ziena®

OPTIMIZATION INC.

KNITRO™ 5.1

User's Manual

Knitro User's Manual

Version 5.1

Richard A. Waltz
Todd D. Plantenga
Ziena Optimization, Inc.
www.ziena.com

July 2007

Contents

1 Introduction

This chapter gives an overview of the KNITRO optimization software package, and details concerning contact and support information.

1.1 Product Overview

KNITRO 5.1 is a software package for finding solutions of continuous, smooth optimization problems, with or without constraints. KNITRO is designed for finding local solutions, but multi-start heuristics are provided for trying to locate the global solution. Although KNITRO is designed for solving large-scale general nonlinear problems, it is efficient at solving all of the following classes of smooth optimization problems:

- unconstrained
- bound constrained
- equality constrained, both linear and nonlinear
- systems of nonlinear equations
- least squares problems, both linear and nonlinear
- linear programming problems (LPs)
- quadratic programming problems (QPs), both convex and nonconvex
- mathematical programs with complementarity constraints (MPCCs)
- general nonlinear constrained problems (NLP), both convex and nonconvex

The KNITRO package provides the following features:

- efficient and robust solution of small or large problems
- derivative-free, 1st derivative, and 2nd derivative options
- option to remain feasible throughout the optimization or not
- 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 (32-bit and 64-bit), Mac OS X (x86 and PowerPC), and Solaris
- programmatic interfaces: C/C++, Fortran, Java, Microsoft Excel
- modeling language interfaces: AMPL, AIMMS, GAMS, Mathematica, Matlab
- thread-safe libraries for easy embedding into application software

1.2 Algorithms Overview

The problems solved by KNITRO have the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) && \text{(1.1a)} \\ \text{subject to} \quad & h(x) = 0 && \text{(1.1b)} \\ & g(x) \leq 0, && \text{(1.1c)} \end{aligned}$$

where $x \in \mathbf{R}^n$. This formulation allows many types of constraints, including bounds on the variables. Complementarity constraints may also be included. KNITRO assumes that the functions $f(x)$, $h(x)$ and $g(x)$ are smooth, although problems with derivative discontinuities can often be solved successfully.

KNITRO implements three state-of-the-art interior-point and active-set methods for solving 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 three 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. For guidance on choosing the best algorithm see section 8.

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 μ. Trust regions and a merit function are used to promote convergence. 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 (1.1) 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 if it encounters difficulties.

Interior/CG algorithm: This method is similar to the Interior/Direct algorithm, except 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 conjugate gradient applied to approximately minimize a quadratic model of the barrier problem. The use of conjugate gradient on large-scale problems allows KNITRO to utilize exact second derivatives without forming the Hessian matrix.

Active Set algorithm: Active set methods solve a sequence of subproblems based on a quadratic model of the original problem. In contrast with interior-point methods, the algorithm seeks active inequalities and follows a more exterior path to the solution. KNITRO implements a sequential linear-quadratic programming (SLQP) algorithm, similar in nature to a sequential quadratic programming method but using linear programming subproblems to estimate the active set. This method may be preferable to interior-point algorithms when a good initial point can be provided; for example, when solving a sequence of related problems. KNITRO can also "crossover" from an interior-point method and apply Active Set to provide highly accurate active set and sensitivity information (see section 9.5).

For a detailed description of the algorithm implemented in Interior/CG see [4] and for the global convergence theory see [1]. The method implemented in Interior/Direct is described in [11]. The

Active Set algorithm is described in [3] and the global convergence theory for this algorithm is in [2]. A summary of the algorithms and techniques implemented in the KNITRO software product is given in [6]. An important component of KNITRO is the HSL routine MA27 [8] which is used to solve the linear systems arising at every iteration of the algorithm. In addition, the Active Set algorithm in KNITRO may make use of the COIN-OR Clp linear programming solver module. The version used in KNITRO may be downloaded from `http://www.ziena.com/clp.html`.

1.3 What's New in Version 5.1

- Two new KNITRO user options, "bar_maxrefactor" and "bar_maxbacktrack", are added for the Interior/Direct algorithm (usually the default algorithm in KNITRO). The options are useful on problems that revert to conjugate gradient (CG) on multiple iterations. KNITRO reverts to CG for robustness, but modifying "bar_maxrefactor" and/or "bar_maxbacktrack" can often improve performance without sacrificing robustness. See section 5.1.

- KNITRO user option "newpoint" is enhanced with built-in capabilities to save the most recent iterate to a file ("newpoint=saveone"), or all iterates to a file ("newpoint=saveall"). The saved iterate is especially useful if progress is slow and the user stops KNITRO before convergence. See section 5.1.

- KNITRO 5.1 ships with the Intel Math Kernel Library (MKL) for BLAS and LAPACK functions. Users can set "blasoption" to be the Intel MKL library, the default KNITRO implementation that is based on netlib, or any suitable BLAS/LAPACK dynamic library. BLAS/LAPACK computations are observed to account for 5-50% of KNITRO'S CPU usage. On an Intel processor, the Intel MKL versions perform BLAS/LAPACK operations roughly 20-30% faster than the default library. See section 5.1.

- Multi-start generation of new start points is improved, and several new KNITRO user options are provided: "ms_maxbndrange", "ms_maxtime_cpu", and "ms_maxtime_real". The first option lets users restrict the search space for start points to a small region around the initial user-provided start point. See sections 5.1 and 9.6.

- An object-oriented C++ test driver is provided in examples/C++. See section 4.3.

- A few KNITRO user options are renamed and simplified (for details, look up the new option names in section 5.1):

  - "initpt" and "shiftinit" are combined into "bar_initpt"
  - "barrule" is renamed to "bar_murule"
  - "feastolabs" is renamed to "feastol_abs"
  - "mu" is renamed to "bar_initmu"
  - "multistart" is renamed to "ms_enable"
  - "opttolabs" is renamed to "opttol_abs"

1.4 Contact and Support Information

KNITRO is licensed and supported by Ziena Optimization, Inc. (`http://www.ziena.com/`). General information regarding KNITRO can be found at the KNITRO website:

`http://www.ziena.com/knitro.html`

For technical support, contact your local distributor. If you purchased KNITRO directly from Ziena, you may send support questions or comments to

info@ziena.com

Questions regarding licensing information or other information about KNITRO can also be sent to

info@ziena.com

2 Installation

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.

If you purchased the Knitro/AMPL solver product, then refer to section 3 and test Knitro as the solver for any smooth optimization model (an AMPL test model is provided with the Knitro distribution).

If you purchased the full Knitro product, then test Knitro by compiling and running one or more programs in the examples directory. Example problems are provided for C, Fortran, and Java interfaces. We recommend understanding these examples and reading section 4 of this manual before proceeding with development of your own application interface.

2.1 Windows

Knitro is supported on Windows 2003, Windows XP SP2, and Windows XP Professional x64. There are compatibility problems with Windows XP SP1 system libraries – users should upgrade to Windows XP SP2. The Knitro 5.1 software package for Windows is delivered as a zipped file ending in .zip, or as a self-extracting executable ending in .exe. For the .zip file, double-click on it and extract all contents to a new folder. For the .exe file, double-click on it and follow the instructions. The self-extracting executable creates start menu shortcuts and an uninstall entry in Add/Remove Programs; otherwise, the two install methods are identical.

The default installation location for Knitro is (assuming your `%HOMEDRIVE%` is `C:`)

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

Unpacking will create a folder named knitro-5.x-z (or knitroampl-5.x-z for the Knitro/AMPL solver product). Contents of the full product distribution are the following:

INSTALL.txt:	A file containing installation instructions.
LICENSE_KNITRO.txt:	A file containing the Knitro license agreement.
README.txt:	A file with instructions on how to get started using Knitro.
KNITRO51-ReleaseNotes.txt:	A file containing 5.1 release notes.
get_machine_ID.exe:	An executable that identifies the machine ID, required for obtaining a Ziena license file.
doc:	A folder containing Knitro documentation, including this manual.
include:	A folder containing the Knitro header file knitro.h.
lib:	A folder containing the Knitro library and object files: knitro_objlib.a, knitro.lib and knitro.dll.
examples:	A folder containing examples of how to use the Knitro API in different programming languages (C, C++, Fortran, Java).

knitroampl:	A folder containing knitroampl.exe (the KNITRO solver for AMPL), instructions, and an example model for testing KNITRO with AMPL.

To activate KNITRO for your computer you will need a valid Ziena license file. If you purchased a floating network license, then refer to the *Ziena License Manager User's Manual*. For a stand-alone license, 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. This will generate a machine ID (five pairs of hexadecimal digits). Email the machine ID to info@ziena.com. Ziena will then send a license file containing the encrypted license text string. Ziena supports a variety of ways to install licenses. The simplest procedure is to copy each license into a file whose name begins with the characters `"ziena_"`. Then place the file in the folder

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

For more installation options and general troubleshooting, read the *Ziena License Manager User's Manual*.

The KNITRO 5.x install locations on Windows are a departure from the 4.0 locations. To run KNITRO 4.0 and 5.x on the same Windows machine, we recommend moving the 4.0 files to the new default location, but make sure the 4.0 and 5.x executables and DLL file are not mixed together when compiling or running an application. We recommend uninstalling KNITRO 4.0 and reinstalling to a new folder

```
C:\Program Files\Ziena\knitro-4.0\
```

The effect of reinstalling should be to move the following KNITRO 4.0 files from their default locations to a new location (assuming your `%HOMEDRIVE%` is `C:`, and `%SystemRoot%` is `C:\Windows`):

`C:\Program Files\knitro-4.0\*`	to `C:\Program Files\Ziena\knitro-4.0\`.
`C:\Windows\knitroampl.exe`	to `C:\Program Files\Ziena\knitro-4.0\`.
`C:\Windows\system32\knitro.dll`	to `C:\Program Files\Ziena\knitro-4.0\lib\`.

KNITRO 4.0 and 5.x also require separate Ziena license files. The Sentinel LM license manager is no longer supported, and 4.0 users must upgrade to use the Ziena license manager. Place both Ziena license files in `C:\Program Files\Ziena\`.

2.2 Unix (Linux, Mac OS X, Solaris)

KNITRO is supported on Linux (32-bit and 64-bit, all distributions), Mac OS X (32-bit x86 and 32-bit PowerPC, Mac OS X 10.4 or higher), and Solaris (SunOS 5.8). The KNITRO 5.1 software package for Unix is delivered as a gzipped tar file. Save this file in a fresh subdirectory on your system. To unpack, type the commands

```
gunzip knitro-5.x-platformname.tar.gz
tar -xvf knitro-5.x-platformname.tar
```

Unpacking will create a directory named knitro-5.x-z (or knitroampl-5.x-z for the KNITRO/AMPL solver product). Contents of the full product distribution are the following:

INSTALL:	A file containing installation instructions.
LICENSE_KNITRO:	A file containing the KNITRO license agreement.
README:	A file with instructions on how to get started using KNITRO.
KNITRO51-ReleaseNotes:	A file containing 5.1 release notes.
get_machine_ID:	An executable that identifies the machine ID, required for obtaining a Ziena license file.
doc:	A directory containing KNITRO documentation, including this manual.
include:	A directory containing the KNITRO header file knitro.h.
lib:	A directory containing the KNITRO library files: libknitro.a and libknitro.so.
examples:	A directory containing examples of how to use the KNITRO API in different programming languages (C, C++, Fortran, Java).
knitroampl:	A directory containing knitroampl (the KNITRO solver for AMPL), instructions, and an example model for testing KNITRO with AMPL.

To activate KNITRO for your computer you will need a valid Ziena license file. If you purchased a floating network license, then refer to the *Ziena License Manager User's Manual.* For a stand-alone license, execute `get_machine_ID`, a program supplied with the distribution. This will generate a machine ID (five pairs of hexadecimal digits). Email the machine ID to info@ziena.com. Ziena will then send a license file containing the encrypted license text string. Ziena supports a variety of ways to install licenses. The simplest procedure is to copy each license into a file whose name begins with the characters `"ziena_"` (please use lower-case letters). Then place the file in your `$HOME` directory. For more installation options and general troubleshooting, read the *Ziena License Manager User's Manual.*

The Mac OS X distribution contains MacIntosh "Universal Binary" objects, which means code runs on both PowerPC and Intel processors. Each object contains natively compiled code for each processor type, and the Mac OS X loader automatically chooses the correct binary for your machine. KNITRO therefore runs at maximum speed on all 32-bit Mac OS X machines, with no emulation.

2.3 Linux Compatibility Issues

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

```
../lib/libknitro.a(.text+0x28808): In function ‘ktr_xeb4’:
: undefined reference to ‘std::__default_alloc_template<true, 0>::deallo
cate(void*, unsigned int)’
../lib/libknitro.a(.text+0x28837): In function ‘ktr_xeb4’:
: undefined reference to ‘std::__default_alloc_template<true, 0>::deallo
cate(void*, unsigned int)’
../lib/libknitro.a(.text+0x290b0): more undefined references to ‘std::__
```

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

This indicates an incompatibility between the libstdc++ library on your Linux distribution and the library that KNITRO was built with. The incompatibilities may be caused by name-mangling differences between versions of the gcc compiler, and by differences in the Application Binary Interface of the two Linux distributions. The best fix is for Ziena to rebuild the KNITRO binaries on the same Linux distribution of your target machine (matching the Linux brand and release, and the gcc/g++ compiler versions). If you see these errors, please contact Ziena at info@ziena.com to correct the problem.

Another Linux link error sometimes seen when using the programs in examples/C is the following:

```
./callback1_dynamic: error while loading shared libraries: ../../lib/libmkl.so:
cannot restore segment prot after reloc: Permission denied
```

This is a security enhanced Linux (SELinux) error message. The Intel Math Kernel Library lib/libmkl.so shipped with KNITRO does not have the proper security identifiers for your distribution of SELinux (the library is loaded with user option "blasoption"). You could disable security enhancements, but a better fix is to change the security identifiers of the library to acceptable values. On Linux Fedora Core 4, an acceptable security type is "texrel_shlib_t"; other Linux distributions are probably similar. The fix is made by changing to the KNITRO lib directory and typing:

```
    chcon -c -v -t texrel_shlib_t libmkl.so
```

3 Using Knitro with the AMPL modeling language

AMPL is a popular modeling language for optimization which 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 see [7] or visit the AMPL web site at:

http://www.ampl.com/

It is straightforward to use Knitro with the AMPL modeling language. 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 (such as a `bin` directory).

Inside of AMPL, to invoke the Knitro solver type:

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

at the prompt. To specify user options, type, for example,

```
ampl:  option knitro_options "maxit=100 alg=2";
```

The above command sets the maximum number of allowable iterations to 100 and chooses the Interior/CG algorithm (described in section 8). 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. See Tables 1-2 at the end of this section for a summary of user specifiable options available in Knitro for use with AMPL. For more detail on these options see section 5. Note that in section 5, user parameters are defined by text names such as "`alg`" and by programming language identifiers such as "`KTR_PARAM_ALG`". In AMPL, parameters are set using only the (lowercase) text names, as specified in Tables 1-2.

3.1 Example Optimization Problem

This section provides an example AMPL model and AMPL session which calls Knitro to solve the problem:

$$\begin{aligned}
\underset{x}{\text{minimize}} \quad & 1000 - x_1^2 - 2x_2^2 - x_3^2 - x_1x_2 - x_1x_3 && (3.2a)\\
\text{subject to} \quad & 8x_1 + 14x_2 + 7x_3 - 56 = 0 && (3.2b)\\
& x_1^2 + x_2^2 + x_3^2 - 25 \geq 0 && (3.2c)\\
& x_1, x_2, x_3 \geq 0 && (3.2d)
\end{aligned}$$

with initial point $x = [x_1, x_2, x_3] = [2, 2, 2]$.

The AMPL model for the above problem is provided with Knitro in a file called testproblem.mod, which is shown below.

AMPL test program file testproblem.mod

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

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

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

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

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

data;

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

The above example displays the ease with which an optimization problem can be expressed in the AMPL modeling language. Below is the AMPL session used to solve this problem with KNITRO. In the example below we set `alg=2` (to use the Interior/CG algorithm), `maxcrossit=2` (to refine the solution using the Active Set algorithm), and `outlev=1` (to limit output from KNITRO). See section 7 for an explanation of the KNITRO output.

AMPL Example

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

KNITRO 5.1: alg=2
maxcrossit=2
outlev=1
```

```
======================================
       Commercial Ziena License
             KNITRO 5.1
       Ziena Optimization, Inc.
       website:  www.ziena.com
       email:   info@ziena.com
======================================

algorithm:    2
maxcrossit:   2
outlev:       1
KNITRO changing bar_murule from AUTO to 1.
KNITRO changing bar_initpt from AUTO to 2.

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

EXIT: LOCALLY OPTIMAL SOLUTION FOUND.

Final Statistics
----------------
Final objective value               =   9.36000000000000e+02
Final feasibility error (abs / rel) =   0.00e+00 / 0.00e+00
Final optimality error  (abs / rel) =   3.55e-15 / 2.22e-16
# of iterations (major / minor)     =          7 /        7
# of function evaluations           =          8
# of gradient evaluations           =          8
# of Hessian evaluations            =          7
Total program time (secs)           =       0.00321 (     0.001 CPU time)
Time spent in evaluations (secs)    =       0.00015
```

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

KNITRO 5.1: LOCALLY OPTIMAL SOLUTION FOUND.
objective 9.360000e+02; feasibility error 0.000000e+00
7 major iterations; 8 function evaluations
ampl:
```

For descriptions of the KNITRO output see section 7. To display the final solution variables `x` and the objective value `obj` through AMPL, use the AMPL `display` command as follows.

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

ampl: display obj;
obj = 936
```

Upon completion, KNITRO displays a message and returns an exit code to AMPL. In the example above KNITRO found a solution, so the message was "`LOCALLY OPTIMAL SOLUTION FOUND`" with exit code of zero (exit code can be seen by typing "`ampl:  display solve_exitcode;`"). If a solution is not found, then KNITRO returns one of the following:

0:	`LOCALLY OPTIMAL SOLUTION FOUND.`
100:	`Current solution estimate cannot be improved.`
101:	`Current solution estimate cannot be improved.  Nearly optimal.`
200:	`Convergence to an infeasible point.  Problem may be locally infeasible.`
300:	`Problem appears to be unbounded.`
400:	`Iteration limit reached.`
401:	`Time limit reached.`
500:	`Invalid input.`
501:	`LP solver error.`
502:	`Evaluation error.`
503:	`Not enough memory.`
504:	`Terminated by user.`
505:	`Unknown termination.`

3.2 Solving with Complementarity Constraints

KNITRO is able to solve mathematical programs with complementarity constraints (MPCCs) through the AMPL interface. 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:

$$x \times y = 0, \quad x \geq 0, \quad y \geq 0. \tag{3.3}$$

The condition above is sometimes expressed more compactly as

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

These constraints must be formulated in a particular way through AMPL in order for KNITRO to effectively deal with them. In particular, complementarity constraints should be modeled using the AMPL `complements` command; e.g.,

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

and they *must* be formulated as one variable complementary to another variable. They *may not* be formulated as a function complementary to a variable or a function complementary to a function. KNITRO will print a warning if functions are used in complementarity constraints, but it is not able to fix the problem. If a complementarity involves a function $F(x)$, for example,

$$0 \leq F(x) \quad \perp \quad x \geq 0,$$

then the user should reformulate the AMPL model by adding a slack variable, as shown below, so that it is formulated as a variable complementary to another variable:

```
var x; var s;
...
constraint_name_a: F(x) = s;
constraint_name_b: 0 <= s complements x >= 0;
```

Be aware that the AMPL presolver sometimes removes complementarity constraints by mistake. Check carefully that the problem definition reported by KNITRO includes all complementarity constraints, or switch off the AMPL presolver to be safe ("`option presolve 0;`").

3.3 Displaying AMPL Variables in KNITRO

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 your 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 5.1: presolve_dbg=2
outlev=0
----- AMPL problem for KNITRO -----
```

```
Objective name:  obj
   0.000000e+00  <=  x[   0]  <=     1.000000e+20  x[1]
   0.000000e+00  <=  x[   1]  <=     1.000000e+20  x[2]
   0.000000e+00  <=  x[   2]  <=     1.000000e+20  x[3]

   2.500000e+01  <=  c[   0]  <=     1.000000e+20  c2
   5.600000e+01  <=  c[   1]  <=     5.600000e+01  c1
-----------------------------------
KNITRO 5.1: LOCALLY OPTIMAL SOLUTION FOUND.
objective 9.360000e+02; feasibility error 7.105427e-15
6 major iterations; 7 function evaluations
```

Table 1: KNITRO user specifiable options for AMPL.

OPTION	DESCRIPTION	DEFAULT
alg algorithm	optimization algorithm used: 0: let KNITRO choose the algorithm 1: Interior/Direct algorithm 2: Interior/CG algorithm 3: Active algorithm	0
bar_initmu	initial value for barrier parameter	1.0e-1
bar_initpt	initial point strategy for barrier algorithms 0: let KNITRO choose the initial point strategy 1: shift the initial point to improve barrier performance 2: do not alter the initial point supplied by the user	0
bar_maxbacktrack	maximum number of linesearch backtracks	3
bar_maxrefactor	maximum number of KKT refactorizations allowed	0
bar_murule	barrier parameter update rule: 0: let KNITRO choose the barrier update rule 1: monotone decrease rule 2: adaptive rule based on centrality measure 3: probing rule 4: safeguarded Mehrotra predictor-corrector type rule 5: Mehrotra predictor-corrector type rule 6: rule based on minimizing a quality function	0
blasoption	specify the BLAS/LAPACK function library to use: 0: use KNITRO built-in functions 1: use Intel Math Kernel Library functions 2: use the dynamic library specified with ''blasoptionlib''	0
debug	enable debugging output: 0: no extra debugging 1: help debug solution of the problem 2: help debug execution of the solver	0
delta	initial trust region radius scaling	1.0e0
feasible	0: allow for infeasible iterates 1: feasible version of KNITRO	0
feasmodetol	tolerance for entering feasible mode	1.0e-4
feastol	feasibility termination tolerance (relative)	1.0e-6
feastol_abs	feasibility termination tolerance (absolute)	0.0e-0
gradopt	gradient computation method: 1: use exact gradients 2: use forward finite-difference approximation 3: use centered finite-difference approximation	1
hessopt	Hessian (Hessian-vector) computation method: 1: use exact Hessian 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	1
honorbnds	0: allow bounds to be violated during the optimization 1: enforce bounds satisfaction of all iterates 2: enforce bounds satisfaction of initial point	2

Table 2: KNITRO user specifiable options for AMPL (continued).

OPTION	DESCRIPTION	DEFAULT
lmsize	number of limited-memory pairs stored in LBFGS approach	10
lpsolver	1: use internal LP solver in active-set algorithm 2: use ILOG-CPLEX LP solver in active-set algorithm (requires valid CPLEX license)	1
maxcgit	maximum allowable conjugate gradient (CG) iterations: 0: automatically set based on the problem size n: maximum of n CG iterations per minor iteration	0
maxcrossit	maximum number of allowable crossover iterations	0
maxit	maximum number of iterations before terminating	10000
maxtime_cpu	maximum CPU time in seconds before terminating	1.0e8
maxtime_real	maximum real time in seconds before terminating	1.0e8
ms_enable	0: multi-start not enabled 1: multi-start enabled	0
ms_maxbndrange	maximum range to vary x when generating start points	1.0e3
ms_maxsolves	maximum number of KNITRO solves during multi-start 0: automatically set the number based on problem size n: make exactly n solves	0
ms_maxtime_cpu	maximum CPU time for multi-start, in seconds	1.0e8
ms_maxtime_real	maximum real time for multi-start, in seconds	1.0e8
newpoint	0: no action 1: save the latest new point to file 'knitro_newpoint.log' 2: append all new points to file 'knitro_newpoint.log'	0
objrange	allowable objective function range	1.0e20
opttol	optimality termination tolerance (relative)	1.0e-6
opttol_abs	optimality termination tolerance (absolute)	0.0e-0
outlev	printing output level: 0: no printing 1: just print summary information 2: print information every 10 major iterations 3: print information at each major iteration 4: print information at each major and minor iteration 5: also print final (primal) variables 6: also print final Lagrange multipliers (sensitivies)	2
outmode	0: direct KNITRO output to standard out (e.g., screen) 1: direct KNITRO output to the file 'knitro.log' 2: print to both the screen and file 'knitro.log'	0
pivot	initial pivot threshold for matrix factorizations	1.0e-8
presolve_dbg	0: no debugging information 2: print the KNITRO problem with AMPL model names	0
scale	0: do not scale the problem 1: perform automatic scaling of functions	1
soc	0: do not allow second order correction steps 1: selectively try second order correction steps 2: always try second order correction steps	1
xtol	stepsize termination tolerance	1.0e-15

4 The Knitro callable library

This section includes information on how to embed and call the Knitro solver from inside a program. Knitro is written in C and C++, with a well-documented application programming interface (API) defined in the file knitro.h. The Knitro product contains example interfaces written in various programming languages under the directory /examples. These are briefly discussed in the following sections (C in 4.2, C++ in 4.3, Java in 4.4, and Fortran in 4.5). 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.

In all languages Knitro runs as a thread-safe module, which means that the calling program can create multiple instances of a Knitro solver in different threads, each instance solving a different problem. This is useful in a multiprocessing environment; for instance, in a web application server.

4.1 Knitro in a C application

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

- `KTR_new()`: create a new Knitro solver context pointer, allocating resources
- `KTR_init_problem()`: load the problem definition into the Knitro solver
- `KTR_solve()`: solve the problem
- `KTR_free()`: delete the Knitro context pointer, releasing allocated resources

The complete C language API is defined in the file knitro.h, provided in the installation under the /include directory. Functions for setting and getting user options are described in sections 5.2 and 5.3. Functions for retrieving Knitro results are described in section 7.2 and illustrated in the examples/C files. The remainder of this section describes in detail the four basic function calls.

```
KTR_context_ptr KTR_new (void)
```

This function must be called first. It returns a pointer to an object (the Knitro "context pointer") that is used in all other calls. If you enable Knitro with the Ziena floating network license handler, then this call also checks out a license and reserves it until `KTR_free()` is called with the context pointer, or the program ends. The contents of the context pointer should never be modified by a calling program.

```
int 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.

The C interface for KNITRO requires the application to define an optimization problem (1.1) in the following general format (for complementarity constraints, see section 10.5):

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) && (4.4a) \\ \text{subject to} \quad & \texttt{cLoBnds} \leq c(x) \leq \texttt{cUpBnds} && (4.4b) \\ & \texttt{xLoBnds} \leq x \leq \texttt{xUpBnds} && (4.4c) \end{aligned}$$

where `cLoBnds` and `cUpBnds` are vectors of length m, and `xLoBnds` and `xUpBnds` are vectors of length n. If constraint i is an equality constraint, set $\texttt{cLoBnds}[i] = \texttt{cUpBnds}[i]$. If constraint i is unbounded from below or above, set $\texttt{cLoBnds}[i]$ or $\texttt{cUpBnds}[i]$ to the value `-KTR_INFBOUND` or `KTR_INFBOUND`, respectively. This constant is defined in `knitro.h` and stands for infinity in the KNITRO code.

To use KNITRO the application must provide routines for evaluating the objective $f(x)$ and constraint functions $c(x)$. For best performance, the application should also provide routines to evaluate first derivatives (gradients of $f(x)$ and $c(x)$), and, ideally, the second derivatives (Hessian of the Lagrangian). First derivatives in the C language API are denoted by `objGrad` and `jac`, where $\texttt{objGrad} = \nabla f(x)$, and `jac` is the $m \times n$ Jacobian matrix of constraint gradients such that the i-th row equals $\nabla c_i(x)$.

The ability to provide exact first derivatives is essential for efficient and reliable performance. Packages like ADOL-C and ADIFOR can help in generating code with derivatives. If the user is unable or unwilling to provide exact first derivatives, KNITRO provides an option that computes approximate first derivatives using finite-differencing (see sections 4.8 and 9.1).

Exact second derivatives are less important, as KNITRO provides several options that substitute quasi-Newton approximations for the Hessian (see section 9.2). However, the ability to provide exact second derivatives often dramatically improves the performance of KNITRO.

Function KTR_init_problem:

```
int KTR_init_problem(  KTR_context_ptr kc,
                       int n,
                       int objGoal,
                       int objType,
                       double * xLoBnds,
                       double * xUpBnds,
                       int m,
                       int * cType,
                       double * cLoBnds,
                       double * cUpBnds,
                       int nnzJ,
                       int * jacIndexVars,
                       int * jacIndexCons,
                       int nnzH,
                       int * hessIndexRows,
                       int * hessIndexCols,
                       double * xInitial,
                       double * lambdaInitial )
```

This function passes 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()` 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

Arguments:

`KTR_context_ptr kc:` is the KNITRO context pointer. Do not modify its contents.

`int n:` is a scalar specifying the number of variables in the problem; i.e., the length of x in (4.4).

`int objGoal:` is the optimization goal.

- `0:` if the goal is to minimize the objective function (`KTR_OBJGOAL_MINIMIZE`)
- `1:` if the goal is to maximize the objective function (`KTR_OBJGOAL_MAXIMIZE`)

`int objType:` is a scalar that describes the type of objective function $f(x)$ in (4.4).

- `0:` if $f(x)$ is a nonlinear function or its type is unknown (`KTR_OBJTYPE_GENERAL`)
- `1:` if $f(x)$ is a linear function (`KTR_OBJTYPE_LINEAR`)
- `2:` if $f(x)$ is a quadratic function (`KTR_OBJTYPE_QUADRATIC`)

`double * 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` (defined in knitro.h).

`double * 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 x_i. If the variable has no upper bound, set `xUpBnds`$[i]$ to be `KTR_INFBOUND` (defined in knitro.h).

`int m:` is a scalar specifying the number of constraints $c(x)$ in (4.4).

`int * cType:` is an array of length `m` that describes the types of the constraint functions $c(x)$ in (4.4).

- `0:` if $c_i(x)$ is a nonlinear function or its type is unknown (`KTR_CONTYPE_GENERAL`)
- `1:` if $c_i(x)$ is a linear function (`KTR_CONTYPE_LINEAR`)
- `2:` if $c_i(x)$ is a quadratic function (`KTR_CONTYPE_QUADRATIC`)

`double * cLoBnds:` is an array of length `m` specifying the lower bounds on the constraints $c(x)$ in (4.4). `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` (defined in knitro.h). If the constraint is an equality, then `cLoBnds`$[i]$ should equal `cUpBnds`$[i]$.

`double * cUpBnds:` is an array of length `m` specifying the upper bounds on the constraints $c(x)$ in (4.4). `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` (defined in knitro.h). If the constraint is an equality, then `cLoBnds`$[i]$ should equal `cUpBnds`$[i]$.

`int nnzJ:` is a scalar specifying the number of nonzero elements in the sparse constraint Jacobian. See section 4.7.

`int * jacIndexVars:` is an array of length `nnzJ` specifying the variable indices of the constraint Jacobian nonzeroes. 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 `KTR_solve()`.

`jacIndexCons`[i] and `jacIndexVars`[i] determine the row numbers and the column numbers, respectively, of the nonzero constraint Jacobian element `jac`[i]. See section 4.7.

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`.

`int * jacIndexCons:` is an array of length `nnzJ` specifying the constraint indices of the constraint Jacobian nonzeroes. 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 `KTR_solve()`.

`jacIndexCons`[i] and `jacIndexVars`[i] determine the row numbers and the column numbers, respectively, of the nonzero constraint Jacobian element `jac`[i]. See section 4.7.

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`.

`int nnzH:` is a scalar specifying the number of nonzero elements in the sparse Hessian of the Lagrangian. Only nonzeroes in the upper triangle (including diagonal nonzeroes) should be counted. See section 4.7.

NOTE: If user option "hessopt" is not set to `KTR_HESSOPT_EXACT`, then Hessian nonzeroes will not be used (see section 5.1). In this case, set `nnzH=0`, and pass NULL pointers for `hessIndexRows` and `hessIndexCols`.

`int * hessIndexRows:` is an array of length `nnzH` specifying the row number indices of the Hessian nonzeroes.

`hessIndexRows`[i] and `hessIndexCols`[i] determine the row numbers and the column numbers, respectively, of the nonzero Hessian element `hess`[i], where `hess` is the array of Hessian elements passed in the call `KTR_solve()`. See section 4.7.

NOTE: Row numbers are in the range 0 .. `n` $-$ 1.

`int * hessIndexCols:` is an array of length `nnzH` specifying the column number indices of the Hessian nonzeroes.

`hessIndexRows`[i] and `hessIndexCols`[i] determine the row numbers and the column numbers, respectively, of the nonzero Hessian element `hess`[i], where `hess` is the array of Hessian elements passed in the call `KTR_solve()`. See section 4.7.

NOTE: Column numbers are in the range 0 .. `n` $-$ 1.

`double * 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`.

`double * lambdaInitial:` is an array of length `m+n` containing an initial guess of the Lagrange multipliers for the constraints $c(x)$ (4.4*b*) and bounds on the variables x (4.4*c*). 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`.

To solve the nonlinear optimization problem (4.4), 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. At a trial point, KNITRO may ask the application to:

`KTR_RC_EVALFC`:	Evaluate f and c at x.
`KTR_RC_EVALGA`:	Evaluate ∇f and ∇c at x.
`KTR_RC_EVALH`:	Evaluate the Hessian matrix of the problem at x and λ.
`KTR_RC_EVALHV`:	Evaluate the Hessian matrix times a vector v at x and λ.

The constants `KTR_RC_*` are return codes defined in `knitro.h`.

The KNITRO C language API has two modes of operation for obtaining problem information: "**callback**" and "**reverse communication**". With callback mode the application provides C language function pointers that KNITRO may call to evaluate the functions, gradients, and Hessians. With reverse communication, the function `KTR_solve()` returns one of the constants listed above to tell the application what it needs, and then waits to be called again with the new problem information. For more details, see section 9.8 (callback mode) and section 9.7 (reverse communication mode). Both modes use `KTR_solve()`.

Function KTR_solve:

```
int KTR_solve(  KTR_context_ptr kc,     /*input*/
                double * x,             /*output*/
                double * lambda,        /*output*/
                int evalStatus,         /*input, reverse comm only*/
                double * obj,           /*input and output*/
                double * c,             /*input, reverse comm only*/
                double * objGrad,       /*input, reverse comm only*/
                double * jac,           /*input, reverse comm only*/
                double * hess,          /*input, reverse comm only*/
                double * hessVector,    /*input, output, rev comm*/
                void * userParams )     /*input, callback only*/
```

Arguments:

`KTR_context_ptr kc:` is the KNITRO context pointer. Do not modify its contents.

`double * x:` is an array of length `n` output by KNITRO. If `KTR_solve()` returns `KTR_RC_OPTIMAL` (zero), then `x` contains the solution.

Reverse communications mode: upon return, `x` contains the value of unknowns at which KNITRO needs more problem information. If user option "newpoint" is set to `KTR_NEWPOINT_USER` (see section 5.1) and `KTR_solve()` returns `KTR_RC_NEWPOINT`, then `x` contains a newly accepted iterate, but not the final solution.

`double * lambda:` is an array of length `m+n` output by KNITRO. If `KTR_solve()` returns zero, then `lambda` contains the multiplier values at the solution. The first `m` components of `lambda` are multipliers corresponding to the constraints specified in $c(x)$, while the last `n` components are multipliers corresponding to the bounds on x.

Reverse communications mode: upon return, `lambda` contains the value of multipliers at which KNITRO needs more problem information.

`int evalStatus:` is a scalar input to KNITRO used only in **reverse communications mode**. A value of zero means the application successfully computed the problem information requested by KNITRO at `x` and `lambda`. A nonzero value means the application failed to compute problem information (e.g., if a function is undefined at the requested value `x`).

`double * obj:` is a scalar holding the value of $f(x)$ at the current `x`. If `KTR_solve()` returns `KTR_RC_OPTIMAL` (zero), then `obj` contains the value of the objective function $f(x)$ at the solution.

Reverse communications mode: if `KTR_solve()` returns `KTR_RC_EVALFC`, then `obj` must be filled with the value of $f(x)$ computed at `x` before `KTR_solve()` is called again.

`double * c:` is an array of length `m` used only in **reverse communications mode**. If `KTR_solve()` returns `KTR_RC_EVALFC`, then `c` must be filled with the value of $c(x)$ computed at `x` before `KTR_solve()` is called again.

`double * objGrad:` is an array of length `n` used only in **reverse communications mode**. If `KTR_solve()` returns `KTR_RC_EVALGA`, then `objGrad` must be filled with the value of $\nabla f(x)$ computed at `x` before `KTR_solve()` is called again.

`double * jac:` is an array of length `nnzJ` used only in **reverse communications mode**. If `KTR_solve()` returns `KTR_RC_EVALGA`, then `jac` must be filled with the constraint Jacobian $\nabla c(x)$ computed at `x` before `KTR_solve()` is called again. Entries are stored according to the sparsity pattern defined in `KTR_init_problem()`.

`double * hess:` is an array of length `nnzH` used only in **reverse communications mode**, and only if option "hessopt" is set to `KTR_HESSOPT_EXACT` (see section 5.1). If `KTR_solve()` returns `KTR_RC_EVALH`, then `hess` must be filled with the Hessian of the Lagrangian computed at `x` and `lambda` before `KTR_solve()` is called again. Entries are stored according to the sparsity pattern defined in `KTR_init_problem()`.

`double * hessVector:` is an array of length `n` used only in **reverse communications mode**, and only if option "hessopt" is set to `KTR_HESSOPT_PRODUCT` (see section 5.1). If `KTR_solve()` returns `KTR_RC_EVALHV`, then the Hessian of the Lagrangian at `x` and `lambda` should be multiplied by `hessVector`, and the result placed in `hessVector` before `KTR_solve()` is called again.

`void * userParams:` is a pointer to a structure used only in **callback mode**. The pointer is provided so the application can pass additional parameters needed for its callback routines. If the application needs no additional parameters, then pass a NULL pointer. See section 9.8 for more details.

Return Value:

The return value of `KTR_solve()` specifies the final exit code from the optimization process. If the return value is zero (`KTR_RC_OPTIMAL`) or negative, then KNITRO has finished solving. In **reverse communications mode** the return value may be positive, in which case it specifies a request for additional problem information, after which the application should call KNITRO again. A detailed description of the possible return values is given in the appendix.

Function KTR_restart:

```
int KTR_restart(  KTR_context_ptr kc,
                  double * x,
                  double * lambda )
```

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 can be modified, and a new initial value can be passed with `KTR_restart()`. 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.

4.2 Examples of calling in C

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, but the working examples provide more complete detail.

Consider the following nonlinear optimization problem from the Hock and Schittkowski test set [9]:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 100 - (x_2 - x_1^2)^2 + (1 - x_1)^2 && (4.5a) \\ \text{subject to} \quad & 1 \le x_1 x_2 && (4.5b) \\ & 0 \le x_1 + x_2^2 && (4.5c) \\ & x_1 \le 0.5. && (4.5d) \end{aligned}$$

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

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

```
#include "knitro.h"

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

KTR_context  *kc;

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

/*---- CREATE A NEW KNITRO SOLVER INSTANCE. */
kc = KTR_new();
```

```
if (kc == NULL)
    {
    printf ("Failed to find a Ziena license.\n");
    return( -1 );
    }
```

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

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

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

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

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

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

/*---- PROVIDE SECOND DERIVATIVE STRUCTURAL INFORMATION. */
hessIndexRows[0] = 0;
```

```
    hessIndexRows[1] = 0;
    hessIndexRows[2] = 1;
    hessIndexCols[0] = 0;
    hessIndexCols[1] = 1;
    hessIndexCols[2] = 1;

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

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

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

Assume for simplicity that the user writes three routines for computing problem information. In examples/C/problemHS15.c these are named computeFC, computeGA, and computeH. To write a driver program using **callback** mode, simply wrap each evaluation routine in a function that matches the KTR_callback() prototype defined in knitro.h. Note that all three wrappers use the same prototype. This is in case the application finds it convenient to combine some of the evaluation steps, as demonstrated in examples/C/callbackExample2.c.

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

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


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

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



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

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

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

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

```
        exit( -1 );
    if ((nHessOpt == KTR_HESSOPT_EXACT) ||
        (nHessOpt == KTR_HESSOPT_PRODUCT))
        {
        if (KTR_set_hess_callback (kc, &callbackEvalHess) != 0)
            exit( -1 );
        }

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

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

To write a driver program using **reverse communications** mode, set up a loop that calls `KTR_solve()` and then computes the requested problem information. The loop continues until `KTR_solve()` returns zero (success), or a negative error code:

```
    /*---- SOLVE THE PROBLEM.  IN REVERSE COMMUNICATIONS MODE, KNITRO
     *---- RETURNS WHENEVER IT NEEDS MORE PROBLEM INFO.  THE CALLING
     *---- PROGRAM MUST INTERPRET KNITRO'S RETURN STATUS AND CONTINUE
     *---- SUPPLYING PROBLEM INFORMATION UNTIL KNITRO IS COMPLETE.
     */
    while (1)
        {
        nStatus = KTR_solve (kc, x, lambda, evalStatus, &obj, c,
                             objGrad, jac, hess, hvector, NULL);

        if      (nStatus == KTR_RC_EVALFC)
            /*---- KNITRO WANTS obj AND c EVALUATED AT THE POINT x. */
            obj = computeFC (x, c);
        else if (nStatus == KTR_RC_EVALGA)
            /*---- KNITRO WANTS objGrad AND jac EVALUATED AT x. */
            computeGA (x, objGrad, jac);
        else if (nStatus == KTR_RC_EVALH)
            /*---- KNITRO WANTS hess EVALUATED AT (x, lambda). */
            computeH (x, lambda, hess);
        else
            /*---- IN THIS EXAMPLE, OTHER STATUS CODES MEAN KNITRO IS
                   FINISHED. */
            break;
        /*---- ASSUME THAT PROBLEM EVALUATION IS ALWAYS SUCCESSFUL.
         *---- IF A FUNCTION OR ITS DERIVATIVE COULD NOT BE EVALUATED
         *---- AT THE GIVEN (x, lambda), THEN SET evalStatus = 1 BEFORE
         *---- CALLING KTR_solve AGAIN. */
        evalStatus = 0;
        }

    if (nStatus != 0)
        printf ("KNITRO failed to solve the problem, final status = %d\n",
                nStatus);

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

This completes the brief overview of creating driver programs to run Knitro in C. Again, more details and options are demonstrated in the programs located in examples/C. Outputs produced when running Knitro are discussed in section 7.

4.3 Knitro in a C++ application

Calling Knitro from a C++ application follows the same outline as a C application. The distribution provides a C++ general test harness in the directory examples/C++. In the example, optimization problems are coded as subclasses of an abstract interface and compiled as separate shared objects. A main driver program dynamically loads a problem and sets up **callback mode** (9.8) 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.

4.4 Knitro in a Java application

Calling Knitro from a Java application follows the same outline as a C application. The optimization problem must be defined in terms of arrays and constants that follow the Knitro API, and then the Java version of `KTR_init_problem()` is called. Java `int` and `double` types map directly to their C counterparts. Having defined the optimization problem, the Java version of `KTR_solve()` is called in **reverse communications mode** (9.7).

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

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

The sample program closely mirrors the structural form of the C reverse communications example described in section 4.2. Refer to that section for more information. See section 4.7 for details on specifying the arrays of partial derivatives that KNITRO needs.

4.5 KNITRO in a Fortran application

Calling KNITRO from a Fortran application follows the same outline as a C application. The optimization problem must be defined in terms of arrays and constants that follow the KNITRO API, and then the Fortran version of `KTR_init_problem()` is called. Fortran `integer` and `double precision` types map directly to C `int` and `double` types. Having defined the optimization problem, the Fortran version of `KTR_solve()` is called in **reverse communications mode** (9.7).

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 code will not be reproduced here, as it closely mirrors the structural form of the C reverse communications example described in section 4.2. 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.

4.6 Compiler Specifications

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

Windows (32-bit x86)

C/C++:	Microsoft Visual Studio C++ 7.1 (.NET 2003 Framework 1.1)
	Microsoft Visual Studio C++ 6.0
Java:	1.4.1_02 from Sun
Fortran:	Intel Visual Fortran 9.0

Windows (64-bit x86_64)

C/C++:	Microsoft Visual Studio C++ 8.0 (.NET 2005 Framework 2.0)
Java:	1.5.0_10 from Sun
Fortran:	Intel Visual Fortran 9.1

Linux (32-bit x86, 64-bit x86_64)	
C/C++:	gcc/g++ (compiler version to match the Linux distribution)
Java:	1.5.0_06 from Sun
Fortran:	g77/g95
Mac OS X (32-bit x86, 32-bit PowerPC)	
C/C++:	gcc/g++ 4.0.1
Java:	1.5.0_06
Solaris (SunOS 5.8 on SPARC)	
C/C++:	Sun Workshop 6 C 5.1
Java:	1.2.2_10 from Sun
Fortran:	Sun Workshop 6 FORTRAN 95 6.0

4.7 Specifying the Jacobian and Hessian matrices

An important issue in using the KNITRO callable library is the ability of the application to specify the Jacobian matrix of the constraints and the Hessian matrix of the Lagrangian function (when using exact Hessians) in sparse form. Below we give an example of how to do this.

Example

Assume we want to use KNITRO to solve the following problem

$$\begin{aligned}
\underset{x}{\text{minimize}} \quad & x_0 + x_1 x_2^3 && (4.6a)\\
\text{subject to} \quad & \cos(x_0) = 0.5 && (4.6b)\\
& 3 \le x_0^2 + x_1^2 \le 8 && (4.6c)\\
& x_0 + x_1 + x_2 \le 10 && (4.6d)\\
& x_0, x_1, x_2 \ge 1. && (4.6e)
\end{aligned}$$

Rewriting in the notation of (4.4), we have

$$\begin{aligned}
f(x) &= x_0 + x_1 x_2^3 && (4.7)\\
c_0(x) &= \cos(x_0) && (4.8)\\
c_1(x) &= x_0^2 + x_1^2 && (4.9)\\
c_2(x) &= x_0 + x_1 + x_2. && (4.10)\\
& && (4.11)
\end{aligned}$$

Computing the Sparse Jacobian Matrix

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

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

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

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

In Knitro, the array `objGrad` stores *all* of the elements of $\nabla f(x)$, while the arrays `jac`, `jacIndexCons`, and `jacIndexVars` store information concerning *only the nonzero* elements of $J(x)$. The array `jac` stores the nonzero values in $J(x)$ evaluated at the current solution estimate x, `jacIndexCons` stores the constraint function (or row) indices corresponding to these values, and `jacIndexVars` stores the variable (or column) indices. There is no restriction on the order in which these elements are stored; however, it is common to store the nonzero elements of $J(x)$ in column-wise fashion. For the example above, the number of nonzero elements `nnzJ` in $J(x)$ is 6, and these arrays are specified as follows in column-wise order.

```
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;
```

The values of `jac` depend on the value of x and change during the solution process. The values of `jacIndexCons` and `jacIndexVars` are set in `KTR_init_problem()` and remain constant.

Computing the Sparse Hessian Matrix

The Hessian of the Lagrangian matrix is defined as

$$H(x, \lambda) = \nabla^2 f(x) + \sum_{i=0}^{m-1} \lambda_i \nabla^2 c_i(x), \tag{4.12}$$

where λ is the vector of Lagrange multipliers (dual variables). For the example defined by problem (4.6), The Hessians (second derivatives) of the objective and constraint functions are given by

$$\begin{aligned} \nabla^2 f(x) &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 3x_2^2 \\ 0 & 3x_2^2 & 6x_1x_2 \end{bmatrix}, \quad \nabla^2 c_0(x) = \begin{bmatrix} -\cos(x_0) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \\ \nabla^2 c_1(x) &= \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \qquad \nabla^2 c_2(x) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \end{aligned}$$

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

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

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

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

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

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

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

The values of `hess` depend on the value of x and change during the solution process. The values of `hessIndexRows` and `hessIndexCols` are set in `KTR_init_problem()` and remain constant.

4.8 Calling without first 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, as described in section 9.1. Even though the application does not evaluate derivatives, it must still provide a sparsity pattern for the constraint Jacobian matrix. Knitro uses the sparsity pattern to speed up linear algebra computations. If the sparsity pattern is unknown, then the application should specify a fully dense pattern (i.e., assume all elements are nonzero).

The code fragment below demonstrates how to define a problem with no derivatives and unknown sparsity pattern. The code is in the C language.

```
/*... define variables, call KTR_new(), etc. ...*/

/*---- DEFINE PROBLEM SIZES.  NOTHING IS KNOWN ABOUT THE DERIVATIVES,
 *---- SO ASSUME THE JACOBIAN IS DENSE AND DO NOT SUPPLY A HESSIAN. */
n = 20;
m = 10;
nnzJ = n * m;
nnzH = 0;

/*... define objType, xLoBnds, xUpBnds, cType, cLoBnds, cUpBnds, etc. ...*/
```

```
/*---- DEFINE DENSE FIRST DERIVATIVE SPARSITY PATTERN. */
k = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
    {
        jacIndexCons[k] = j;
        jacIndexVars[k] = i;
        k++;
    }

/*---- INSTRUCT KNITRO TO COMPUTE FIRST DERIVATIVE ESTIMATES
 *---- AND APPROXIMATE THE HESSIAN. */
if (KTR_set_int_param (kc, KTR_PARAM_GRADOPT, KTR_GRADOPT_CENTRAL) != 0)
    { /*... an error occurred ...*/ }
if (KTR_set_int_param (kc, KTR_PARAM_HESSOPT, KTR_HESSOPT_LBFGS) != 0)
    { /*... an error occurred ...*/ }

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

/*... call KTR_solve(), etc. ...*/
```

5 User options in Knitro

Knitro 5.1 offers a number of user options for modifying behavior of the solver. Each option takes a value that may be an integer, double precision number, or character string. Options are usually identified by a string name (for example, "`algorithm`"), but programmatic interfaces also identify options by an integer value associated with a C language macro defined in the file `knitro.h` (for example, "`KTR_PARAM_ALG`"). This section lists all user options in alphabetical order, identified by the string name and the macro definition. Sections 5.2 and 5.3 provide instructions on how to set and modify user options.

5.1 Description of Knitro user options

`algorithm (KTR_PARAM_ALG):` Indicates which algorithm to use to solve the problem (see section 8).

`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`

`bar_initmu (KTR_PARAM_BAR_INITMU):` Specifies the initial value for the barrier parameter μ used with the barrier algorithms. This option has no effect on the Active Set algorithm.

Default value: `1.0e-1`

`bar_initpt (KTR_PARAM_BAR_INITPT):` Indicates whether an initial point strategy is used with barrier algorithms. If the "`honorbnds`" or "`feasible`" options are enabled, then a request to shift may be ignored. This option has no effect on the Active Set algorithm.

`0 (auto):`	Let Knitro automatically choose the strategy.
`1 (yes):`	Shift the initial point to improve barrier algorithm performance.
`2 (no):`	Do no alter the initial point supplied by the user.

Default value: `0`

`bar_maxbacktrack (KTR_PARAM_BAR_MAXBACKTRACK):` Indicates the maximum allowable number of backtracks during the linesearch of the Interior/Direct algorithm before reverting to a CG step. Increasing this value will make the Interior/Direct algorithm less likely to take CG steps. If the Interior/Direct algorithm is taking a large number of CG steps (as indicated by a positive value for "*CG its*" in the output), this may improve performance. This option has no effect on the Active Set algorithm.

Default value: `3`

`bar_maxrefactor (KTR_PARAM_BAR_MAXREFACTOR):` Indicates the maximum number of refactorizations of the KKT system per iteration of the Interior/Direct algorithm before reverting to a CG step. 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 "*CG its*" in the output), this may improve performance. This option has no effect on the Active Set algorithm.

Default value: `0`

`bar_murule (KTR_PARAM_BAR_MURULE):` Indicates which strategy to use for modifying the barrier parameter μ in the barrier algorithms (see section 8). 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`

`blasoption (KTR_PARAM_BLASOPTION):` 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.
`2 (dynamic):`	Use the dynamic library specified with option "`blasoptionlib`".

Default value: `0`

NOTE: BLAS and LAPACK functions from the Intel Math Kernel Library (MKL) 8.1 are provided with the KNITRO distribution. The MKL is available for Windows (32-bit and 64-bit), Linux (32-bit and 64-bit), and Mac OS X (32-bit x86); it is *not* available for Solaris or Mac OS X PowerPC. The MKL is not included with the free student edition of KNITRO.

The MKL is provided in the KNITRO `lib` directory and is loaded at runtime by KNITRO. The operating system's load path must be configured to find this directory or the MKL will fail to load. Section 5.4 explains how to do this.

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_summ*.txt`. Some optimization problems are observed to spend less than 1% of 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 built-in functions are based on standard netlib routines (`www.netlib.org`). The Intel MKL functions are 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 reduce the CPU time in BLAS/LAPACK operations by 20-30%.

If your machine uses security enhanced Linux (SELinux), you may see errors when loading the Intel MKL. Refer to section 2.3 for more information.

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`".

`blasoptionlib (KTR_PARAM_BLASOPTIONLIB)`: Specifies a dynamic library name that contains object code for BLAS/LAPACK functions. The library must implement all the functions declared in the file `include/blas_lapack.h`. The source file `blasAcmlExample.c` in `examples/C` provides a wrapper for the AMD Core Math Library (ACML), suitable for machines with an AMD processor. Instructions are given in the file for creating a BLAS/LAPACK dynamic library from the ACML.

NOTE: This option has no effect unless "`blasoption=2`".

`debug (KTR_PARAM_DEBUG)`: 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`" equals zero.

`0 (none)`:	No debugging output.
`1 (problem)`:	Print algorithm information to `kdbg*.log` output files.
`2 (execution)`:	Print program execution information.

Default value: `0`

`delta (KTR_PARAM_DELTA)`: Specifies the initial trust region radius scaling factor used to determine the initial trust region size.

Default value: `1.0e0`

`feasible (KTR_PARAM_FEASIBLE)`: Specifies whether to use the feasible version of KNITRO, which will force iterates to satisfy inequality constraints (see section 9.3). This option and the "`honorbnds`" option may be useful in applications where functions are undefined outside the region defined by inequalities.

`0 (no)`:	Iterates may be infeasible.
`1 (yes)`:	Iterates must satisfy inequality constraints once they become sufficiently feasible.

Default value: `0`

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

The feasible version of KNITRO will force iterates to strictly satisfy inequalities, but does not require satisfaction of equality constraints at intermediate iterates. 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

$$cl + tol \leq c(x) \leq cu - tol \tag{5.13}$$

The constant *tol* is determined by the option "`feasmodetol`". See section 9.3 for details.

`feasmodetol (KTR_PARAM_FEASMODETOL)`: Specifies the tolerance in equation (5.13) that determines whether KNITRO will force subsequent iterates to remain feasible. The tolerance applies to all inequality constraints in the problem. This option has no effect if option "`feasible=no`".

Default value: `1.0e-4`

`feastol (KTR_PARAM_FEASTOL)`: 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. See section 6 for more information.

Default value: `1.0e-6`

`feastol_abs (KTR_PARAM_FEASTOLABS)`: 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. See section 6 for more information.

Default value: `0.0e0`

`gradopt (KTR_PARAM_GRADOPT)`: Specifies how to compute the gradients of the objective and constraint functions. See section 9.1 for more information.

`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.

`hessopt (KTR_PARAM_HESSOPT)`: Specifies how to compute the (approximate) Hessian of the Lagrangian. See section 9.2 for more information.

`1 (exact)`:	User provides a routine for computing the exact Hessian.
`2 (bfgs)`:	KNITRO computes a (dense) quasi-Newton BFGS Hessian.
`3 (sr1)`:	KNITRO computes a (dense) quasi-Newton SR1 Hessian.
`4 (finite_diff)`:	KNITRO computes Hessian-vector products using finite-differences.
`5 (product)`:	User provides a routine to compute the Hessian-vector products.

`6 (lbfgs):` KNITRO computes a limited-memory quasi-Newton BFGS Hessian (its size is determined by the option "`lmsize`").

Default value: `1`

NOTE: Options "`hessopt=4`" and "`hessopt=5`" are not available with the Interior/Direct algorithm.

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`" is recommended. 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. 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 ($n < 1000$) since they require working with a dense Hessian approximation. Option "`hessopt=6`" should be used for large problems. See section 9.2 for more information.

`honorbnds (KTR_PARAM_HONORBNDS):` Indicates whether or not to enforce satisfaction of simple variable bounds throughout the optimization (see section 9.4). This option and the "`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`

`lmsize (KTR_PARAM_LMSIZE):` 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 when "`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. See section 9.2 for more details.

Default value: `10`

`lpsolver (KTR_PARAM_LPSOLVER):` Indicates which linear programming simplex solver the KNITRO Active Set algorithm uses when solving internal LP subproblems. This option has no effect on the Interior/Direct and Interior/CG algorithms.

`1 (internal):` KNITRO uses its default LP solver.

`2 (cplex):` KNITRO uses ILOG-CPLEX, provided the user has a valid CPLEX license. The CPLEX library is loaded dynamically after `KTR_solve()` is called.

Default value: `1`

If "`lpsolver=cplex`" then the CPLEX shared object library or DLL must reside in the operating system's load path (see section 5.4). If this option is selected, KNITRO will automatically look for (in order): 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 (see section 5.4). For example, to specifically load the Windows CPLEX library `cplex90.dll`, make sure the directory containing the library is part of the `PATH` environment variable, and call the following (also be sure to check the return status of this call):

```
KTR_set_char_param_by_name (kc, "cplexlibname", "cplex90.dll");
```

`maxcgit (KTR_PARAM_MAXCGIT):` 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`

`maxcrossit (KTR_PARAM_MAXCROSSIT):` 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 "`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. See section 9.5 for more information.

Default value: `0`

`maxit (KTR_PARAM_MAXIT):` Specifies the maximum number of major iterations before termination.

Default value: `10000`

`maxtime_cpu (KTR_PARAM_MAXTIMECPU):` Specifies, in seconds, the maximum allowable CPU time before termination.

Default value: `1.0e8`

`maxtime_real (KTR_PARAM_MAXTIMEREAL):` Specifies, in seconds, the maximum allowable real time before termination.

Default value: `1.0e8`

`ms_enable or multistart (KTR_PARAM_MULTISTART):` Indicates whether KNITRO will solve from multiple start points to find a better local minimum. See section 9.6 for details.

`0 (no):`	KNITRO solves from a single initial point.
`1 (yes):`	KNITRO solves using multiple start points.

Default value: `0`

`ms_maxbndrange (KTR_PARAM_MSMAXBNDRANGE):` Specifies the maximum range that each variable can take when determining new start points. If a variable has upper and lower bounds and the difference between them is less than "`ms_maxbndrange`", 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_maxbndrange`", then new start point values are restricted by the option. If x_i is such a variable, then all initial values satisfy

$$x_i^0 - \texttt{ms\_maxbndrange}/2 \le x_i \le x_i^0 + \texttt{ms\_maxbndrange}/2,$$

where x_i^0 is the initial value of x_i provided by the user. This option has no effect unless "`ms_enable=yes`".

Default value: `1000.0`

`ms_maxsolves (KTR_PARAM_MSMAXSOLVES):` 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 $\min(200, 10N)$, where N is the number of variables in the problem.
n:	Try $n > 0$ start points.

Default value: `0`

`ms_maxtime_cpu (KTR_PARAM_MSMAXTIMECPU):` Specifies, in seconds, the maximum allowable CPU time before termination. The limit applies to the operation of KNITRO since multi-start began; in contrast, the value of "`maxtime_cpu`" limits how long KNITRO iterates from a single start point. Therefore, "`ms_maxtime_cpu`" should be greater than "`maxtime_cpu`". This option has no effect unless "`ms_enable=yes`".

Default value: `1.0e8`

`ms_maxtime_real (KTR_PARAM_MSMAXTIMEREAL):` 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`

`newpoint (KTR_PARAM_NEWPOINT):` Specifies additional action to take after every major iteration. Major iterations of KNITRO result in a new point that is closer to a solution. The new point includes values of `x` and Lagrange multipliers `lambda`.

`0 (none):`	KNITRO takes no additional action.
`1 (saveone):`	KNITRO writes `x` and `lambda` to the file knitro_newpoint.log. Previous contents of the file are overwritten.

`2 (saveall):` KNITRO appends `x` and `lambda` to the file knitro_newpoint.log. *Warning:* this option can generate a very large file. All major iterates, including the start point, crossover points, and the final solution are saved. Each iterate also prints the objective value at the new point, except the initial start point.

`3 (user):` If using callback mode (see section 9.8) and a user callback function is defined with `KTR_set_newpoint_callback()`, then KNITRO will invoke the callback function after every major iteration. If using reverse communications mode (see section 9.7), then KNITRO will return to the driver level after every major iteration with `KTR_solve()` returning the integer value defined by `KTR_RC_NEWPOINT` (6).

Default value: `0`

`objrange (KTR_PARAM_OBJRANGE):` Specifies the extreme limits of the objective function for purposes of determining unboundedness. If the magnitude of the objective function becomes greater than "`objrange`" for a feasible iterate, then the problem is determined to be unbounded and KNITRO proceeds no further.

Default value: `1.0e20`

`opttol (KTR_PARAM_OPTTOL):` 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. See section 6 for more information.

Default value: `1.0e-6`

`opttol_abs (KTR_PARAM_OPTTOLABS):` 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. See section 6 for more information.

Default value: `0.0e0`

`outlev (KTR_PARAM_OUTLEV):` Controls the level of output produced by KNITRO.

`0 (none):` Printing of all output is suppressed.

`1 (summary):` Print only summary information.

`2 (majorit10):` Print information every 10 major iterations, where a major iteration is defined by a new solution estimate.

`3 (majorit):` Print information at each major iteration.

`4 (allit):` Print information at each major and minor iteration, where a minor iteration is defined by a trial iterate.

`5 (allit_x):` Print all the above, and the values of the solution vector `x`.

`6 (all):` Print all the above, and the values of the constraints `c` at `x` and the Lagrange multipliers `lambda`.

Default value: `2`

`outmode (KTR_PARAM_OUTMODE):` 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`

`pivot (KTR_PARAM_PIVOT):` Specifies the initial pivot threshold used in factorization routines. The value should be in the range [0 .. 0.5] with higher values resulting in more pivoting (more stable factorizations). Values less than 0 will be set to 0 and values larger than 0.5 will be set to 0.5. If "`pivot`" is non-positive, initially no pivoting will be performed. Smaller values may improve the speed of the code but higher values are recommended for more stability (for example, if the problem appears to be very ill-conditioned).

Default value: `1.0e-8`

`scale (KTR_PARAM_SCALE):` Performs a scaling of the objective and constraint functions based on their values at the initial point. If scaling is performed, all internal computations, including the stopping tests, are based on the scaled values.

`0 (no):`	No scaling is performed.
`1 (yes):`	KNITRO is allowed to scale the objective function and constraints.

Default value: `1`

`soc (KTR_PARAM_SOC):` 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`

`xtol (KTR_PARAM_XTOL):` The optimization process will terminate if the relative change in all components of the solution point estimate is less than "`xtol`". If using the Interior/Direct or Interior/CG algorithm and the barrier parameter is still large, KNITRO will first try decreasing the barrier parameter before terminating.

Default value: `1.0e-15`

5.2 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. (Note that the AMPL interface to KNITRO cannot read such a file. Other modeling environments may be able to read an options file – please check with the modeling vendor.)

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
```

The options file is read into KNITRO by calling the following function before invoking `KTR_solve()`:

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

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

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

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

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

For example:

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

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

Most user options can be specified with either a numeric value or a string value. The individual user options and their possible numeric values are described in section 5.1. String values are listed in the comments of the file examples/C/knitro.opt provided with the distribution.

5.3 Setting options through function calls

The functions for setting user options have the form:

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

for setting integer valued parameters, or

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

for setting double precision valued parameters.

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

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

NOTE: User parameters cannot be set after beginning the optimization process; i.e., after making the first call to `KTR_solve()`. Some options cannot be set after calling `KTR_init_problem()`.

5.4 Loading 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 5.1.0 is installed at its default location)

```
> set PATH=%PATH%;C:\Program Files\Ziena\knitro-5.1.0-z\lib
```

On Mac OS X, type (assuming KNITRO 5.1.0 is installed at `/tmp`)

```
> export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:/tmp/knitro-5.1.0-z/lib
```

If you run a Unix bash shell, then type (assuming KNITRO 5.1.0 is installed at `/tmp`)

```
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/tmp/knitro-5.1.0-z/lib
```

If you run a Unix csh or tcsh shell, then type (assuming KNITRO 5.1.0 is installed at `/tmp`)

```
> setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/tmp/knitro-5.1.0-z/lib
```

6 KNITRO termination test and optimality

The first-order conditions for identifying a locally optimal solution of the problem (1.1) are:

$$\begin{aligned}
\nabla_x \mathcal{L}(x,\lambda) = \nabla f(x) + \sum_{i\in\mathcal{E}} \lambda_i \nabla h_i(x) + \sum_{i\in\mathcal{I}} \lambda_i \nabla g_i(x) &= 0 && (6.14)\\
\lambda_i g_i(x) &= 0, \quad i \in \mathcal{I} && (6.15)\\
h_i(x) &= 0, \quad i \in \mathcal{E} && (6.16)\\
g_i(x) &\leq 0, \quad i \in \mathcal{I} && (6.17)\\
\lambda_i &\geq 0, \quad i \in \mathcal{I} && (6.18)
\end{aligned}$$

where $\mathcal{E}$ and $\mathcal{I}$ represent the sets of indices corresponding to the equality constraints and inequality constraints respectively, and λ_i is the Lagrange multiplier corresponding to constraint i. In KNITRO we define the feasibility error (`Feas err`) at a point x^k to be the maximum violation of the constraints (6.16), (6.17), i.e.,

$$\text{Feas err} = \max_{i\in\mathcal{E}\cup\mathcal{I}} (0, |h_i(x^k)|, g_i(x^k)), \qquad (6.19)$$

while the optimality error (`Opt err`) is defined as the maximum violation of the first two conditions (6.14), (6.15),

$$\text{Opt err} = \max_{i\in\mathcal{I}} (\|\nabla_x \mathcal{L}(x^k,\lambda^k)\|_\infty, \min(|\lambda_i^k g_i(x^k)|, |\lambda_i^k|, |g_i(x^k)|)). \qquad (6.20)$$

The last optimality condition (6.18) is enforced explicitly throughout the optimization. In order to take into account problem scaling in the termination test, the following scaling factors are defined

$$\begin{aligned}
\tau_1 &= \max(1, |h_i(x^0)|, g_i(x^0)), && (6.21)\\
\tau_2 &= \max(1, \|\nabla f(x^k)\|_\infty), && (6.22)
\end{aligned}$$

where x^0 represents the initial point.

For unconstrained problems, the scaling (6.22) is not effective since $\|\nabla f(x^k)\|_\infty \to 0$ as a solution is approached. Therefore, for unconstrained problems only, the following scaling is used in the termination test

$$\tau_2 = \max(1, \min(|f(x^k)|, \|\nabla f(x^0)\|_\infty)), \qquad (6.23)$$

in place of (6.22).

KNITRO stops and declares `LOCALLY OPTIMAL SOLUTION FOUND` if the following stopping conditions are satisfied:

$$\begin{aligned}
\text{Feas err} &\leq \max(\tau_1 * \texttt{feastol}, \texttt{feastol\_abs}) && (6.24)\\
\text{Opt err} &\leq \max(\tau_2 * \texttt{opttol}, \texttt{opttol\_abs}) && (6.25)
\end{aligned}$$

where "`feastol`", "`opttol`", "`feastol_abs`", and "`opttol_abs`" are constants defined by user options (see section 5).

This stopping test is designed to give the user much flexibility in deciding when the solution returned by KNITRO is accurate enough. One can use a scaled stopping test (which is the recommended default option) by setting "`feastol_abs`" and "`opttol_abs`" equal to `0.0e0`. Likewise, an absolute stopping test can be enforced by setting "`feastol`" and "`opttol`" equal to `0.0e0`.

Unbounded problems

Since by default, KNITRO uses a relative/scaled stopping test it is possible for the optimality conditions to be satisfied within the tolerances given by (6.24)-(6.25) for an unbounded problem. For example, if $\tau_2 \to \infty$ while the optimality error (6.20) stays bounded, condition (6.25) will eventually be satisfied for some "`opttol>0`". If you suspect that your problem may be unbounded, using an absolute stopping test will allow KNITRO to detect this.

7 Knitro output and solution information

This section provides information on understanding the Knitro output and accessing solution information.

7.1 Understanding Knitro output

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`").

This section describes Knitro outputs at various levels. We examine the output that results from running examples/C/callback2_static to solve problemHS15.c.

Display of Nondefault Options:

Knitro first prints the banner displaying the Ziena license type and version of Knitro that is installed. It then lists all user options which are different from their default values (see section 5 for the default user option settings). If nothing is listed in this section then it must be that all user options are set to their default values. Lastly, Knitro prints messages that describe how it resolved user options that were set to `AUTOMATIC` values. For example, if option "`algorithm=auto`", then Knitro prints the algorithm that it chooses.

```
======================================
       Commercial Ziena License
             KNITRO 5.1.0
       Ziena Optimization, Inc.
       website:  www.ziena.com
       email:   info@ziena.com
======================================

outlev:       6
KNITRO changing algorithm from AUTO to 1.
KNITRO changing bar_murule from AUTO to 1.
KNITRO changing bar_initpt 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 determined two 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 providing the exact Hessian).

```
Problem Characteristics
-----------------------
Objective goal:  Minimize
Number of variables:                     2
```

```
      bounded below:                  0
      bounded above:                  1
      bounded below and above:        0
      fixed:                          0
      free:                           1
  Number of constraints:              2
      linear equalities:              0
      nonlinear equalities:           0
      linear inequalities:            0
      nonlinear inequalities:         2
      range:                          0
  Number of nonzeros in Jacobian:     4
  Number of nonzeros in Hessian:      3
```

Display of Iteration Information:

Next, if "`outlev`" is greater than 2, Knitro prints columns of data reflecting detailed information about individual iterations during the solution process. A major iteration is defined as a step which generates a new solution estimate (i.e., a successful step). A minor iteration is one which generates a trial step (which may either be accepted or rejected).

If "`outlev=2`", this data is printed every 10 major iterations, and on the final iteration. If "`outlev=3`", this data is printed every major iteration. If "`outlev`" is greater than 3, information is printed for every major and minor iteration.

```
   Iter(maj/min)  Res   Objective      Feas err    Opt err     ||Step||    CG its
  --------------- ---  -------------  ----------  ----------  ----------  -------
        0/      0 ---   9.090000e+02   3.000e+00
        1/      1 Acc   7.989784e+02   2.878e+00   9.096e+01   6.566e-02        0
        2/      2 Acc   4.232342e+02   2.554e+00   5.828e+01   2.356e-01        0
        3/      3 Acc   1.457686e+01   9.532e-01   3.088e+00   1.909e+00        0
                4 Rej   3.227542e+02   9.532e-01   3.088e+00   1.483e+01        1
                5 Rej   1.803608e+03   9.532e-01   3.088e+00   7.330e+00        1
                6 Rej   1.176121e+03   9.532e-01   3.088e+00   3.576e+00        1
                7 Rej   4.249636e+02   9.532e-01   3.088e+00   1.698e+00        1
        4/      8 Acc   1.235269e+02   7.860e-01   3.818e+00   7.601e-01        1
        5/      9 Acc   3.993788e+02   3.022e-02   1.795e+01   1.186e+00        0
        6/     10 Acc   3.924231e+02   2.924e-02   1.038e+01   1.856e-02        0
        7/     11 Acc   3.158787e+02   0.000e+00   6.905e-02   2.373e-01        0
        8/     12 Acc   3.075530e+02   0.000e+00   6.888e-03   2.255e-02        0
        9/     13 Acc   3.065107e+02   0.000e+00   6.397e-05   2.699e-03        0
       10/     14 Acc   3.065001e+02   0.000e+00   4.457e-07   2.714e-05        0
```

The meaning of each column is described below.

`Iter:`	Iteration number (major/minor).
`Res:`	The step result. The values in this column indicate whether or not the step attempted during the iteration was accepted (`Acc`) or rejected (`Rej`) by the merit function. If the step was rejected, the solution estimate was not updated. (This information is only printed if "`outlev`" is greater than 3).

`Objective:`	Gives the value of the objective function at the trial iterate.
`Feas err:`	Gives a measure of the feasibility violation at the trial iterate (see section 6).
`Opt err:`	Gives a measure of the violation of the Karush-Kuhn-Tucker (KKT) (first-order) optimality conditions (not including feasibility) (see section 6).
`||Step||:`	The 2-norm length of the step (i.e., the distance between the trial iterate and the old iterate).
`CG its:`	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 (see section 6), the message will look as follows:

```
EXIT: LOCALLY OPTIMAL SOLUTION FOUND.
```

See the appendix for a list of possible termination messages and a description of their meaning and the corresponding value returned by `KTR_solve()`.

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               =   3.06500096351765e+02
Final feasibility error (abs / rel) =   0.00e+00 / 0.00e+00
Final optimality error  (abs / rel) =   4.46e-07 / 3.06e-08
# of iterations (major / minor)     =         10 /       14
# of function evaluations           =         15
# of gradient evaluations           =         11
# of Hessian evaluations            =         10
Total program time (secs)           =       0.00136 (     0.000 CPU time)
Time spent in evaluations (sec)     =       0.00012
```

Display of Solution Vector and Constraints:

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

```
Constraint Vector                  Lagrange Multipliers
-----------------                  ---------------------
c[       0] =   1.00000006873e+00,   lambda[       0] =  -7.00000062964e+02
```

```
c[       1] =   4.50000096310e+00,   lambda[       1] =  -1.07240081095e-05

Solution Vector
---------------
x[       0] =   4.99999972449e-01,   lambda[       2] =   7.27764067199e+01
x[       1] =   2.00000024766e+00,   lambda[       3] =   0.00000000000e+00

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

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.

Users can generate a file containing iterates and/or solution points with option "`newpoint`". The output file is called knitro_newpoint.txt. See section 5 for details.

7.2 Accessing solution information

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

The `KTR_solve()` function (see section 4) returns the final value of the objective function in `obj`, the final (primal) solution vector in the array `x` and the final values of the Lagrange multipliers (or dual variables) in the array `lambda`. The solution status code is given by the return value from `KTR_solve()`.

In addition, information related to the final statistics can be retrieved through the following function calls:

```
int KTR_get_number_FC_evals (const KTR_context_ptr  kc);
```

This function call returns the number of function evaluations requested by `KTR_solve()`. It returns a negative number if there is a problem with `kc`.

```
int KTR_get_number_GA_evals (const KTR_context_ptr  kc);
```

This function call returns the number of gradient evaluations requested by `KTR_solve()`. It returns a negative number if there is a problem with `kc`.

```
int KTR_get_number_H_evals (const KTR_context_ptr  kc);
```

This function call returns the number of Hessian evaluations requested by `KTR_solve()`. It returns a negative number if there is a problem with `kc`.

```
int KTR_get_number_HV_evals (const KTR_context_ptr  kc);
```

This function call returns the number of Hessian-vector products requested by `KTR_solve()`. It returns a negative number if there is a problem with `kc`.

```
int KTR_get_number_major_iters (const KTR_context_ptr  kc);
```

This function returns the number of major iterations made by `KTR_solve()`. It returns a negative number if there is a problem with `kc`.

```
int KTR_get_number_minor_iters (const KTR_context_ptr  kc);
```

This function returns the number of minor iterations made by `KTR_solve()`. It returns a negative number if there is a problem with `kc`.

```
double KTR_get_abs_feas_error (const KTR_context_ptr  kc);
```

This function returns the absolute feasibility error at the solution. See 6 for a detailed definition of this quantity. It returns a negative number if there is a problem with `kc`.

```
double KTR_get_abs_opt_error (const KTR_context_ptr  kc);
```

This function returns the absolute optimality error at the solution. See 6 for a detailed definition of this quantity. It returns a negative number if there is a problem with `kc`.

8 Algorithm Options

8.1 Automatic

KNITRO provides three different algorithms for solving problems. See section 1.2 for an overview of the methods. By default, KNITRO automatically tries to choose the best algorithm for a given problem based on problem characteristics.

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.

8.2 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 (see section 9.2). It may not be used with Hessian options 4 or 5, which do not supply a full Hessian matrix. The Interior/Direct algorithm may be used with the "`feasible`" option.

8.3 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, and with the "`feasible`" option. 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.

8.4 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`".

NOTE: The "`feasible`" option (see section 9.3) is not available for use with the Active Set algorithm. The method works with all Hessian options.

9 Other KNITRO special features

This section describes in more detail some of the most important features of KNITRO. It provides some guidance on which features to use so that KNITRO runs most efficiently for the problem at hand.

9.1 First derivative and gradient check options

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`" (see section 5).

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`" (see section 5). 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.

Gradient Checks
If the user supplies a routine for computing exact gradients, KNITRO can easily check them against finite-difference gradient approximations. To do this, modify your application and replace the call to `KTR_solve()` with `KTR_check_first_ders()`, then run the application. KNITRO will call the user routine for exact gradients, compute finite-difference approximations, and print any differences that exceed a given threshold. KNITRO also checks that the sparse constraint Jacobian has all nonzero elements defined. The check can be made with forward or centered differences. A sample driver is provided in examples/C/checkDersExample.c. Small differences between exact and finite-difference approximations are to be expected (see comments in examples/C/checkDersExample.c). It is best to check the gradient at different points, and to avoid points where partial derivatives happen to equal zero.

9.2 Second derivative options

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

(Dense) Quasi-Newton BFGS
The quasi-Newton BFGS option uses gradient information to compute a symmetric, *positive-definite*

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

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

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

NOTE: This option may not be used when "`algorithm=1`".

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, Hv, and returns the result in `hessVector`. This option is chosen by setting user option "`hessopt=5`".

NOTE: This option may not be used when "`algorithm=1`".

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.

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`".

9.3 Feasible version

KNITRO offers an option "`feasible`" that forces iterates to stay feasible with respect to *inequality* constraints. The option does not enforce feasibility with respect to *equality* constraints, as this would impact performance too much. KNITRO satisfies inequalities by switching to a *feasible mode* of operation, which alters the manner in which iterates are computed. The theory behind feasible mode is described in [5].

The initial point must satisfy 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. We say *sufficient* satisfaction occurs at a point x if it is true for all inequalities that

$$cl + tol \leq c(x) \leq cu - tol \tag{9.26}$$

The constant $tol > 0$ is determined by the option "`feasmodetol`"; its default value is `1.0e-4`.

Feasible mode becomes active once an iterate x satisfies (9.26) for all inequality constraints. If the initial point satisfies (9.26), then every iterate will be feasible.

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

9.4 Honor Bounds

In some applications, the user may want to enforce that the initial point and all subsequent iterates satisfy the simple bounds $bl \leq x \leq bu$. 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`".

9.5 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 "`maxcrossit`" user option. If this parameter is greater than 0, then KNITRO will attempt to perform "`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 "`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 "`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 "`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 "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.

9.6 Multi-start

Nonlinear optimization problems (1.1) 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 described in section 6. Default KNITRO behavior is to return the first locally optimal point found. KNITRO 5.1 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".

The multi-start procedure generates new start points by randomly selecting components of x that satisfy lower and upper bounds on the variables. KNITRO finds a local optimum from each start point using the same problem definition and user options. The final solution returned from KTR_solve() is the local optimum with the best objective function value. If you wish to see details of the local optimization process for each start point, then set "outlev" to at least 4.

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

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 option "ms_maxbndrange". This keeps new start points within a specified distance of the initial start point. See section 5.1 for details.

In most cases the user would like to obtain the *global optimum* to (1.1); 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. See section 10.6 for more discussion.

9.7 Reverse communication mode for invoking KNITRO

The reverse communication mode of KNITRO returns control to the user at the driver level whenever a function, gradient, or Hessian evaluation is needed, making it easy to embed the KNITRO solver into an application. In addition, this mode allows applications to monitor or stop the progress of the KNITRO solver after each iteration, based on any criteria the user desires.

If the return value from `KTR_solve()` is 0 or negative, the optimization is finished (see Appendix A). If the return value is positive, KNITRO requires that some task be performed by the user at the driver level before re-entering `KTR_solve()`. Referring to the optimization problem formulation given in (4.4), the action to take for possible positive return values are:

`KTR_RC_EVALFC (1):`	Evaluate functions $f(x)$ and $c(x)$ and re-enter `KTR_solve()`.
`KTR_RC_EVALGA (2):`	Evaluate gradient $\nabla f(x)$ and the constraint Jacobian matrix and re-enter `KTR_solve()`.
`KTR_RC_EVALH (3):`	Evaluate the Hessian $H(x, \lambda)$ and re-enter `KTR_solve()`.
`KTR_RC_EVALHV (7):`	Evaluate the Hessian $H(x, \lambda)$ times a vector and re-enter `KTR_solve()`.
`KTR_RC_NEWPOINT (6):`	KNITRO has just computed a new solution estimate, and the function and gradient values are up-to-date. The user may provide routines to perform some task. Then the application must re-enter `KTR_solve()` so that KNITRO can begin a new major iteration. `KTR_RC_NEWPOINT` is only returned if user option "`newpoint=user`".

Section 4.2 describes an example program that uses the reverse communications mode.

9.8 Callback mode for invoking KNITRO

The callback mode of KNITRO requires the user to supply several function pointers that KNITRO calls when it needs new function, gradient or Hessian values, or to execute a user-provided newpoint routine. For convenience, every one of these callback routines takes the same list of arguments. If your callback requires additional parameters, you are encouraged to create a structure containing them and pass its address as the `userParams` pointer. KNITRO does not modify or dereference the `userParams` pointer, so it is safe to use for this purpose. Section 4.2 describes an example program that uses the callback mode.

The C language prototype for the KNITRO callback function is defined in knitro.h:

```
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);
```

The callback functions for evaluating the functions, gradients and Hessian or for performing some newpoint task, are set as described below. Each user callback routine should return an `int` value of 0 if successful, or a negative value to indicate that an error occurred during execution of the user-provided function. Section 4.2 describes example program that uses the callback mode.

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

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

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

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

NOTE: To enable "newpoint" callbacks, set "`newpoint=user`".

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, the user can define a callback function to handle all output. This callback function can be set as shown below.

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

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

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

10 Special problem classes

This section describes 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.

10.1 Linear programming problems (LPs)

A linear program (LP) is an optimization problem where the objective function and all the constraint functions are linear.

KNITRO has built in specializations for efficiently solving LPs. However, KNITRO is unable to automatically detect whether or not a problem is an LP. In order for KNITRO to detect that a problem is an LP, you must specify this by setting the value of `objType` to `KTR_OBJTYPE_LINEAR` and all values of the array `cType` to `KTR_CONTYPE_LINEAR` in the function call to `KTR_init_problem()` (see section 4). If this is not done, KNITRO will not apply special treatment to the LP and will typically be less efficient in solving the LP.

10.2 Quadratic programming problems (QPs)

A quadratic program (QP) is an optimization problem where the objective function is quadratic and all the constraint functions are linear.

KNITRO has built in specializations for efficiently solving QPs. However, KNITRO is unable to automatically detect whether or not a problem is a QP. In order for KNITRO to detect that a problem is a QP, you must specify this by setting the value of `objType` to `KTR_OBJTYPE_QUADRATIC` and all values of the array `cType` to `KTR_CONTYPE_LINEAR` in the function call to `KTR_init_problem()` (see section 4). If this is not done, KNITRO will not apply special treatment to the QP and will typically be less efficient in solving the QP.

Typically, these specialization will only help on convex QPs.

10.3 Systems of Nonlinear Equations

KNITRO is effective at solving systems of nonlinear equations. To solve a square system of nonlinear equations using KNITRO one should specify the nonlinear equations as equality constraints (1.1*b*), and specify the objective function (1.1*a*) as zero (i.e., $f(x) = 0$).

10.4 Least Squares Problems

There are two ways of using KNITRO for solving problems in which the objective function is a sum of squares of the form

$$f(x) = \frac{1}{2}\sum_{j=1}^{q} r_j(x)^2.$$

If the value of the objective function at the solution is not close to zero (the large residual case), the least squares structure of f can be ignored and the problem can be solved as any other optimization problem. Any of the KNITRO options can be used.

On the other hand, if the optimal objective function value is expected to be small (small residual case) then KNITRO can implement the Gauss-Newton or Levenberg-Marquardt methods which only require first derivatives of the residual functions, $r_j(x)$, and yet converge rapidly. To do so, the user need only define the Hessian of f to be

$$\nabla^2 f(x) = J(x)^T J(x),$$

where

$$J(x) = \left[\frac{\partial r_j}{\partial x_i} \right]_{\substack{j=1,2,\ldots,q \\ i=1,2,\ldots,n}} .$$

The actual Hessian is given by

$$\nabla^2 f(x) = J(x)^T J(x) + \sum_{j=1}^{q} r_j(x) \nabla^2 r_j(x);$$

the Gauss-Newton and Levenberg-Marquardt approaches consist of ignoring the last term in the Hessian.

KNITRO will behave like a Gauss-Newton method by setting "`algorithm=1`", and will be very similar to the classical Levenberg-Marquardt method when "`algorithm=2`". For a discussion of these methods see, for example, [10].

10.5 Mathematical programs with complementarity constraints (MPCCs)

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 which enforces that two variables are *complementary* to each other, i.e., the variables x_1 and x_2 are complementary if the following conditions hold

$$x_1 \times x_2 = 0, \quad x_1 \geq 0, \quad x_2 \geq 0. \tag{10.27}$$

The condition above, is sometimes expressed more compactly as

$$0 \leq x_1 \quad \perp \quad x_2 \geq 0.$$

One could also have more generally, that a particular constraint is complementary to another constraint or a constraint is complementary to a variable. However, by adding slack variables, a complementarity constraint can always be expressed as two variables complementary to each other, and KNITRO requires that you express complementarity constraints in this form. For example, if you have two constraints $c_1(x)$ and $c_2(x)$ which are complementary

$$c_1(x) \times c_2(x) = 0, \quad c_1(x) \geq 0, \quad c_2(x) \geq 0,$$

you can re-write this as two equality constraints and two complementary variables, s_1 and s_2 as follows:

$$\begin{aligned} s_1 &= c_1(x) && (10.28) \\ s_2 &= c_2(x) && (10.29) \\ s_1 \times s_2 &= 0, \quad s_1 \geq 0, \quad s_2 \geq 0. && (10.30) \end{aligned}$$

Intuitively, a complementarity constraint is a way to model a constraint which is combinatorial in nature since, for example, the conditions in (10.27) imply that either x_1 or x_2 must be 0 (both may be 0 as well). Without special care, these type of constraints may cause problems for nonlinear optimization solvers because problems which contain these types of constraints fail to satisfy constraint qualifications which 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 apply some special care to them internally.

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

Prototype:

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

Arguments:

`KTR_context *kc:` is a pointer to a structure which holds all the relevant information about a particular problem instance.

`int numCompConstraints:` is a scalar specifying the number of complementarity constraints to be added to the problem (i.e., the number of pairs of variables which are complementary to each other).

`int *indexList1:` is an array of length `numCompConstraints` specifying the variable indices for the first set of variables in the pairs of complementary variables.

`int *indexList2:` is an array of length `numCompConstraints` specifying the variable indices for the second set of variables in the pairs of complementary variables.

The call to `KTR_addcompcons()` must occur after the call to `KTR_init_problem()`, but before the first call to `KTR_solve()`. Below we provide a simple example of how to define the KNITRO data structures to specify a problem which includes complementarity constraints.

Example: [1]

Assume we want to solve the following MPEC with KNITRO.

$$\begin{aligned}
\underset{x}{\text{minimize}} \quad & f(x) = (x_0-5)^2 + (2x_1+1)^2 && (10.31a)\\
\text{subject to} \quad & c_0(x) = 2(x_1-1) - 1.5x_0 + x_2 - 0.5x_3 + x_4 = 0 && (10.31b)\\
& c_1(x) = 3x_0 - x_1 - 3 \geq 0 && (10.31c)\\
& c_2(x) = -x_0 + 0.5x_1 + 4 \geq 0 && (10.31d)
\end{aligned}$$

[1] An MPEC from J.F. Bard, Convex two-level optimization, *Mathematical Programming* 40(1), 15-27, 1988.

$$
\begin{aligned}
& c_3(x) = -x_0 - x_1 + 7 \geq 0 && (10.31e)\\
& x_i \geq 0, i = 0..4 && (10.31f)\\
& c_1(x)x_2 = 0 && (10.31g)\\
& c_2(x)x_3 = 0 && (10.31h)\\
& c_3(x)x_4 = 0. && (10.31i)
\end{aligned}
$$

It is easy to see that the last 3 constraints (along with the corresponding non-negativity conditions) represent complementarity constraints. Expressing this in compact notation, we have:

$$
\begin{aligned}
\underset{x}{\text{minimize}} \quad & f(x) = (x_0 - 5)^2 + (2x_1 + 1)^2 && (10.32a)\\
\text{subject to} \quad & c_0(x) = 0 && (10.32b)\\
& 0 \leq c_1(x) \perp x_2 \geq 0 && (10.32c)\\
& 0 \leq c_2(x) \perp x_3 \geq 0 && (10.32d)\\
& 0 \leq c_3(x) \perp x_4 \geq 0 && (10.32e)\\
& x_0 \geq 0, x_1 \geq 0. && (10.32f)
\end{aligned}
$$

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 problem (10.31) as

$$
\begin{aligned}
\underset{x}{\text{minimize}} \quad & f(x) = (x_0 - 5)^2 + (2x_1 + 1)^2 && (10.33a)\\
\text{subject to} \quad & c_0(x) = 2(x_1 - 1) - 1.5x_0 + x_2 - 0.5x_3 + x_4 = 0 && (10.33b)\\
& \tilde{c_1}(x) = 3x_0 - x_1 - 3 - x_5 = 0 && (10.33c)\\
& \tilde{c_2}(x) = -x_0 + 0.5x_1 + 4 - x_6 = 0 && (10.33d)\\
& \tilde{c_3}(x) = -x_0 - x_1 + 7 - x_7 = 0 && (10.33e)\\
& x_i \geq 0, i = 0..7 && (10.33f)\\
& x_2 \perp x_5 && (10.33g)\\
& x_3 \perp x_6 && (10.33h)\\
& x_4 \perp x_7. && (10.33i)
\end{aligned}
$$

Now that the problem is in a form suitable for KNITRO, we define the problem for KNITRO by using `c`, `cLoBnds`, and `cUpBnds` for (10.33*b*)-(10.33*e*), and `xLoBnds`, `xUpBnds` for (10.33*f*) to specify the normal constraints and bounds in the usual way for KNITRO. We use `indexList1`, `indexList2` and the `KTR_addcompcons()` function call to specify the complementarity constraints (10.33*g*)-(10.33*i*). These arrays are specified as follows for (10.33).

```
n = 8;                   /* number of variables */
m = 4;                   /* number of regular constraints */
numCompConstraints = 3; /* number of complementarity constraints */

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

```
c[3] = -x[0] - x[1] + 7 - x[7];

cLoBnds[0] = 0;  cUpBnds[0] = 0;
cLoBnds[1] = 0;  cUpBnds[1] = 0;
cLoBnds[2] = 0;  cUpBnds[2] = 0;
cLoBnds[3] = 0;  cUpBnds[3] = 0;

xLoBnds[0] = 0;  xUpBnds[0] = KTR_INFBOUND;
xLoBnds[1] = 0;  xUpBnds[1] = KTR_INFBOUND;
xLoBnds[2] = 0;  xUpBnds[2] = KTR_INFBOUND;
xLoBnds[3] = 0;  xUpBnds[3] = KTR_INFBOUND;
xLoBnds[4] = 0;  xUpBnds[4] = KTR_INFBOUND;
xLoBnds[5] = 0;  xUpBnds[5] = KTR_INFBOUND;
xLoBnds[6] = 0;  xUpBnds[6] = KTR_INFBOUND;
xLoBnds[7] = 0;  xUpBnds[7] = KTR_INFBOUND;

indexList1[0] = 2;   indexList2[0] = 5;
indexList1[1] = 3;   indexList2[1] = 6;
indexList1[2] = 4;   indexList2[2] = 7;
```

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

When using KNITRO through a particular modeling language, only some modeling languages allow for the identification of complementarity constraints. 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.

10.6 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 which attempts to find multiple local solutions in the hopes of locating the global solution. See section 9.6 for information on trying to find the globally optimal solution using the KNITRO multi-start feature.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] R. H. Byrd, J. Nocedal, and R.A. Waltz. KNITRO: An integrated package for nonlinear optimization. In G. di Pillo and M. Roma, editors, *Large-Scale Nonlinear Optimization*, pages 35–59. Springer, 2006.

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

[8] Harwell Subroutine Library. *A catalogue of subroutines (HSL 2002).* AEA Technology, Harwell, Oxfordshire, England, 2002.

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

[10] J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer Series in Operations Research. Springer, 1999.

[11] 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.

Appendix A
Solution Status Codes

0: `EXIT: LOCALLY OPTIMAL SOLUTION FOUND.`

KNITRO found a locally optimal point which satisfies the stopping criterion (see section 6 for more detail on how this is defined). If the problem is convex (for example, a linear program), then this point corresponds to a globally optimal solution.

-1: `EXIT: Iteration limit reached.`

The iteration limit was reached before being able to satisfy the required stopping criteria.

-2: `EXIT: Convergence to an infeasible point.`
`Problem may be locally infeasible.`

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. If this occurs for a variety of initial points, it is likely the problem is infeasible.

-3: `EXIT: Problem appears to be unbounded.`
`Iterate is feasible and objective magnitude > 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.

-4: `EXIT: Relative change in solution estimate < xtol.`

The relative change in the solution estimate is less than that specified by the paramater `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. If the current point is feasible, it is possible it may be optimal, however the stopping tests cannot be satisfied (perhaps because of degeneracy, ill-conditioning or bad scaling).

-5: `EXIT: Current solution estimate cannot be improved. Point appears to be optimal, but desired accuracy 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.

-6: `EXIT: Time limit reached.`

The time limit was reached before being able to satisfy the required stopping criteria.

-50 to -60:

Termination values in this range imply some input error. If `outlev>0` details of this error will be printed to standard output or the file `knitro.log` depending on the value of `outmode`.

-90: `EXIT: Callback function error.`

This termination value indicates that an error (i.e., negative return value) occurred in a user provided callback routine.

-97: `EXIT: 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.

-98: `EXIT: 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.

-99: `EXIT: Not enough memory available to solve problem.`

This termination value indicates that there was not enough memory available to solve the problem.

Appendix B
Migrating to Knitro 5.x

Migrating to Knitro 5.x

Knitro 5.x is NOT backwards compatible with previous versions of Knitro. However, it should be a simple process to migrate from Knitro 4.0 to 5.x as the primary data structures have not changed. Whereas Knitro 4.0 solved problems through a sequence of three function calls:

- `KTR_new()`
- `KTR_solve()`
- `KTR_free()`

Knitro 5.x uses a sequence of four function calls:

- `KTR_new()`
- `KTR_init_problem()`
- `KTR_solve()`
- `KTR_free()`

Here, `KTR_init_problem()` is a new function call used to pass in the optimization problem definition. There is also a new function call `KTR_restart()` for re-solving the same problem with a different initial point or different user option settings.

Summary of API changes:

- The argument list in the function call `KTR_new()` has changed. No arguments are passed.
- A new function `KTR_init_problem()` was created to pass information which describes the structure of your problem.
- The argument list in the function call `KTR_solve()` has changed. Many variables which used to be passed through `KTR_solve()` are now passed through `KTR_init_problem()`.
- A new argument `objGoal` was created to specify whether the problem is formulated as a minimization problem or a maximization problem. Pass the argument to `KTR_init_problem()`.
- Many variable names have changed to be more descriptive (although their structure is the same).
- New functions of the form

  `KTR_get_*`

  were created for retrieving solution information (see section 7).

See section 4 for detailed information on using the Knitro 5.x API. In addition numerous sample programs are provided in the `examples` directory of the distribution.