KNITRO User's Manual

Version 3.0

Richard A. Waltz* Jorge Nocedal*

1 April 2003

Technical Report OTC 2003/5
Optimization Technology Center
Northwestern University

*Electrical and Computer Engineering Department, Northwestern University, Evanston IL 60208. These authors were supported by National Science Foundation grants CCR-9987818, CCR-0219438, ATM-0086579, STTR-25637 and by Department of Energy grant DE-FG02-87ER25047-A004.

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 3.0 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,
- quadratic programming problems,
- 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 feasible and infeasible versions,
- Both iterative and direct approaches for computing steps,
- Interfaces: AMPL, C/C++, Fortran,
- 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

$$\min_{x} \quad f(x) \tag{1.1a}$$

$$\text{s.t.} \quad h(x) = 0 \tag{1.1b}$$

$$g(x) \leq 0. \tag{1.1c}$$

This allows many forms of constraints, including bounds on the variables. KNITRO requires that the functions $f(x)$, $h(x)$ and $g(x)$ be smooth functions.

KNITRO implements an interior method (also known as barrier method) for nonlinear programming. The nonlinear programming problem is replaced by a series of barrier subproblems controlled by a barrier parameter μ. 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. In the version known as KNITRO-CG each step is the sum of two components: (i) a normal step whose objective is to improve feasibility; and (ii) a tangential step that aims toward optimality. The tangential 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 KNITRO-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 KNITRO-CG procedure.

KNITRO-CG is the default, but we encourage the user to try both options to determine which one is more suitable for the application at hand. KNITRO-CG is recommended for problems in which the Hessian of the Lagrangian is not very sparse whereas KNITRO-DIRECT may be more effective on ill-conditioned problems.

For a detailed description of the algorithm implemented in KNITRO see [3] and for the global convergence theory see [2]. The method implemented in KNITRO-DIRECT is described in [7]. An important component of KNITRO is the HSL routine `MA27` [6] which is used to solve the linear systems arising at every iteration of the algorithm.

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 at

`http://www.ziena.com/knitro/kindex.htm`

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`

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 UNIX platforms are described below.

2.1 Windows

Save the downloaded file (KNITRO.3.0.zip) in a fresh subdirectory on your system. To install, first, unzip this file using a Windows extraction tool (such as WinZip) to create the directory KNITRO.3.0 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 Windows dynamically linked libraries (dll's) needed for running KNITRO, and the precompiled KNITRO-AMPL executable file, `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.

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. An example problem is provided in both the `C` and `Fortran` interface subdirectories. It is recommended to run and understand this example problem before proceeding, as this problem contains 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.

2.2 UNIX/Linux

Save the downloaded file (KNITRO.3.0.tar.gz) in a fresh subdirectory on your system. To install, first type

```
gunzip KNITRO.3.0.tar.gz
```

to produce a file KNITRO.3.0.tar. Then, type

```
tar -xvf KNITRO.3.0.tar
```

to create the directory KNITRO.3.0 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`.
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.

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. An example problem is provided in both the `C` and `Fortran` interface subdirectories. It is recommended to run and understand this example problem before proceeding, as this problem contains 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 [5] 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` 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;
```

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

```
option knitro_options ‘‘maxit=100 direct=1’’;
```

The above command would set the maximum number of allowable iterations to 100 and choose the KNITRO-DIRECT algorithm (see section 8). See Table 1 for a summary of all available user specifiable options in KNITRO. For more detail on these options see section 5.

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

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

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 `direct=1` (to use the KNITRO-DIRECT algorithm), and `opttol=1e-8` (to tighten the optimality stopping tolerance).

AMPL Example

```
ampl: reset;
ampl: option solver knitro;
ampl: option knitro_options "direct=1 opttol=1e-8";
ampl: model testproblem.mod;
ampl: solve;

KNITRO 3.0: 04/01/03

: direct=1
opttol=1e-8
```

```
******************
*   KNITRO 3.0   *
******************

Nondefault Options
------------------
 direct      =   1
 opttol      =  1.00E-08

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   Res   Objective    Feas err  Opt err   ||Step||  CG its     mu
------  ---  ------------  --------  --------  --------  ------  --------
     0       9.760000E+02  1.30E+01
     1  Acc  9.712755E+02  9.66E+00  6.03E+00  8.97E-01       1  1.00E-01
     2  Acc  9.445610E+02  7.06E-01  6.95E+00  5.45E+00       1
     3  Acc  9.392542E+02  7.11E-15  2.45E+00  7.89E-01       2
     4  Acc  9.361107E+02  7.11E-15  1.45E-01  4.84E-01       0
     5  Acc  9.360409E+02  0.00E+00  2.10E-02  4.79E-03       0  2.00E-02
     6  Acc  9.360004E+02  0.00E+00  2.10E-04  4.03E-03       0  2.00E-04
     7  Acc  9.360000E+02  0.00E+00  2.00E-06  4.16E-05       0  2.00E-06
     8  Acc  9.360000E+02  0.00E+00  2.00E-08  3.99E-07       0  2.00E-08

EXIT: 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) =     2.00E-08 / 1.25E-09
# of iterations                      =          8
# of function evaluations           =          9
# of gradient evaluations           =          9
# of Hessian evaluations            =          8
Total program time (sec)             =     0.01

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

KNITRO 3.0: OPTIMAL SOLUTION FOUND.
ampl:
```

OPTION	DESCRIPTION	DEFAULT
delta	initial trust region radius	1.0e0
direct	0: always use iterative approach to generate steps 1: allow for direct factorization steps	0
feasible	0: allow for infeasible iterates 1: feasible version of KNITRO	0
feastol	feasibility termination tolerance (relative)	1.0e-6
feastol_abs	feasibility termination tolerance (absolute)	0.0e-0
gradopt	gradient computation method: 1: use exact gradients (only option available for AMPL interface)	1
hessopt	Hessian (Hessian-vector) computation method: 1: use exact Hessian 2: use dense quasi-Newton BFGS Hessian approximation 3: use dense quasi-Newton SR1 Hessian approximation 4: compute Hessian-vector products via finite differencing 5: compute exact Hessian-vector products (user-must provide routine to use this option) 6: use limited-memory BFGS Hessian approximation	1
initpt	0: do not use any initial point strategies 1: use initial point strategy	0
iprint	printing output level: 0: no printing 1: just print summary information 2: print information at each iteration 3: also print final (primal) variables 4: also print final constraint values and Lagrange multipliers	2
maxit	maximum number of iterations before terminating	1000
mu	initial barrier parameter value	1.0e-1
nout	where to direct output: 6: screen 90: output file 'knitro.out'	6
opttol	optimality termination tolerance (relative)	1.0e-6
opttol_abs	optimality termination tolerance (absolute)	0.0e-0
pivot	initial pivot threshold for matrix factorizations	1.0e-8
soc	0: do not allow second order correction steps 1: allow for second order correction steps	1

Table 1: KNITRO user specifiable options for AMPL.

4 The KNITRO callable library

4.1 Calling KNITRO from a C application

For the C interface the user must include the KNITRO header file `knitro.h` in the C source code which calls KNITRO. In addition the user is required to provide a maximum of four C functions. The user may name these functions as he or she wishes but in our example we use the following names:

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

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

`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 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.

$$\min_x \quad f(x) \tag{4.3a}$$
$$\text{s.t.} \quad cl \leq c(x) \leq cu \tag{4.3b}$$
$$bl \leq x \leq bu. \tag{4.3c}$$

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

NOTE: KNITRO defines infinite upper and lower bounds using the `double` values $\pm$`1.0e+20`. Any bounds smaller in magnitude than `1.0e+20` 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, user_problem.c and knitro.spc provided with the distribution for an example of how to specify the above functions and call the KNITRO solver function, `KNITROsolver`, to solve a user-defined problem in C. The file user_problem.c contains examples of the functions - `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 `KNITROsolver`.

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

```
KNITROsolver  (&n, &m, &f, x, bl, bu, c, cl, cu, equatn,
               linear, &nnzj, cjac, indvar, indfun, lambda,
               &nnz_w, w, w_row, w_col, vector, &specs,
               &info, &status)
```

NOTE: Because of the reverse communication/interactive implementation of the KNITRO callable library (see section 9.6), and for uniformity and simplicity, all scalar variables passed to `KNITROsolver` are passed by reference (using the `&` syntax) rather than the C standard of passing scalars by value.

The following variables are passed to the KNITRO solver function `KNITROsolver`:

`n:` is a variable of type `int`.

`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.

`m:` is a variable of type `int`.

`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.

`f:` is a variable of type `double` which 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 `status` $= 1$ or 4 the user must evaluate `f` at the current value of `x` before re-entering `KNITROsolver`.

`x:` is an array of type `double` and 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.

`bl:` is an array of type `double` and length `n`.

`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 the large negative number `-1.0e+20`.

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

`bu:` is an array of type `double` and length `n`.

`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 the large positive number `1.0e+20`.

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

`c:` is an array of type `double` and 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 `status` = 1 or 4 the user must evaluate `c` at the current value of `x` before re-entering `KNITROsolver`.

`cl:` is an array of type `double` and length `m`.

`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 the large negative number `-1.0e+20`. If the constraint is an equality constraint `cl`[i] should equal `cu`[i].

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

`cu:` is an array of type `double` and length `m`.

`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 the large positive number `1.0e+20`. If the constraint is an equality constraint `cl`[i] should equal `cu`[i].

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

`equatn:` is an array of type `int` and length `m`.

`On initial entry:` `equatn`[i] must be set by the user to 1 if constraint `c`[i] is an equality constraint and 0 otherwise.

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

NOTE: The array `equatn` overrides the settings of the bounds `cl` and `cu`. If `equatn`[i] = 1 but `cl`[i] does not equal `cu`[i], then the upper bound, `cu`[i], will be ignored and constraint i will be treated as an equality constraint of the form `c`[i] = `cl`[i].

`linear:` is an array of type `int` and length `m`.

`On initial entry:` `linear`[i] must be set by the user to 1 if constraint `c`[i] is a linear constraint and 0 otherwise.

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

`nnzj:` is a variable of type `int`.

`On initial entry:` `nnzj` must be set by the user to the number of nonzeros in the Jacobian matrix `cjac` which contains **both** the gradient of the objective function `f` and the constraint gradients in sparse form.

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

`cjac:` is an array of type `double` and length `nnzj`. The first part contains the nonzero elements of the gradient of the objective function; the second part contains the nonzero elements of the Jacobian of the constraints.

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

`On exit:` `cjac` holds the current approximation to the optimal gradient values. If `status` = 2 or 4 the user must evaluate `cjac` at the current value of `x` before re-entering `KNITROsolver`.

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

NOTE: C array indexing starts with indice 0. Therefore, the j-th variable corresponds to C array element `x[j-1]`.

`On initial entry:` `indvar` need not be specified on initial entry but must be specified with the first evaluation of `cjac` (it may also be specified before initial entry if desired).

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

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

NOTE: C array indexing starts with indice 0. Therefore, the k-th constraint corresponds to C array element `c[k-1]`.

`On initial entry:` `indfun` need not be specified on initial entry but must be specified with the first evaluation of `cjac` (it may also be specified before initial entry if desired).

`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 gradient and Jacobian elements specified in `cjac`[i].

`lambda:` is an array of type `double` and length `m+n`.

`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`.

`nnz_w:` is a variable of type `int`.

`On initial entry:` `nnz_w` 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).

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

NOTE: If `hessopt` $= 4, 5$ or 6 then the Hessian of the Lagrangian is not explicitly stored and one can set `nnz_w` $= 0$.

`w:` is an array of type `double` and dimension `nnz_w` containing the nonzero elements of the Hessian of the Lagrangian which is defined as

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

Only the nonzero elements of the upper triangle are stored.

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

`On exit:` `w` contains the current approximation of the optimal Hessian of the Lagrangian. If `status` $= 3$ (and `hessopt=1`) the user must evaluate `w` at the current values of `x` and `lambda` before re-entering `KNITROsolver`.

`w_row:` is an array of type `int` and length `nnz_w`. `w_row`$[i]$ stores the row number of the nonzero element `w`$[i]$. (NOTE: Row numbers range from 1 to n).

`On initial entry:` `w_row` need not be specified on initial entry but must be specified with the first evaluation of `w` (it may also be specified before initial entry if desired).

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

`w_col:` is an array of type `int` and length `nnz_w`. `w_col`$[i]$ stores the column number of the nonzero element `w`$[i]$. (NOTE: Column numbers range from 1 to n).

`On initial entry:` `w_col` need not be specified on initial entry but must be specified with the first evaluation of `w` (it may also be specified before initial entry if desired).

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

`vector:` is an array of type `double` and length `n` which is only used if the user is providing a function for computing exact Hessian-vector products.

`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 `KNITROsolver` function.

`On exit:` If `status` = 3 (and `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.

`specs:` is a C structure with 16 elements which holds the user options; see section 5 for more details.

`info:` is a variable of type `int` which indicates the exit status of the optimization routine.

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

`On exit:` `info` returns an integer which specifies the termination status of the code (see section 7 for details).

`info` = 0: termination without error

`info` < 0: termination with an error

`status:` is a variable of type `int` which keeps track of the reverse communication status.

`On initial entry:` Initialize `status = 0`.

`On exit:` `status` contains information on which task needs to be performed at the driver level before re-entry.

0	Initialization state
1 or 4	Evaluate functions `f` and `c`
2 or 4	Evaluate gradients `cjac`
3	Evaluate Hessian `w` or Hessian-vector product
5	Optimization finished

4.2 Example

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

$$\min_x \quad 1000 - x_1^2 - 2x_2^2 - x_3^2 - x_1x_2 - x_1x_3 \tag{4.5a}$$

$$\text{s.t.} \quad 8x_1 + 14x_2 + 7x_3 - 56 = 0 \tag{4.5b}$$

$$x_1^2 + x_2^2 + x_3^2 - 25 \geq 0 \tag{4.5c}$$

$$x_1, x_2, x_3 \geq 0 \tag{4.5d}$$

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

A sample C program for calling the KNITRO function `KNITROsolver` and output are contained in the following pages.

C driver program

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

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

/* User-defined functions */

#ifdef __cplusplus
extern "C" {
#endif

void setup(int *n, int *m, double *x, double *bl, double *bu,
           int *equatn, int *linear, double *cl, double *cu);

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

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

void evalhess(int *n, int *m, double *x, double *lambda,
              int *nnz_w, double *w, int *w_row, int *w_col);

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

#ifdef __cplusplus
}
#endif

main()
{
  /* Declare variables which are passed to function KNITROsolver. */
  int *indvar, *indfun, *w_row, *w_col, *equatn, *linear;
  double *x, *c, *cjac, *lambda, *w, *bl, *bu, *cl, *cu, *vector;
  int n, m, nnzj, nnz_w, info, status;
```

```
double f;
struct spectype specs;

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

/* Declare other variables. */
int i, j, k;
double *tmp;

/* Declare some flag values used by KNITRO. */
int initial       = 0;
int eval_fc       = 1;
int eval_ga       = 2;
int eval_w        = 3;
int eval_x0       = 4;
int finished      = 5;

/* Read in values from the specs file 'knitro.spc'. */
specs = KNITROspecs();

/* Set n, m, nnzj, nnz_w */
n = 3;
m = 2;
nnzj = 9;
switch (specs.hessopt)
  {
  case 1:
    /* User-supplied exact Hessian */
    nnz_w = 5;
    break;
  case 2: case 3:
    /* dense quasi-Newton */
    nnz_w = (n*n - n)/2 + n;
    break;
  case 4: case 5: case 6:
    /* Hessian-vector product or limited-memory BFGS */
    nnz_w = 0;
    break;
  default: printf("ERROR: Bad value for hessopt. (%d)\n", specs.hessopt);
    exit(1);
  }
```

```
/* Allocate arrays that get passed to KNITROsolver */
x         = (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));
w         = (double *)malloc(nnz_w*sizeof(double));
w_row     = (int *)malloc(nnz_w*sizeof(int));
w_col     = (int *)malloc(nnz_w*sizeof(int));
vector    = (double *)malloc(n*sizeof(double));
bl        = (double *)malloc(n*sizeof(double));
bu        = (double *)malloc(n*sizeof(double));
equatn    = (int *)malloc(m*sizeof(int));
linear    = (int *)malloc(m*sizeof(int));
cl        = (double *)malloc(m*sizeof(double));
cu        = (double *)malloc(m*sizeof(double));
tmp       = (double *)malloc(n*sizeof(double));

/* Array only needed if performing gradient check */
if (specs.gradopt == 4 || specs.gradopt == 5)
  cjacfd    = (double *)malloc(nnzj*sizeof(double));

setup(&n, &m, x, bl, bu, equatn, linear, cl, cu);

status = initial;
info = 0;

while(status != finished){

  /* evaluate function and constraint values */
  if ((status == eval_fc) || (status == eval_x0)){
    evalfc(&n, &m, x, &f, c);
  }

  /* evaluate objective and constraint gradients */
  if ((status == eval_ga) || (status == eval_x0)){
    switch (specs.gradopt)
      {
      case 1:
        /* User-supplied exact gradient. */
        evalga(&n, &m, x, &nnzj, cjac, indvar, indfun);
```

```
        break;
      case 2: case 3:
        /* Compute finite difference gradient
           (specify sparsity pattern first). */
        i = 0;
        for (j = 0; j <= m; j++) {
          for (k = 1; k <= n; k++) {
            indfun[i] = j;
            indvar[i] = k;
            i++;
          }
        }
        KNITROgradfd(&n, &m, x, &nnzj, cjac, indvar, indfun,
                     &f, c, specs.gradopt);
        break;
      case 4: case 5:
        /* Perform gradient check using finite difference gradient */
        evalga(&n, &m, x, &nnzj, cjac, indvar, indfun);
        KNITROgradfd(&n, &m, x, &nnzj, cjacfd, indvar, indfun,
                     &f, c, specs.gradopt);
        KNITROgradcheck(&nnzj, cjac, cjacfd, &fderror);
        printf("Finite difference gradient error = %e\n", fderror);
        break;
      default: printf("ERROR: Bad value for gradopt. (%d)\n", specs.gradopt);
        exit(1);
      }
  }

  /* Evaluate user-supplied Lagrange Hessian (or Hessian-vector product) */
  if (status == eval_w){
    if (specs.hessopt == 1)
      evalhess(&n, &m, x, lambda, &nnz_w, w, w_row, w_col);
    else if (specs.hessopt == 5)
      evalhessvec(&n, &m, x, lambda, vector, tmp);
  }

  /* Call KNITRO solver routine */

  KNITROsolver(&n, &m, &f, x, bl, bu, c, cl, cu, equatn, linear,
               &nnzj, cjac, indvar, indfun, lambda, &nnz_w, w,
               w_row, w_col, vector, &specs, &info, &status);
}
```

```
  free(x);
  free(c);
  free(cjac);
  free(indvar);
  free(indfun);
  free(lambda);
  free(w);
  free(w_row);
  free(w_col);
  free(vector);
  free(bl);
  free(bu);
  free(equatn);
  free(linear);
  free(cl);
  free(cu);
  free(tmp);

  return 0;
}
```

User-defined C Functions

```
void setup(int *n, int *m, double *x, double *bl, double *bu,
           int *equatn, int *linear, double *cl, double *cu)
{
  /* Function that gives data on the problem. */

  int i;
  double zero, two, biginf;

  zero   = 0.0e0;
  two    = 2.0e0;
  biginf = 1.0e20;

  /* 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] = biginf;
  }

  /* equatn[i] = 1 if i-th constraint is an equality,
   *           = 0 if it is an inequality.
   * Analogously for the parameter linear.
   */

  equatn[0] = 1;
  equatn[1] = 0;

  linear[0] = 1;
  linear[1] = 0;

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

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

  cl[1] = zero;
  cu[1] = biginf;
}
/*******************************************************/
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, double *x, int *nnzj, double *cjac,
            int *indvar, int *indfun)
{
  /* Routine for computing gradient values in sparse form.
```

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

/* Gradient of the objective function. */

cjac[0]   = -2.0e0*x[0]-x[1]-x[2];
indvar[0] = 1;
indfun[0] = 0;

cjac[1]   = -4.0e0*x[1]-x[0];
indvar[1] = 2;
indfun[1] = 0;

cjac[2]   = -2.0e0*x[2]-x[0];
indvar[2] = 3;
indfun[2] = 0;

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

cjac[3]   = 8.0e0;
indvar[3] = 1;
indfun[3] = 1;

cjac[4]   = 14.0e0;
indvar[4] = 2;
indfun[4] = 1;

cjac[5]   = 7.0e0;
indvar[5] = 3;
indfun[5] = 1;

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

cjac[6]   = 2.0e0*x[0];
indvar[6] = 1;
indfun[6] = 2;

cjac[7]   = 2.0e0*x[1];
indvar[7] = 2;
indfun[7] = 2;

cjac[8]   = 2.0e0*x[2];
```

```
  indvar[8] = 3;
  indfun[8] = 2;
}
/*************************************************/
void evalhess(int *n, int *m, double *x, double *lambda,
              int *nnz_w, double *w, int *w_row, int *w_col)
{
  /*
   * Compute the Hessian of the Lagrangian.
   *
   * NOTE: Only the NONZEROS of the UPPER TRIANGLE (including
   *       diagonal) of this matrix should be stored.
   */

  w[0]     = 2.0e0*(lambda[1]-1.0e0);
  w_row[0] = 1;
  w_col[0] = 1;

  w[1]     = -1.0e0;
  w_row[1] = 1;
  w_col[1] = 2;

  w[2]     = -1.0e0;
  w_row[2] = 1;
  w_col[2] = 3;

  w[3]     = 2.0e0*(lambda[1]-2.0e0);
  w_row[3] = 2;
  w_col[3] = 2;

  w[4]     = 2.0e0*(lambda[1]-1.0e0);
  w_row[4] = 3;
  w_col[4] = 3;
}
/*************************************************/
void evalhessvec(int *n, int *m, 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 3.0   *
******************

Nondefault Options
------------------

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   Res   Objective    Feas err  Opt err   ||Step||  CG its     mu
------  ---  ------------  --------  --------  --------  ------  --------
     0       9.760000E+02  1.30E+01
     1  Acc  9.688061E+02  7.19E+00  6.92E+00  1.51E+00       1  1.00E-01
```

```
     2  Acc  9.397651E+02  1.95E+00  2.99E+00  5.66E+00       1
     3  Acc  9.323614E+02  1.66E+00  1.65E-01  1.24E+00       2
     4  Acc  9.361994E+02  1.70E-16  1.02E-01  2.39E-01       2
     5  Acc  9.360402E+02  1.80E-15  2.02E-02  1.59E-02       2  2.00E-02
     6  Acc  9.360004E+02  5.69E-16  2.51E-04  4.02E-03       1  2.00E-04
     7  Acc  9.360000E+02  2.40E-15  2.00E-06  3.79E-05       2  2.00E-06

EXIT: OPTIMAL SOLUTION FOUND.

Final Statistics
----------------
Final objective value               =     9.36000004001378E+02
Final feasibility error (abs / rel) =     2.40E-15 / 1.84E-16
Final optimality error  (abs / rel) =     2.00E-06 / 1.25E-07
# of iterations                     =          7
# of function evaluations           =          8
# of gradient evaluations           =          8
# of Hessian evaluations            =          7
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

For the Fortran interface the user is required to provide a maximum of four Fortran subroutines. The user may name these subroutines as he or she wishes but in our example we use the following names:

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

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

`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.

$$\min_x \quad f(x) \tag{4.6a}$$
$$\text{s.t.} \quad cl \leq c(x) \leq cu \tag{4.6b}$$
$$bl \leq x \leq bu. \tag{4.6c}$$

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

NOTE: KNITRO defines infinite upper and lower bounds using the `DOUBLE PRECISION` values $\pm$`1.0d+20`. Any bounds smaller in magnitude than `1.0d+20` 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 knitro.spc provided with the distribution for an example of how to specify the above subroutines and call `KNITROsolverF` to solve a user-defined problem in Fortran. The file user_problem.f contains examples of the subroutines - `setup`, `evalfc`, `evalgA`, `evalHess` and `evalHessvec` - while driverFortran.f is

an example driver file for the Fortran interface which calls these subroutines as well as the KNITRO solver function for the Fortran interface, `KNITROsolverF`.

The KNITRO solver is invoked via a call to the routine `KNITROsolverF` which has the following calling sequence:

```
 call KNITROsolverF  (n, m, f, x, bl, bu, c, cl, cu,
&                    equatn, linear, nnzj, cjac, indvar, indfun,
&                    lambda, nnz_w, w, w_row, w_col, vector,
&                    ispecs, dspecs, info, status)
```

Most of the information and instruction on calling the KNITRO solver from within a Fortran program is the same as for the C interface (see section 4.4) 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.
- The C structure `specs` used to hold the user options is replaced in the Fortran interface by two arrays of size 10 – the `INTEGER` array `ispecs` and the `DOUBLE PRECISION` array `dspecs`. The first nine elements of `ispecs` should hold the values corresponding to the first nine elements in the C structure `specs` which are integer. Likewise, the first seven elements of `dspecs` should hold the values of the tenth through sixteenth elements in the C structure `specs` which are double. Element 10 of `ispecs` and elements 8-10 of `dspecs` are currently unused and do not need to be specified.
- 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 when applied to a Fortran program.

4.5 Using KNITRO with the SIF/CUTEr interface

CUTEr (Constrained and Unconstrained Testing Environment) is a testing environment and interface for optimization solvers. For a detailed description of the CUTEr package see [1]. The CUTEr interface reads problems written in SIF format. There are already nearly 1000 problems in the CUTEr collection (with derivatives provided) which are extremely valuable for benchmarking optimization software. Moreover, if one can formulate his or her own problem in SIF format, then one can use the CUTEr interface with KNITRO to solve that problem.

The CUTEr interface for KNITRO is a Fortran interface so the call to the KNITRO subroutine `KNITROsolverF` is the same as that described in the Fortran interface section (section 4.4) and won't be repeated here.

However, it differs from the Fortran interface in that CUTEr provides it's own routines for evaluating functions and sparse gradients and Hessians.

5 The KNITRO specifications file (knitro.spc)

The KNITRO specs file, knitro.spc 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). The parameters which can be changed are stored in a structure of type `spectype` which has the following format:

```
struct spectype {
  int nout;
  int iprint;
  int maxit;
  int hessopt;
  int gradopt;
  int initpt;
  int soc;
  int feasible;
  int direct;
  double feastol;
  double opttol;
  double feastol_abs;
  double opttol_abs;
  double delta;
  double pivot;
  double mu;
};
```

The individual components of this structure are described below. To change the value of a parameter, just overwrite the current value specified for that parameter in knitro.spc and save the modified knitro.spc file. The parameter value may also be specified in the driver *after* the file knitro.spc is read. However, specifying these parameters in the driver requires that one recompile the driver before these changes take affect.

The following options are specified in the KNITRO specifications file knitro.spc.

The integer valued components are:

`nout:` specifies where to direct the output.

- `6:` output is directed to the screen
- `90:` (or some value other than 6) output is sent to a file called knitro.out.

Default value: `6`

`iprint:` controls the level of output.

`0:`	printing of all output is suppressed
`1:`	only summary information is printed
`2:`	information at each iteration is printed
`3:`	in addition, the values of the solution vector `x` are printed
`4:`	in addition, the values of the constraints `c` and Lagrange multipliers `lambda` are printed

Default value: `2`

`maxit:` specifies the maximum number of iterations before termination.

Default value: `1000`

`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: Typically 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 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 `hessopt=4` and `hessopt=5` may only be used when `direct=0`. See section 9.2 for more detail on second derivative options.

`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:` gradient check performed with forward finite differences
- `5:` gradient check performed 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.

`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`

NOTE: This option may only be used when `direct=1`.

`soc:` indicates whether or not to use the second order correction option.

- `0:` No second order correction steps are attempted.
- `1:` Second order correction steps may be attempted on some iterations.

Default value: `1`

`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,

$$cl + 10^{-4} \leq c(x) \leq cu - 10^{-4} \tag{5.7}$$

(for $cl \neq cu$), the feasible version of KNITRO ensures that all subsequent iterates strictly satisfy the *inequality* constraints. However, the iterates may not be feasible with respect to the *equality* constraints. If the initial point is infeasible (or not

sufficiently feasible according to (5.7)) 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 `direct=0`. See section 9.3 for more details.

`direct:` indicates whether or not to use the Direct Solve option to compute a step (see section 8).

- `0:` KNITRO will always compute steps using an iterative (Conjugate Gradient approach).
- `1:` KNITRO will attempt to compute a step using a direct factorization of the KKT (primal-dual) matrix.

Default value: `0`

The double precision valued options are:

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

Default value: `1.0e-6`

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

Default value: `1.0e-6`

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

Default value: `0.0e0`

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

Default value: `0.0e0`

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

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

Default value: `1.0e0`

`pivot:` specifies the initial pivot threshold used in the factorization routine. The value must be in the range [-0.5 0.5] with higher values resulting in more pivoting (more stable factorization). Values less than -0.5 will be set to -0.5 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.

Default value: `1.0e-8`

`mu:` specifies the initial barrier parameter value.

Default value: `1.0e-1`

6 KNITRO Termination Test and Optimality

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

$$\nabla_x \mathcal{L}(x,\lambda) = \nabla f(x) + \sum_{i\in\mathcal{E}} \lambda_i \nabla h_i(x) + \sum_{i\in\mathcal{I}} \lambda_i \nabla g_i(x) = 0 \tag{6.8}$$

$$\lambda_i g_i(x) = 0, \quad i \in \mathcal{I} \tag{6.9}$$

$$h_i(x) = 0, \quad i \in \mathcal{E} \tag{6.10}$$

$$g_i(x) \leq 0, \quad i \in \mathcal{I} \tag{6.11}$$

$$\lambda_i \geq 0, \quad i \in \mathcal{I} \tag{6.12}$$

where $\mathcal{E}$ and $\mathcal{I}$ represent the sets of indices corresponding to the equality constraints and inequality constraints respectively, and λ_i is the Lagrange multiplier corresponding to constraint i. In KNITRO we define the feasibility error (`Feas err`) at a point x^k to be the maximum violation of the constraints (6.10), (6.11), i.e.,

$$\text{Feas err} = \max_i(0, |h_i(x^k)|, g_i(x^k)), \tag{6.13}$$

while the optimality error (`Opt err`) is defined as the maximum violation of the first two conditions (6.8), (6.9),

$$\text{Opt err} = \max_{i\in\mathcal{I}}(\|\nabla_x \mathcal{L}(x^k,\lambda^k)\|_\infty, |\lambda_i g_i(x^k)|). \tag{6.14}$$

The last optimality condition (6.12) is enforced explicitly throughout the optimization. In order to take into account problem scaling in the termination test, the following scaling factors are defined

$$\tau_1 = \max(1, |h_i(x^0)|, g_i(x^0)), \tag{6.15}$$

$$\tau_2 = \max(1, \|\nabla f(x^k)\|_\infty), \tag{6.16}$$

where x^0 represents the initial point

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

$$\text{Feas err} \leq \max(\tau_1 * \text{feastol}, \text{feastol\_abs}) \tag{6.17}$$

$$\text{Opt err} \leq \max(\tau_2 * \text{opttol}, \text{opttol\_abs}) \tag{6.18}$$

where `feastol, opttol, feastol_abs` and `opttol_abs` are user-defined options which can be specified in the file `knitro.spc` (see section 5). For systems of nonlinear equations or feasibility problems only stopping condition (6.17) is used (see section 9.4).

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 `feastol_abs` and `opttol_abs` equal to `0.0e0`. Likewise, an absolute stopping test can be enforced by setting `feastol` and `opttol` equal to `0.0e0`.

7 KNITRO Output

If `iprint=0` then all printing of output is suppressed. For the default printing output level (`iprint=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:
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.

`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 attempted step.

`CG its:` The number of Projected Conjugate Gradient (CG) iterations required to approximately solve the tangential subproblem.

`mu:` The value of the barrier parameter (only printed at the beginning of a new barrier subproblem).

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. Below is a list of possible termination messages and a description of their meaning and corresponding `info` value.

0: `EXIT: 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).

-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 infeasible.`

The algorithm has converged to a stationary point of infeasibility. This happens when the problem is infeasible, but may also occur on occasion for feasible problems with nonlinear constraints. 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.

-50 to -99:

Termination values in this range imply some input error.

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 `iprint=3`, the values of the solution vector are printed after the final statistics. If `iprint=4`, 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 KNITRO-CG

Since KNITRO was designed with the idea of solving large problems, by default KNITRO uses 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.

8.2 KNITRO-DIRECT

The Conjugate Gradient approach may suffer when the Hessian of the Lagrangian is ill-conditioned. In this case 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 an option which allows the algorithm to take direct steps by setting `direct=1`. This option will try to take a direct step at each iteration and will only fall back on the standard iterative step if the direct step is suspected to be of poor quality, or if negative curvature is detected.

Using the Direct version of KNITRO may result in substantial improvements over the default KNITRO-CG when the problem is ill-conditioned (as evidenced by KNITRO-CG taking a large number of Conjugate Gradient iterations). It also may be more efficient for small problems. 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 Direct Solver version of KNITRO requires the explicit storage of a Hessian matrix, this version can only be used with Hessian options, `hessopt=1, 2, 3` or `6`. It may not be used with Hessian options, `hessopt=4` or `5`, which only provide Hessian-vector products. Also, the Direct version of KNITRO cannot be used with the Feasible version.

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 particular problem the user wishes to solve.

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 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 `gradopt=2` and calling the routine `KNITROgradfd` (`KNITROgradfdF` 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 `KNITROgradfd` 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 `gradopt=3` and calling the routine `KNITROgradfd` (`KNITROgradfdF` 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 `KNITROgradfd` 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 `KNITROgradcheck` (`KNITROgradcheckF` in the Fortran interface) after calling the user-supplied gradient routine and the finite-difference routine `KNITROgradfd`. See the example problem provided in the `C` and `Fortran` directories 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 `KNITROgradcheck` 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 `gradopt=4`, and to perform a gradient check with *centered* finite-differences set `gradopt=5`.

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

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 many cases. This option stores a *dense* quasi-Newton Hessian approximation so it is only recommended for small to medium problems ($n < 2000$). The quasi-Newton BFGS option can be chosen by setting options value `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 ($n < 2000$). The quasi-Newton SR1 option can be chosen by setting options value `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 `hessopt=4`. This option is generally only recommended if the exact gradients are provided.

NOTE: This option may only be used when `direct=0`.

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, Wv, and stores the result in `vector`. This option can be chosen by setting options value `hessopt=5`.

NOTE: This option may only be used when `direct=0`.

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 `hessopt=6`.

9.3 Feasible version

The default version of KNITRO allows intermediate iterates to be infeasible (violate the constraints). However, in some applications it is important that the iterates try to stay feasible. For example the user may want to stop the solver before optimality has been reached and be assured that the approximate solution is feasible or there may be some constraints which are undefined outside the feasible region which make it necessary to stay feasible.

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 `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

$$cl + 10^{-4} \leq c(x) \leq cu - 10^{-4} \tag{9.19}$$

for *all* inequality constraints (i.e., for $cl \neq cu$). If the initial point does not satisfy (9.19) 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 [4].

NOTE: This option may only be used when `direct=0`.

9.4 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 in KNITRO (i.e., constraints with $cl = cu$). If the number of equality constraints equals (or exceeds) the number of problem variables, KNITRO will treat the problem as a system of nonlinear equations (or feasibility problem). In this case, KNITRO will ignore the objective function, if one is provided, and just search for a point which satisfies the constraints declaring `OPTIMAL SOLUTION FOUND` if one is found. Since the problem, is treated as a feasibility problem, Lagrange multiplier estimates are not computed.

9.5 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 `direct=1`, and will be very similar to the classical Levenberg-Marquardt method when `direct=0`. For a discussion of these methods see, for example, [8].

9.6 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 and over the definition of these routines. In addition, this feature makes it easy for the user 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.

NOTE: In default mode, the indication that KNITRO has just computed a new approximate solution or iterate is when it returns to the driver level to compute the function and constraint *gradients* (i.e., when `status=2`). Therefore, if the user wishes to insert a routine which performs a particular task at each new iterate, this routine should be called immediately after the gradient computation routine at the driver level.

However, if Hessian-vector products are being computed via finite-differences of the gradients (i.e., `hessopt=4`), then a gradient computation occurs whenever a Hessian-vector product is needed so a gradient computation does not signify a new iterate in this case.

References

[1] I. Bongartz, A. R. Conn, N. I. M. Gould, and Ph. L. Toint. CUTE: Constrained and Unconstrained Testing Environment. *ACM Transactions on Mathematical Software*, 21(1):123–160, 1995.

[2] R. H. Byrd, J. Ch. Gilbert, and J. Nocedal. A trust region method based on interior point techniques for nonlinear programming. *Mathematical Programming*, 89(1):149–185, 2000.

[3] R. 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.

[4] R. H. Byrd, J. Nocedal, and R. A. Waltz. Feasible interior methods using slacks for nonlinear optimization. Technical Report OTC 2000/11, Optimization Technology Center, Northwestern University, Evanston, IL, USA, March 2000. To appear in Computational Optimization and Applications.

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

[6] Harwell Subroutine Library. *A catalogue of subroutines (HSL 2000)*. AEA Technology, Harwell, Oxfordshire, England, 2002.

[7] J.L. Morales, D. Orban, J. Nocedal, and R. A. Waltz. A hybrid interior algorithm for nonlinear optimization. Technical Report 2003-6, Optimization Technology Center, Northwestern University, Evanston, Il, USA, June 2003.

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