positive value is specified, that complementarity constraint will use either the standard Knitro scaling (`KN_SCALE_USER_INTERNAL`), or no scaling (`KN_SCALE_USER_NONE`). Returns 0 if OK, nonzero if error.

KN_set_obj_scaling()

```

int KNITRO_API KN_set_obj_scaling (      KN_context_ptr  kc,
const double                          objScaleFactor);

```

Set a scaling value for the objective function:

```
objScaled = objScaleFactor * obj
```

This scaling factor should try to represent the “typical” value of the **inverse** of the objective function value obj so that the scaled objective ($objScaled$) used internally by Knitro is close to one. The value for `objScaleFactor` should be positive. If a non-positive value is specified, then the objective will use either the standard Knitro scaling (`KN_SCALE_USER_INTERNAL`), or no scaling (`KN_SCALE_USER_NONE`). Returns 0 if OK, nonzero if error.

KN_set_var_names()

```

int KNITRO_API KN_set_var_names (      KN_context_ptr  kc,
const KNINT      nV,
const KNINT * const indexVars,
char * const     xNames[]);
int KNITRO_API KN_set_var_names_all (  KN_context_ptr  kc,
char * const     xNames[]);
int KNITRO_API KN_set_var_name (      KN_context_ptr  kc,
const KNINT      indexVars,
char * const     xName);

```

Set names for model variables passed in by the user/modeling language so that Knitro can internally print out these variable names. Knitro makes a local copy of all inputs, so the application may free memory after the call. Returns 0 if OK, nonzero if error.

KN_set_con_names()

```

int KNITRO_API KN_set_con_names (      KN_context_ptr  kc,
const KNINT      nC,
const KNINT * const indexCons,
char * const     cNames[]);
int KNITRO_API KN_set_con_names_all (  KN_context_ptr  kc,
char * const     cNames[]);
int KNITRO_API KN_set_con_name (      KN_context_ptr  kc,
const KNINT      indexCon,
char * const     cName);

```

Set names for model constraints passed in by the user/modeling language so that Knitro can internally print out these constraint names. Knitro makes a local copy of all inputs, so the application may free memory after the call. Returns

0 if OK, nonzero if error.

KN_set_compccon_names()

```
int KNITRO_API KN_set_compccon_names (      KN_context_ptr  kc,
                                           const KNINT      nCC,
                                           const KNINT * const indexCompCons,
                                           char * const     ccNames[]);
int KNITRO_API KN_set_compccon_names_all (  KN_context_ptr  kc,
                                           char * const     ccNames[]);
int KNITRO_API KN_set_compccon_name (      KN_context_ptr  kc,
                                           const int        indexCompCon,
                                           char * const     ccName);
```

Set names for model complementarity constraints passed in by the user/modeling language so that Knitro can internally print out these complementarity constraint names. Knitro makes a local copy of all inputs, so the application may free memory after the call. Returns 0 if OK, nonzero if error.

KN_set_obj_name()

```
int KNITRO_API KN_set_obj_name (      KN_context_ptr  kc,
                                      const char * const objName);
```

Set name for model objective passed in by the user/modeling language so that Knitro can internally print out the objective name. Returns 0 if OK, nonzero if error.

KN_set_var_honorbnds()

```
int KNITRO_API KN_set_var_honorbnds (      KN_context_ptr  kc,
                                           const KNINT      nV,
                                           const KNINT * const indexVars,
                                           const int * const  xHonorBnds);
int KNITRO_API KN_set_var_honorbnds_all (  KN_context_ptr  kc,
                                           const int * const  xHonorBnds);
int KNITRO_API KN_set_var_honorbnd (      KN_context_ptr  kc,
                                           const KNINT      indexVar,
                                           const int        xHonorBnd);
```

This API function can be used to identify which variables should satisfy their variable bounds throughout the optimization process (`KN_HONORBND_ALWAYS`). The user option `KN_PARAM_HONORBND` can be used to set ALL variables to honor their bounds. This routine takes precedence over the setting of `KN_PARAM_HONORBND` and is used to customize the settings for individual variables. Knitro makes a local copy of all inputs, so the application may free memory after the call. Returns 0 if OK, nonzero if error.

KN_set_mip_branching_priorities()

```
int KNITRO_API KN_set_mip_branching_priorities
(      KN_context_ptr  kc,
  const KNINT      nV,
  const KNINT * const indexVars,
  const int * const  xPriorities);
int KNITRO_API KN_set_mip_branching_priorities_all
(      KN_context_ptr  kc,
  const int * const  xPriorities);
int KNITRO_API KN_set_mip_branching_priority
(      KN_context_ptr  kc,
  const KNINT      indexVar,
  const int        xPriority);
```

This function can be used to set the branching priorities for integer variables when using the MIP features in Knitro. You must first set the types of variables (e.g. by calling `KN_set_var_types()`) before calling this function so that integer variables are marked. 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. Values for continuous variables are ignored. Knitro makes a local copy of all inputs, so the application may free memory after the call. Returns 0 if OK, nonzero if error.

`KN_set_mip_intvar_strategies()`

```
int KNITRO_API KN_set_mip_intvar_strategies
(
    KN_context_ptr kc,
    const KNINT      nV,
    const KNINT * const indexVars,
    const int  * const xStrategies);
int KNITRO_API KN_set_mip_intvar_strategies_all
(
    KN_context_ptr kc,
    const int * const xStrategies);
int KNITRO_API KN_set_mip_intvar_strategy
(
    KN_context_ptr kc,
    const KNINT      indexVar,
    const int         xStrategy);
```

Set strategies for dealing with individual integer variables. Possible strategy values include:

<code>KN_MIP_INTVAR_STRATEGY_NONE</code>	0 (default)
<code>KN_MIP_INTVAR_STRATEGY_RELAX</code>	1
<code>KN_MIP_INTVAR_STRATEGY_MPEC</code>	2 (binary variables only)

The parameter `indexVars` should be an index value corresponding to an integer variable (nothing is done if the index value corresponds to a continuous variable), and `xStrategies` should correspond to one of the strategy values listed above. The default strategy is `KN_MIP_INTVAR_STRATEGY_NONE`, and the strategy `KN_MIP_INTVAR_STRATEGY_MPEC` can only be applied to binary variables. Returns 0 if OK, nonzero if error.

3.5.14 Solving

`KN_solve()`

```
int KNITRO_API KN_solve (KN_context_ptr kc);
```

Call Knitro to solve the problem. The return value indicates the final exit code from the optimization process:

0:	the final solution is verified optimal to specified tolerances;
-100 to -109:	a feasible solution was found (but not verified optimal);
-200 to -209:	Knitro terminated at an infeasible point;
-300	: the problem was determined to be unbounded;
-400 to -409:	Knitro terminated because it reached a pre-defined limit (a feasible point was found before reaching the limit);
-410 to -419:	Knitro terminated because it reached a pre-defined limit (no feasible point was found before reaching the limit);
-500 to -599:	Knitro terminated with an input error or some non-standard error.

A detailed description of the possible return values is given in [Return codes](#).

3.5.15 Reading model/solution properties

`KN_get_number_vars()`

```
int KNITRO_API KN_get_number_vars (const KN_context_ptr kc,
                                   int * const nV);
```

Retrieve the number of variables `nV` in the model. Returns 0 if OK, nonzero if error.

`KN_get_number_cons()`

```
int KNITRO_API KN_get_number_cons (const KN_context_ptr kc,
                                    int * const nC);
```

Retrieve the number of constraints `nC` in the model. Returns 0 if OK, nonzero if error.

`KN_get_number_rsds()`

```
int KNITRO_API KN_get_number_rsds (const KN_context_ptr kc,
                                    int * const nR);
```

Retrieve the number of residuals `nR` in the model. Returns 0 if OK, nonzero if error.

`KN_get_number_FC_evals()`

```
int KNITRO_API KN_get_number_FC_evals (const KN_context_ptr kc,
                                        int * const numFCevals);
```

Return the number of function evaluations requested by `KN_solve()` in `numFCevals`. One evaluation count includes a single evaluation of the objective and all the constraints defined via callbacks (whether evaluated altogether in one callback or evaluated using several separate callbacks). Returns 0 if OK, nonzero if error.

`KN_get_number_GA_evals()`

```
int KNITRO_API KN_get_number_GA_evals (const KN_context_ptr kc,
                                        int * const numGAevals);
```

Return the number of gradient evaluations requested by `KN_solve()` in `numGAevals`. One evaluation count includes a single evaluation of the first derivatives of the objective and all the constraints defined via gradient callbacks (whether evaluated altogether in one callback or evaluated using several separate callbacks). Returns 0 if OK, nonzero if error.

`KN_get_number_H_evals()`

```
int KNITRO_API KN_get_number_H_evals (const KN_context_ptr kc,
                                       int * const numHevals);
```

Return the number of Hessian evaluations requested by `KN_solve()` in `numHevals`. One evaluation count includes a single evaluation of all the components of the Hessian of the Lagrangian matrix defined via callbacks (whether evaluated altogether in one callback or evaluated using several separate callbacks). Returns 0 if OK, nonzero if error.

`KN_get_number_HV_evals()`

```
int KNITRO_API KN_get_number_HV_evals (const KN_context_ptr kc,
                                        int * const numHVevals);
```

Return the number of Hessian-vector products requested by `KN_solve()` in `numHVevals`. One evaluation count includes a single evaluation of the product of the Hessian of the Lagrangian matrix with a vector submitted by Knitro (whether evaluated altogether in one callback or evaluated using several separate callbacks). Returns 0 if OK, nonzero if error.

KN_get_solution()

```
int KNITRO_API KN_get_solution (const KN_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.

KN_get_obj_value()

```
int KNITRO_API KN_get_obj_value (const KN_context_ptr kc,
                                 double * const obj);
```

Return the value of the objective $obj(x)$ in *obj*. Returns 0 if call is successful; <0 if there is an error.

KN_get_obj_type()

```
int KNITRO_API KN_get_obj_type (const KN_context_ptr kc,
                                int * const objType);
```

Return the type (e.g. KN_OBJTYPE_GENERAL, KN_OBJTYPE_LINEAR, KN_OBJTYPE_QUADRATIC, etc.) of the objective $obj(x)$ in *objType*. Returns 0 if call is successful; <0 if there is an error.

KN_get_con_values()

```
int KNITRO_API KN_get_con_values (const KN_context_ptr kc,
                                  const KNINT nC,
                                  const KNINT * const indexCons,
                                  double * const c);
int KNITRO_API KN_get_con_values_all (const KN_context_ptr kc,
                                      double * const c);
int KNITRO_API KN_get_con_value (const KN_context_ptr kc,
                                 const KNINT indexCon,
                                 double * const c);
```

Return the values of the constraint vector $c(x)$ in *c*. The array *c* must be allocated by the user. Returns 0 if call is successful; <0 if there is an error.

KN_get_con_types()

```
int KNITRO_API KN_get_con_types (const KN_context_ptr kc,
                                 const KNINT nC,
                                 const KNINT * const indexCons,
                                 int * const cTypes);
int KNITRO_API KN_get_con_types_all (const KN_context_ptr kc,
                                     int * const cTypes);
int KNITRO_API KN_get_con_type (const KN_context_ptr kc,
                                const KNINT indexCon,
                                int * const cType);
```

Return the types (e.g. KN_CONTYPE_GENERAL, KN_CONTYPE_LINEAR, KN_CONTYPE_QUADRATIC, etc.) of the constraint vector $c(x)$ in *cTypes*. The array *cTypes* must be allocated by the user. Returns 0 if call is successful; <0 if there is an error.

KN_get_rsd_values()

```

int KNITRO_API KN_get_rsd_values (const KN_context_ptr kc,
                                const KNINT nR,
                                const KNINT * const indexRsds,
                                double * const r);
int KNITRO_API KN_get_rsd_values_all (const KN_context_ptr kc,
                                       double * const r);
int KNITRO_API KN_get_rsd_value (const KN_context_ptr kc,
                                 const KNINT indexRsd,
                                 double * const r);

```

Return the values of the residual vector $r(x)$ in r . The array r must be allocated by the user. Returns 0 if call is successful; <0 if there is an error.

KN_get_number_iters()

```

int KNITRO_API KN_get_number_iters (const KN_context_ptr kc,
                                    int * const numIters);

```

Return the number of iterations made by `KN_solve()` in `numIters`. Returns 0 if OK, nonzero if error. For continuous problems only.

KN_get_number_cg_iters()

```

int KNITRO_API KN_get_number_cg_iters (const KN_context_ptr kc,
                                        int * const numCGiters);

```

Return the number of conjugate gradients (CG) iterations made by `KN_solve()` in `numCGiters`. Returns 0 if OK, nonzero if error. For continuous problems only.

KN_get_abs_feas_error()

```

int KNITRO_API KN_get_abs_feas_error (const KN_context_ptr kc,
                                       double * const absFeasError);

```

Return the absolute feasibility error at the solution in `absFeasError`. Refer to the Knitro manual section *Termination criteria* for a detailed definition of this quantity. Returns 0 if OK, nonzero if error. For continuous problems only.

KN_get_rel_feas_error()

```

int KNITRO_API KN_get_rel_feas_error (const KN_context_ptr kc,
                                       double * const relFeasError);

```

Return the relative feasibility error at the solution in `relFeasError`. Refer to the Knitro manual section *Termination criteria* for a detailed definition of this quantity. Returns 0 if OK, nonzero if error. For continuous problems only.

KN_get_abs_opt_error()

```

int KNITRO_API KN_get_abs_opt_error (const KN_context_ptr kc,
                                       double * const absOptError);

```

Return the absolute optimality error at the solution in `absOptError`. Refer to the Knitro manual section *Termination criteria* for a detailed definition of this quantity. Returns 0 if OK, nonzero if error. For continuous problems only.

KN_get_rel_opt_error()

```

int KNITRO_API KN_get_rel_opt_error (const KN_context_ptr kc,
                                       double * const relOptError);

```

Return the relative optimality error at the solution in `relOptError`. Refer to the Knitro manual section *Termination criteria* for a detailed definition of this quantity. Returns 0 if OK, nonzero if error. For continuous problems only.

KN_get_objgrad_values()

```
int KNITRO_API KN_get_objgrad_nnz (const KN_context_ptr kc,
                                   KNINT * const nnz);
int KNITRO_API KN_get_objgrad_values (const KN_context_ptr kc,
                                       KNINT * const indexVars,
                                       double * const objGrad);
```

Return the values of the objective gradient vector in `indexVars` and `objGrad`. The objective gradient values returned correspond to the non-zero sparse objective gradient indices provided by the user. The arrays `indexVars` and `objGrad` must be allocated by the user. The size of these arrays is obtained by first calling `KN_get_objgrad_nnz()`. Returns 0 if call is successful; <0 if there is an error. For continuous problems only.

KN_get_objgrad_values_all()

```
int KNITRO_API KN_get_objgrad_values_all (const KN_context_ptr kc,
                                           double * const objGrad);
```

Return the values of the full (dense) objective gradient in `objGrad`. The array `objGrad` must be allocated by the user (the size is equal to the total number of variables in the problem). Returns 0 if call is successful; <0 if there is an error. For continuous problems only.

KN_get_jacobian_values()

```
int KNITRO_API KN_get_jacobian_nnz (const KN_context_ptr kc,
                                     KNLONG * const nnz);
int KNITRO_API KN_get_jacobian_values (const KN_context_ptr kc,
                                       KNINT * const indexCons,
                                       KNINT * const indexVars,
                                       double * const jac);
```

Return the values of the constraint Jacobian in `indexCons`, `indexVars`, and `jac`. The Jacobian values returned correspond to the non-zero sparse Jacobian indices provided by the user. The arrays `indexCons`, `indexVars`, and `jac` must be allocated by the user. The size of these arrays is obtained by first calling `KN_get_jacobian_nnz()`. Returns 0 if call is successful; <0 if there is an error. For continuous problems only.

KN_get_rsd_jacobian_values()

```
int KNITRO_API KN_get_rsd_jacobian_nnz (const KN_context_ptr kc,
                                         KNLONG * const nnz);
int KNITRO_API KN_get_rsd_jacobian_values (const KN_context_ptr kc,
                                           KNINT * const indexRsds,
                                           KNINT * const indexVars,
                                           double * const rsdJac);
```

Return the values of the residual Jacobian in `indexRsds`, `indexVars`, and `rsdJac`. The Jacobian values returned correspond to the non-zero sparse Jacobian indices provided by the user. The arrays `indexRsds`, `indexVars` and `rsdJac` must be allocated by the user. The size of these arrays is obtained by first calling `KN_get_rsd_jacobian_nnz()`. Returns 0 if call is successful; <0 if there is an error. For continuous least-squares problems only.

KN_get_hessian_values()

```
int KNITRO_API KN_get_hessian_nnz (const KN_context_ptr kc,
                                    KNLONG * const nnz);
int KNITRO_API KN_get_hessian_values (const KN_context_ptr kc,
```

```

KNINT * const indexVars1,
KNINT * const indexVars2,
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. `KN_HESSOPT_EXACT` (presolver on or off), or;
2. `KN_HESSOPT_BFGS` or `KN_HESSOPT_SR1`, but only with the Knitro presolver off (i.e. `KN_PRESOLVE_NONE`).
3. Solving a least squares model with the Gauss-Newton Hessian and the Gauss-Newton Hessian is explicitly computed and stored in Knitro.

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 `indexVars1`, `indexVars2`, and `hess` are left unmodified, and the routine has return code 1.

Note that in case 2 above (`KN_HESSOPT_BFGS` or `KN_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 `KN_HESSOPT_EXACT` (case 1 above) the Hessian values returned correspond to the non-zero sparse Hessian indices provided by the user.

The arrays `indexVars1`, `indexVars2` and `hess` must be allocated by the user. The size of these arrays is obtained by first calling `KN_get_hessian_nnz()`. 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.

KN_get_mip_number_nodes()

```

int KNITRO_API KN_get_mip_number_nodes (const KN_context_ptr kc,
                                       int * const numNodes);

```

Return the number of nodes processed in the MIP solve in `numNodes`. Returns 0 if OK, nonzero if error.

KN_get_mip_number_solves()

```

int KNITRO_API KN_get_mip_number_solves (const KN_context_ptr kc,
                                         int * const numSolves);

```

Return the number of continuous subproblems processed in the MIP solve in `numSolves`. Returns 0 if OK, nonzero if error.

KN_get_mip_abs_gap()

```

int KNITRO_API KN_get_mip_abs_gap (const KN_context_ptr kc,
                                   double * const absGap);

```

Return the final absolute integrality gap in the MIP solve in `absGap`. Refer to the Knitro manual section *Termination criteria* for a detailed definition of this quantity. Set to `KN_INFINITY` if no incumbent (i.e., integer feasible) point found. Returns 0 if OK, nonzero if error.

KN_get_mip_rel_gap()

```

int KNITRO_API KN_get_mip_rel_gap (const KN_context_ptr kc,
                                   double * const relGap);

```

Return the final absolute integrality gap in the MIP solve in `relGap`. Refer to the Knitro manual section *Termination criteria* for a detailed definition of this quantity. Set to `KN_INFINITY` if no incumbent (i.e., integer feasible) point found. Returns 0 if OK, nonzero if error.

`KN_get_mip_incumbent_obj()`

```
int KNITRO_API KN_get_mip_incumbent_obj (const KN_context_ptr kc,
                                       double * const incumbentObj);
```

Return the objective value of the MIP incumbent solution in `incumbentObj`. Set to `KN_INFINITY` if no incumbent (i.e., integer feasible) point found. Returns 0 if incumbent solution exists and call is successful; 1 if no incumbent (i.e., integer feasible) exists; <0 if there is an error.

`KN_get_mip_relaxation_bnd()`

```
int KNITRO_API KN_get_mip_relaxation_bnd (const KN_context_ptr kc,
                                          double * const relaxBound);
```

Return the value of the current MIP relaxation bound in `relaxBound`. Returns 0 if OK, nonzero if error.

`KN_get_mip_lastnode_obj()`

```
int KNITRO_API KN_get_mip_lastnode_obj (const KN_context_ptr kc,
                                        double * const lastNodeObj);
```

Return the objective value of the most recently solved MIP node subproblem in `lastNodeObj`. Returns 0 if OK, nonzero if error.

`KN_get_mip_incumbent_x()`

```
int KNITRO_API KN_get_mip_incumbent_x (const KN_context_ptr kc,
                                       double * const x);
```

Return the MIP incumbent solution in `x` if one exists. Returns 0 if incumbent solution exists and call is successful; 1 if no incumbent (i.e., integer feasible) point exists and leaves `x` unmodified; <0 if there is an error.

3.5.16 Problem definition defines

KN_INFINITY

```
#define KN_INFINITY DBL_MAX
```

Use to set infinite variable and constraint bounds in Knitro.

KN_PARAMTYPE

```
#define KN_PARAMTYPE_INTEGER 0
#define KN_PARAMTYPE_FLOAT 1
#define KN_PARAMTYPE_STRING 2
```

Possible parameter types.

KN_OBJGOAL

```
#define KN_OBJGOAL_MINIMIZE 0
#define KN_OBJGOAL_MAXIMIZE 1
```

Possible objective goals for the solver (`objGoal` in `KN_set_obj_goal()`).

KN_OBJTYPE

```
#define KN_OBJTYPE_CONSTANT -1
#define KN_OBJTYPE_GENERAL 0
#define KN_OBJTYPE_LINEAR 1
#define KN_OBJTYPE_QUADRATIC 2
```

Possible values for the objective type.

KN_CONTYPE

```
#define KN_CONTYPE_CONSTANT -1
#define KN_CONTYPE_GENERAL 0
#define KN_CONTYPE_LINEAR 1
#define KN_CONTYPE_QUADRATIC 2
#define KN_CONTYPE_CONIC 3
```

Possible values for the constraint type.

KN_RSDDTYPE

```
#define KN_RSDDTYPE_CONSTANT -1
#define KN_RSDDTYPE_GENERAL 0
#define KN_RSDDTYPE_LINEAR 1
```

Possible values for the residual type.

KN_CCTYPE

```
#define KN_CCTYPE_VARVAR 0
#define KN_CCTYPE_VARCON 1 /* NOT SUPPORTED YET */
#define KN_CCTYPE_CONCON 2 /* NOT SUPPORTED YET */
```

Possible values for the complementarity constraint type (ccTypes in *KN_set_compcons()*). Currently only KN_CCTYPE_VARVAR is supported. The other types will be supported in future releases.

KN_VARTYPE

```
#define KN_VARTYPE_CONTINUOUS 0
#define KN_VARTYPE_INTEGER 1
#define KN_VARTYPE_BINARY 2
```

Possible values for the variable type (xTypes in *KN_set_var_types()*).

KN_VAR_

```
#define KN_VAR_LINEAR 1 /*-- LINEAR ONLY EVERYWHERE */
```

Possible values for enabling bits to set variable properties via *KN_set_var_properties()*.

KN_OBJ_

```
#define KN_OBJ_CONVEX 1 /*-- CONVEX OBJECTIVE */
#define KN_OBJ_CONCAVE 2 /*-- CONCAVE OBJECTIVE */
#define KN_OBJ_CONTINUOUS 4 /*-- OBJECTIVE IS CONTINUOUS */
#define KN_OBJ_DIFFERENTIABLE 8 /*-- (ONCE) DIFFERENTIABLE OBJECTIVE */
#define KN_OBJ_TWICE_DIFFERENTIABLE 16 /*-- TWICE DIFFERENTIABLE OBJECTIVE */
#define KN_OBJ_NOISY 32 /*-- OBJECTIVE FUNCTION IS NOISY */
#define KN_OBJ_NONDETERMINISTIC 64 /*-- OBJECTIVE IS NONDETERMINISTIC */
```

Possible values for bit flags used to set objective function properties via *KN_set_obj_properties()*.

KN_CON_

```
#define KN_CON_CONVEX          1 /*-- CONVEX CONSTRAINT */
#define KN_CON_CONCAVE        2 /*-- CONCAVE CONSTRAINT */
#define KN_CON_CONTINUOUS      4 /*-- CONSTRAINT IS CONTINUOUS */
#define KN_CON_DIFFERENTIABLE  8 /*-- (ONCE) DIFFERENTIABLE CONSTRAINT */
#define KN_CON_TWICE_DIFFERENTIABLE 16 /*-- TWICE DIFFERENTIABLE CONSTRAINT */
#define KN_CON_NOISY           32 /*-- CONSTRAINT FUNCTION IS NOISY */
#define KN_CON_NONDETERMINISTIC 64 /*-- CONSTRAINT IS NONDETERMINISTIC */
```

Possible values for bit flags used to set constraint function properties via `KN_set_con_properties()`.

KN_DENSE

```
#define KN_DENSE                -1 /*-- GENERIC DENSE (e.g. FOR ARRAYS) */
#define KN_DENSE_ROWMAJOR       -2 /*-- DENSE MATRIX IN ROW MAJOR ORDER */
#define KN_DENSE_COLMAJOR       -3 /*-- DENSE MATRIX IN COLUMN MAJOR ORDER */
```

Possible values for dense arrays or matrices.

KN_RC_

```
#define KN_RC_EVALFC           1 /*-- OBJECTIVE AND CONSTRAINT FUNCTIONS */
#define KN_RC_EVALGA           2 /*-- OBJ. GRADIENT AND CONSTRAINT JACOBIAN */
#define KN_RC_EVALH            3 /*-- HESSIAN OF THE LAGRANGIAN */
#define KN_RC_EVALHV           7 /*-- HESSIAN-VECTOR PRODUCT */
#define KN_RC_EVALH_NO_F       8 /*-- NO OBJECTIVE COMPONENT INCLUDED */
#define KN_RC_EVALHV_NO_F      9 /*-- NO OBJECTIVE COMPONENT INCLUDED */
#define KN_RC_EVALR            10 /*-- RESIDUAL FUNCTIONS (LEAST SQUARES) */
#define KN_RC_EVALRJ           11 /*-- RESIDUAL JACOBIAN (LEAST SQUARES) */
#define KN_RC_EVALFCGA         12 /*-- BOTH FUNCTIONS AND GRADIENTS */
```

Possible evaluation request codes for evaluation callbacks.

3.6 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.

KN_RC_OPTIMAL_OR_SATISFACTORY

```
#define KN_RC_OPTIMAL_OR_SATISFACTORY 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.

KN_RC_NEAR_OPT

```
#define KN_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.

KN_RC_FEAS_XTOL

```
#define KN_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.

KN_RC_FEAS_NO_IMPROVE

```
#define KN_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.

KN_RC_FEAS_FTOL

```
#define KN_RC_FEAS_FTOL        -103
```

Primal feasible solution; the optimization terminated because the relative change in the objective function is less than that specified by the parameter *ftol* for *ftol_iters* consecutive iterations. To try to get more accuracy one may decrease *ftol* and/or increase *ftol_iters*. If *ftol* 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.

KN_RC_INFEASIBLE

```
#define KN_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.

KN_RC_INFEAS_XTOL

```
#define KN_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.

KN_RC_INFEAS_NO_IMPROVE

```
#define KN_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.

KN_RC_INFEAS_MULTISTART

```
#define KN_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.

KN_RC_INFEAS_CON_BOUNDS

```
#define KN_RC_INFEAS_CON_BOUNDS -204
```

The constraint bounds have been determined to be infeasible.

KN_RC_INFEAS_VAR_BOUNDS

```
#define KN_RC_INFEAS_VAR_BOUNDS -205
```

The variable bounds have been determined to be infeasible.

KN_RC_UNBOUNDED

```
#define KN_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.

KN_RC_ITER_LIMIT_FEAS

```
#define KN_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*.

KN_RC_TIME_LIMIT_FEAS

```
#define KN_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*.

KN_RC_FEVAL_LIMIT_FEAS

```
#define KN_RC_FEVAL_LIMIT_FEAS -402
```

The function evaluation limit was reached before being able to satisfy the required stopping criteria. A feasible point was found. The function evaluation limit can be increased through the user option *maxfevals*.

KN_RC_MIP_EXH_FEAS

```
#define KN_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.

KN_RC_MIP_TERM_FEAS

```
#define KN_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”.

KN_RC_MIP_SOLVE_LIMIT_FEAS

```
#define KN_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*.

KN_RC_MIP_NODE_LIMIT_FEAS

```
#define KN_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*.

KN_RC_ITER_LIMIT_INFEAS

```
#define KN_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*.

KN_RC_TIME_LIMIT_INFEAS

```
#define KN_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*.

KN_RC_FEVAL_LIMIT_INFEAS

```
#define KN_RC_FEVAL_LIMIT_INFEAS     -412
```

The function evaluation limit was reached before being able to satisfy the required stopping criteria. No feasible point was found. The function evaluation limit can be increased through the user option *maxfevals*.

KN_RC_MIP_EXH_INFEAS

```
#define KN_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.

KN_RC_MIP_SOLVE_LIMIT_INFEAS

```
#define KN_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*.

KN_RC_MIP_NODE_LIMIT_INFEAS

```
#define KN_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*.

KN_RC_CALLBACK_ERR

```
#define KN_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.

KN_RC_LP_SOLVER_ERR

```
#define KN_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.

KN_RC_EVAL_ERR

```
#define KN_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.

KN_RC_OUT_OF_MEMORY

```
#define KN_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.

KN_RC_USER_TERMINATION

```
#define KN_RC_USER_TERMINATION -504
```

Knitro has been terminated by the user.

Other codes

```
#define KN_RC_OPEN_FILE_ERR -505
#define KN_RC_BAD_N_OR_F -506 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_CONSTRAINT -507 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_JACOBIAN -508 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_HESSIAN -509 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_CON_INDEX -510 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_JAC_INDEX -511 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_HESS_INDEX -512 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_CON_BOUNDS -513 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_VAR_BOUNDS -514 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_ILLEGAL_CALL -515 /*-- KNITRO CALL IS OUT OF SEQUENCE */
#define KN_RC_BAD_KCPTR -516 /*-- KNITRO PASSED A BAD KC POINTER */
#define KN_RC_NULL_POINTER -517 /*-- KNITRO PASSED A NULL ARGUMENT */
#define KN_RC_BAD_INIT_VALUE -518 /*-- APPLICATION INITIAL POINT IS BAD */
#define KN_RC_LICENSE_ERROR -520 /*-- LICENSE CHECK FAILED */
#define KN_RC_BAD_PARAMINPUT -521 /*-- INVALID PARAMETER INPUT */
#define KN_RC_LINEAR_SOLVER_ERR -522 /*-- ERROR IN LINEAR SOLVER */
#define KN_RC_DERIV_CHECK_FAILED -523 /*-- DERIVATIVE CHECK FAILED */
#define KN_RC_DERIV_CHECK_TERMINATE -524 /*-- DERIVATIVE CHECK TERMINATE */
#define KN_RC_OVERFLOW_ERR -525 /*-- INTEGER OVERFLOW ERROR */
#define KN_RC_BAD_SIZE -526 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_VARIABLE -527 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_VAR_INDEX -528 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_OBJECTIVE -529 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_OBJ_INDEX -530 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_RESIDUAL -531 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_BAD_RSD_INDEX -532 /*-- PROBLEM DEFINITION ERROR */
#define KN_RC_INTERNAL_ERROR -600 /*-- CONTACT support-knitro@artelys.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`.

3.7 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: The *hessopt* user option cannot be changed after calling *KN_solve()*. You must first call *KN_free()* and then reload the model before changing *hessopt* and solving again.

Note: In the pre-Knitro 11.0 API, user option names begin with *KTR_*, instead of *KN_*.

3.7.1 Index

User options are defined in the `knitro.h` and summarized in the following index. To see a more detailed description of an individual option and its possible values click on the option name. The importance of each option is related to its category (General, Derivatives, etc...), 1 being the most important parameters.

General options

Option name	Importance	Purpose
<i>algorithm</i>	1	Indicates which algorithm to use to solve the problem
<i>blasoption</i>	2	Specifies the BLAS/LAPACK function library to use for basic vector and matrix computations
<i>blasoptionlib</i>	3	Specifies a dynamic library name that contains object code for BLAS/LAPACK functions
<i>bndrange</i>	3	Specifies max limits on the magnitude of constraint and variable bounds
<i>cg_maxit</i>	2	Determines the maximum allowable number of inner conjugate gradient (CG) iterations
<i>cg_pmem</i>	3	Specifies number of nonzero elements per hessian column when computing preconditioner
<i>cg_precond</i>	2	Specifies whether or not to apply preconditioning during CG iterations in barrier algorithms
<i>cg_stoptol</i>	3	Relative stopping tolerance for CG subproblems
<i>convex</i>	1	Apply specializations/tunings often beneficial for convex models
<i>datacheck</i>	2	Specifies whether to perform more extensive data checks
<i>delta</i>	3	Specifies the initial trust region radius scaling factor
<i>eval_fcga</i>	3	Specifies that gradients are provided together with functions in one callback
<i>honorbnds</i>	1	Indicates whether or not to enforce satisfaction of simple variable bounds
<i>initpenalty</i>	3	Initial penalty value used in Knitro merit function
<i>linesearch_maxtrials</i>	3	Indicates the maximum allowable number of trial points during the linesearch
<i>linesearch</i>	2	Indicates which linesearch strategy to use for the Interior/Direct or SQP algorithm
<i>linsolver_ooc</i>	3	Indicates whether to use Intel MKL PARDISO out-of-core solve of linear systems
<i>linsolver</i>	2	Indicates which linear solver to use to solve linear systems arising in Knitro algorithms
<i>linsolver_pivottol</i>	3	Specifies the initial pivot threshold used in factorization routines
<i>objrange</i>	3	Specifies the extreme limits of the objective function for purposes of determining unboundedness
<i>presolve_tol</i>	3	Determines the tolerance used by the Knitro presolver
<i>presolve</i>	1	Determine whether or not to use the Knitro presolver
<i>restarts</i>	2	Specifies whether to enable automatic restarts
<i>restarts_maxit</i>	3	Maximum number of iterations before restarting when restarts are enabled
<i>scale</i>	1	Specifies whether to perform problem scaling
<i>soc</i>	3	Specifies whether or not to try second order corrections (SOC)

Derivatives options

Option name	Importance	Purpose
<i>derivcheck</i>	1	Determine whether or not to perform a derivative check on the model
<i>derivcheck_terminate</i>	3	Determine whether or not to terminate after the derivative check
<i>derivcheck_tol</i>	3	Specifies the relative tolerance used for detecting derivative errors
<i>derivcheck_type</i>	3	Specifies whether to use forward or central finite differencing for the derivative checker
<i>gradopt</i>	1	Specifies how to compute the gradients of the objective and constraint functions
<i>hessian_no_f</i>	3	Determines whether or not to allow Knitro to request Hessian evaluations without the objective component included.
<i>hessopt</i>	1	Specifies how to compute the (approximate) Hessian of the Lagrangian
<i>lmsize</i>	2	Specifies the number of limited memory pairs stored when approximating the Hessian

Termination options

Option name	Importance	Purpose
<i>feastol</i>	1	Specifies the final relative stopping tolerance for the feasibility error
<i>feastol_abs</i>	3	Specifies the final absolute stopping tolerance for the feasibility error
<i>fstopval</i>	2	Used to implement a custom stopping condition based on the objective function value
<i>ftol</i>	2	The optimization process will terminate if feasible and the relative change in the objective function is less than ftol
<i>ftol_iters</i>	3	Number of consecutive feasible iterations where the relative change in the objective function is less than ftol before Knitro stops
<i>infeastol</i>	2	Specifies the (relative) tolerance used for declaring infeasibility of a model
<i>maxfevals</i>	2	Specifies the maximum number of function evaluations before termination.
<i>maxit</i>	1	Specifies the maximum number of iterations before termination
<i>maxtime_cpu</i>	2	Specifies, in seconds, the maximum allowable CPU time before termination
<i>maxtime_real</i>	2	Specifies, in seconds, the maximum allowable real time before termination
<i>opttol</i>	1	Specifies the final relative stopping tolerance for the KKT (optimality) error
<i>opttol_abs</i>	1	Specifies the final absolute stopping tolerance for the KKT (optimality) error
<i>xtol</i>	1	The optimization process will terminate if the relative change of the solution point estimate is less than xtol
<i>xtol_iters</i>	3	Number of consecutive iterations where change of the solution point estimate is less than xtol before Knitro stops

Barrier options

Option name	Importance	Purpose
<i>bar_conic_enable</i>	1	Enable special treatments for conic constraints in the Interior/Direct algorithm
<i>bar_directiterations</i>	1	Controls the maximum number of consecutive conjugate gradient (CG) steps
<i>bar_feasible</i>	1	Specifies whether special emphasis is placed on getting and staying feasible
<i>bar_feasmodetol</i>	3	Specifies the tolerance in equation that determines whether Knitro will force subsequent iterates to remain feasible
<i>bar_initmu</i>	2	Specifies the initial value for the barrier parameter μ used
<i>bar_initpi_mpec</i>	3	Specifies the initial value for the MPEC penalty parameter π
<i>bar_initpt</i>	2	Indicates initial point strategy for x, slacks and multipliers
<i>bar_maxcrossit</i>	3	Specifies the maximum number of crossover iterations before termination
<i>bar_maxrefact</i>	3	Indicates the maximum number of refactorizations of the KKT system per iteration
<i>bar_murule</i>	1	Indicates which strategy to use for modifying the barrier parameter μ
<i>bar_penaltycon</i>	2	Indicates whether a penalty approach is applied to the constraints
<i>bar_penaltyrule</i>	3	Indicates which penalty parameter strategy to use for determining whether or not to accept a trial iterate
<i>bar_refinement</i>	3	Specifies whether to try to refine the barrier solution for better precision
<i>bar_relaxcons</i>	2	Indicates whether a relaxation approach is applied to the constraints
<i>bar_slackboundpush</i>	3	Indicates minimum amount by which initial slack variables are pushed inside the bounds
<i>bar_switchobj</i>	3	Indicates objective function used when the barrier algorithms switch to a pure feasibility phase
<i>bar_switchrule</i>	3	Indicates whether or not the barrier algorithms will allow switching from an optimality phase to a pure feasibility phase
<i>bar_watchdog</i>	3	Specifies whether to enable watchdog heuristic

Active-set options

Option name	Importance	Purpose
<i>act_lpalg</i>	3	Indicates which algorithm to use for linear programming (LP) subproblems (only when using Cplex or Xpress)
<i>act_lpfeastol</i>	3	Feasibility tolerance for the linear programming solver in the Knitro Active Set or SQP algorithms
<i>act_lppenalty</i>	1	Indicate whether to use penalty formulation for linear programming subproblems
<i>act_lppresolve</i>	3	Controls presolve for linear programming subproblems
<i>act_lpsolver</i>	1	Indicates which linear programming solver the Knitro Active Set or SQP algorithms use
<i>act_parametric</i>	1	Solve parametric linear programming subproblems instead of standard LPs
<i>act_qpalg</i>	1	Indicates which algorithm to use to solve quadratic programming (QP) subproblems
<i>cplexlibname</i>	3	Name of the Xpress library when <i>act_lpsolver</i> =KN_ACT_LPSOLVER_CPLEX
<i>xpresslibname</i>	3	Name of the Xpress library when <i>act_lpsolver</i> =KN_ACT_LPSOLVER_XPRESS

MIP options

Option name	Importance	Purpose
<i>mip_branchrule</i>	1	Specifies which branching rule to use for MIP branch and bound procedure
<i>mip_debug</i>	2	Specifies debugging level for MIP solution
<i>mip_gub_branch</i>	3	Specifies whether or not to branch on generalized upper bounds (GUBs)
<i>mip_heuristic</i>	1	Specifies which MIP heuristic search approach to apply
<i>mip_heuristic_maxit</i>	2	Specifies the maximum number of iterations to allow for MIP heuristic

Table 3.5 – continued from previous page

Option name	Importance	Purpose
<i>mip_heuristic_terminate</i>	2	Specifies the condition for terminating the MIP heuristic
<i>mip_implications</i>	2	Specifies whether or not to add constraints to the MIP derived from logical implications
<i>mip_integer_tol</i>	3	Specifies the threshold for deciding whether or not a variable is determined to be integer
<i>mip_integral_gap_abs</i>	1	The absolute integrality gap stop tolerance for MIP
<i>mip_integral_gap_rel</i>	1	The relative integrality gap stop tolerance for MIP
<i>mip_intvar_strategy</i>	2	Specifies how to handle integer variables
<i>mip_knapsack</i>	2	Specifies rules for adding MIP knapsack cuts
<i>mip_lpalg</i>	2	Specifies which algorithm to use for any linear programming (LP) subproblem solves
<i>mip_maxnodes</i>	2	Specifies the maximum number of nodes explored (0 means no limit)
<i>mip_maxsolves</i>	3	Specifies the maximum number of subproblem solves allowed (0 means no limit)
<i>mip_maxtime_cpu</i>	2	Specifies the maximum allowable CPU time in seconds for the complete MIP solve
<i>mip_maxtime_real</i>	1	Specifies the maximum allowable real time in seconds for the complete MIP solve
<i>mip_method</i>	1	Specifies which MIP method to use
<i>mip_nodealg</i>	1	Specifies which algorithm to use for standard node subproblem solves in MIP
<i>mip_outinterval</i>	1	Specifies node printing interval for <i>mip_outlevel</i> when <i>mip_outlevel</i> > 0
<i>mip_outlevel</i>	1	Specifies how much MIP information to print
<i>mip_outsub</i>	3	Specifies MIP subproblem solve debug output control
<i>mip_pseudoinit</i>	3	Specifies the method used to initialize pseudo-costs
<i>mip_relaxable</i>	2	Specifies whether integer variables are relaxable
<i>mip_rootalg</i>	2	Specifies which algorithm to use for the root node solve in MIP
<i>mip_rounding</i>	2	Specifies the MIP rounding rule to apply
<i>mip_selectdir</i>	2	Specifies the MIP node selection direction rule for choosing the next node in the branch and bound
<i>mip_selectrule</i>	1	Specifies the MIP select rule for choosing the next node in the branch and bound
<i>mip_strong_candlim</i>	3	Specifies the maximum number of candidates to explore for MIP strong branching
<i>mip_strong_level</i>	3	Specifies the maximum number of tree levels on which to perform MIP strong branching
<i>mip_strong_maxit</i>	3	Specifies the maximum number of iterations to allow for MIP strong branching
<i>mip_terminate</i>	1	Specifies conditions for terminating the MIP algorithm

Multi-algorithm options

Option name	Importance	Purpose
<i>ma_maxtime_cpu</i>	3	Specifies the maximum allowable CPU time before termination for the multi-algorithm procedure
<i>ma_maxtime_real</i>	2	Specifies the maximum allowable real time before termination for the multi-algorithm procedure
<i>ma_outsub</i>	1	Enable writing algorithm output to files for the multi-algorithm procedure
<i>ma_terminate</i>	1	Define the termination condition for the multi-algorithm procedure

Multistart options

Option name	Importance	Purpose
<i>ms_deterministic</i>	2	Indicates whether Knitro multi-start procedure will be deterministic
<i>ms_enable</i>	1	Indicates whether Knitro will solve from multiple start points to find a better local minimum
<i>ms_maxbndrange</i>	2	Specifies the maximum range that an unbounded variable can take when determining new start points
<i>ms_maxsolves</i>	1	Specifies how many start points to try in multi-start
<i>ms_maxtime_cpu</i>	3	Specifies, in seconds, the maximum allowable CPU time before termination
<i>ms_maxtime_real</i>	2	Specifies, in seconds, the maximum allowable real time before termination
<i>ms_num_to_save</i>	2	Specifies the number of distinct feasible points to save in a file named
<i>ms_outsub</i>	2	Enable writing algorithm output to files for the parallel multistart procedure
<i>ms_savetol</i>	2	Specifies the tolerance for deciding if two feasible points are distinct
<i>ms_seed</i>	2	Seed value used to generate random initial points in multi-start
<i>ms_startptrange</i>	2	Specifies the maximum range that each variable can take when determining new start points
<i>ms_terminate</i>	1	Specifies the condition for terminating multi-start
<i>par_mnumthreads</i>	2	Specify the number of threads to use for multistart

Parallelism options

Option name	Importance	Purpose
<i>par_blasnumthreads</i>	2	Specify the number of threads to use for BLAS operations
<i>par_concurrent_evals</i>	1	Determines whether or not function and derivative evaluations can take place concurrently in parallel
<i>par_lnumthreads</i>	2	Specify the number of threads to use for linear system solve operations
<i>par_numthreads</i>	1	Specify the number of threads to use for parallel (excluding BLAS) computing features

Output options

Option name	Importance	Purpose
<i>debug</i>	2	Controls the level of debugging output
<i>newpoint</i>	2	Specifies additional action to take after every iteration in a solve of a continuous problem
<i>out_csvinfo</i>	3	Specifies whether to create <i>knitro_solve.csv</i> information file
<i>out_csvname</i>	3	Specify non-default filename when using <i>out_csvinfo</i>
<i>out_hints</i>	2	Print diagnostic hints (e.g. on user option settings) after solving
<i>outappend</i>	2	Specifies whether output should be started in a new file, or appended to existing files
<i>outdir</i>	2	Specifies a single directory as the location to write all output files
<i>outlev</i>	1	Controls the level of output produced by Knitro
<i>outmode</i>	1	Specifies where to direct the output from Knitro
<i>outname</i>	2	Specify filename (default <i>knitro.log</i>) when directing output to a file via <i>outmode</i>

Tuner options

Option name	Importance	Purpose
<code>tuner</code>	1	Indicates whether to invoke the Knitro-Tuner
<code>tuner_maxtime_cpu</code>	2	Specifies the maximum allowable CPU time before terminating the Knitro-Tuner
<code>tuner_maxtime_real</code>	1	Specifies the maximum allowable real time before terminating the Knitro-Tuner
<code>tuner_optionsfile</code>	1	Can be used to specify the location of a Tuner options file
<code>tuner_outsub</code>	2	Enable writing additional Tuner subproblem solve output to files for the Knitro-Tuner procedure
<code>tuner_terminate</code>	1	Define the termination condition for the Knitro-Tuner procedure

3.7.2 General options

algorithm

`KN_PARAM_ALG`

```
#define KN_PARAM_ALGORITHM      1003
#define KN_PARAM_ALG            1003
# define KN_ALG_AUTOMATIC      0
# define KN_ALG_AUTO           0
# define KN_ALG_BAR_DIRECT     1
# define KN_ALG_BAR_CG         2
# define KN_ALG_ACT_CG         3
# define KN_ALG_ACT_SQP        4
# define KN_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

blasoption

`KN_PARAM_BLASOPTION`

```
#define KN_PARAM_BLASOPTION     1042
# define KN_BLASOPTION_KNITRO  0
# define KN_BLASOPTION_INTEL   1
# define KN_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. 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*. 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

KN_PARAM_BLASOPTIONLIB

```
#define KN_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*.

Note: This option has no effect unless *blasoption* = 2.

bndrange

KN_PARAM_BNDRANGE

```
#define KN_PARAM_BNDRANGE           1112
```

Specifies max limits on the magnitude of constraint and variable bounds. Any constraint or variable bounds whose magnitude is greater than or equal to *bndrange* will be treated as infinite by Knitro. Using very large, finite bounds is discouraged (and is generally an indication of a poorly scaled model).

Default value: 1.0e20

cg_maxit

KN_PARAM_CG_MAXIT

```
#define KN_PARAM_CG_MAXIT          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

cg_pmem

KN_PARAM_CG_PMEM

```
#define KN_PARAM_CG_PMEM          1103
```

Specifies the amount of nonzero elements per column of the Hessian of the Lagrangian which are retained when computing the incomplete Cholesky preconditioner.

- n At most $n > 0$ nonzero elements per column.

Default value: 10

cg_precond

KN_PARAM_CG_PRECOND

```
#define KN_PARAM_CG_PRECOND      1041
# define KN_CG_PRECOND_NONE      0
# define KN_CG_PRECOND_CHOL      1
```

Specifies whether an incomplete Cholesky preconditioner is applied during CG iterations in barrier algorithms.

- 0 (no) Not applied.
- 1 (chol) Preconditioner is applied.

Default value: 0

cg_stoptol

KN_PARAM_CG_STOPTOL

```
#define KN_PARAM_CG_STOPTOL      1099
```

Specifies the relative stopping tolerance used for the conjugate gradient (CG) subproblem solves.

Default value: 1.0e-2

convex

KN_PARAM_CONVEX

```
#define KN_PARAM_CONVEX          1114
# define KN_CONVEX_AUTO          0
# define KN_CONVEX_YES           1
```

Declare the problem as convex by setting `KN_CONVEX_YES`. Otherwise, Knitro will try to determine this automatically, but may only be able to do so for simple model forms. If your model is convex, setting this option to `KN_CONVEX_YES` will cause Knitro to apply specializations and tunings that are often beneficial for convex models to speed up the solution. Currently this option is only active for the Interior/Direct algorithm, but may be applied to other algorithms in the future.

Default value: 0

KN_PARAM_DATACHECK

```
#define KN_PARAM_DATACHECK      1087
# define KN_DATACHECK_NO        0
# define KN_DATACHECK_YES      1
```

Specifies whether to perform more extensive data checks to look for errors in the problem input to Knitro (in particular, this option looks for errors in the sparse Jacobian and/or sparse Hessian structure). The datacheck may have a non-trivial cost for large problems. It is turned on by default, but can be turned off for improved speed.

Default value: 1

delta

KN_PARAM_DELTA

```
#define KN_PARAM_DELTA          1020
```

Specifies the initial trust region radius scaling factor used to determine the initial trust region size.

Default value: 1.0e0

eval_fcga

KN_PARAM_EVAL_FCGA

```
#define KN_PARAM_EVAL_FCGA      1116
# define KN_EVAL_FCGA_NO        0
# define KN_EVAL_FCGA_YES      1
```

Use this option to tell Knitro that you are providing the first derivatives (i.e. gradients) in the same callback routine used for your function evaluations.

Default value: 0

honorbnds

KN_PARAM_HONORBND

```
#define KN_PARAM_HONORBND      1002
# define KN_HONORBND_NO        0
# define KN_HONORBND_ALWAYS    1
# define KN_HONORBND_INITPT    2
```

Indicates whether or not to enforce satisfaction of simple variable bounds throughout the optimization. The API function `KN_set_var_honorbnds()` can be used to set this option for each variable individually. 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

initpenalty

KN_PARAM_INITPENALTY

```
#define KN_PARAM_INITPENALTY 1097
```

Specifies the initial penalty parameter used in the Knitro merit functions. The Knitro merit functions are used to balance improvements in the objective function versus improvements in feasibility. A larger initial penalty value places more weight initially on feasibility in the merit function.

Default value: 1.0e1

linesearch

KN_PARAM_LINESEARCH

```
#define KN_PARAM_LINESEARCH 1095
# define KN_LINESEARCH_AUTO 0
# define KN_LINESEARCH_BACKTRACK 1
# define KN_LINESEARCH_INTERPOLATE 2
```

Indicates which linesearch strategy to use for the Interior/Direct or SQP algorithm to search for a new acceptable iterate. This option has no effect on the Interior/CG or Active Set algorithm.

- 0 (auto) Let Knitro automatically choose the strategy.
- 1 (backtrack) Use a simple backtracking scheme.
- 2 (interpolate) Use a cubic interpolation scheme.

Default value: 0

linesearch_maxtrials

KN_PARAM_LINESEARCH_MAXTRIALS

```
#define KN_PARAM_LINESEARCH_MAXTRIALS 1044
```

Indicates the maximum allowable number of trial points during the linesearch of the Interior/Direct or SQP algorithm before treating the linesearch step as a failure and generating a new step.

This option has no effect on the Interior/CG or Active Set algorithm.

Default value: 3

linsolver

KN_PARAM_LINSOLVER

```
#define KN_PARAM_LINSOLVER          1057
# define KN_LINSOLVER_AUTO          0
# define KN_LINSOLVER_INTERNAL      1
# define KN_LINSOLVER_HYBRID        2
# define KN_LINSOLVER_DENSEQR       3
# define KN_LINSOLVER_MA27          4
# define KN_LINSOLVER_MA57          5
# define KN_LINSOLVER_MKLPARDISO     6
# define KN_LINSOLVER_MA97          7
# define KN_LINSOLVER_MA86          8
```

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 (parallel, deterministic) sparse symmetric indefinite solver.
- 7 (ma97) Use the HSL MA97 (parallel, deterministic) sparse symmetric indefinite solver.
- 8 (ma86) Use the HSL MA86 (parallel, non-deterministic) sparse symmetric indefinite solver.

Default value: 0

Note: The QR linear solver, the HSL MA57/MA86/MA97 linear solvers and the Intel MKL PARDISO solver all make frequent use of Basic Linear Algebra Subroutines (BLAS) for internal linear algebra operations. If using any of these 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. You may also achieve speedups using multi-threaded BLAS with these solvers by setting `par_numthreads>1` or `par_blasnumthreads>1` when using the solvers.

Additionally, the HSL solvers MA86 and MA97 and the Intel MKL PARDISO solver are specifically designed to exploit parallelism (beyond BLAS parallelism) to achieve speedups on large problems. You may try setting `par_numthreads>1` or `par_lsnnumthreads>1` (with `par_blasnumthreads=1`) when using these solvers, to obtain greater speedups.

`linsolver_ooc`

`KN_PARAM_LINSOLVER_OOC`

```
#define KN_PARAM_LINSOLVER_OOC      1076
# define KN_LINSOLVER_OOC_NO        0
# define KN_LINSOLVER_OOC_MAYBE     1
# define KN_LINSOLVER_OOC_YES       2
```

Indicates whether to use Intel MKL PARDISO out-of-core solve of linear systems when `linsolver = mkl-pardiso`.

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.

`linsolver_pivottol`

`KN_PARAM_LINSOLVER_PIVOTTOL`

```
#define KN_PARAM_LINSOLVER_PIVOTTOL 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 `linsolver_pivottol` 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

`objrange`

`KN_PARAM_OBJRANGE`

```
#define KN_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

`presolve`

`KN_PARAM_PRESOLVE`

```
#define KN_PARAM_PRESOLVE 1059
# define KN_PRESOLVE_NONE 0
# define KN_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

KN_PARAM_PRESOLVE_TOL

```
#define KN_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

restarts

KN_PARAM_RESTARTS

```
#define KN_PARAM_RESTARTS          1100
```

Specifies whether or not to enable automatic restarts in Knitro. When enabled, if a Knitro algorithm seems to be converging slowly or not converging, the algorithm will automatically restart, which may help with convergence.

- 0 No automatic restarts allowed.
- n At most $n > 0$ automatic restarts may be performed.

Default value: 0

restarts_maxit

KN_PARAM_RESTARTS_MAXIT

```
#define KN_PARAM_RESTARTS_MAXIT    1101
```

When restarts are enabled, this option can be used to specify a maximum number of iterations before enforcing a restart.

- 0 No iteration limit on restarts enforced.
- n At most $n > 0$ iterations are allowed without convergence before enforcing an automatic restart, if restarts are enabled.

Default value: 0

scale

KN_PARAM_SCALE

```
#define KN_PARAM_SCALE              1017
# define KN_SCALE_NEVER             0
# define KN_SCALE_NO                0
# define KN_SCALE_USER_INTERNAL     1
# define KN_SCALE_USER_NONE        2
# define KN_SCALE_INTERNAL          3
```

Specifies whether to perform problem scaling of the objective function, constraint functions, or possibly variables.

If scaling is performed, internal computations, including some aspects of the optimality tests, are based on the scaled values, though the feasibility error is always computed in terms of the original, unscaled values.

- 0 (no) No scaling is performed.
- 1 (user_internal) User provided scaling is used if defined, otherwise Knitro internal scaling is applied.
- 2 (user_none) User provided scaling is used if defined, otherwise no scaling is applied.
- 3 (internal) Knitro internal scaling is applied.

Default value: 1

soc**KN_PARAM_SOC**

```
#define KN_PARAM_SOC          1019
# define KN_SOC_NO           0
# define KN_SOC_MAYBE       1
# define KN_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

3.7.3 Derivatives options

derivcheck**KN_PARAM_DERIVCHECK**

```
#define KN_PARAM_DERIVCHECK   1080
# define KN_DERIVCHECK_NONE  0
# define KN_DERIVCHECK_FIRST 1
# define KN_DERIVCHECK_SECOND 2
# define KN_DERIVCHECK_ALL    3
```

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 only.
- 2 (second) Check second derivatives (i.e. the Hessian) only.
- 3 (all) Check both first and second derivatives.

Default value: 0

derivcheck_terminate

KN_PARAM_DERIVCHECK_TERMINATE

```
#define KN_PARAM_DERIVCHECK_TERMINATE 1088
# define KN_DERIVCHECK_STOPERROR      1
# define KN_DERIVCHECK_STOPALWAYS     2
```

Determine whether to always terminate after the derivative check or only when the derivative checker detects a possible error.

- 1 (error) Terminate only when an error is detected.
- 2 (always) Always terminate when the derivative check is finished.

Default value: 1

derivcheck_tol**KN_PARAM_DERIVCHECK_TOL**

```
#define KN_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**KN_PARAM_DERIVCHECK_TYPE**

```
#define KN_PARAM_DERIVCHECK_TYPE      1081
# define KN_DERIVCHECK_FORWARD        1
# define KN_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

gradopt**KN_PARAM_GRADOPT**

```
#define KN_PARAM_GRADOPT              1007
# define KN_GRADOPT_EXACT              1
# define KN_GRADOPT_FORWARD            2
# define KN_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

KN_PARAM_HESSIAN_NO_F

```
#define KN_PARAM_HESSIAN_NO_F      1062
# define KN_HESSIAN_NO_F_FORBID    0
# define KN_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

KN_PARAM_HESSOPT

```
#define KN_PARAM_HESSOPT          1008
# define KN_HESSOPT_EXACT         1
# define KN_HESSOPT_BFGS         2
# define KN_HESSOPT_SR1          3
# define KN_HESSOPT_PRODUCT_FINDIFF 4
# define KN_HESSOPT_PRODUCT      5
# define KN_HESSOPT_LBFGS        6
# define KN_HESSOPT_GAUSS_NEWTON 7
```

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 (product_findiff) 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`).
- 7 (gauss_newton) Knitro computes a Gauss-Newton approximation of the hessian (available for least-squares only, and default value for least-squares)

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.

lmsize

KN_PARAM_LMSIZE

```
#define KN_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

3.7.4 Termination options

feastol

KN_PARAM_FEASTOL

```
#define KN_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

KN_PARAM_FEASTOLABS

```
#define KN_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

fstopval

KN_PARAM_FSTOPVAL

```
#define KN_PARAM_FSTOPVAL 1086
```

Used to implement a custom stopping condition based on the objective function value. Knitro will stop and declare that a satisfactory solution was found if a feasible objective function value at least as good as the value specified by *fstopval* is achieved. This stopping condition is only active when the absolute value of *fstopval* is less than *objrange*.

Default value: KN_INFINITY

ftol

KN_PARAM_FTOL

```
#define KN_PARAM_FTOL 1090
```

The optimization process will terminate if the relative change in the objective function is less than *ftol* for *ftol_iters* consecutive *feasible* iterations.

Default value: 1.0e-15

ftol_iters

KN_PARAM_FTOL_ITERS

```
#define KN_PARAM_FTOL_ITERS 1091
```

The optimization process will terminate if the relative change in the objective function is less than *ftol* for *ftol_iters* consecutive *feasible* iterations.

Default value: 5

infeastol

KN_PARAM_INFEASTOL

```
#define KN_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

maxfevals

KN_PARAM_MAXFEVALS

```
#define KN_PARAM_MAXFEVALS 1085
```

Specifies the maximum number of function evaluations before termination. Values less than zero imply no limit.

Default value: -1 (unlimited)

maxit

KN_PARAM_MAXIT

```
#define KN_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**KN_PARAM_MAXTIMECPU**

```
#define KN_PARAM_MAXTIMECPU 1024
```

Specifies, in seconds, the maximum allowable CPU time before termination.

Default value: 1.0e8

maxtime_real**KN_PARAM_MAXTIMEREAL**

```
#define KN_PARAM_MAXTIMEREAL 1040
```

Specifies, in seconds, the maximum allowable real time before termination.

Default value: 1.0e8

opttol**KN_PARAM_OPTTOL**

```
#define KN_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**KN_PARAM_OPTTOLABS**

```
#define KN_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

xtol

KN_PARAM_XTOL

```
#define KN_PARAM_XTOL 1030
```

The optimization process will terminate if the relative change in all components of the solution point estimate is less than *xtol* for *xtol_iters*. consecutive iterations. 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-12

xtol_iters

KN_PARAM_XTOL_ITERS

```
#define KN_PARAM_XTOL_ITERS 1094
```

The optimization process will terminate if the relative change in the solution estimate is less than *xtol* for *xtol_iters* consecutive iterations. If set to 0, Knitro chooses this value based on the solver and context. Currently Knitro sets this value to 3 unless the MISQP algorithm is being used, in which case the value is set to 1 by default.

Default value: 0

3.7.5 Barrier options

bar_conic_enable

KN_PARAM_BAR_CONIC_ENABLE

```
#define KN_PARAM_BAR_CONIC_ENABLE 1113
# define KN_BAR_CONIC_ENABLE_NONE 0
# define KN_BAR_CONIC_ENABLE_SOC 1
```

Enable special treatments for conic constraints when using the Interior/Direct algorithm (has no affect when using the Interior/CG algorithm).

- 0 (none) Do not apply any special treatment for conic constraints.
- 1 (soc) Apply special treatments for any Second Order Cone (SOC) constraints identified in the model.

Default value: 0

bar_directinterval

KN_PARAM_BAR_DIRECTINTERVAL

```
#define KN_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**KN_PARAM_BAR_FEASIBLE**

```
#define KN_PARAM_BAR_FEASIBLE      1006
# define KN_BAR_FEASIBLE_NO        0
# define KN_BAR_FEASIBLE_STAY      1
# define KN_BAR_FEASIBLE_GET       2
# define KN_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**KN_PARAM_BAR_FEASMODETOL**

```
#define KN_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**KN_PARAM_BAR_INITMU**

```
#define KN_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_initpi_mpec

KN_PARAM_BAR_INITPI_MPEC

```
#define KN_PARAM_BAR_INITPI_MPEC    1093
```

Specifies the initial value for the MPEC penalty parameter π used when solving problems with complementarity constraints using the barrier algorithms. If this value is non-positive, then Knitro uses an internal formula to initialize the MPEC penalty parameter.

Default value: 0.0

bar_initpt

KN_PARAM_BAR_INITPT

```
#define KN_PARAM_BAR_INITPT          1009
# define KN_BAR_INITPT_AUTO          0
# define KN_BAR_INITPT_CONVEX        1
# define KN_BAR_INITPT_NEARBND       2
# define KN_BAR_INITPT_CENTRAL       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 (convex) Initialization designed for convex models.
- 2 (nearbnd) Initialization strategy that stays closer to the bounds.
- 3 (central) Initialization strategy that is more central on double-bounded variables.

Default value: 0

bar_maxcrossit

KN_PARAM_BAR_MAXCROSSIT

```
#define KN_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

KN_PARAM_BAR_MAXREFACTOR

```
#define KN_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

KN_PARAM_BAR_MURULE

```
#define KN_PARAM_BAR_MURULE          1004
# define KN_BAR_MURULE_AUTOMATIC    0
# define KN_BAR_MURULE_AUTO         0
# define KN_BAR_MURULE_MONOTONE     1
# define KN_BAR_MURULE_ADAPTIVE     2
# define KN_BAR_MURULE_PROBING      3
# define KN_BAR_MURULE_DAMPMPMC     4
# define KN_BAR_MURULE_FULLMPC      5
# define KN_BAR_MURULE_QUALITY      6
```

Indicates which strategy to use for modifying the barrier parameter μ 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**KN_PARAM_BAR_PENCONS**

```
#define KN_PARAM_BAR_PENCONS          1050
# define KN_BAR_PENCONS_AUTO          0
# define KN_BAR_PENCONS_NONE          1
# define KN_BAR_PENCONS_ALL           2
# define KN_BAR_PENCONS_EQUALITIES    3
```

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.
- 3 (equalities) Apply a penalty approach to equality constraints only.

Default value: 0

bar_penaltyrule**KN_PARAM_BAR_PENRULE**

```
#define KN_PARAM_BAR_PENRULE          1049
# define KN_BAR_PENRULE_AUTO          0
# define KN_BAR_PENRULE_SINGLE        1
# define KN_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**KN_PARAM_BAR_REFINEMENT**

```
#define KN_PARAM_BAR_REFINEMENT       1079
# define KN_BAR_REFINEMENT_NO         0
# define KN_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

KN_PARAM_BAR_RELAXCONS

```
#define KN_PARAM_BAR_RELAXCONS      1077
# define KN_BAR_RELAXCONS_NONE      0
# define KN_BAR_RELAXCONS_EQS       1
# define KN_BAR_RELAXCONS_INEQS     2
# define KN_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_slackboundpush

KN_PARAM_BAR_SLACKBOUNDPUSH

```
#define KN_PARAM_BAR_SLACKBOUNDPUSH  1102
```

Specifies the amount by which the barrier slack variables are initially pushed inside the bounds. A smaller value may be preferable when warm-starting from a point close to the solution.

Default value: 1.0e-1

bar_switchobj

KN_PARAM_BAR_SWITCHOBJ

```
#define KN_PARAM_BAR_SWITCHOBJ      1104
# define KN_BAR_SWITCHOBJ_NONE      0
# define KN_BAR_SWITCHOBJ_SCALARPROX 1
# define KN_BAR_SWITCHOBJ_DIAGPROX  2
```

Indicates which objective function to use when the barrier algorithms switch to a pure feasibility phase.

- 0 (none) No (or zero) objective.
- 1 (scalarprox) Proximal point objective with scalar weighting.
- 2 (diagprox) Proximal point objective with diagonal weighting.

Default value: 1

bar_switchrule

KN_PARAM_BAR_SWITCHRULE

```
#define KN_PARAM_BAR_SWITCHRULE      1061
# define KN_BAR_SWITCHRULE_AUTO      0
# define KN_BAR_SWITCHRULE_NEVER     1
# define KN_BAR_SWITCHRULE_MODERATE  2
# define KN_BAR_SWITCHRULE_AGGRESSIVE 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 (moderate) Allow switches to feasibility phase.
- 3 (aggressive) Use a more aggressive switching rule.

Default value: 0

bar_watchdog

KN_PARAM_BAR_WATCHDOG

```
#define KN_PARAM_BAR_WATCHDOG        1089
# define KN_BAR_WATCHDOG_NO          0
# define KN_BAR_WATCHDOG_YES         1
```

Specifies whether to enable watchdog heuristic for barrier algorithms. In general, enabling the watchdog heuristic makes the barrier algorithms more likely to accept trial points. Specifically, the watchdog heuristic may occasionally accept trial points that increase the merit function, provided that subsequent iterates decrease the merit function.

Default value: 0

3.7.6 Active-set options

act_lpalg

KN_PARAM_ACT_LPALG

```
#define KN_PARAM_ACT_LPALG           1109
# define KN_ACT_LPALG_DEFAULT        0
# define KN_ACT_LPALG_PRIMAL         1
# define KN_ACT_LPALG_DUAL           2
# define KN_ACT_LPALG_BARRIER       3
```

Indicates which algorithm to use to solve linear programming (LP) subproblems when using the Knitro Active Set or SQP algorithms.

This option is currently only active when using the CPLEX(R) or Xpress(R) LP solvers chosen via *act_lpsolver*.

This option has no effect on the Interior/Direct and Interior/CG algorithms.

- 0 (default) use the default algorithm for the chosen LP solver.
- 1 (primal) use a primal simplex algorithm.
- 2 (dual) use a dual simplex algorithm.

- 3 (barrier) use a barrier/interior-point algorithm.

Default value: 0

act_lpfeastol

KN_PARAM_ACT_LPFEASTOL

```
#define KN_PARAM_ACT_LPFEASTOL      1098
```

Specifies the feasibility tolerance used for linear programming subproblems solved when using the Active Set or SQP algorithms.

Default value: 1.0e-8

act_lppenalty

KN_PARAM_ACT_LPPENALTY

```
#define KN_PARAM_ACT_LPPENALTY      1111
# define KN_ACT_LPPENALTY_ALL        1
# define KN_ACT_LPPENALTY_NONLINEAR  2
# define KN_ACT_LPPENALTY_DYNAMIC    3
```

Indicates whether to use a penalty formulation for linear programming subproblems in the Knitro Active Set or SQP algorithms.

- 1 (all) penalize all constraints.
- 2 (nonlinear) penalize only nonlinear constraints.
- 3 (dynamic) dynamically choose which constraints to penalize.

Default value: 1

act_lppresolve

KN_PARAM_ACT_LPPRESOLVE

```
#define KN_PARAM_ACT_LPPRESOLVE      1110
# define KN_ACT_LPPRESOLVE_OFF        0
# define KN_ACT_LPPRESOLVE_ON         1
```

Indicates whether to apply a presolve for linear programming subproblems in the Knitro Active Set or SQP algorithms.

- 0 (off) presolve turned off for LP subproblems.
- 1 (on) presolve turned on for LP subproblems.

Default value: 0

act_lpsolver

KN_PARAM_ACT_LPSOLVER

```
#define KN_PARAM_ACT_LPSOLVER        1012
# define KN_ACT_LPSOLVER_INTERNAL     1
# define KN_ACT_LPSOLVER_CPLEX        2
# define KN_ACT_LPSOLVER_XPRESS       3
```

Indicates which linear programming simplex solver the Knitro Active Set or SQP algorithms use 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 KN_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 KN_solve() is called.

Default value: 1

If `act_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):

```
KN_set_char_param_by_name (kc, "cplexlibname", "cplex90.dll");
```

If `act_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.

act_parametric

KN_PARAM_ACT_PARAMETRIC

```
#define KN_PARAM_ACT_PARAMETRIC      1107
# define KN_ACT_PARAMETRIC_NO        0
# define KN_ACT_PARAMETRIC_MAYBE     1
# define KN_ACT_PARAMETRIC_YES       2
```

Indicates whether to use a parametric approach when solving linear programming (LP) subproblems when using the Knitro Active Set or SQP algorithms. A parametric approach will solve a sequence of closely related LPs instead of one LP. It may increase the cost of an active-set iteration, but perhaps lead to convergence in fewer iterations.

- 0 (no) do not use a parametric solve (i.e. solve a single LP).
- 1 (maybe) use a parametric solve sometimes.
- 2 (yes) always try a parametric solve.

Default value: 1

act_qpalg

KN_PARAM_ACT_QPALG

```
#define KN_PARAM_ACT_QPALG          1092
# define KN_ACT_QPALG_AUTO          0
# define KN_ACT_QPALG_BAR_DIRECT    1
# define KN_ACT_QPALG_BAR_CG        2
# define KN_ACT_QPALG_ACT_CG        3
```

Indicates which algorithm to use to solve quadratic programming (QP) subproblems when using the Knitro Active Set or SQP algorithms.

This option has no effect on the Interior/Direct and Interior/CG algorithms.

- 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.

Default value: 0

cplexlibname

KN_PARAM_CPLEXLIB

```
#define KN_PARAM_CPLEXLIB          1048
```

See option *act_lpsolver*.

xpresslibname

KN_PARAM_XPRESSLIB

```
#define KN_PARAM_XPRESSLIB          1069
```

See option *act_lpsolver*.

3.7.7 MIP options

mip_branchrule

KN_PARAM_MIP_BRANCHRULE

```
#define KN_PARAM_MIP_BRANCHRULE    2002
# define KN_MIP_BRANCH_AUTO        0
# define KN_MIP_BRANCH_MOSTFRAC    1
# define KN_MIP_BRANCH_PSEUDOCOST  2
# define KN_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 (pseudocost) 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

KN_PARAM_MIP_DEBUG

```
#define KN_PARAM_MIP_DEBUG          2013
# define KN_MIP_DEBUG_NONE         0
# define KN_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

KN_PARAM_MIP_GUB_BRANCH

```
#define KN_PARAM_MIP_GUB_BRANCH     2015 /*-- BRANCH ON GENERALIZED BOUNDS */
# define KN_MIP_GUB_BRANCH_NO      0
# define KN_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

KN_PARAM_MIP_HEURISTIC

```
#define KN_PARAM_MIP_HEURISTIC     2022
# define KN_MIP_HEURISTIC_AUTO     0
# define KN_MIP_HEURISTIC_NONE     1
# define KN_MIP_HEURISTIC_FEASPUMP 2
# define KN_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

KN_PARAM_MIP_HEURISTIC_MAXIT

```
#define KN_PARAM_MIP_HEUR_MAXIT      2023
```

Specifies the maximum number of iterations to allow for MIP heuristic, if one is enabled.

Default value: 100

mip_heuristic_terminate**KN_PARAM_MIP_HEUR_TERMINATE**

```
#define KN_PARAM_MIP_HEUR_TERMINATE  2033
# define KN_MIP_HEUR_TERMINATE_FEASIBLE 1
# define KN_MIP_HEUR_TERMINATE_LIMIT   2
```

Specifies the condition for terminating the MIP heuristic.

- 1 (feasible) Terminate at first feasible point or iteration limit (whichever comes first).
- 2 (limit) Always run to the iteration limit.

Default value: 1

mip_implications**KN_PARAM_MIP_IMPLICATNS**

```
#define KN_PARAM_MIP_IMPLICATNS      2014 /*-- USE LOGICAL IMPLICATIONS */
# define KN_MIP_IMPLICATNS_NO        0
# define KN_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**KN_PARAM_MIP_INTEGERTOL**

```
#define KN_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**KN_PARAM_MIP_INTGAPABS**

```
#define KN_PARAM_MIP_INTGAPABS       2004
```

The absolute integrality gap stop tolerance for MIP.

Default value: 1.0e-6

mip_integral_gap_rel

KN_PARAM_MIP_INTGAPREL

```
#define KN_PARAM_MIP_INTGAPREL      2005
```

The relative integrality gap stop tolerance for MIP.

Default value: 1.0e-6

mip_intvar_strategy

KN_PARAM_MIP_INTVAR_STRATEGY

```
#define KN_PARAM_MIP_INTVAR_STRATEGY 2030
# define KN_MIP_INTVAR_STRATEGY_NONE 0
# define KN_MIP_INTVAR_STRATEGY_RELAX 1
# define KN_MIP_INTVAR_STRATEGY_MPEC 2
```

Specifies how to handle integer variables.

- 0 (none) No special treatment applied.
- 1 (relax) Relax all integer variables.
- 2 (mpec) Convert all binary variables to complementarity constraints.

Default value: 0

mip_knapsack

KN_PARAM_MIP_KNAPSACK

```
#define KN_PARAM_MIP_KNAPSACK      2016 /*-- KNAPSACK CUTS */
# define KN_MIP_KNAPSACK_NO        0 /*-- NONE */
# define KN_MIP_KNAPSACK_INEQ      1 /*-- ONLY FOR INEQUALITIES */
# define KN_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

KN_PARAM_MIP_LPALG

```
#define KN_PARAM_MIP_LPALG      2019
# define KN_MIP_LPALG_AUTO      0
# define KN_MIP_LPALG_BAR_DIRECT 1
# define KN_MIP_LPALG_BAR_CG    2
# define KN_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

KN_PARAM_MIP_MAXNODES

```
#define KN_PARAM_MIP_MAXNODES 2021
```

Specifies the maximum number of nodes explored (0 means no limit).

Default value: 100000

mip_maxsolves

KN_PARAM_MIP_MAXSOLVES

```
#define KN_PARAM_MIP_MAXSOLVES 2008
```

Specifies the maximum number of subproblem solves allowed (0 means no limit).

Default value: 200000

mip_maxtime_cpu

KN_PARAM_MIP_MAXTIMECPU

```
#define KN_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

KN_PARAM_MIP_MAXTIMEREAL

```
#define KN_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

KN_PARAM_MIP_METHOD

```
#define KN_PARAM_MIP_METHOD      2001
# define KN_MIP_METHOD_AUTO      0
# define KN_MIP_METHOD_BB       1
# define KN_MIP_METHOD_HQG      2
# define KN_MIP_METHOD_MISQP    3
```

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).
- 3 (MISQP) Use mixed-integer SQP method (allows for non-relaxable integer variables).

Default value: 0

mip_nodealg

KN_PARAM_MIP_NODEALG

```
#define KN_PARAM_MIP_NODEALG    2032
# define KN_MIP_NODEALG_AUTO    0
# define KN_MIP_NODEALG_BAR_DIRECT 1
# define KN_MIP_NODEALG_BAR_CG  2
# define KN_MIP_NODEALG_ACT_CG  3
# define KN_MIP_NODEALG_ACT_SQP 4
# define KN_MIP_NODEALG_MULTI   5
```

Specifies which algorithm to use for standard node subproblem solves in MIP (same options as *algorithm* user option).

Default value: 0

mip_outinterval

KN_PARAM_MIP_OUTINTERVAL

```
#define KN_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

KN_PARAM_MIP_OUTLEVEL

```
#define KN_PARAM_MIP_OUTLEVEL    2010
# define KN_MIP_OUTLEVEL_NONE    0
# define KN_MIP_OUTLEVEL_ITERS   1
```

```
# define KN_MIP_OUTLEVEL_ITERSTIME      2
# define KN_MIP_OUTLEVEL_ROOT          3
```

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.
- 3 (root) Also print detailed log from root node solve.

Default value: 1

mip_outsub

KN_PARAM_MIP_OUTSUB

```
#define KN_PARAM_MIP_OUTSUB             2012
# define KN_MIP_OUTSUB_NONE            0
# define KN_MIP_OUTSUB_YES             1
# define KN_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

KN_PARAM_MIP_PSEUDOINIT

```
#define KN_PARAM_MIP_PSEUDOINIT        2026
# define KN_MIP_PSEUDOINIT_AUTO        0
# define KN_MIP_PSEUDOINIT_AVE        1
# define KN_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_relaxable

KN_PARAM_MIP_RELAXABLE

```
#define KN_PARAM_MIP_RELAXABLE      2031
# define KN_MIP_RELAXABLE_NONE      0
# define KN_MIP_RELAXABLE_ALL       1
```

Specifies Whether integer variables are relaxable.

- 0 (none) Integer variables are not relaxable.
- 1 (all) All integer variables are relaxable.

Default value: 1

mip_rootalg

KN_PARAM_MIP_ROOTALG

```
#define KN_PARAM_MIP_ROOTALG        2018
# define KN_MIP_ROOTALG_AUTO        0
# define KN_MIP_ROOTALG_BAR_DIRECT  1
# define KN_MIP_ROOTALG_BAR_CG      2
# define KN_MIP_ROOTALG_ACT_CG      3
# define KN_MIP_ROOTALG_ACT_SQP     4
# define KN_MIP_ROOTALG_MULTI       5
```

Specifies which algorithm to use for the root node solve in MIP (same options as *algorithm* user option).

Default value: 0

mip_rounding

KN_PARAM_MIP_ROUNDING

```
#define KN_PARAM_MIP_ROUNDING        2017
# define KN_MIP_ROUND_AUTO           0
# define KN_MIP_ROUND_NONE           1 /*-- DO NOT ATTEMPT ROUNDING */
# define KN_MIP_ROUND_HEURISTIC      2 /*-- USE FAST HEURISTIC */
# define KN_MIP_ROUND_NLP_SOME       3 /*-- SOLVE NLP IF LIKELY TO WORK */
# define KN_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) No rounding heuristic is used.
- 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_selectdir

KN_PARAM_MIP_SELECTDIR

```
#define KN_PARAM_MIP_SELECTDIR      2034
# define KN_MIP_SELECTDIR_DOWN       0
# define KN_MIP_SELECTDIR_UP         1
```

Specifies the MIP node selection direction rule (for tiebreakers) for choosing the next node in the branch and bound tree.

- 0 (down) Choose the *down* (i.e. \leq) node first.
- 1 (up) Choose the *up* (i.e. \geq) node first.

Default value: 0

mip_selectrule

KN_PARAM_MIP_SELECTRULE

```
#define KN_PARAM_MIP_SELECTRULE      2003
# define KN_MIP_SEL_AUTO              0
# define KN_MIP_SEL_DEPTHFIRST       1
# define KN_MIP_SEL_BESTBOUND        2
# define KN_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

KN_PARAM_MIP_STRONG_CANDLIM

```
#define KN_PARAM_MIP_STRONG_CANDLIM  2028
```

Specifies the maximum number of candidates to explore for MIP strong branching.

Default value: 10

mip_strong_level

KN_PARAM_MIP_STRONG_LEVEL

```
#define KN_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

KN_PARAM_MIP_STRONG_MAXIT

```
#define KN_PARAM_MIP_STRONG_MAXIT     2027
```

Specifies the maximum number of iterations to allow for MIP strong branching solves.

Default value: 1000

mip_terminate**KN_PARAM_MIP_TERMINATE**

```
#define KN_PARAM_MIP_TERMINATE      2020
# define KN_MIP_TERMINATE_OPTIMAL  0
# define KN_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

3.7.8 Multi-algorithm options

ma_maxtime_cpu**KN_PARAM_MA_MAXTIMECPU**

```
#define KN_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**KN_PARAM_MA_MAXTIMEREAL**

```
#define KN_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**KN_PARAM_MA_OUTSUB**

```
#define KN_PARAM_MA_OUTSUB          1067
# define KN_MA_OUTSUB_NONE         0
# define KN_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

KN_PARAM_MA_TERMINATE

```
#define KN_PARAM_MA_TERMINATE      1063
# define KN_MA_TERMINATE_ALL      0
# define KN_MA_TERMINATE_OPTIMAL  1
# define KN_MA_TERMINATE_FEASIBLE 2
# define KN_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

3.7.9 Multistart options

ms_deterministic

KN_PARAM_MSDETERMINISTIC

```
#define KN_PARAM_MSDETERMINISTIC  1078
# define KN_MSDETERMINISTIC_NO    0
# define KN_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

KN_PARAM_MULTISTART

```
#define KN_PARAM_MULTISTART      1033
# define KN_MULTISTART_NO        0
# define KN_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

KN_PARAM_MS_MAXBNDRANGE

```
#define KN_PARAM_MS_MAXBNDRANGE 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

$$\max\{b_i^L, x_i^0 - \text{ms_maxbndrange}/2\} \leq x_i \leq \min\{b_i^U, x_i^0 + \text{ms_maxbndrange}/2\},$$

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

KN_PARAM_MS_MAXSOLVES

```
#define KN_PARAM_MS_MAXSOLVES 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

KN_PARAM_MS_MAXTIMECPU

```
#define KN_PARAM_MS_MAXTIMECPU 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

KN_PARAM_MS_MAXTIMEREAL

```
#define KN_PARAM_MS_MAXTIMEREAL 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**KN_PARAM_MSNUMTOSAVE**

```
#define KN_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**KN_PARAM_MS_OUTSUB**

```
#define KN_PARAM_MS_OUTSUB        1068
# define KN_MS_OUTSUB_NONE        0
# define KN_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**KN_PARAM_MSSAVETOL**

```
#define KN_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**KN_PARAM_MSSEED**

```
#define KN_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**KN_PARAM_MSSTARTPTRANGE**

```
#define KN_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 or equal to *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 *ms_startptrange*, then new start point values are restricted by the option. If x_i is such a variable, then all initial values satisfy

$$\max\{b_i^L, x_i^0 - \tau\} \leq x_i \leq \min\{b_i^U, x_i^0 + \tau\},$$

$$\tau = \min\{\text{ms_startptrange}/2, \text{ms_maxbndrange}/2\}$$

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

KN_PARAM_MSTERMINATE

```
#define KN_PARAM_MSTERMINATE 1054
# define KN_MSTERMINATE_MAXSOLVES 0
# define KN_MSTERMINATE_OPTIMAL 1
# define KN_MSTERMINATE_FEASIBLE 2
# define KN_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

par_msnthreads

KN_PARAM_PAR_MSNUMTHREADS

```
#define KN_PARAM_PAR_MSNUMTHREADS 3005
# define KN_PAR_MSNUMTHREADS_AUTO 0
```

Specify the number of threads to use for multistart (when *ms_enable* = 1).

- 0 (auto) Let Knitro choose the number of threads (currently sets *par_msnthreads* to *par_numthreads*).
- $n > 0$ Use n threads for the multistart (solve n problems in parallel).

Default value: 0

3.7.10 Parallelism options

`par_blasnumthreads`

`KN_PARAM_PAR_BLASNUMTHREADS`

```
#define KN_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`

`KN_PARAM_PAR_CONCURRENT_EVALS`

```
#define KN_PARAM_PAR_CONCURRENT_EVALS 3002
# define KN_PAR_CONCURRENT_EVALS_NO 0
# define KN_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_lnumthreads`

`KN_PARAM_PAR_LSNUMTHREADS`

```
#define KN_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`

`KN_PARAM_PAR_NUMTHREADS`

```
#define KN_PARAM_PAR_NUMTHREADS 3001
```

Specify the number of threads to use for parallel (excluding BLAS) computing features (see *Parallelism*).

Default value: 1

3.7.11 Output options

`debug`

KN_PARAM_DEBUG

```
#define KN_PARAM_DEBUG          1031
# define KN_DEBUG_NONE          0
# define KN_DEBUG_PROBLEM       1
# define KN_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

newpoint

KN_PARAM_NEWPOINT

```
#define KN_PARAM_NEWPOINT      1001
# define KN_NEWPOINT_NONE     0
# define KN_NEWPOINT_SAVEONE  1
# define KN_NEWPOINT_SAVEALL  2
```

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 λ . The “newpoint” feature in Knitro is currently only available for continuous problems (solved via `KN_solve()`).

- 0 (none) Knitro takes no additional action.
- 1 (saveone) Knitro writes x and λ to the file `knitro_newpoint.log`. Previous contents of the file are overwritten.
- 2 (saveall) Knitro appends x and λ 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.

Default value: 0

out_csvinfo

KN_PARAM_OUT_CSVINFO

```
#define KN_PARAM_OUT_CSVINFO   1096
# define KN_OUT_CSVINFO_NO     0
# define KN_OUT_CSVINFO_YES    1
```

Controls whether or not to generates a file `knitro_solve.csv` containing solve information in comma separated format.

- 0 (no) No solution information file is generated.
- 1 (yes) The `knitro_solve.csv` solution file is generated.

Default value: 0

out_csvname

KN_PARAM_OUT_CSVNAME

```
#define KN_PARAM_OUT_CSVNAME 1106
```

Use to specify a custom csv filename when using *out_csvinfo*.

Default value: knitro_solve.csv

out_hints

KN_PARAM_OUT_HINTS

```
#define KN_PARAM_OUT_HINTS 1115
# define KN_OUT_HINTS_NO 0
# define KN_OUT_HINTS_YES 1
```

Specifies whether to print diagnostic hints (e.g. about user option settings) after solving.

- 0 (no) Do not print any hints.
- 1 (yes) Print diagnostic hints on occasion.

Default value: 1

outappend

KN_PARAM_OUTAPPEND

```
#define KN_PARAM_OUTAPPEND 1046
# define KN_OUTAPPEND_NO 0
# define KN_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

outdir

KN_PARAM_OUTDIR

```
#define KN_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.

outlev

KN_PARAM_OUTLEV

```
#define KN_PARAM_OUTLEV          1015
# define KN_OUTLEV_NONE          0
# define KN_OUTLEV_SUMMARY       1
# define KN_OUTLEV_ITER_10       2
# define KN_OUTLEV_ITER          3
# define KN_OUTLEV_ITER_VERBOSE  4
# define KN_OUTLEV_ITER_X        5
# define KN_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 λ .

Default value: 2

outmode

KN_PARAM_OUTMODE

```
#define KN_PARAM_OUTMODE          1016
# define KN_OUTMODE_SCREEN       0
# define KN_OUTMODE_FILE         1
# define KN_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

outname

KN_PARAM_OUTNAME

```
#define KN_PARAM_OUTNAME          1105
```

Use to specify a custom filename when output is written to a file using `outmode`.

Default value: `knitro.log`

3.7.12 Tuner options

tuner

KN_PARAM_TUNER

```
#define KN_PARAM_TUNER          1070
#  define KN_TUNER_OFF          0
#  define KN_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_maxtime_cpu**KN_PARAM_TUNER_MAXTIMECPU**

```
#define KN_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**KN_PARAM_TUNER_MAXTIMEREAL**

```
#define KN_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_optionsfile**KN_PARAM_TUNER_OPTIONSFILE**

```
#define KN_PARAM_TUNER_OPTIONSFILE  1071
```

Can be used to specify the location of a Tuner options file (see *The Knitro-Tuner*).

Default value: NULL

tuner_outsub**KN_PARAM_TUNER_OUTSUB**

```
#define KN_PARAM_TUNER_OUTSUB      1074
#  define KN_TUNER_OUTSUB_NONE      0
#  define KN_TUNER_OUTSUB_SUMMARY    1
#  define KN_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

KN_PARAM_TUNER_TERMINATE

```
#define KN_PARAM_TUNER_TERMINATE    1075
#  define KN_TUNER_TERMINATE_ALL     0
#  define KN_TUNER_TERMINATE_OPTIMAL 1
#  define KN_TUNER_TERMINATE_FEASIBLE 2
#  define KN_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

3.8 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_summary.csv`, `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 files are generated.

3.9 Knitro 10.x and Earlier Callable Library API

All functions offered by the Knitro callable library are listed here.

3.9.1 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 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.

3.9.2 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_lsq_init_problem()` or `KTR_mip_init_problem()`, and cannot 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_param_by_name()

```
int KNITRO_API KTR_set_param_by_name
(KTR_context_ptr kc, const char * const name, const double value);
```

Set an integer or 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 type of the 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 the 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 the 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_set_compcons()`. 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_lsq_init_problem()` / `KTR_mip_init_problem()` and after any calls to `KTR_set_compcons()`. It must be called before calling `KTR_solve()` / `KTR_mip_solve()`. Returns 0 if OK, nonzero if error.

`KTR_set_var_scalings()`

```
int KNITRO_API KTR_set_var_scalings
(
    KTR_context_ptr kc,
    const double * const xScaleFactors,
    const double * const xScaleCenters);
```

Set an array of variable scaling and centering values (one for each variable) to perform a linear scaling:

```
x[i] = xScaleFactors[i] * xScaled[i] + xScaleCenters[i]
```

for each variable. These scaling factors should try to represent the “typical” values of the x variables so that the scaled variables ($xScaled$) used internally by Knitro are close to one. The values for `xScaleFactors` should be positive. If a non-positive value is specified, that variable will not be scaled. This routine must be called after calling `KTR_init_problem()` / `KTR_lsq_init_problem()` / `KTR_mip_init_problem()` and before calling `KTR_solve()` / `KTR_mip_solve()`. Returns 0 if OK, nonzero if error.

`KTR_set_con_scalings()`

```
int KNITRO_API KTR_set_con_scalings
(
    KTR_context_ptr kc,
    const double * const cScaleFactors,
    const double * const ccScaleFactors);
```

Set an array of constraint scaling values (one for each constraint) to perform a scaling:

```
cScaled[i] = cScaleFactors[i] * c[i]
```

for each constraint. These scaling factors should try to represent the “typical” values of the **inverse** of the constraint values c so that the scaled constraints ($cScaled$) used internally by Knitro are close to one. Scaling factors for standard constraints can be provided with `cScaleFactors`, while scalings for complementarity constraints can be specified with `ccScaleFactors`. The values for `cScaleFactors` / `ccScaleFactors` should be positive. If a non-positive value is specified, that constraint will use either the standard Knitro scaling (`KTR_SCALE_USER_INTERNAL`), or no scaling (`KTR_SCALE_USER_NONE`). This routine must be called after calling `KTR_init_problem()` / `KTR_lsq_init_problem()` / `KTR_mip_init_problem()` and before calling `KTR_solve()` / `KTR_mip_solve()`. Returns 0 if OK, nonzero if error.

`KTR_set_obj_scaling()`

```
int KNITRO_API KTR_set_obj_scaling
(
    KTR_context_ptr kc,
    const double objScaleFactor);
```

Set a scaling value for the objective function:

```
objScaled = objScaleFactor * obj
```

This scaling factor should try to represent the “typical” value of the **inverse** of the objective function value obj so that the scaled objective ($objScaled$) used internally by Knitro is close to one. The value for `objScaleFactor` should be positive. If a non-positive value is specified, then the objective will use either the standard Knitro scaling (`KTR_SCALE_USER_INTERNAL`), or no scaling (`KTR_SCALE_USER_NONE`). This routine must be called after

calling `KTR_init_problem()` / `KTR_lsq_init_problem()` / `KTR_mip_init_problem()` and 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_lsq_init_problem()` / `KTR_mip_init_problem()` and before calling `KTR_solve()` / `KTR_mip_solve()`. Returns 0 if OK, nonzero if error.

`KTR_set_linearvars()`

```
int KNITRO_API KTR_set_linearvars
(
    KTR_context_ptr kc,
    const int * const linearVars);
```

This API function can be used to identify which variables only appear linearly in the model (`KTR_LINEARVAR_YES`). This information can be used by Knitro to perform more extensive preprocessing. If a variable appears nonlinearly in any constraint or the objective (or if the user does not know) then it should be marked as `KTR_LINEARVAR_NO`. 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_lsq_init_problem()` / `KTR_mip_init_problem()` and before calling `KTR_solve()` / `KTR_mip_solve()`. Returns 0 if OK, nonzero if error.

`KTR_set_honorbnds()`

```
int KNITRO_API KTR_set_honorbnds
(
    KTR_context_ptr kc,
    const int * const honorBnds);
```

This API function can be used to identify which variables should satisfy their variable bounds throughout the optimization process (`KTR_HONORBND_ALWAYS`). The user option `KTR_PARAM_HONORBND` can be used to set ALL variables to honor their bounds. This routine takes precedence over the setting of `KTR_PARAM_HONORBND` and is used to customize the settings for individual variables. 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_lsq_init_problem()` / `KTR_mip_init_problem()` and before calling `KTR_solve()` / `KTR_mip_solve()`. Returns 0 if OK, nonzero if error.

3.9.3 Problem modification

`KTR_set_compcns()`

```
int KNITRO_API KTR_set_compcns (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 only be called once to set all the complementarity constraints in the model at one time. 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.

3.9.4 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.) A single call to `KTR_solve()`

The typical calling sequence is:

```
KTR_new
KTR_set_xxx_callback (set all the necessary callbacks)
KTR_init_problem
KTR_set_xxx_param (set any number of parameters)
KTR_solve
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_set_xxx_callback (set all the necessary callbacks)
KTR_init_problem
KTR_set_xxx_param (set any number of parameters)
KTR_solve
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
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 * xLoBnds,
const double * xUpBnds,
const int      m,
const int      * cType,
const double * cLoBnds,
const double * cUpBnds,
const int      nnzJ,
const int      * jacIndexVars,
const int      * jacIndexCons,
const int      nnzH,
const int      * hessIndexRows,
const int      * hessIndexCols,
const double * xInitial,
const double * 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()`, `KTR_lsq_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`, `KTR_OBJTYPE_CONSTANT`).
- `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 to *KTR_solve()*.
-

Note: Row numbers are in the range $0, \dots, n - 1$.

- *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 $0, \dots, n - 1$.

- *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_OR_SATISFACTORY`, then *x* contains the solution.
Reverse communications mode (**deprecated**): 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 (**deprecated**): 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 (**deprecated**). 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_OR_SATISFACTORY`, then *obj* contains the value of the objective function *f(x)* at the solution.
- *c* is an array of length *m* used only in reverse communications mode (**deprecated**). Set to NULL for callback mode.
- *objGrad* is an array of length *n* used only in reverse communications mode (**deprecated**). Set to NULL for callback mode.
- *jac* is an array of length *nnzJ* used only in reverse communications mode (**deprecated**). Set to NULL for callback mode.
- *hess* is an array of length *nnzH* used only in reverse communications mode (**deprecated**), and only if option *hessopt* is set to `KTR_HESSOPT_EXACT`. Set to NULL for callback mode.
- *hessVector* is an array of length *n* used only in reverse communications mode (**deprecated**), and only if option *hessopt* is set to `KTR_HESSOPT_PRODUCT`. Set to NULL for callback mode.
- *userParams* is a pointer to a structure used in callback functions. 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. 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_lsq_init_problem()`

```
int KNITRO_API KTR_lsq_init_problem(KTR_context_ptr      kc,
                                    const int             n,
                                    const double * const xLoBnds,
                                    const double * const xUpBnds,
                                    const int             m,
                                    const int * const     rType,
                                    const int             nnzJ,
                                    const int * const     jacIndexVars,
                                    const int * const     jacIndexRes,
                                    const double * const xInitial,
                                    const double * const lambdaInitial);
```

`KTR_lsq_init_problem()` is used to initialize a nonlinear least squares problem.

This function only varies from `KTR_init_problem()` by the use of arguments specific to least squares problems, namely:

- `m` is the number of residuals
- `rType` is an array of length `m` that describes the types of the residuals (`KTR_RESTYPE_GENERAL` or `KTR_RESTYPE_LINEAR`)

Returns 0 if OK, nonzero if error.

`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_set_intvar_strategy()

```

int KNITRO_API KTR_mip_set_intvar_strategy
(
    KTR_context_ptr kc,
    const int      xIndex,
    const int      xStrategy);
    
```

Set strategies for dealing with individual integer variables. Possible strategy values include:

```

KTR_MIP_INTVAR_STRATEGY_NONE    0
KTR_MIP_INTVAR_STRATEGY_RELAX   1
KTR_MIP_INTVAR_STRATEGY_MPEC    2
    
```

The parameter *xIndex* should be an index value corresponding to an integer variable (nothing is done if the index value corresponds to a continuous variable), and *xStrategy* should correspond to one of the strategy values listed above. The default strategy is `KTR_MIP_INTVAR_STRATEGY_NONE`, and the strategy `KTR_MIP_INTVAR_STRATEGY_MPEC` can only be applied to binary variables. 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()`.

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 $\sqrt{\text{eps}}$ for forward finite differences, and $\text{eps}^{1/3}$ for central finite differences. Use this function to overwrite the default values. Array `relStepSizes` has length n . Any zero values will use Knitro default values, while non-zero values will overwrite default values. 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.

3.9.5 Callbacks

To solve a nonlinear optimization problem, Knitro needs the application to supply information at various trial points. Knitro specifies a trial point with a new vector of variable values x , and sometimes a corresponding vector of Lagrange multipliers λ . This information needs to be provided by the application through **callback** functions. The application provides C language function pointers that Knitro may call to evaluate the functions, gradients, and Hessians at the trial points.

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_PRODUCT_FINDIFF`, 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*. The user may use `KTR_RC_USER_TERMINATION` as a return value to stop the execution (for example, if the new point matches a criteria calculated in `KTR_newpt_callback`). In this case the Knitro final return code will be `KTR_RC_USER_TERMINATION` (“Knitro has been terminated by the user”).

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 λ 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 multistart procedure. On input, arguments x and λ 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.

3.9.6 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.

3.9.7 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_CONSTANT -1
#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()*).

KTR_LINEARVAR

```
#define KTR_LINEARVAR_NO 0
#define KTR_LINEARVAR_YES 1
```

Possible values to indicate whether a variable appears only in linear terms in the problem. Used by *KTR_set_linearvars()* function.

Artelys Knitro User's Manual is copyrighted 2001-2018 by Artelys.