# KNITRO 4.0

## User's Manual

# KNITRO User's Manual

**Version 4.0**

Richard A. Waltz
Ziena Optimization, Inc.
Northwestern University

October 2004

# Contents

# 1 Introduction

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

## 1.1 KNITRO Overview

KNITRO 4.0 is a software package for finding local solutions of continuous, smooth optimization problems, with or without constraints. Even though KNITRO has been designed for solving large-scale general nonlinear problems, it is efficient for solving all of the following classes of smooth optimization problems:

- unconstrained,
- bound constrained,
- equality constrained,
- systems of nonlinear equations,
- least squares problems,
- linear programming problems (LPs),
- quadratic programming problems (QPs),
- general (inequality) constrained problems.

The KNITRO package provides the following features:

- Efficient and robust solution of small or large problems,
- Derivative-free, 1st derivative and 2nd derivative options,
- Both interior-point (barrier) and active-set optimizers,
- Both feasible and infeasible versions,
- Both iterative and direct approaches for computing steps,
- Interfaces: AMPL, C/C++, Fortran, GAMS, Matlab, Microsoft Excel, Visual Basic
- Reverse communication design for more user control over the optimization process and for easily embedding KNITRO within another piece of software.

The problems solved by KNITRO have the form

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

This allows many forms of constraints, including bounds on the variables. 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 both state-of-the-art interior-point and active-set methods for solving nonlinear optimization problems. In the interior method (also known as a barrier method), the nonlinear programming problem is replaced by a series of barrier sub-problems controlled by a barrier parameter $\mu$. The algorithm uses trust regions and a merit function to promote convergence. The algorithm performs one or more minimization steps on each barrier problem, then decreases the barrier parameter, and repeats the process until the original problem (1.1) has been solved to the desired accuracy.

KNITRO provides two procedures for computing the steps within the interior point approach. In the version known as `Interior/CG` each step is computed using a projected conjugate gradient iteration. This approach differs from most interior methods proposed in the literature in that it does not compute each step by solving a linear system involving the KKT (or primal-dual) matrix. Instead, it factors a projection matrix, and uses the conjugate gradient method, to approximately minimize a quadratic model of the barrier problem.

The second procedure for computing the steps, which we call `Interior/Direct`, always attempts to compute a new iterate by solving the primal-dual KKT matrix using direct linear algebra. In the case when this step cannot be guaranteed to be of good quality, or if negative curvature is detected, then the new iterate is computed by the `Interior/CG` procedure.

KNITRO also implements an active-set sequential linear-quadratic programming (SLQP) algorithm which we call `Active`. This method is similar in nature to a sequential quadratic programming method but uses linear programming sub-problems to estimate the active-set at each iteration. This active-set code may be preferable when a good initial point can be provided, for example, when solving a sequence of related 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.*

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 [9]. The `Active` algorithm is described in [3] and the global convergence theory for this algorithm is in [2]. An important component of KNITRO is the HSL routine `MA27` [7] which is used to solve the linear systems arising at every iteration of the algorithm. In addition, the `Active` algorithm in KNITRO may make use of the COIN-OR Clp linear programming solver module. The version used in KNITRO 4.0 may be downloaded from `http://www.ziena.com/clp.html`.

## 1.2 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

support-knitro@ziena.com

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

info-knitro@ziena.com

# 2 Installing

Instructions for installing the KNITRO package on both Windows and Linux/UNIX platforms are described below.

## 2.1 Windows

Download and save the installer (KNITRO-4.0.exe) on your computer. Double click on the installer to run it. It will prompt you for a place to install and create some start menu shortcuts. Afterwards, you can delete the installer. By default KNITRO will be installed in the folder `C:\Program Files\knitro-4.0`.

If you are using the full version of KNITRO you will have to activate it for your computer. Use the "Locking Code" start menu item to determine your computer's code to send to Ziena. We will send you a license key for it.

| | |
|---|---|
| INSTALL: | A file containing installation instructions. |
| LICENSE: | A file containing the KNITRO license agreement. |
| README: | A file with instructions on how to get started using KNITRO. |
| doc: | Directory containing KNITRO documentation including this manual. |
| include: | Directory containing the KNITRO header file `knitro.h`. |
| lib: | Directory containing the KNITRO library file `knitro.lib`. |
| ampl: | Directory containing files and instructions for using KNITRO with AMPL and an example AMPL model. |
| C: | Directory containing instructions and an example for calling KNITRO from a C (or C++) program. |
| fortran: | Directory containing instructions and an example for calling KNITRO from a Fortran program. |
| callback: | Directory containing instructions and an example for calling KNITRO from a C (or C++) program using the callback feature. |

Before beginning, view the `INSTALL`, `LICENSE` and `README` files. To get started, go inside the interface folder which you would like to use and view the `README` file inside of that folder. Example problems are provided in the `C`, `callback` and `Fortran` interface folders. It is recommended to run and understand these example problems before proceeding, as these problems contain all the essential features of using the KNITRO callable library. The `Ampl` interface folder contains instructions for using KNITRO with AMPL and an example of how to formulate a problem in AMPL. More detailed information on using KNITRO with the various interfaces can be found in the KNITRO manual `knitroman` in the `doc` folder.

For instructions on using KNITRO with the Visual Studio environment refer to Section 4.5.

## 2.2 Linux/UNIX

Save the downloaded file (KNITRO.4.0.tar.gz) in a fresh subdirectory on your system. To install, first type

```
gunzip KNITRO.4.0.tar.gz
```

to produce a file KNITRO.4.0.tar. Then, type

```
tar -xvf KNITRO.4.0.tar
```

to create the directory KNITRO.4.0 containing the following files and subdirectories:

| | |
|---|---|
| INSTALL: | A file containing installation instructions. |
| LICENSE: | A file containing the KNITRO license agreement. |
| README: | A file with instructions on how to get started using KNITRO. |
| bin: | Directory containing the precompiled KNITRO/AMPL executable file, `knitro-ampl`. |
| doc: | Directory containing KNITRO documentation including this manual. |
| include: | Directory containing the KNITRO header file `knitro.h`. |
| lib: | Directory containing the KNITRO library files, `libknitro.a` and `libknitro.so`. |
| ampl: | Directory containing files and instructions for using KNITRO with AMPL and an example AMPL model. |
| C: | Directory containing instructions and an example for calling KNITRO from a C (or C++) program. |
| fortran: | Directory containing instructions and an example for calling KNITRO from a Fortran program. |
| callback: | Directory containing instructions and an example for calling KNITRO from a C (or C++) program using the callback feature. |

Before beginning, view the `INSTALL`, `LICENSE` and `README` files. To get started, go inside the interface subdirectory which you would like to use and view the `README` file inside of that directory. Example problems are provided in the `C`, `callback` and `Fortran` interface subdirectories. It is recommended to run and understand these example problems before proceeding, as these problems contain all the essential features of using the KNITRO callable library. The `Ampl` interface subdirectory contains instructions for using KNITRO with AMPL and an example of how to formulate a problem in AMPL. More detailed information on using KNITRO with the various interfaces can be found in the KNITRO manual `knitroman` in the `doc` subdirectory.

# 3 AMPL interface

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 ones job of formulating and modeling a problem much simpler. For a description of AMPL see [6] 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 and that the KNITRO/AMPL executable file `knitro-ampl` resides in the current directory or in a directory which is specified in ones `PATH` environment variable (such as a `bin` directory).

Inside of AMPL, to invoke the KNITRO solver type:

```
option solver knitro-ampl;
```

at the prompt. Likewise, to specify user options one would type, for example,

```
option knitro_options ‘‘maxit=100 alg=2’’;
```

The above command would set the maximum number of allowable iterations to 100 and choose the `Interior/CG` algorithm (see section 8). See Tables 1-2 for a summary of all available user specifiable options in KNITRO for use with AMPL. For more detail on these options see section 5. Note, that in section 5, user parameters for the callable library are of the form "`KTR_PARAM_NAME`". In AMPL, parameters are set using only the (lowercase) "`name`" as specified in Tables 1-2.

**NOTE:** AMPL will often perform a reordering of the variables and constraints defined in the AMPL model (and may also simplify the form of the problem using a presolve). The output printed by KNITRO will correspond to this reformulated problem. To view values of the variables and constraints in the order and form corresponding to the original AMPL model, use the AMPL `display` command.

Below is an example AMPL model and AMPL session which calls KNITRO to solve the problem:

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

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

Assume the AMPL model for the above problem is defined 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.
#

# 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 one can express an optimization problem 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), `opttol=1e-8` (to tighten the optimality stopping tolerance) and `outlev=3` (to print output at each major iteration). See section 7 for an explanation of the output.

**AMPL Example**

```
ampl: reset;
ampl: option solver knitro-ampl;
```

```
ampl: option knitro_options "alg=2 opttol=1e-8 outlev=3";
ampl: model testproblem.mod;
ampl: solve;

KNITRO 4.0.0: 10/20/04

: alg=2
opttol=1e-8
outlev=3

=======================================
             KNITRO 4.0.0
       Ziena Optimization, Inc.
       website:  www.ziena.com
       email:   info@ziena.com
=======================================

algorithm:   2
opttol:      1e-08
outlev:      3

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

  Iter      Objective      Feas err    Opt err     ||Step||    CG its
--------  -------------   ---------   ---------   ---------   --------
       0   9.760000e+02   1.300e+01
       1   9.688061e+02   7.190e+00   6.919e+00   1.513e+00          1
       2   9.397651e+02   1.946e+00   2.988e+00   5.659e+00          1
```

```
       3   9.323614e+02   1.659e+00   5.003e-03   1.238e+00          2
       4   9.361994e+02   1.421e-14   9.537e-03   2.395e-01          2
       5   9.360402e+02   7.105e-15   1.960e-03   1.591e-02          2
       6   9.360004e+02   0.000e+00   1.597e-05   4.017e-03          1
       7   9.360000e+02   7.105e-15   1.945e-07   3.790e-05          2
       8   9.360000e+02   0.000e+00   1.925e-09   3.990e-07          1

EXIT: LOCALLY OPTIMAL SOLUTION FOUND.

Final Statistics
----------------
Final objective value               =   9.36000000040000e+02
Final feasibility error (abs / rel) =   0.00e+00 / 0.00e+00
Final optimality error  (abs / rel) =   1.92e-09 / 1.20e-10
# of iterations (major / minor)     =          8 /        8
# of function evaluations           =          9
# of gradient evaluations           =          9
# of Hessian evaluations            =          8
Total program time (sec)            =         0.00

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

KNITRO 4.0.0: LOCALLY OPTIMAL SOLUTION FOUND.
ampl:
```

See section 8 for algorithm descriptions and section 9 for other special features. A user running AMPL can skip section 4.

| OPTION | DESCRIPTION | DEFAULT |
|---|---|---|
| alg | optimization algorithm used:<br>0: automatic algorithm selection<br>1: Interior/Direct algorithm<br>2: Interior/CG algorithm<br>3: Active algorithm | 0 |
| barrule | barrier paramater update rule:<br>0: automatic barrier rule chosen<br>1: monotone decrease rule<br>2: adaptive rule<br>3: probing rule<br>4: safeguarded Mehrotra predictor-corrector type rule<br>5: Mehrotra predictor-corrector type rule | 0 |
| delta | initial trust region radius scaling | 1.0e0 |
| feasible | 0: allow for infeasible iterates<br>1: feasible version of KNITRO | 0 |
| feasmodetol | tolerance for entering feasible mode | 1.0e-4 |
| feastol | feasibility termination tolerance (relative) | 1.0e-6 |
| feastolabs | feasibility termination tolerance (absolute) | 0.0e-0 |
| gradopt | gradient computation method:<br>1: use exact gradients<br>(only option available for AMPL interface) | 1 |
| hessopt | Hessian (Hessian-vector) computation method:<br>1: use exact Hessian<br>2: use dense quasi-Newton BFGS Hessian approximation<br>3: use dense quasi-Newton SR1 Hessian approximation<br>4: compute Hessian-vector products via finite differencing<br>5: compute exact Hessian-vector products (not available with AMPL interface)<br>6: use limited-memory BFGS Hessian approximation | 1 |
| honorbnds | 0: allow bounds to be violated during the optimization<br>1: enforce satisfaction of simple bounds always | 0 |
| initpt | 0: do not use any initial point strategies<br>1: use initial point strategy | 0 |
| islp | 0: do not treat the problem as an LP<br>1: treat the problem as an LP (if determined by AMPL) | 1 |
| isqp | 0: do not treat the problem as a QP<br>1: treat the problem as a QP (if determined by AMPL) | 1 |

Table 1: KNITRO user specifiable options for AMPL.

| OPTION | DESCRIPTION | DEFAULT |
|---|---|---|
| lpsolver | 1: use internal LP solver in active-set algorithm<br>2: use ILOG-CPLEX LP solver in active-set algorithm<br>(requires valid CPLEX license) | 1 |
| maxcgit | maximum allowable conjugate gradient (CG) iterations:<br>0: automatically set based on the problem size<br>$n$: maximum of $n$ CG iterations per minor iteration | 0 |
| maxit | maximum number of iterations before terminating | 10000 |
| maxtime | maximum CPU time in seconds before terminating | 1.0e8 |
| mu | initial barrier parameter value | 1.0e-1 |
| objrange | allowable objective function range | 1.0e20 |
| opttol | optimality termination tolerance (relative) | 1.0e-6 |
| opttolabs | optimality termination tolerance (absolute) | 0.0e-0 |
| outlev | printing output level:<br>0: no printing<br>1: just print summary information<br>2: print information every 10 major iterations<br>3: print information at each major iteration<br>4: print information at each major and minor iteration<br>5: also print final (primal) variables<br>6: also print final constraint values and Lagrange multipliers | 2 |
| outmode | where to direct output:<br>0: print to standard out (e.g., screen)<br>1: print to file 'knitro.out'<br>2: both screen and file 'knitro.out' | 0 |
| pivot | initial pivot threshold for matrix factorizations | 1.0e-8 |
| scale | 0: do not scale the problem<br>1: perform automatic scaling of functions | 1 |
| shiftinit | shift the initial point to satisfy the bounds | 1 |
| soc | 0: do not allow second order correction steps<br>1: selectively try second order correction steps<br>2: always try second order correction steps | 1 |
| xtol | stepsize termination tolerance | 1.0e-15 |

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

# 4 The KNITRO callable library

This section includes information on how to embed and call the KNITRO optimizer from inside a program.

## 4.1 Calling KNITRO from a C application

The KNITRO callable library can be used to solve an optimization problem from a `C` code through a sequence of three function calls which we summarize below:

- `KTR_new()`: get license and create KNITRO problem context pointer
- `KTR_solve()`: call the KNITRO optimizer to solve the problem
- `KTR_free()`: delete KNITRO problem context pointer and free memory and license

If the user wishes to free all the temporary memory, without releasing the licensing or destroying the context pointer, this can be done through a call to the function `KTR_free_tempwork()`.

The prototypes and a detailed summary of the functions used for creating a problem object, freeing temporary memory and destroying a problem object are described below. Later on we will describe in detail the primary KNITRO solver function `KTR_solve()`.

```
KTR_context_ptr KTR_new(int install_interrupt_handler)
```

This function should be called first. It returns an object that is used for every other call. The KNITRO license is reserved until the last `KTR_context_ptr` has `KTR_free()` called for it, or the program ends. If `install_interrupt_handler` is nonzero, KNITRO will be able to free a network license on abnormal program termination.

```
void KTR_free_tempwork(KTR_context_ptr kc)
```

This function will free work memory and close files associated with a context. The context pointer can still be used to solve another problem, and keeps all its settings. The context pointer still retains the license, if a floating license is used.

```
void KTR_free(KTR_context_ptr *kc_handle)
```

This function call will free the context pointer. You actually pass the address of your pointer so that KNITRO can clobber it to NULL. This helps to avoid mistakes. If this was the last context, the license is freed.

The KNITRO `C` API has two major modes of operation called "callback" and "reverse communication". With callback the user provides KNITRO with function pointers for it to evaluate the functions, gradients, and Hessian (or to insert other user-defined routines). With reverse communication, the KNITRO solve function `KTR_solve()` returns with a positive return value if it needs an evaluation, then is to be called again. See sections 9.7 and 9.8 for more details on these two modes of operations.

For the `C` interface the user must include the KNITRO header file knitro.h in the `C` source code which calls KNITRO. Programs using the KNITRO 3.1 API, will need to include knitro3.h which contains the depreciated declarations.

To use KNITRO the user must provide routines for evaluating the objective and constraint functions, the first derivatives (i.e., gradients), and optionally, the second derivatives (i.e., Hessian). The ability to provide exact first derivatives is essential for efficient and reliable performance. However, if the user is unable or unwilling to provide exact first derivatives, KNITRO provides routines in the file KNITROgrad.c which will compute approximate first derivatives using finite-differencing.

Exact Hessians (or second derivatives) are less important. However, the ability to provide exact second derivatives may often dramatically improve the performance of KNITRO. Packages like ADIFOR and ADOL-C can help in coding these routines.

In the example program provided with the distribution, routines for setting up the problem and for evaluating functions, gradients and the Hessian are provided in the file user_problem.c as listed below.

`sizes:` This routine sets the problem sizes.

`setup:` This routine provides data about the problem.

`evalfc:` A routine for evaluating the objective function (1.1$a$) and constraints (1.1$b$)-(1.1$c$).

`evalga:` A routine for evaluating the gradient of the objective function and the Jacobian matrix of the constraints in sparse form.

`evalhess/evalhessvec:` A routine for evaluating the Hessian of the Lagrangian function in sparse form or alternatively the Hessian-vector products.

Another file driverC.c provides example code for calling the above functions in conjunction with KNITRO. The function `evalhess` is only needed if the user wishes to provide exact Hessian computations and likewise the function `evalhessvec` is only needed if the user wishes to provide exact Hessian-vector products. Otherwise approximate Hessians or Hessian-vector products can be computed internally by KNITRO.

The C interface for KNITRO requires the user to define an optimization problem using the following general format.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) && \text{(4.3a)} \\ \text{subject to} \quad & cl \leq c(x) \leq cu && \text{(4.3b)} \\ & bl \leq x \leq bu. && \text{(4.3c)} \end{aligned}$$

**NOTE:** If constraint $i$ is an equality constraint, set $cl(i) = cu(i)$.

**NOTE:** KNITRO defines infinite upper and lower bounds using the constant `KTR_INFBOUND` set to `1.0e+20` in the header file `knitro.h`. Any bounds smaller in magnitude than `KTR_INFBOUND` will be considered finite and those that are equal to this value or larger in magnitude will be considered infinite.

Please refer to section 4.2 and to the files driverC.c and user_problem.c provided with the distribution for an example of how to specify the above functions and call the KNITRO solver function, `KTR_solve()`, to solve a user-defined problem in C. The file user_problem.c contains examples of the functions - `sizes`, `setup`, `evalfc`, `evalga`, `evalhess` and `evalhessvec` - while driverC.c is an example driver file for the C interface which calls these functions as well as the KNITRO solver function `KTR_solve()`.

The KNITRO solver is invoked via a call to the function `KTR_solve()` which has the following calling sequence:

```
int  KTR_solve  (KTR_context *kc,
                 double *f,
                 int ftype,
                 int n,
                 double *x,
                 double *bl,
                 double *bu,
                 double *fgrad,
                 int m,
                 double *c,
                 double *cl,
                 double *cu,
                 int *ctype,
                 int nnzj,
                 double *cjac,
                 int *indvar,
                 int *indfun,
                 double *lambda,
                 int nnzh,
                 double *hess,
                 int *hrow,
                 int *hcol,
                 double *vector,
                 void *user
)
```

The following arguments are passed to the KNITRO solver function `KTR_solve()`:

**Arguments:**

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

`double *f:` is a scalar that holds the value of the objective function at the current $x$.

`On initial entry: f` need not be set by the user.

`On exit: f` holds the current approximation to the optimal objective value. If the return value from `KTR_solve()` = `KTR_RC_EVALFC` or `KTR_RC_EVALX0` the user must evaluate `f` at the current value of `x` before re-entering `KTR_solve()`.

**NOTE**: Because of the reverse communication/interactive implementation of the KNITRO callable library (see section 9.7), the scalar argument `f` is passed by reference to `KTR_solve()` (using the `&` syntax).

`int ftype:` is a scalar that describes the type of the objective function.

`On initial entry: ftype` should be initialized by the user as follows:

0 if `f` is a nonlinear function or nothing is known about `f`

1 if `f` is a linear function

2 if `f` is a quadratic function

`On exit: ftype` is not altered by the function.

`int n:` is a scalar specifying the number of variables.

`On initial entry: n` must be set by the user to the number of variables (i.e., the length of $x$).

`On exit: n` is not altered by the function.

`double *x:` is an array of length `n`. It is the solution vector.

`On initial entry: x` must be set by the user to an initial estimate of the solution.

`On exit: x` contains the current approximation to the solution.

`double *bl:` is an array of length `n` specifying the lower bounds on `x`.

`On initial entry: bl`$[i]$ must be set by the user to the lower bound of the corresponding $i$-th variable `x`$[i]$. If there is no such bound, set it to be `-KTR_INFBOUND`. The value `KTR_INFBOUND` is defined in the header file `knitro.h`.

`On exit: bl` is not altered by the function.

`double *bu:` is an array of length `n` specifying the upper bounds on `x`.

On initial entry: bu[$i$] must be set by the user to the upper bound of the corresponding $i$-th variable x[$i$]. If there is no such bound, set it to be KTR_INFBOUND. The value KTR_INFBOUND is defined in the header file knitro.h.

On exit: bu is not altered by the function.

double *fgrad: is an array of length n specifying the gradient of f.

On initial entry: fgrad need not be set by the user.

On exit: fgrad holds the current approximation to the optimal objective gradient value. If the return value from KTR_solve() = KTR_RC_EVALGA or KTR_RC_EVALX0 the user must evaluate fgrad at the current value of x before re-entering KTR_solve().

int m: is a scalar specifying the number of constraints.

On initial entry: m must be set by the user to the number of general constraints (i.e., the length of $c(x)$).

On exit: m is not altered by the function.

double *c: is an array of length m. It holds the values of the general equality and inequality constraints at the current $x$. (It excludes fixed variables and simple bound constraints which are specified using bl and bu.)

On initial entry: c need not be set by the user.

On exit: c holds the current approximation to the optimal constraint values. If the return value from KTR_solve() = KTR_RC_EVALFC or KTR_RC_EVALX0 the user must evaluate c at the current value of x before re-entering KTR_solve().

double *cl: is an array of length m specifying the lower bounds on c.

On initial entry: cl[$i$] must be set by the user to the lower bound of the corresponding $i$-th constraint c[$i$]. If there is no such bound, set it to be -KTR_INFBOUND. The value KTR_INFBOUND is defined in the header file knitro.h. If the constraint is an equality constraint cl[$i$] should equal cu[$i$].

On exit: cl is not altered by the function.

double *cu: is an array of length m specifying the upper bounds on c.

On initial entry: cu[$i$] must be set by the user to the upper bound of the corresponding $i$-th constraint c[$i$]. If there is no such bound, set it to be KTR_INFBOUND. The value KTR_INFBOUND is defined in the header file knitro.h. If the constraint is an equality constraint cl[$i$] should equal cu[$i$].

`On exit:` `cu` is not altered by the function.

`int *ctype:` is an array of length `m` that describes the type of the constraint functions.

`On initial entry:` `ctype` should be initialized by the user as follows:

0 if `ctype`[$i$] is a nonlinear function or nothing is known about `c`[$i$]
1 if `ctype`[$i$] is a linear function
2 if `ctype`[$i$] is a quadratic function

`On exit:` `ctype` is not altered by the function.

`int nnzj:` is a scalar specifying the number of nonzero elements in the sparse constraint Jacobian.

`On initial entry:` `nnzj` must be set by the user to the number of nonzero elements in the sparse constraint Jacobian `cjac`.

`On exit:` `nnzj` is not altered by the function.

`double *cjac:` is an array of length `nnzj` which contains the nonzero elements of the constraint gradients. There is no restriction on the ordering of these elements.

`On initial entry:` `cjac` need not be set by the user.

`On exit:` `cjac` holds the current approximation to the optimal constraint gradients. If the return value from `KTR_solve()` = `KTR_RC_EVALGA` or `KTR_RC_EVALX0` the user must evaluate `cjac` at the current value of `x` before re-entering `KTR_solve()`.

`int *indvar:` is an array of length `nnzj`. It is the index of the variables. If `indvar`[$i$]=$j$, then `cjac`[$i$] refers to the $j$-th variable, where $j = 0..n-1$.

**NOTE**: C array indexing starts with index 0. Therefore, the $j$-th variable corresponds to C array element `x[j]`.

`On initial entry:` `indvar` should be specified on initial entry.

`On exit:` Once specified `indvar` is not altered by the function.

`int *indfun:` is an array of type of length `nnzj`. If `indfun`[$i$]=$k$, then `cjac`[$i$] refers to the $k$-th constraint, where $k = 0..m-1$.

**NOTE**: C array indexing starts with index 0. Therefore, the $k$-th constraint corresponds to C array element `c[k]`.

`On initial entry:` `indfun` should be specified on initial entry.

`On exit:` Once specified `indfun` is not altered by the function.

`indfun`[$i$] and `indvar`[$i$] determine the row numbers and the column numbers respectively of the nonzero constraint Jacobian elements specified in `cjac`[$i$].

**NOTE**: See section 4.6 for an example of how to specify the arrays `cjac`, `indvar` and `indfun` which hold the elements of the constraint Jacobian matrix in sparse form.

`double *lambda:` is an array of length `m+n` holding the Lagrange multipliers for the constraints `c` (4.3*b*) and bounds on the variables `x` (4.3*c*) respectively.

`On initial entry:` `lambda` need not be set by the user.

`On exit:` `lambda` contains the current approximation of the optimal Lagrange multiplier values. The first `m` components of `lambda` are the multipliers corresponding to the constraints specified in `c` while the last `n` components are the multipliers corresponding to the bounds on `x`.

`int nnzh:` is a scalar specifying the number of nonzero elements in the upper triangle of the sparse Hessian of the Lagrangian.

`On initial entry:` `nnzh` must be set by the user to the number of nonzero elements in the upper triangle of the Hessian of the Lagrangian function (including diagonal elements) if the user is providing exact Hessian (i.e., `KTR_PARAM_HESSOPT`=1).

`On exit:` `nnzh` is not altered by the function.

**NOTE:** If `KTR_PARAM_HESSOPT` $= 2 - 6$, then the Hessian of the Lagrangian is not explicitly stored and one should set `nnzh` $= 0$.

`double *hess:` is an array of length `nnzh` containing the nonzero elements of the upper triangle of the Hessian of the Lagrangian which is defined as

$$\nabla^2 f(x) + \sum_{i=0..m-1} \lambda_i \nabla^2 c(x)_i. \tag{4.4}$$

**Only the nonzero elements of the upper triangle are stored**.

`On initial entry:` `hess` need not be set by the user.

`On exit:` `hess` contains the current approximation of the optimal Hessian of the Lagrangian. If the return value from `KTR_solve()` = `KTR_RC_EVALH` (and `KTR_PARAM_HESSOPT=1`) the user must evaluate `hess` at the current values of `x` and `lambda` before re-entering `KTR_solve()`.

**NOTE:** This array should only be specified and allocated when `KTR_PARAM_HESSOPT = 1`.

`int *hrow:` is an array of length `nnzh`. `hrow`[$i$] stores the row number of the nonzero element `hess`[$i$]. (NOTE: Row numbers range from 0 to $n - 1$).

`On initial entry:` `hrow` should be specified on initial entry.

`On exit:` Once specified `hrow` is not altered by the function.

**NOTE:** This array should only be specified and allocated when `KTR_PARAM_HESSOPT = 1`.

`int *hcol:` is an array of length `nnzh`. `hcol`$[i]$ stores the column number of the nonzero element `hess`$[i]$. (NOTE: Column numbers range from 0 to $n-1$).

`On initial entry:` `hcol` should be specified on initial entry.

`On exit:` Once specified `hcol` is not altered by the function.

**NOTE:** This array should only be specified and allocated when `KTR_PARAM_HESSOPT = 1`.

**NOTE**: See section 4.6 for an example of how to specify the arrays `hess`, `hrow` and `hcol` which hold the elements of the Hessian of the Lagrangian matrix in sparse form.

`double *vector:` is an array of length `n` which is used for the Hessian-vector product option `KTR_PARAM_HESSOPT=5`.

`On initial entry:` `vector` need not be specified on the *initial* entry but may need to be set by the user on future calls to the `KTR_solve()` function.

`On exit:` If the return value from `KTR_solve()` = `KTR_RC_EVALH` and `KTR_PARAM_HESSOPT=5`, `vector` holds the vector which will be multiplied by the Hessian and on re-entry `vector` must hold the desired Hessian-vector product supplied by the user.

**NOTE:** This array should only be specified and allocated when `KTR_PARAM_HESSOPT = 5`.

`void *user:` is a pointer to a structure which a user can use to defined additional paramaters needed for a callback routine. This argument is only used in the callback feature. See section 9.8 for more details.

**Return Value:**

The return value of `KTR_solve()` specifies the exit code from the optimization and also keeps track of the reverse communication re-entry status. If the return value is 0 or negative, then KNITRO has finished the optimization and should be terminated. If the return value is strictly positive, then KNITRO is requesting that the user re-enter `KTR_solve()` after performing some task. Below is listed a description of the positive *re-entry* return values.

For a detailed description of all the possible non-positive *termination* values, see Appendix A:.

Re-entry values:

`KTR_RC_EVALFC` **(1)** Evaluate functions `f` and `c` and re-enter `KTR_solve()`.

`KTR_RC_EVALGA` **(2)** Evaluate gradients `fgrad` and `cjac` and re-enter `KTR_solve()`.

`KTR_RC_EVALH` **(3)** Evaluate Hessian `hess` or Hessian-vector product and re-enter `KTR_solve()`.

`KTR_RC_EVALXO` **(4)** Evaluate functions `f` and `c` and gradients `fgrad` and `cjac` and re-enter `KTR_solve()`.

`KTR_RC_NEWPOINT` **(6)** KNITRO has just computed a new solution estimate. The user may provide routine(s) if desired to perform some task (see section 9.7) before KNITRO begins a new major iteration.

## 4.2 Example

The following example demonstrates how to call the KNITRO solver using C to solve the problem:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & 1000 - x_1^2 - 2x_2^2 - x_3^2 - x_1x_2 - x_1x_3 && \text{(4.5a)} \\ \text{subject to} \quad & 8x_1 + 14x_2 + 7x_3 - 56 = 0 && \text{(4.5b)} \\ & x_1^2 + x_2^2 + x_3^2 - 25 \geq 0 && \text{(4.5c)} \\ & x_1, x_2, x_3 \geq 0 && \text{(4.5d)} \end{aligned}$$

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

A sample C program for solving the above problem using the KNITRO Interior/CG algorithm and the output from the optimization are contained in the following pages. This sample problem is included in the distribution.

### C driver program

```
/* ------------------------------------------------------------------ */
/*                KNITRO DRIVER FOR C/C++ INTERFACE                   */
/* ------------------------------------------------------------------ */

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

```
/* User-defined functions */

#ifdef __cplusplus
extern "C" {
#endif

void sizes(int *n, int *m, int *nnzj, int *nnzh, int hessopt);

void setup(int n, int m, int nnzj, int nnzh,
           double *x, double *bl, double *bu,
           int *ftype, int *ctype, double *cl, double *cu,
           int *indvar, int *indfun, int *hrow, int *hcol,
           int gradopt, int hessopt);

void evalfc(int n, int m, double *x, double *f, double *c);

void evalga(int n, int m, int nnzj, double *x, double *fgrad, double *cjac);

void evalhess(int n, double *x, double *lambda, double *hess);

void evalhessvec(int n, double *x, double *lambda,
                 double *vector, double *tmp);

#ifdef __cplusplus
}
#endif

main()
{
  /* Declare variables which are passed to function KTR_solve(). */
  int *indvar, *indfun, *hrow, *hcol, *ctype;
  double *x, *fgrad, *c, *cjac, *lambda, *hess, *bl, *bu, *cl, *cu, *vector;
  int n, m, nnzj, nnzh, ftype;
  double f;

  /* Variables used for gradient check */
  double *fgradfd, *cjacfd;
  double fderror;

  /* Declare other variables. */
  int i, j, k, errorflag, status;
  int hessopt, gradopt;
  double *tmp;
```

```
KTR_context *kc;

/* Create a new problem instance */
/* (install_interrupt_handler) */
kc = KTR_new(1);
if(kc == NULL) {
  fprintf (stderr, "Failure to get license.\n");
  exit(1);
}

/* Three redundant examples of setting a parameter */
errorflag = KTR_set_int_param_by_name(kc,"algorithm", KTR_ALG_IPCG);
if (errorflag) {
  fprintf (stderr, "Failure to set param. %d file=%s line=%d\n",
           errorflag,__FILE__,__LINE__);
  exit(1);
}
errorflag = KTR_set_int_param(kc,KTR_PARAM_ALGORITHM, KTR_ALG_IPCG);
if (errorflag) {
  fprintf (stderr, "Failure to set param. %d file=%s line=%d\n",
           errorflag,__FILE__,__LINE__);
  exit(1);
}
errorflag = KTR_set_char_param_by_name(kc,"algorithm", "cg");
if (errorflag) {
  fprintf (stderr, "Failure to set param. %d file=%s line=%d\n",
           errorflag,__FILE__,__LINE__);
  exit(1);
}

/* Here we read and write our whole config to a file, for easy */
/* runtime changing and debugging. */
/* (uncomment the lines below to read or write an options file) */
/*KTR_load_param_file(kc,"knitro.opt");*/
/*KTR_save_param_file(kc,"knitro.opt");*/

/* Get the Hessian option used */
errorflag = KTR_get_int_param(kc,KTR_PARAM_HESSOPT,&hessopt);
if (errorflag) {
  fprintf (stderr, "Failure to get param. %d file=%s line=%d\n",
           errorflag,__FILE__,__LINE__);
  exit(1);
```

```
}

/* Set n, m, nnzj, nnzh */
sizes(&n, &m, &nnzj, &nnzh, hessopt);

/* Allocate arrays that get passed to KTR_solve */
x         = (double *)malloc(n*sizeof(double));
fgrad     = (double *)malloc(n*sizeof(double));
c         = (double *)malloc(m*sizeof(double));
cjac      = (double *)malloc(nnzj*sizeof(double));
indvar    = (int *)malloc(nnzj*sizeof(int));
indfun    = (int *)malloc(nnzj*sizeof(int));
lambda    = (double *)malloc((m+n)*sizeof(double));
bl        = (double *)malloc(n*sizeof(double));
bu        = (double *)malloc(n*sizeof(double));
ctype     = (int *)malloc(m*sizeof(int));
cl        = (double *)malloc(m*sizeof(double));
cu        = (double *)malloc(m*sizeof(double));

/* Arrays only needed for exact Hessian */
if (hessopt == 1) {
  hess      = (double *)malloc(nnzh*sizeof(double));
  hrow      = (int *)malloc(nnzh*sizeof(int));
  hcol      = (int *)malloc(nnzh*sizeof(int));
}

/* Arrays only needed for exact Hessian-vector products */
if (hessopt == 5) {
  vector    = (double *)malloc(n*sizeof(double));
  tmp       = (double *)malloc(n*sizeof(double));
}

/* Arrays only needed if performing gradient check */
errorflag = KTR_get_int_param(kc,KTR_PARAM_GRADOPT,&gradopt);
if (errorflag) {
  fprintf (stderr, "Failure to get param. %d file=%s line=%d\n",
           errorflag,__FILE__,__LINE__);
  exit(1);
}
if (gradopt == 4 || gradopt == 5) {
  fgradfd   = (double *)malloc(n*sizeof(double));
  cjacfd    = (double *)malloc(nnzj*sizeof(double));
}
```

```
/* Initialize values which specify the problem */
setup(n, m, nnzj, nnzh, x, bl, bu, &ftype, ctype, cl, cu,
      indvar, indfun, hcol, hrow, gradopt, hessopt);

status = KTR_RC_INITIAL;

/* This is the main loop where we keep calling KTR_solve() until it */
/* stops returning a positive number. */
/* Positive returns mean we need to evaluate our objective function, */
/* gradient, or hessian again. */
do {  /* loop until done (i.e, staus <=0) */

  /* Evaluate function and constraint values */
  if ((status == KTR_RC_EVALFC) || (status == KTR_RC_EVALX0)){
    evalfc(n, m, x, &f, c);
  }

  /* Evaluate objective and constraint gradients */
  if ((status == KTR_RC_EVALGA) || (status == KTR_RC_EVALX0)){
    switch (gradopt)
      {
      case 1:
        /* User-supplied exact gradient. */
        evalga(n, m, nnzj, x, fgrad, cjac);
        break;
      case 2: case 3:
        /* Compute finite difference gradient */
        KTR_gradfd(n, m, x, fgrad, nnzj, cjac, indvar, indfun,
                   f, c, gradopt);
        break;
      case 4: case 5:
        /* Perform gradient check using finite difference gradient */
        evalga(n, m, nnzj, x, fgrad, cjac);
        KTR_gradfd(n, m, x, fgradfd, nnzj, cjacfd, indvar, indfun,
                   f, c, gradopt);
        fderror = KTR_gradcheck(n, fgrad, fgradfd, nnzj, cjac, cjacfd);
        printf("Finite difference gradient error = %e\n", fderror);
        break;
      default: printf("ERROR: Bad value for gradopt. (%d)\n", gradopt);
        exit(1);
      }
  }
```

```
  /* Evaluate user-supplied Lagrange Hessian (or Hessian-vector product) */
  if (status == KTR_RC_EVALH){
    if (hessopt == 1)
      evalhess(n, x, lambda, hess);
    else if (hessopt == 5)
      evalhessvec(n, x, lambda, vector, tmp);
  }

  /* We now have a new solution estimate.  If newpoint feature
     enabled, provide newpoint routine(s) below. */
  if (status == KTR_RC_NEWPOINT) {
    /* The user may insert newpoint routines here if desired. */
  }

  /* Call KNITRO solver routine */
  status = KTR_solve(kc, &f, ftype, n, x, bl, bu, fgrad, m, c,
                     cl, cu, ctype, nnzj, cjac, indvar, indfun,
                     lambda, nnzh, hess, hrow, hcol, vector, NULL);

} while (status > 0); /* KNITRO done if status <=  0. */

/* Optimization finished */

/* Delete problem instance */
KTR_free(&kc);

free(x);
free(fgrad);
free(c);
free(cjac);
free(indvar);
free(indfun);
free(lambda);
free(bl);
free(bu);
free(ctype);
free(cl);
free(cu);
if (hessopt == 1) {
  free(hess);
  free(hrow);
  free(hcol);
```

```
  }
  if (hessopt == 5) {
    free(vector);
    free(tmp);
  }
  if (gradopt == 4 || gradopt == 5) {
    free(fgradfd);
    free(cjacfd);
  }
  return 0;
}
```

**User-defined C Functions**

```
#include "knitro.h"

void sizes(int *n, int *m, int *nnzj, int *nnzh, int hessopt)
{
  /* Set n, m, nnzj, nnzh */
  *n = 3;
  *m = 2;
  *nnzj = 6;
  if (hessopt == 1) {
    /* User-supplied exact Hessian */
    *nnzh = 5;
  } else {
    /* No need to specify or allocate hess,hrow,hcol */
    *nnzh = 0;
  }
  return;
}

void setup(int n, int m, int nnzj, int nnzh,
           double *x, double *bl, double *bu,
           int *ftype, int *ctype, double *cl, double *cu,
           int *indvar, int *indfun, int *hrow, int *hcol,
           int gradopt, int hessopt)
{
  /* Function that gives data on the problem. */

  int i;
  double zero, two;

  zero   = 0.0e0;
  two    = 2.0e0;

  /* Initial point. */
  for (i=0; i<n; i++) {
    x[i] = two;
  }

  /* Bounds on the variables. */
  for (i=0; i<n; i++) {
    bl[i] = zero;
    bu[i] = KTR_INFBOUND;
```

```
}

/* Indicate type of objective and constraints
 0: general nonlinear or unknown
 1: linear
 2: quadratic */

*ftype = 2;
ctype[0] = 1;
ctype[1] = 2;

/* Bounds on the constraints ( cl <= c(x) <= cu ). */

cl[0] = zero;
cu[0] = zero;

cl[1] = zero;
cu[1] = KTR_INFBOUND;

/* Specify sparsity info for constraint Jacobian */

indvar[0] = 0;
indfun[0] = 0;

indvar[1] = 1;
indfun[1] = 0;

indvar[2] = 2;
indfun[2] = 0;

indvar[3] = 0;
indfun[3] = 1;

indvar[4] = 1;
indfun[4] = 1;

indvar[5] = 2;
indfun[5] = 1;

/* Specify sparsity info for Hessian (if using exact Hessian)
 *
 * NOTE: Only the NONZEROS of the UPPER TRIANGLE (including
 *       diagonal) of this matrix should be stored.
```

```
   */

  if (hessopt == 1) {
    hrow[0] = 0;
    hcol[0] = 0;

    hrow[1] = 0;
    hcol[1] = 1;

    hrow[2] = 0;
    hcol[2] = 2;

    hrow[3] = 1;
    hcol[3] = 1;

    hrow[4] = 2;
    hcol[4] = 2;
  }

}
/*******************************************************/
void evalfc(int n, int m, double *x, double *f, double *c)
{
  /* Objective function and constraint values for the
   * given problem.
   */

  *f = 1.0e3 - x[0]*x[0] - 2.0e0*x[1]*x[1] - x[2]*x[2]
    - x[0]*x[1] - x[0]*x[2];

  /* Equality constraint. */
  c[0] = 8.0e0*x[0] + 14.0e0*x[1] + 7.0e0*x[2] - 56.0e0;

  /* Inequality constraint. */
  c[1] = x[0]*x[0] + x[1]*x[1] + x[2]*x[2] - 25.0e0;
}
/*******************************************************/
void evalga(int n, int m, int nnzj,
            double *x, double *fgrad, double *cjac)
{
  /* Routine for computing gradient values.  Objective gradient
   * "fgrad" specified in dense array of size n, constraint gradients
   * "cjac" in sparse form.
```

```
   *
   * NOTE: Only NONZERO constraint gradient elements should be specified.
   */

  /* Gradient of the objective function (dense). */

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

  /* Gradient of the first constraint, c[0]. */

  cjac[0]   = 8.0e0;
  cjac[1]   = 14.0e0;
  cjac[2]   = 7.0e0;

  /* Gradient of the second constraint, c[1]. */

  cjac[3]   = 2.0e0*x[0];
  cjac[4]   = 2.0e0*x[1];
  cjac[5]   = 2.0e0*x[2];

}
/*****************************************************/
void evalhess(int n, double *x, double *lambda, double *hess)
{
  /*
   * Compute the Hessian of the Lagrangian.
   *
   * NOTE: Only the NONZEROS of the UPPER TRIANGLE (including
   *       diagonal) of this matrix should be stored.
   */

  hess[0] = 2.0e0*(lambda[1]-1.0e0);
  hess[1] = -1.0e0;
  hess[2] = -1.0e0;
  hess[3] = 2.0e0*(lambda[1]-2.0e0);
  hess[4] = 2.0e0*(lambda[1]-1.0e0);

}
/*****************************************************/
void evalhessvec(int n, double *x, double *lambda,
                 double *vector, double *tmp)
```

```
{
  /*
   * Compute the Hessian of the Lagrangian times "vector"
   * and store the result in "vector".
   */

  int i;

  tmp[0] = 2.0e0*(lambda[1]-1.0e0)*vector[0] - vector[1] - vector[2];
  tmp[1] = -vector[0] + 2.0e0*(lambda[1]-2.0e0)*vector[1];
  tmp[2] = -vector[0] + 2.0e0*(lambda[1]-1.0e0)*vector[2];

  for (i=0; i<n; i++) {
    vector[i] = tmp[i];
  }

}
/*************************************************/
```

**Output**

```
=====================================
             KNITRO 4.0.0
       Ziena Optimization, Inc.
       website:  www.ziena.com
       email:   info@ziena.com
=====================================

algorithm:   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

  Iter      Objective      Feas err    Opt err     ||Step||    CG its
--------  -------------   ---------   ---------   ---------   --------
       0   9.760000e+02   1.300e+01
       6   9.360004e+02   2.384e-15   1.597e-05   4.017e-03          1

EXIT: LOCALLY OPTIMAL SOLUTION FOUND.

Final Statistics
----------------
Final objective value               =   9.36000414931828e+02
Final feasibility error (abs / rel) =   2.38e-15 / 1.83e-16
Final optimality error  (abs / rel) =   1.60e-05 / 9.98e-07
# of iterations (major / minor)     =          6 /        6
# of function evaluations           =          7
# of gradient evaluations           =          7
# of Hessian evaluations            =          6
Total program time (sec)            =       0.00

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

### 4.3 Calling KNITRO from a C++ application

Calling KNITRO from a C++ application requires little or no modification of the C example in the previous section. The KNITRO header file `knitro.h` already includes the necessary `extern C` statements (if called from a C++ code) for the KNITRO callable functions.

### 4.4 Calling KNITRO from a Fortran application

To use KNITRO the user must provide routines for evaluating the objective and constraint functions, the first derivatives (i.e., gradients), and optionally, the second derivatives (i.e., Hessian). The ability to provide exact first derivatives is essential for efficient and reliable performance. However, if the user is unable or unwilling to provide exact first derivatives, KNITRO provides Fortran routines in the file `KNITROgradF.f` which will compute approximate first derivatives using finite-differencing.

Exact Hessians (or second derivatives) are less important. However, the ability to provide exact second derivatives may often dramatically improve the performance of KNITRO. Packages like ADIFOR and ADOL-C can help in coding these routines.

In the example program provided with the distribution, routines for setting up the problem and for evaluating functions, gradients and the Hessian are provided in the file `user_problem.f` as listed below.

`sizes:` This routine sets the problem sizes.

`setup:` This routine provides data about the problem.

`evalfc:` A routine for evaluating the objective function (1.1$a$) and constraints (1.1$b$)-(1.1$c$).

`evalga:` A routine for evaluating the gradient of the objective function and the Jacobian matrix of the constraints in sparse form.

`evalhess/evalhessvec:` A routine for evaluating the Hessian of the Lagrangian function in sparse form or alternatively the Hessian-vector products.

The subroutine `evalhess` is only needed if the user wishes to provide exact Hessian computations and likewise the subroutine `evalhessvec` is only needed if the user wishes to provide exact Hessian-vector products. Otherwise approximate Hessians or Hessian-vector products can be computed internally by KNITRO.

The Fortran interface for KNITRO requires the user to define an optimization problem using the following general format.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) && \text{(4.6a)} \\ \text{subject to} \quad & cl \leq c(x) \leq cu && \text{(4.6b)} \\ & bl \leq x \leq bu. && \text{(4.6c)} \end{aligned}$$

**NOTE:** If constraint $i$ is an equality constraint, set $cl(i) = cu(i)$.

**NOTE:** KNITRO defines infinite upper and lower bounds using the parameter `KTR_INFBOUND` which is defined in the header file `knitro.h` to have the value `1.0e+20`. When using the Fortran interface, be sure that bounds which are meant to be infinite are set at least as large as `KTR_INFBOUND`. Any bounds smaller in magnitude than `KTR_INFBOUND` will be considered finite and those that are equal to this value or larger in magnitude will be considered infinite.

Please refer to the files driverFortran.f, user_problem.f and fortranglue.c provided with the distribution for an example of how to specify the above subroutines and use KNITRO to solve a user-defined problem in Fortran. The file user_problem.f contains examples of the subroutines - `sizes`, `setup`, `evalfc`, `evalga`, `evalhess` and `evalhessvec`. The file driverFortran.f is an example driver file for the Fortran interface which calls an interface/gateway routine fortranglue.c which in turn calls the KNITRO solver function `KTR_solve()` described in section 4.1.

Most of the information and instruction on calling the KNITRO solver from a Fortran program is the same as for the C interface (see section 4.1) so that information will not be repeated here.

The following exceptions apply:

- C variables of type `int` should be specified as `INTEGER` in Fortran.
- C variables of type `double` should be specified as `DOUBLE PRECISION` in Fortran.
- Fortran arrays start at index value 1, whereas C arrays start with index value 0. Therefore, one must increment the array indices (*but not the array values*) by 1 from the values given in section 4.1 when applied to a Fortran program.

## 4.5 Building KNITRO with a Microsoft Visual C Developer Studio Project

KNITRO comes with an example makefile for building programs on Microsoft operating systems. This can be done using the `nmake` command.

Alternatively, people may wish to use a Visual Project of Microsoft Developer Studio which is an Integrated Development Environment (IDE) that includes editor, documentation, and compiler in one unified interface. Users must create their own project, but the steps are documented below.

1. Start Microsoft Developer Studio.
2. Select the File->New... menu.
3. Choose the appropriate project type. Since we are going to use the example C program which calls `printf()`, we should choose "Win32 Console Application". Give the project a name and notice the full path in the location. Click OK.
4. We want "an empty project" and click Finish.
5. Outside of Developer Studio. Copy the three source files (`driverC.c`, `KNITROgrad.c`, and `user_problem.c` ) from C:\Program Files\knitro-4.0\C\ into the project folder.
6. Back in Developer Studio add those three files to the project with the Project->Add-to-Project->Files... menu.
7. Again with the Project->Add-to-Project->Files... menu, add C:\Program Files\knitro-4.0\include\knitro.lib
8. Select the Project->Settings... menu.
9. In the upper left drop-down change "Win32 Debug" to "All Configurations". On the C/C++ tab select the Category Preprocessor and add C:\Program Files\knitro-4.0\include to the "Additional include directories". Click OK.
10. Select the Build->Execute menu.

## 4.6 Specifying the Jacobian and Hessian matrices in sparse form

An important issue in using the KNITRO callable library is the ability of the user to specify the Jacobian matrix of the constraints and the Hessian 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

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & x_0 + x_1 x_2^3 && (4.7a)\\ \text{subject to} \quad & \cos(x_0) = 0.5 && (4.7b)\\ & 3 \le x_0^2 + x_1^2 \le 8 && (4.7c)\\ & x_0 + x_1 + x_2 \le 10 && (4.7d)\\ & x_0, x_1, x_2 \ge 1. && (4.7e) \end{aligned}$$

Referring to (4.3), we have

$$\begin{aligned} f(x) &= x_0 + x_1 x_2^3 && (4.8)\\ c_0(x) &= \cos(x_0) && (4.9)\\ c_1(x) &= x_0^2 + x_1^2 && (4.10)\\ c_2(x) &= x_0 + x_1 + x_2. && (4.11)\\ & && (4.12) \end{aligned}$$

**Computing the Sparse Jacobian Matrix**

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

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

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

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

In KNITRO, the array `fgrad` stores all of the elements of $\nabla f(x)$, while the arrays `cjac`, `indfun`, and `indvar` store information concerning *only the nonzero* elements of $J(x)$. The array `cjac` stores the nonzero values in $J(x)$ evaluated at the current solution estimate $x$, `indfun` stores the function (or row) indices corresponding to these values and `indvar` stores the variable (or column) indices corresponding to these values. There is no restrictions 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 would be specified as follows in column-wise order.

```
cjac[0] = -sin(x[0]);  indfun[0] = 0; indvar[0] = 0;
cjac[1] = 2*x[0];      indfun[1] = 1; indvar[1] = 0;
```

```
cjac[2] = 1;          indfun[2] = 2; indvar[2] = 0;
cjac[3] = 2*x[1];     indfun[3] = 1; indvar[3] = 1;
cjac[4] = 1;          indfun[4] = 2; indvar[4] = 1;
cjac[5] = 1;          indfun[5] = 2; indvar[5] = 2;
```

As is evident, the values of `cjac` depend on the value of $x$ and may change while the values of `indfun` and `indvar` are constant.

### Computing the Sparse Hessian Matrix

The Hessian of the Lagrangian matrix is defined as

$$H(x,\lambda) = \nabla^2 f(x) + \sum_{i=0..m-1} \lambda_i \nabla^2 c(x)_i, \tag{4.13}$$

where $\lambda$ is the vector of Lagrange multipliers (dual variables). For the example defined by problem (4.7), The Hessians (second derivatives) of the objective and constraint functions are given by

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

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

By 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, we only store 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 `hrow` and `hcol` 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`, `hrow` and `hcol` would look as follows.

```
hess[0] = -lambda[0]*cos(x[0]) + 2*lambda[1];  hrow[0] = 0; hcol[0] = 0;
hess[1] = 2*lambda[1];                         hrow[1] = 1; hcol[1] = 1;
hess[2] = 3*x[2]*x[2];                         hrow[2] = 1; hcol[2] = 2;
hess[3] = 6*x[1]*x[2];                         hrow[3] = 2; hcol[3] = 2;
```

# 5 User options in KNITRO

## 5.1 Description of KNITRO user options

The following user options are available in KNITRO 4.0.

The integer valued options are:

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

- `0`: KNITRO will automatically try to choose the best algorithm based on the problem characteristics.
- `1`: KNITRO will use the Interior/Direct algorithm.
- `2`: KNITRO will use the Interior/CG algorithm.
- `3`: KNITRO will use the Active algorithm.

*Default value*: `0`

`KTR_PARAM_BARRULE`: indicates which strategy to use for modifying the barrier parameter in the interior point code. Some strategies are only available for the Interior/Direct algorithm. (see section 8).

- `0 (AUTOMATIC)`: KNITRO will automatically choose the rule for updating the barrier parameter.
- `1 (MONOTONE)`: KNITRO will monotonically decrease the barrier parameter.
- `2 (ADAPTIVE)`: KNITRO uses an adaptive rule based on the complementarity gap to determine the value of the barrier parameter at every iteration.
- `3 (PROBING)`: KNITRO uses a probing (affine-scaling) step to dynamically determine the barrier parameter value at each iteration.
- `4 (DAMPMPC)`: KNITRO uses a Mehrotra predictor-corrector type rule to determine the barrier parameter with safeguards on the corrector step.
- `5 (FULLMPC)`: KNITRO uses a Mehrotra predictor-corrector type rule to determine the barrier parameter without safeguards on the corrector step.

*Default value*: `0`

**NOTE**: Only strategies 0-2 are available for the Interior/CG algorithm. All strategies are available for the Interior/Direct algorithm. The last two strategies are typically recommended for linear programs or convex quadratic programs. This parameter is not applicable to the Active algorithm.

`KTR_PARAM_NEWPOINT`: specifies whether or not the new-point feature is enabled. If enabled, KNITRO returns control to the driver level with the return value from `KTR_solve()` = `KTR_RC_NEWPOINT` after a new solution estimate has been obtained and quantities have been updated (see section 9.7).

- `0`: KNITRO will *not* return to the driver level after completing a successful iteration.
- `1`: KNITRO will return to the driver level with the return value from `KTR_solve()` = `KTR_RC_NEWPOINT` after completing a successful iteration.

*Default value*: `0`

`KTR_PARAM_FEASIBLE`: indicates whether or not to use the feasible version of KNITRO.

- `0`: Iterates may be infeasible.
- `1`: Given an initial point which *sufficiently* satisfies all *inequality* constraints as defined by,

$$cl + tol \leq c(x) \leq cu - tol \tag{5.14}$$

(for $cl \neq cu$), the feasible version of KNITRO ensures that all subsequent solution estimates strictly satisfy the *inequality* constraints. However, the iterates may not be feasible with respect to the *equality* constraints. The tolerance $tol > 0$ in (5.14) for determining when the feasible mode is active is determined by the double precision parameter `KTR_PARAM_FEASMODETOL` described below. This tolerance (i.e. `KTR_PARAM_FEASMODETOL`) must be strictly positive. That is, in order to enter feasible mode, the point given to KNITRO must be strictly feasible with respect to the inequality constraints.

If the initial point is infeasible (or not sufficiently feasible according to (5.14)) with respect to the *inequality* constraints, then KNITRO will run the infeasible version until a point is obtained which sufficiently satisfies all the *inequality* constraints. At this point it will switch to feasible mode.

*Default value*: `0`

**NOTE**: This option can be used only when `KTR_PARAM_ALG=2`. See section 9.3 for more details.

`KTR_PARAM_GRADOPT`: specifies how to compute the gradients of the objective and constraint functions, and whether to perform a gradient check.

- `1`: user will provide a routine for computing the exact gradients

`2:` gradients computed by forward finite-differences

`3:` gradients computed by central finite differences

`4:` gradients provided by user checked with forward finite differences

`5:` gradients provided by user checked with 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. For more information on these options see section 9.1.

`KTR_PARAM_HESSOPT:` specifies how to compute the (approximate) Hessian of the Lagrangian.

`1:` user will provide a routine for computing the exact Hessian

`2:` KNITRO will compute a (dense) quasi-Newton BFGS Hessian

`3:` KNITRO will compute a (dense) quasi-Newton SR1 Hessian

`4:` KNITRO will compute Hessian-vector products using finite-differences

`5:` user will provide a routine to compute the Hessian-vector products

`6:` KNITRO will compute a limited-memory quasi-Newton BFGS Hessian

*Default value*: `1`

**NOTE**: If exact Hessians (or exact Hessian-vector products) cannot be provided by the user but exact gradients are provided and are not too expensive to compute, option 4 above is typically recommended. The finite-difference Hessian-vector option is comparable in terms of robustness to the exact Hessian option (*assuming exact gradients are provided*) and typically not too much slower in terms of time if gradient evaluations are not the dominant cost.

However, if exact gradients cannot be provided (i.e. finite-differences are used for the first derivatives), or gradient evaluations are expensive, it is recommended to use one of the quasi-Newton options, in the event that the exact Hessian is not available. Options 2 and 3 are only recommended for small problems ($n < 1000$) since they require working with a dense Hessian approximation. Option 6 should be used in the large-scale case.

**NOTE**: Options `KTR_PARAM_HESSOPT=4` and `KTR_PARAM_HESSOPT=5` are not available when `KTR_PARAM_ALG=1`. See section 9.2 for more detail on second derivative options.

`KTR_PARAM_HONORBNDS`: indicates whether or not to enforce satisfaction of the simple bounds (4.3*c*) throughout the optimization (see section 9.4).

- `0`: KNITRO does not enforce that the bounds on the variables are satisfied at intermediate iterates.
- `1`: KNITRO enforces that the initial point and all subsequent solution estimates satisfy the bounds on the variables (4.3*c*).

*Default value*: `0`

`KTR_PARAM_INITPT`: indicates whether an initial point strategy is used.

- `0`: No initial point strategy is employed.
- `1`: Initial values for the variables are computed. This option may be recommended when an initial point is not provided by the user, as is typically the case in linear and quadratic programming problems.

*Default value*: `0`

`KTR_PARAM_ISLP`: indicates whether or not the problem is a linear program.

- `0`: KNITRO will not recognize the problem as a linear program.
- `1`: KNITRO will assume the problem is a linear program and may perform some specializations.

*Default value*: `0`

`KTR_PARAM_ISQP`: indicates whether or not the problem is a quadratic program.

- `0`: KNITRO will not recognize the problem as a quadratic program.
- `1`: KNITRO will assume the problem is a quadratic program and may perform some specializations.

*Default value*: `0`

`KTR_PARAM_LPSOLVER`: indicates which linear programming simplex solver the KNITRO active-set algorithm uses to solve the LP subproblems.

- `1`: KNITRO uses an internal LP solver.
- `2`: KNITRO uses ILOG-CPLEX provided the user has a valid CPLEX license. The CPLEX shared object library or dll must reside in the current working directory or a directory specified in the library load path in order to be run-time loadable. By default, if this option is selected, KNITRO will look for either the shared object libraries or dlls for CPLEX 8.0 or CPLEX 9.0. If you would like KNITRO to look for a different CPLEX library,

this can be specified using the KTR_set_char_param_by_name() function, using the character parameter cplexlibname. For example, if you wanted to specifically tell KNITRO to look for the library cplex90.dll this could be done through the command

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

*Default value*: 1

KTR_PARAM_MAXCGIT: Determines the maximum allowable number of inner conjugate gradient (CG) iterations per KNITRO minor iteration.

- 0: KNITRO automatically determines an upper bound on the number of allowable CG iterations based on the problem size.
- $n$: At most $n$ CG iterations may be performed during one KNITRO minor iteration, where $n > 0$.

*Default value*: 0

KTR_PARAM_MAXIT: specifies the maximum number of major iterations before termination.

*Default value*: 10000

KTR_PARAM_OUTLEV: controls the level of output.

- 0: printing of all output is suppressed
- 1: only summary information is printed
- 2: information every 10 major iterations is printed where a major iteration is defined by a new solution estimate
- 3: information at each major iteration is printed
- 4: information is printed at each major and minor iteration where a minor iteration is defined by a trial iterate
- 5: in addition, the values of the solution vector x are printed
- 6: in addition, the values of the constraints c and Lagrange multipliers lambda are printed

*Default value*: 2

KTR_PARAM_OUTMODE: specifies where to direct the output.

- 0: output is directed to standard out (e.g., screen)
- 1: output is sent to a file named knitro.out

`2:` output is directed to both the screen and file knitro.out

*Default value*: `0`

`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 scaling is performed.

`1:` The objective function and constraints may be scaled.

*Default value*: `1`

`KTR_PARAM_SHIFTINIT:` Determines whether or not the interior-point algorithm in KNITRO shifts the given initial point to satisfy the variable bounds (4.3*c*).

`0:` KNITRO will not shift the given initial point to satisfy the variable bounds before starting the optimization.

`1:` KNITRO will shift the given initial point.

*Default value*: `1`

`KTR_PARAM_SOC:` indicates whether or not to use the second order correction (SOC) option. A second order correction may be beneficial for problems with highly nonlinear constraints.

`0:` No second order correction steps are attempted.

`1:` Second order correction steps may be attempted on some iterations.

`2:` Second order correction steps are always attempted if the original step is rejected and there are nonlinear constraints.

*Default value*: `1`

The double precision valued options are:

`KTR_PARAM_DELTA:` specifies the initial trust region radius scaling factor used to determine the initial trust region size.

*Default value*: `1.0e0`

`KTR_PARAM_FEASMODETOL:` specifies the tolerance in (5.14) by which the iterate must be feasible with respect to the inequality constraints before the feasible mode becomes active. This option is only relevant when `KTR_PARAM_FEASIBLE`=1.

*Default value*: `1.0e-4`

`KTR_PARAM_FEASTOL`: specifies the final relative stopping tolerance for the feasibility error. Smaller values of `KTR_PARAM_FEASTOL` result in a higher degree of accuracy in the solution with respect to feasibility.

*Default value*: `1.0e-6`

`KTR_PARAM_FEASTOLABS`: specifies the final absolute stopping tolerance for the feasibility error. Smaller values of `KTR_PARAM_FEASTOLABS` result in a higher degree of accuracy in the solution with respect to feasibility.

*Default value*: `0.0e0`

**NOTE**: For more information on the stopping test used in KNITRO see section 6.

`KTR_PARAM_MAXTIME`: specifies, in seconds, the maximum allowable time before termination.

*Default value*: `1.0e8`

`KTR_PARAM_MU`: specifies the initial barrier parameter value for the interior-point algorithms.

*Default value*: `1.0e-1`

`KTR_PARAM_OBJRANGE`: determines the allowable range of values for the objective function for determining unboundedness. If the magnitude of the objective function is greater than `KTR_PARAM_OBJRANGE` and the iterate is feasible, then the problem is determined to be unbounded.

*Default value*: `1.0e20`

`KTR_PARAM_OPTTOL`: specifies the final relative stopping tolerance for the KKT (optimality) error. Smaller values of `KTR_PARAM_OPTTOL` result in a higher degree of accuracy in the solution with respect to optimality.

*Default value*: `1.0e-6`

`KTR_PARAM_OPTTOLABS`: specifies the final absolute stopping tolerance for the KKT (optimality) error. Smaller values of `KTR_PARAM_OPTTOLABS` result in a higher degree of accuracy in the solution with respect to optimality.

*Default value*: `0.0e0`

**NOTE**: For more information on the stopping test used in KNITRO see section 6.

`KTR_PARAM_PIVOT`: specifies the initial pivot threshold used in the factorization routine. The value should be in the range [0 0.5] with higher values resulting in more pivoting (more stable factorization). 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`

KTR_PARAM_XTOL: The optimization will terminate when the relative change in the solution estimate is less than KTR_PARAM_XTOL. If using an interior-point 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 certain parameters without needing to recompile the code when using the KNITRO callable library. (This file has no effect when using the AMPL interface to KNITRO.)

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, one could use the following options file. In this example, let's call the options file knitro.opt (although any name will do):

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

In order for the options file to be read by KNITRO the following function call must be specified in the driver:

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

*Example:*

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

Likewise, the options used in a given optimization may be written to a file called knitro.opt through the function call:

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

*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 C and Fortran directories. Note that this file is only provided as a sample: it is not read by KNITRO (unless the user specifies the function call for reading this file).

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. Optional string values for many of the options are indicated in the example knitro.opt file provided with the distribution.

## 5.3 Setting options through function calls

The functions for setting user parameters have the form:

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

for setting integer valued parameters or

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

for setting double precision valued parameters.

*Example:*

The `The Interior/CG` algorithm can be chosen through the following function call.

```
status = KTR_set_int_param(kc,KTR_PARAM_ALG, 2);
```

Similarly, the optimality tolerance can be set through the function call:

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

**NOTE**: User parameters should only be set at the very beginning of the optimization before the call to `KTR_solve()` and should not be modified at all during the course of the optimization.

# 6 KNITRO Termination Test and Optimality

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

$$\nabla_x \mathcal{L}(x,\lambda) = \nabla f(x) + \sum_{i\in\mathcal{E}} \lambda_i \nabla h_i(x) + \sum_{i\in\mathcal{I}} \lambda_i \nabla g_i(x) = 0 \tag{6.15}$$

$$\lambda_i g_i(x) = 0, \quad i \in \mathcal{I} \tag{6.16}$$

$$h_i(x) = 0, \quad i \in \mathcal{E} \tag{6.17}$$

$$g_i(x) \leq 0, \quad i \in \mathcal{I} \tag{6.18}$$

$$\lambda_i \geq 0, \quad i \in \mathcal{I} \tag{6.19}$$

where $\mathcal{E}$ and $\mathcal{I}$ represent the sets of indices corresponding to the equality constraints and inequality constraints respectively, and $\lambda_i$ 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 (6.17), (6.18), i.e.,

$$\text{Feas err} = \max_{i\in\mathcal{E}\cup\mathcal{I}}(0, |h_i(x^k)|, g_i(x^k)), \tag{6.20}$$

while the optimality error (`Opt err`) is defined as the maximum violation of the first two conditions (6.15), (6.16),

$$\text{Opt err} = \max_{i\in\mathcal{I}}(\|\nabla_x \mathcal{L}(x^k,\lambda^k)\|_\infty, |\lambda_i g_i(x^k)|, |\lambda_i|, |g_i(x^k)|). \tag{6.21}$$

The last optimality condition (6.19) is enforced explicitly throughout the optimization. In order to take into account problem scaling in the termination test, the following scaling factors are defined

$$\tau_1 = \max(1, |h_i(x^0)|, g_i(x^0)), \tag{6.22}$$

$$\tau_2 = \max(1, \|\nabla f(x^k)\|_\infty), \tag{6.23}$$

where $x^0$ represents the initial point.

For unconstrained problems, the scaling (6.23) 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

$$\tau_2 = \max(1, \min(|f(x^k)|, \|\nabla f(x^0)\|_\infty)), \tag{6.24}$$

in place of (6.23).

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

$$\text{Feas err} \leq \max(\tau_1 * \texttt{KTR\_PARAM\_FEASTOL}, \texttt{KTR\_PARAM\_FEASTOLABS}) \tag{6.25}$$

$$\text{Opt err} \leq \max(\tau_2 * \texttt{KTR\_PARAM\_OPTTOL}, \texttt{KTR\_PARAM\_OPTTOLABS}) \tag{6.26}$$

where `KTR_PARAM_FEASTOL`, `KTR_PARAM_OPTTOL`, `KTR_PARAM_FEASTOLABS` and `KTR_PARAM_OPTTOLABS` are user-defined 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 purely scaled stopping test (which is the recommended default option) by setting `KTR_PARAM_FEASTOLABS` and `KTR_PARAM_OPTTOLABS` equal to `0.0e0`. Likewise, an absolute stopping test can be enforced by setting `KTR_PARAM_FEASTOL` and `KTR_PARAM_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 for an unbounded problem. For example, if $\tau_2 \rightarrow \infty$ while the optimality error (6.21) stays bounded, condition (6.26) will eventually be satisfied for some `KTR_PARAM_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

If KTR_PARAM_OUTLEV=0 then all printing of output is suppressed. For the default printing output level (KTR_PARAM_OUTLEV=2) the following information is given:

**Nondefault Options**:
This output lists all user options (see section 5) which are different from their default values. If nothing is listed in this section then all user options are set to their default values.

**Problem Characteristics**:
The output begins with a description of the problem characteristics.

**Iteration Information**:
A major iteration, in the context of KNITRO, 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). After the problem characteristic information there are columns of data reflecting information about each iteration of the run. Below is a description of the values contained under each column header:

`Iter:` Iteration number.

`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 KTR_PARAM_OUTLEV>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.

`Opt Err:` Gives a measure of the violation of the Karush-Kuhn-Tucker (KKT) (first-order) optimality conditions (not including feasibility).

`||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.

If KTR_PARAM_OUTLEV=2, information is printed every 10 major iterations. If KTR_PARAM_OUTLEV=3 information is printed at each major iteration. If KTR_PARAM_OUTLEV=4 in addition to printing iteration information on all the major iterations (i.e., accepted steps), the same information will be printed on all minor iterations as well.

**Termination Message**: At the end of the run a termination message is printed indicating whether or not the optimal solution was found and if not, why the code terminated. See Appendix A: for a list of possible termination messages and a description of their meaning and corresponding return value.

**Final Statistics**:

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

**Solution Vector/Constraints**:

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

# 8 Algorithm Options

## 8.1 Automatic

By default, KNITRO will automatically try to choose the best optimizer for the given problem based on the problem characteristics.

## 8.2 Interior/Direct

If the Hessian of the Lagrangian is ill-conditioned or the problem does not have a large-dense Hessian, it may be advisable to compute a step by directly factoring the KKT (primal-dual) matrix rather than using an iterative approach to solve this system. KNITRO offers the Interior/Direct optimizer which allows the algorithm to take direct steps by setting `KTR_PARAM_ALG=1`. This option will try to take a direct step at each iteration and will only fall back on the iterative step if the direct step is suspected to be of poor quality, or if negative curvature is detected.

Using the Interior/Direct optimizer may result in substantial improvements over Interior/CG when the problem is ill-conditioned (as evidenced by Interior/CG taking a large number of Conjugate Gradient iterations). We encourage the user to try both options as it is difficult to predict in advance which one will be more effective on a given problem.

**NOTE:** Since the Interior/Direct algorithm in KNITRO requires the explicit storage of a Hessian matrix, this version can only be used with Hessian options, `KTR_PARAM_HESSOPT=1, 2, 3` or `6`. It may not be used with Hessian options, `KTR_PARAM_HESSOPT=4` or `5`, which only provide Hessian-vector products. Also, the Interior/Direct optimizer cannot be used with the feasible option (`KTR_PARAM_FEASIBLE=1`).

## 8.3 Interior/CG

Since KNITRO was designed with the idea of solving large problems, the Interior/CG optimizer in KNITRO offers an iterative Conjugate Gradient approach to compute the step at each iteration. This approach has proven to be efficient in most cases and allows KNITRO to handle problems with large, dense Hessians, since it does not require factorization of the Hessian matrix. The Interior/CG algorithm can be chosen by setting `KTR_PARAM_ALG=2`. It can use any of the Hessian options as well as the feasible option.

## 8.4 Active

KNITRO 4.0 introduces a new active-set Sequential Linear-Quadratic Programing (SLQP) optimizer. This optimizer is particular advantageous 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 the preferred algorithm for detecting infeasible problems quickly. The Active algorithm can be chosen by setting `KTR_PARAM_ALG=3`. It can use any of the Hessian options.

# 9 Other KNITRO special features

This section describes in more detail some of the most important features of KNITRO and 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 (in dense format) and constraint gradients (in sparse form). It is *highly* recommended that the user provide at least 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 or desirable 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. This option can be invoked by choosing `KTR_PARAM_GRADOPT=2` and calling the routine `KTR_gradfd` (`KTR_gradfdF` for the Fortran interface) at the driver level when asked to evaluate the problem gradients. See the example problem provided with the distribution, or the example in section 4.2 to see how the function `KTR_gradfd` is called.

*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. This option can be invoked by choosing `KTR_PARAM_GRADOPT=3` and calling the routine `KTR_gradfd` (`KTR_gradfdF` for the Fortran interface) at the driver level when asked to evaluate the problem gradients. The centered finite-difference approximation may often be more accurate than the forward finite-difference approximation. However, it is more expensive since it requires twice as many function evaluations to compute. See the example problem provided with the distribution, or the example in section 4.2 to see how the function `KTR_gradfd` is called.

*Gradient Checks*
If the user is supplying a routine for computing exact gradients, but would like to compare these gradients with finite-difference gradient approximations as an error check this can easily be done in KNITRO. To do this, one must call the gradient check routine `KTR_gradcheck` (`KTR_gradcheckF` in the Fortran interface) after calling the user-supplied gradient routine and the finite-difference routine `KTR_gradfd`. See the example problem provided in the `C` directory in the distribution, or section 4.2 of this document for an example of how this is used. The call to the gradient check routine `KTR_gradcheck` will print the relative difference between the user-supplied gradients and the finite-difference gradient approximations. If this value is very small, then it is likely the user-supplied gradients are correct, whereas

a large value may be indicative of an error in the user-supplied gradients. To perform a gradient check with *forward* finite-differences set `KTR_PARAM_GRADOPT=4`, and to perform a gradient check with *centered* finite-differences set `KTR_PARAM_GRADOPT=5`.

**NOTE:** The user must supply the sparsity pattern for the function and constraint gradients to the finite-difference routine `KTR_gradfd`. Therefore, the vectors `indfun` and `indvar` must be specified prior to calling `KTR_gradfd`.

**NOTE:** The finite-difference gradient computation routines are provided for the user's convenience in the file `KNITROgrad.c`. (`KNITROgradF.f` for Fortran). These routines require a call to the user-supplied routine for evaluating the objective and constraint functions. Therefore, these routines may need to be modified to ensure that the call to this user-supplied routine is correct.

## 9.2 Second derivative options

The default version of KNITRO assumes that the user can provide exact second derivatives to compute the Hessian of the Lagrangian function. If the user 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 can be chosen by setting options value `KTR_PARAM_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 can be chosen by setting options value `KTR_PARAM_HESSOPT=3`.

*Finite-difference Hessian-vector product option*
If the problem is large and gradient evaluations are not the dominate 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 can be chosen by setting options value `KTR_PARAM_HESSOPT=4`. This option is generally only recommended if the exact gradients are provided.

**NOTE**: This option may not be used when `KTR_PARAM_ALG=1`.

*Exact Hessian-vector products*
In some cases the user may have a large, dense Hessian which makes it impractical to store or work with the Hessian directly, but the user may be able to provide a routine for evaluating exact Hessian-vector products. KNITRO provides the user with this option. If this option is selected, the user can provide a routine callable at the driver level which given a vector $v$ stored in `vector`, computes the Hessian-vector product, $Hv$, and stores the result in `vector`. This option can be chosen by setting options value `KTR_PARAM_HESSOPT=5`.

**NOTE**: This option may not be used when `KTR_PARAM_ALG=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 only stores a limited number of gradient vectors used to approximate the Hessian. In general it requires more iterations to converge than the dense quasi-Newton BFGS approach but will be much more efficient on large-scale problems. This option can be chosen by setting options value `KTR_PARAM_HESSOPT=6`.

## 9.3 Feasible version

KNITRO offers the user the option of forcing intermediate iterates to stay feasible with respect to the *inequality* constraints (it does not enforce feasibility with respect to the *equality* constraints however). Given an initial point which is *sufficiently* feasible with respect to all inequality constraints and selecting `KTR_PARAM_FEASIBLE = 1`, forces all the iterates to strictly satisfy the inequality constraints throughout the solution process. For the feasible mode to become active the iterate $x$ must satisfy

$$cl + tol \leq c(x) \leq cu - tol \tag{9.27}$$

for *all* inequality constraints (i.e., for $cl \neq cu$). The tolerance $tol > 0$ by which an iterate must be strictly feasible for entering the feasible mode is determined by the parameter `KTR_PARAM_FEASMODETOL` which is `1.0e-4` by default. If the initial point does not satisfy (9.27) then the default infeasible version of KNITRO will run until it obtains a point which is sufficiently feasible with respect to all the inequality constraints. At this point it will switch to the feasible version of KNITRO and all subsequent iterates will be forced to satisfy the inequality constraints.

For a detailed description of the feasible version of KNITRO see [5].

**NOTE:** This option may only be used when `KTR_PARAM_ALG=2`.

## 9.4 Honor Bounds

By default KNITRO does not enforce that the simple bounds on the variables (4.3*c*) are satisfied throughout the optimization process. Rather, satisfaction of these bounds is only enforced at the solution. In some applications, however, the user may want to enforce that the initial point and all intermediate iterates satisfy the bounds $bl \leq x \leq bu$. This can be enforced by setting `KTR_PARAM_HONORBNDS=1`.

## 9.5 Solving Systems of Nonlinear Equations

KNITRO is quite 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 (i.e., constraints with $cl = cu$), and specify the objective function (4.3*a*) as zero (i.e., $f(x) = 0$).

## 9.6 Solving 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

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

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

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

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 `KTR_PARAM_ALG=1`, and will be very similar to the classical Levenberg-Marquardt method when `KTR_PARAM_ALG=2`. For a discussion of these methods see, for example, [8].

## 9.7 Reverse Communication and Interactive Usage of KNITRO

The reverse communication design of KNITRO returns control to the user at the driver level whenever a function, gradient, or Hessian evaluation is needed, thus giving the user complete control over the definition of these routines. In addition, this feature makes it easy for users to insert their own routines to monitor the progress of the algorithm after each iteration or to stop the optimization whenever the user wishes based on whatever criteria the user desires. If the return value from `KTR_solve()` is 0 or negative, the optimization is finished (0 indicates successful completion, whereas a negative return value indicates unsuccessful completion). Otherwise, 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()`. Below are a description of the possible positive return values:

`KTR_RC_EVALFC` **(1)** Evaluate functions `f` and `c` and re-enter `KTR_solve()`.

`KTR_RC_EVALGA` **(2)** Evaluate gradients `fgrad` and `cjac` and re-enter `KTR_solve()`.

`KTR_RC_EVALH` **(3)** Evaluate Hessian `hess` or Hessian-vector product and re-enter `KTR_solve()`.

`KTR_RC_EVALXO` **(4)** Evaluate functions `f` and `c` and gradients `fgrad` and `cjac` and re-enter `KTR_solve()`.

`KTR_RC_NEWPOINT` **(6)** KNITRO has just computed a new solution estimate. The user may provide routine(s) if desired to perform some task before KNITRO begins a new major iteration (requires that `KTR_PARAM_NEWPOINT=1`).

By setting the user parameter `KTR_PARAM_NEWPOINT=1`, KNITRO will return control to the driver level with the return value from `KTR_solve()` = `KTR_RC_NEWPOINT` every time a new approximate solution has been obtained. At this point the functions and gradients are up-to-date and the user may enter his or her own routines to monitor progress or display some results before re-entering KNITRO to start a new iteration.

## 9.8 Callbacks

In the callback feature, the user gives KNITRO several function pointers that KNITRO uses when it needs new function, gradient or Hessian values or to execute a user-provided new-point routine.

For convenience, every one of these callback routines receives every parameter. If your callback requires additional parameters, you are encouraged to create a structure of them, and pass its address with the `user` pointer. KNITRO does not modify or dereference the `user` pointer, so it is safe to use for this purpose. Below is the prototype for the KNITRO callback function:

```
/* define callback prototype */
typedef int KTR_callback(KTR_context_ptr kc,
```

```
    double *f, int ftype, int n, double *x, double *bl, double *bu,
    double *fgrad,
    int m, double *c, double *cl, double *cu, int *ctype, int nnzj,
    double *cjac, int *indvar, int *indfun, double *lambda,
    int nnzh, double *hess, int *hrow, int *hcol, double *vector, void
    *user );
```

The callback functions for evaluating the functions, gradients and Hessian or for performing some newpoint task, can be set as described below. The user callback routines themselves should return an `int` value where any non-negative value indicates a successful return from the callback function and any negative value indicates that an error occurred in the user-provided callback function.

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

/* This callback should modify fgrad and cjac */
int KTR_set_grad_callback(KTR_context_ptr kc,KTR_callback *func);

/* This callback should modify hess or vector, */
/* depending on which hessopt you are using. */
int KTR_set_hess_callback(KTR_context_ptr kc,KTR_callback *func);

/* This callback should modify nothing. */
/* It can be used for updating your graphical user inteface. */
int KTR_set_newpoint_callback(KTR_context_ptr kc,KTR_callback *func);
```

**NOTE**: In order for the "newpoint" callback to be operational, the user must set `KTR_PARAM_NEWPOINT=1`.

**NOTE**: In order to use the finite-difference gradient options with the callback feature you will need to set the gradient evaluation routine to be the finite-difference routine `KTR_grad` provided with the distribution, and pass in the necessary extra parameters through the `user` structure pointer.

KNITRO also provides a special callback function for output printing. By default KNITRO prints to stdout or a `knitro.out` file. This is controlled with the `KTR_PARAM_OUTMODE` option. Alternatively, the user can define a callback function which handles this 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);
```

For an example of how to use the callback feature in KNITRO, please see the callback example provided with the distribution.

## 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. Technical Report OTC 2002/5, Optimization Technology Center, Northwestern University, Evanston, IL, USA, 2002.

[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. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press, 1993.

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

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

[9] 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. Technical Report 2003-6, Optimization Technology Center, Northwestern University, Evanston, IL, USA, June 2003. To appear in Mathematical Programming, Series A.

# 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.`

The objective function appears to be decreasing without bound, while satisfying the constraints.

**-4:** `EXIT: Current point cannot be improved.`

No more progress can be made. If the current point is feasible it is likely it may be optimal, however the stopping tests cannot be satisfied (perhaps because of degeneracy, ill-conditioning or bad scaling).

**-5:** `EXIT: Current point 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 `KTR_PARAM_OUTLEV>0` details of this error will be printed to standard output or the file `knitro.out` 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: Upgrading from KNITRO 3.x

### Migrating from 3.x to 4.0

KNITRO 3.x is backwardly compatible with KNITRO 4.0. That is, an old KNITRO 3.x driver file which calls the KNITRO 3.x `KNITROsolver` function should work with the KNITRO 4.0 libraries provided the `knitro3.h` header file is included in the old KNITRO 3.x driver file. The `knitro3.h` header file holds all the old KNITRO 3.x depreciated declarations.

In moving from KNITRO 3.x to 4.0, you will need to change your linker from `f77` to `g++`, because KNITRO 4.0 no longer has any Fortran but now includes some C++ code. Also you will need to link against the `pthread` and the `dl` libraries (i.e., add `-lpthread` and `-ldl` to the link command).

In order to use most of the new features and functionality of KNITRO 4.0, one must migrate to the new 4.0 API. For a quick overview of the major KNITRO 4.0 API changes, please see the section on 4.0 API Changes below.

### A Summary of 4.0 API Changes

- In KNITRO 4.0, the arrays `cjac/indfun/indvar` now only hold the sparse elements of the constraint gradients. The objective function gradient is stored in a new argument called `fgrad` which is a dense array of size `n`.
- The argument `nnzj` now only refers to the number of nonzero constraint gradient elements (i.e., the number of nonzeros in the new `cjac`). Therefore `nnzj` will be less that in KNITRO 3.x (the nonzeros in the objective gradient were included in the KNITRO 3.x value of `nnzj`). Please be sure to change this value.
- The arrays `indvar/indfun` used to describe the sparse Jacobian elements now use C indexing (0-based) rather than Fortran indexing so they are decremented by one compared to the old value. For example `indfun[i]=0` refers to the first constraint `c[0]`, and `indvar[i]=0` refers to the first variable `x[0]`.
- The sparse Hessian values which used to be described by `nnz_w/w/w_row/w_col` are now called `nnzh/hess/hrow/hcol`. As with the Jacobian indexing, the arrays `hrow/hcol` now use C indexing (0-based).
- The user no longer needs to (and in fact should not) allocate the arrays `hess/hrow/hcol` if not using exact Hessians. If not using exact Hessians, these arrays should be set to `NULL`. Likewise, if not using exact Hessian-vector products, the argument `vector` should not be allocated and should be set to `NULL`.
- The old argument `linear` has been changed to `ctype`. It has the following meaning:

  0: constraint is general nonlinear or nothing is known about it
  1: constraint is linear
  2: constraint is quadratic

- There is a new argument called `ftype` which has the same meaning for the objective function.
- KNITRO 4.0 has a new user options file format which is more flexible than the previous KNITRO 3.x `knitro.spc` file. Using this new options file, the user defines a keyword and a value to set an option. A sample options file is provided in the distribution.
- KNITRO 4.0 is threadsafe.