HiOp – User Guide

version 0.4.1

by

Cosmin G. Petra and **Nai-Yuan Chiang**

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

7000 East Avenue,
Livermore, CA 94550, USA.

Oct 15, 2017
Updated April 07, 2021

Technical report LLNL-SM-743591

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Contents

1 Introduction

This document describes the `HiOp` suite of HPC optimization solvers for some large-scale nonconvex nonlinear programming problems (NLPs). Two main classes of optimization problems are supported. First class consists of NLPs with extremely large number of variables but with a relatively small number of general constraints; the solver for these problems is a memory-distributed, MPI-based quasi-Newton interior-point solver using limited-memory approximations for the Hessians. The second class of problems consists of NLPs that have dense and sparse blocks, for which a "Newton" interior-point solver is available toghether with a specialized, so called mixed dense-sparse (MDS) linear algebra capable of achieving good performance on host-device, *i.e.*, CPU-GPU, computing hardware.

This document includes instructions on how to obtain and build `HiOp` and a description of its interface, user options, and use as an optimization library. Guidelines on how is best to use the solver for parallel computations are also provided. The document generally targets users of `HiOp`, but also contains information relevant to potential developers or advanced users; these are strongly encouraged to also read the paper on the computational approach implemented in `HiOp` [2].

While the MPI quasi-Newton solver of `Hiop` targets DAE- and PDE-constrained optimization problems formulated in a "reduced-space" approach, it can be used for general nonconvex nonlinear optimization as well. For efficiency considerations, it is recommended to *use quasi-Newton `Hiop` for NLPs that have a relatively small number of general constraints*, say less than 100; note that there are no restrictions on the number of bounds constraints, *e.g.*, one can specify simple bounds on any, and potentially all the decision variables without affecting the computational efficiency. The minimizers computed by `HiOp` satisfies *local* first-order optimality conditions.

The goal of quasi-Newton solver of `HiOp` is to remove the parallelization limitations of existing state-of-the-art solvers for nonlinear programming (NLP) and match/surpass the parallel scalability of the underlying PDE or DAE solver. Such limitation occurs whenever the dimensionality of the optimization space is as large as the dimensionality of the discretization of the differential systems of equations governing the optimization. In these cases, the use of existing NLP solvers results in i. considerable long time spent in optimization, which affects the parallel scalability, and/or ii. memory requirements beyond the memory capacity of the computational node that runs the optimization. `HiOp` removes these scalability/parallelization bottlenecks (for certain optimization problems described above) by offering interface for a *memory-distributed* specification of the problem and parallelizing the optimization search using specialized parallel linear algebra technique.

The general computational approach in `HiOp` is to use existing state-of-the-art NLP algorithms and develop linear algebra kernels tailored to the specific of this class of problems. `HiOp` is based on an interior-point line search filter method [4, 5] and follows the implementation details from [6], which is the implementation paper for IPOPT open-source NLP solver. The quasi-Newton approach is based on limited-memory secant approximations of the Hessian [1], which is generalized as required by the specific of interior-point methods for constrained optimization problems [2]. The specialized linear algebra decomposition is obtained by using a Schur-complement reduction that leverages the fact that the quasi-Newton Hessian matrix has a small number of dense blocks that border a low-rank update of a diagonal matrix. The technique is described in [2]. The Newton interior-point solver of `HiOp` uses linear algebra specialized to the particular form of the MDS NLPs supported by this solver, for more details consult Section 3.3.

The C++ parallel implementation in `HiOp` is lightweight and portable since it is expressed

and implemented only in terms of parallel (multi-)vector operations (implemented internally using BLAS level 1 and level 2 operations and MPI for communication) and BLAS level 3 and LAPACK operations for small dense matrices.

By using multithreadead BLAS and LAPACK libraries, *e.g.*, INTEL MKL, GotoBlas, Atlas, etc, additional, intra-node parallelism can be achieved. These libraries are usually machine/hardware specific and available for a variety of computer architectures. A list of BLAS/LAPACK implementations can be found at `https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms#Implementations`.

2 Installing/building `HiOp`

`HiOp` is available on Lawrence Livermore National Laboratory (LLNL) github's page at `https://github.com/LLNL/hiop`. `HiOp` can be obtained by cloning the repository or by downloading the release archive(s). To clone from the repository, one needs to simply run

```
> git clone https://github.com/LLNL/hiop.git
```

2.1 Prerequisites

`HiOp` is written in C++ (C++98) and requires a C++ compiler. Also it requires BLAS and LAPACK, and, optionally, MPI. Not having MPI enabled in `HiOp` results in `HiOp` running on one processor only; still, significant multi-core parallelization can be obtained by using multithreaded BLAS and LAPACK. The CMake-based build system of `HiOp` generally detects these prerequisites automatically.

At this point the build system only supports macOS and Linux operating systems. On the other hand, other than the build system, `HiOp`'s code is platform independent and should run fine on Windows as well.

2.2 Building, testing, and installing `HiOp`

The build system is based on CMake. Up-to-date detailed information about `HiOp` custom builds and installs are kept at `https://github.com/LLNL/hiop`.

A quick way to build and code is run the following commands in the 'build/' directory in the root `HiOp` directory:

```
> cmake ..
> make all
> make test
> make install
```

This will compile, build the static library and example executables, perform a couple of tests to detect potential issues during the installation, and will install `HiOp`'s header and the static library in the root directory under '_build_defaultDist/'

2.3 Support of host-device computations using (generic)CPU-(NVIDIA)GPU hardware

Starting version 0.3, `HiOp` offers support for offloading computations to NVIDIA GPU accelerators. This feature is available only when solving NLPs in the mixed dense-sparse (MDS) form. First,

support for GPUs should be enabled during the build by using `-DHIOP_USE_GPU` option with `cmake`, which will result in using the GPU accelerators for the internal linear solves; in addition, the options `-DHIOP_USE_RAJA` and `-DHIOP_USE_UMPIRE` will employ RAJA portability abstraction to perform the remaining linear algebra computations on the device or on the host (with OpenMP acceleration).

`HiOp`'s `cmake` build system is quite versatile to find the dependencies required to offload computations to the device GPUs since was developed and tested on a few GPU-enabled HPC platforms at Oak Ridge, Lawrence Livermore, and Pacific Northwest National Laboratories. These dependencies consist of CUDA library version 10.1 or later and a recent Magma linear solver library (as well as a physical NVIDIA GPU device). `HiOp` offers an extensive build support for using customized NVIDIA libraries and/or Magma solver as well as for advanced troubleshooting. The user is referred to `cmake/FindHiopCudaLibraries.cmake` and `cmake/FindMagma.cmake` scripts.

⚠ **Note:** Installing NVIDIA CUDA (and likely the NVIDIA driver) and/or building Magma can be quite challenging. The user is encouraged to rely on preinstalled versions of these, as they are available via `module` utility on virtually all high-performance computing machines. An example of how to satisfy all the GPU dependencies on Summit supercomputer at Oak Ridge National Lab with a one commands are available at `https://github.com/LLNL/hiop/blob/master/README_summit.md`.

2.4 Building extra features

To build the documentation for `HiOp`, enable the `HIOP_BUILD_DOCUMENTATION` option when configuring. This option can only be enabled if a `doxygen` executable is available in the path. This option adds the `make` targets `doc` and `install_doc` which build and install the documentation respectively. When installed, `html` and LaTeX/pdf documentation may be found under `<install prefix>/doc/html` and `<install prefix>/doc/html`, respectively.

To build every configuration of HiOp for testing purposes, the build script has an option `./BUILD.sh --full-build-matrix`. See the testing section of `README_developers.md` for more information.

Additional `HiOp` features not yet mentioned may be found in the top of the top-level `CMakeLists.txt` file with a brief description.

3 Interfacing with `HiOp`

Once `HiOp` is built, it can be used as the optimization solver in your application through the `HiOp`'s C++ interfaces and by linking with the static library. A shared dynamic load library can be also built using `HIOP_BUILD_SHARED` option with cmake. There are three types of nonlinear optimization or NLP formulations currently supported by `HiOp`. They are described and discussed by the subsequent sections.

3.1 The NLP with *dense* constraints formulation requiring *up to first-order* derivative information

A first class of problems supported by `HiOp` consists of nonlinear nonconvex NLP with *dense* constraints of the form

$$\min_{x\in\mathbb{R}^n} \quad f(x) \tag{1}$$

$$\text{s.t.} \quad c(x) = c_E \qquad [y_c] \tag{2}$$

$$[v_l] \qquad d_l \le d(x) \le d_u \qquad [v_u] \tag{3}$$

$$[z_u] \qquad x_l \le x \le x_u \qquad [z_u] \tag{4}$$

Here $f : \mathbb{R}^n \to \mathbb{R}$, $c : \mathbb{R}^n \to \mathbb{R}^{m_E}$, $d : \mathbb{R}^n \to \mathbb{R}^{m_I}$. The bounds appearing in the inequality constraints (3) are assumed to be $d^l \in \mathbb{R}^{m_I} \cup \{-\infty\}$, $d^u \in \mathbb{R}^{m_I} \cup \{+\infty\}$, $d^l_i < d^u_i$, and at least of one of d^l_i and d^u_i are finite for all $i \in \{1, \ldots, m_I\}$. The bounds in (4) are such that $x^l \in \mathbb{R}^n \cup \{-\infty\}$, $x^u \in \mathbb{R}^n \cup \{+\infty\}$, and $x^l_i < x^u_i$, $i \in \{1, \ldots, n\}$. The quantities insides brackets are the Lagrange multipliers of the constraints. Whenever a bound is infinite, the corresponding multiplier is by convention zero.

The following quantities are required by `HiOp`:

D1 objective and constraint functions $f(x)$, $c(x)$, $d(x)$;

D2 the first-order derivatives of the above: $\nabla f(x)$, $Jc(x)$, $Jd(x)$;

D3 the simple bounds x_l and x_u, the inequalities bounds: d_l and d_u, and the right-hand size of the equality constraints c_E.

3.1.1 The C++ interface

The above optimization problem (1)-(4) can be specified by using the C++ interface, namely by deriving and providing an implementation for the `hiop::hiopInterfaceDenseConstraints` abstract class.

We present next the methods of this abstract class that needs to be implemented in order to specify the parts D1-D3 of the optimization problem.

⚠ **Note:** All the functions that return `bool` should return `false` when an error occurs, otherwise should return `true`.

3.1.2 Specifying the optimization problem

All the methods of this section are "pure" virtual in `hiop::hiopInterfaceDenseConstraints` abstract class and need to be provided by the user implementation.

```
1 bool get_prob_sizes(long long& n, long long& m);
```

Provides the number of decision variables and the number of constraints ($m = m_E + m_I$).

```
1 bool get_vars_info(const long long& n, double *xlow, double* xupp,
2                    NonlinearityType* type);
```

Provides the lower and upper bounds x_l and x_u on the decision variables. When a variable (let us say the i^{th}) has no lower or/and upper bounds, the i^{th} entry of xlow and/or xupp should be less than $-1e20$ or/and larger than $1e20$, respectively. The last argument is not used and can set to any value of the enum `hiop::hiopInterfaceDenseConstraints::NonlinearityType`.

```
bool get_cons_info(const long long& m, double* clow, double* cupp,
                   NonlinearityType* type);
```

Similar to the above, but for the inequality bounds d_l and d_u. For equalities, set the corresponding entries in clow and cupp equal to the desired value (from c_E).

```
bool eval_f(const long long& n,
            const double* x, bool new_x,
            double& obj_value);
```

Implement this method to compute the function value $f(x)$ in `obj_value` for the provided decision variables x. The input argument `new_x` specifies whether the variables x have been changed since the previous call of one of the `eval_` methods. Use this argument to "buffer" the objective and gradients function and derivative evaluations when this is possible.

```
bool eval_grad_f(const long long& n,
                 const double* x, bool new_x,
                 double* gradf);
```

Same as above but for $\nabla f(x)$.

```
 bool eval_cons(const long long& n, const long long& m,
                const long long& num_cons,
                const long long* idx_cons, const double* x,
                bool new_x, double* cons);
```

Implement this method to provide the value of the constraints $c(x)$ and/or $d(x)$. The input parameter `num_cons` specifies how many constraints (out of `m`) needs to evaluated; `idx_cons` array specifies the indexes, which are zero-based, of the constraints and is of size `num_cons`. These values should be provided in `cons`, which is also an array of size `num_cons`.

```
bool
eval_Jac_cons(const long long& n, const long long& m,
              const long long& num_cons, const long long* idx_cons,
              const double* x, bool new_x,
              double* Jac);
```

Implement this method to provide the Jacobian of a subset of the constraints $c(x)$ and/or $d(x)$ in `Jac`; as for `eval_cons`, this subset is specified by the array of row indexes `idx_cons`. The array `Jac` should contain the Jacobian row-wise, meaning that the each row of the Jacobian is contiguous in memory and starts right after the previous row.

3.1.3 Specifying the inter-process/memory distribution of the problem

`HiOp` uses *data parallelism*, meaning that the data [D1]-[D3] of the optimization problem is distributed across processes (MPI ranks). It is **crucial** to understand the data distribution scheme in order to use `HiOp`'s interface properly.

The general rule of thumb is to distribute any data of the problem with storage depending on n, namely the decision variables x and their bounds x_l and x_u, the gradient $\nabla f(x)$, and the Jacobians $Jc(x)$ and $Jd(x)$. The Jacobians, which are assumed to be dense matrices with n columns, are distributed column-wise.

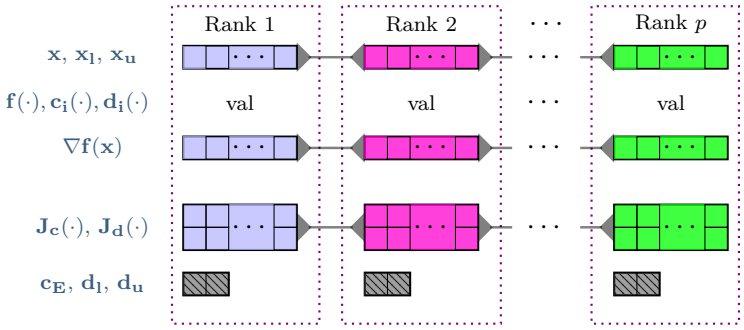

Figure 1: Depiction of the distribution of the data of the optimization problem (1)-(4) across MPI ranks. The vectors and matrices with storage dependent on the number of optimization variables are distributed. Other data, *i.e.*, scalar function values or vectors of small size (shown in dashed dark grey boxes), are replicated on each rank.

⚠ **Note:** All the `eval_` functions of the C++ interface provides local array slices of the above mentioned distributed data to the application code that implements `HiOp`'s C++ interface. The size of these local slices is the "local size" (specified by the application code through the `get_vecdistrib_info` method explained below) and is different from the "global size" n and parameter `n` of methods.

⚠ **Note:** Since the Jacobians are distributed column-wise, the implementer should populate the `Jac` argument of `eval_Jac_cons` with the "local" columns.

On the other hand, the problem's data that does not have storage depending on n, is not distributed; instead, it is replicated on all ranks. Such data consist of c_E, d_l, d_u and the evaluations of $c(x)$ and $d(x)$.

```
1 bool get_MPI_comm(MPI_Comm& comm_out) ;
```

Use this method to specify the MPI communicator to be used by `HiOp`. It has a default implementation that will provide `MPI_COMM_WORLD`.

```
1 bool get_vecdistrib_info(long long global_n, long long* cols);
```

Use this method to specify the data distribution of the data of the problem that has storage depending on n. `HiOp` will call the implementation of this method to obtain the partitioning/distribution of an hypothetical vector of size `global_n` across the MPI ranks. The array `cols` is of dimension number of ranks plus one and should be populated such that `cols[r]` and `cols[r+1]-1`

specify the start and end indexes of the slice stored on rank r in the hypothetical vector. It has a default implementation that will returns `false`, indicating that `HiOp` should run in serial.

⚠ **Note:** `HiOp` also uses `get_vecdistrib_info` to obtain the information about the Jacobians' distribution across MPI ranks (this is possible since they are column-wise distributed).

Examples of how to use these functions can be found in the standalone drivers in `src/Drivers/` under the `HiOp`'s root directory.

3.1.4 Calling `HiOp` for a `hiopInterfaceDenseConstraints` formulation

Once an implementation of the `hiop::hiopInterfaceDenseConstraints` abstract interface class containing the user's NLP representation is available, the user code needs to create a `HiOp` problem formulation that encapsulate the NLP representation, instantiate an optimization algorithm class, and start the numerical optimization process. Assuming that the NLP representation is implemented in a class named `NlpEx1` (deriving `hiop::hiopInterfaceDenseConstraints`), the aforementioned sequence of steps can be performed by:

```
#include "NlpEx1.hpp"                //the NLP representation class
#include "hiopInterface.hpp"    //HiOP encapsulation of the NLP
#include "hiopAlgFilterIPM.hpp"      //solver class
using namespace hiop;
...
NlpEx1 nlp_interface();                      //instantiate your NLP representation class
hiopNlpDenseConstraints nlp(nlp_interface); //create HiOP encapsulation
nlp.options.SetNumericValue("mu0", 0.01);  //set initial value for  barrier parameter
hiopAlgFilterIPM solver(&nlp);              //create a solver object
hiopSolveStatus status = solver.run();      //numerical optimization
double obj_value = solver.getObjective();  //get objective
...
```

Various output quantities of the numerical optimization phase (*e.g.*, the optimal objective value and (primal) solution, status of the numerical optimization process, and solve statistics) can be retrieved from `HiOp`'s `hiopAlgFilterIPM` solver object. Most commonly used such methods are:

```
double getObjective() const;
void getSolution(double* x) const;
hiopSolveStatus getSolveStatus() const;
int getNumIterations() const;
```

The standalone drivers `nlpDenseCons_ex1`, `nlpDenseCons_ex2`, and `nlpDenseCons_ex3` inside directory `src/Drivers/` under the `HiOp`'s root directory contain more detailed examples of the use of `HiOp`.

3.2 General sparse NLPs requiring *up to second-order* derivative information

The sparse NLP formulation supports sparse optimization problems and requires Hessians of the objective and constraints in addition to gradients/Jacobian of the objective/constraints.

$$\min_{x\in\mathbb{R}^n} \quad f(x) \tag{5}$$

$$\text{s.t.} \quad c(x) = c_E \qquad [y_c] \tag{6}$$

$$[v_l] \qquad d_l \le d(x) \le d_u \qquad [v_u] \tag{7}$$

$$[z_u] \qquad x_l \le x \le x_u \qquad [z_u] \tag{8}$$

Here $f : \mathbb{R}^n \to \mathbb{R}$, $c : \mathbb{R}^n \to \mathbb{R}^{m_E}$, $d : \mathbb{R}^n \to \mathbb{R}^{m_I}$. The bounds appearing in the inequality constraints (3) are assumed to be $d^l \in \mathbb{R}^{m_I} \cup \{-\infty\}$, $d^u \in \mathbb{R}^{m_I} \cup \{+\infty\}$, $d^l_i < d^u_i$, and at least of one of d^l_i and d^u_i are finite for all $i \in \{1, \ldots, m_I\}$. The bounds in (4) are such that $x^l \in \mathbb{R}^n \cup \{-\infty\}$, $x^u \in \mathbb{R}^n \cup \{+\infty\}$, and $x^l_i < x^u_i$, $i \in \{1, \ldots, n\}$. The quantities insides brackets are the Lagrange multipliers of the constraints. Whenever a bound is infinite, the corresponding multiplier is by convention zero.

The following quantities are required by `HiOp`:

D1 objective and constraint functions $f(x)$, $c(x)$, $d(x)$;

D2 the first-order derivatives of the above: $\nabla f(x)$, $Jc(x)$, $Jd(x)$;

D3 The Hessian of the Lagrangian

$$\nabla^2 L(x) = \lambda_0 \nabla^2 f(x) + \sum_{i=1}^{m_E} \lambda^E_i \nabla^2 c_i(x) + \sum_{i=1}^{m_I} \lambda^I_i \nabla^2 d_i(x). \tag{9}$$

D4 the simple bounds x_l and x_u, the inequalities bounds: d_l and d_u, and the right-hand size of the equality constraints c_E.

3.2.1 C++ interface to solve sparse NLPs

The above optimization problem (5)-(8) can be specified by using the C++ interface, namely by deriving and providing an implementation for the `hiop::hiopInterfaceSparse` abstract class.

We present next the methods of this abstract class that needs to be implemented in order to specify the parts D1-D4 required to solve a sparse NLP problem.

⚠ **Note:** All the functions that have a `bool` return type should return `false` when an error occurs, otherwise should return `true`.

⚠ **Note:** `hiop::hiopInterfaceSparse` runs only in non-distributed/non-MPI mode. Intraprocess acceleration can be obtained using OpenMP or CUDA.

3.2.2 Specifying the optimization problem

All the methods of this section are "pure" virtual in `hiop::hiopInterfaceSparse` abstract class and need to be provided by the user implementation.

```
1 bool get_prob_sizes(long long& n, long long& m);
```

Provides the number of decision variables and the number of constraints ($m = m_E + m_I$).

```
bool get_vars_info(const long long& n, double *xlow, double* xupp,
                   NonlinearityType* type);
```

Provides the lower and upper bounds x_l and x_u on the decision variables. When a variable (let us say the i^{th}) has no lower or/and upper bounds, the i^{th} entry of xlow and/or xupp should be less than $-1e20$ or/and larger than $1e20$, respectively. The last argument is not used and can set to any value of the enum `hiop::hiopInterface::NonlinearityType`.

```
bool get_cons_info(const long long& m, double* clow, double* cupp,
                   NonlinearityType* type);
```

Similar to the above, but for the inequality bounds d_l and d_u. For equalities, set the corresponding entries in clow and cupp equal to the desired value (from c_E).

```
bool eval_f(const long long& n,
            const double* x, bool new_x,
            double& obj_value);
```

Implement this method to compute the function value $f(x)$ in `obj_value` for the provided decision variables x. The input argument `new_x` specifies whether the variables x have been changed since the previous call of one of the `eval_` methods. Use this argument to "buffer" the objective and gradients function and derivative evaluations when this is possible.

```
bool eval_grad_f(const long long& n,
                 const double* x, bool new_x,
                 double* gradf);
```

Same as above but for $\nabla f(x)$.

```
 bool eval_cons(const long long& n, const long long& m,
                const long long& num_cons,
                const long long* idx_cons, const double* x,
                bool new_x, double* cons);
```

Implement this method to provide the value of the constraints $c(x)$ and/or $d(x)$. The input parameter `num_cons` specifies how many constraints (out of `m`) needs to evaluated; `idx_cons` array specifies the indexes, which are zero-based, of the constraints and is of size `num_cons`. These values should be provided in `cons`, which is also an array of size `num_cons`.

```
bool
eval_Jac_cons(const long long& n, const long long& m,
              const long long& num_cons, const long long* idx_cons,
              const double* x, bool new_x,
              const int& nnzJacS, int* iJacS, int* jJacS, double* MJacS);
```

Implement this method to provide the Jacobian of a subset of the constraints $c(x)$ and/or $d(x)$ in `Jac`; this subset is specified by the array `idx_cons`. The last three arguments should be used to

specify the Jacobian information in sparse triplet format. `iJacS` and `jJacS` needs to be jointly sorted: by indexes in `iJacS` and, for equal (row) indexes in `iJacS`, by indexes in `jJacS`.

Notes for implementer of this method:

2. When `iJacS` and `jJacS` are non-null, the implementer should provide the (i, j) indexes in these arrays.
3. When `MJacS` is non-null, the implementer should provide the values corresponding to entries specified by `iJacS` and `jJacS`.
4. `iJacS` and `jJacS` are both either non-null or null during the same call.
5. The pair (`iJacS`, `jJacS`) and `MJacS` can be both non-null during the same call or only one of them non-null; but they will not be both null.

```
bool eval_Jac_cons(const long long& n, const long long& m,
                   const double* x, bool new_x,
                   const int& nnzJacS, int* iJacS, int* jJacS, double* MJacS);
```

Evaluates the Jacobian of equality and inequality constraints *in one call*.

⚠ **Note:** HiOp will call this method whenever the implementer/user returns `false` from the previous, "two-calls" `eval_Jac_cons`. We remark that the two-calls method should return `false` during both calls (for equalities and inequalities) made to it by `HiOp` in order to let `HiOp` know that the Jacobian should be evaluated using the one-call callback listed above.

The main difference from the above `eval_Jac_cons` is that the implementer/user of this method does not have to split the constraints into equalities and inequalities; instead, HiOp does this internally.

Parameters:

- first four: number of variables, number of constraints, (primal) variables at which the Jacobian should be evaluated, and boolean flag indicating whether the variables `x` have changed since a previous call to any of the function and derivative evaluations.
- `nnzJacS`, `iJacS`, `jJacS`, `MJacS`: number of nonzeros, (i, j) indexes, and nonzero values of the sparse Jacobian matrix. `iJacS` and `jJacS` needs to be jointly sorted: by indexes in `iJacS` and, for equal (row) indexes in `iJacS`, by indexes in `jJacS`.

⚠ **Note:** Notes 1-5 from the previous, two-call `eval_Jac_cons` applies here as well.

```
bool eval_Hess_Lagr(const long long& n, const long long& m,
                    const double* x, bool new_x, const double& obj_factor,
                    const double* lambda, bool new_lambda,
                    const long long& nsparse, const long long& ndense,
                    const int& nnzHSS, int* iHSS, int* jHSS, double* MHSS)
```

Evaluates the Hessian of the Lagrangian function as a sparse matrix in triplet format.

⚠ **Note:** Notes 1-5 from `eval_Jac_cons` apply to arrays `iHSS`, `jHSS`, and `MHSS` that stores the sparse part of the Hessian.

⚠ **Note:** The array `lambda` contains first the multipliers of the equality constraints followed by the multipliers of the inequalities.

3.2.3 Calling HiOp for a hiopInterfaceSparse formulation

Once the sparse NLP is coded, the user code needs to create a HiOp problem formulation that encapsulate the NLP representation, instantiate an optimization algorithm class, and start the numerical optimization process. Assuming that the NLP representation is implemented in a class named NlpEx6 (that derives from hiop::hiopInterfaceSparse), the aforementioned sequence of steps can be performed by:

```
#include "NlpEx6.hpp"                //the NLP representation class
#include "hiopInterface.hpp"    //HiOP encapsulation of the NLP
#include "hiopAlgFilterIPM.hpp"      //solver class
using namespace hiop;
...
NlpEx6 nlp_interface();                      //instantiate your NLP representation class
hiopNlpDenseConstraints nlp(nlp_interface); //create HiOP encapsulation
nlp.options.SetNumericValue("mu0", 0.01);  //set a non-default initial value for barrier parameter
hiopAlgFilterIPM solver(&nlp);              //create a solver object
hiopSolveStatus status = solver.run();      //numerical optimization
double obj_value = solver.getObjective();  //get objective
...
```

Various output quantities of the numerical optimization phase (*e.g.*, the optimal objective value and (primal) solution, status of the numerical optimization process, and solve statistics) can be retrieved from HiOp's hiopAlgFilterIPM solver object. Most commonly used such methods are:

```
double getObjective() const;
void getSolution(double* x) const;
hiopSolveStatus getSolveStatus() const;
int getNumIterations() const;
```

The standalone drivers nlpSparse_ex6 and nlpSparse_ex7 inside directory src/Drivers/ under the HiOp's root directory contain more detailed examples of the use of the sparse NLP interface of HiOp.

3.3 NLPs in the mixed dense-sparse (MDS) form

A second class of optimization problems supported by HiOp consists of nonlinear, possibly non-convex optimization problems that explicitly partition the optimization variables into so-called "dense" and "sparse" variables, x_d and x_s, respectively; this problem can be expressed compactly as

$$\min_{x_d \in \mathbb{R}^{n_d}, x_s \in \mathbb{R}^{n_s}} \quad f(x_d, x_s) \tag{10}$$

$$\text{s.t.} \quad c(x_d, x_s) = c_E, \tag{11}$$

$$d^l \le d(x_d, x_s) \le d^u, \tag{12}$$

$$x_d^l \le x_d \le x_d^u,\ x_s^l \le x_s \le x_s^u. \tag{13}$$

Here $f : \mathbb{R}^n \to \mathbb{R}$, $c : \mathbb{R}^n \to \mathbb{R}^{m_E}$, and $d : \mathbb{R}^n \to \mathbb{R}^{m_I}$, where n denotes the total number of variables, $n = n_d + n_s$. The bounds appearing in the inequality constraints (12) are assumed

to be $d^l \in \mathbb{R}^{m_I} \cup \{-\infty\}$, $d^u \in \mathbb{R}^{m_I} \cup \{+\infty\}$, $d^l_i < d^u_i$, and at least of one of d^l_i and d^u_i are finite for each $i \in \{1, \ldots, m_I\}$. The vector bounds x^l_d, x^u_d, x^l_s, and x^u_s in (13) need to satisfy identical requirements. For the rest of the paper m will denote $m_E + m_I$, *i.e.*, the total number of constraints excepting the simple bounds constraints (13).

The salient idea behind mixed dense-sparse problems of the form (10)-(13) is that the explicit partitioning of the optimization variables and a couple of (block) structural properties of the functions $f(\cdot)$, $c(\cdot)$, and $d(\cdot)$, which are elaborated below, allow orchestrating the computations of the optimization algorithm to heavily rely on matrix and vector *dense* kernels and to reduce the reliance on sparse linear algebra kernels.

As mentioned above we make a couple of assumptions on the block structure of the derivatives:

A1. The "cross-term" Hessian matrices $\nabla^2_{x_d x_s} f$, $\nabla^2_{x_s x_d} f$, $\nabla^2_{x_d x_s} c$, $\nabla^2_{x_s x_d} c$, $\nabla^2_{x_d x_s} d$, and $\nabla^2_{x_s x_d} d$ are zero;

A2. The Hessian matrix $\nabla^2_{x_s x_s} L$ has a sparsity pattern that allows *computationally efficient* inversion of (or solving with) the matrix $\nabla^2_{x_s x_s} L + D_{x_s}$ where D_{x_s} is a diagonal matrix with positive diagonal entries; in our target applications, namely, optimal power flow problems, $\nabla^2_{x_s x_s} L$ is a diagonal matrix with nonnegative entries.

The optimization problem (10)–(13) is transformed internally by `HiOp` to an equivalent form that is more amenable to the use of interior-point methods as described on [3, Section 3]. Furthermore, HiOp implements the filter line-search interior-point algorithm of Wächter and Biegler [5, 4] (also implemented by IPOPT [6]) and makes explicit use of second-order derivatives/Hessians.

Starting version 0.3, `HiOp` offers support for NVIDIA GPU computations acceleration. This feature is available only when solving NLPs in the mixed dense-sparse (MDS) form and should be enabled during the build by using `-DHIOP_USE_GPU` option with `cmake`. `HiOp`'s `cmake` build system is quite versatile to find the dependencies required to offload computations to the device GPUs since was developed and tested on a few GPU-enabled HPC platforms at Oak Ridge, Lawrence Livermore, and Pacific Northwestern National Laboratories. These dependencies consist of CUDA library version 10.1 or later and a recent Magma linear solver library (as well as an NVIDIA GPU). If offloading computations to the device is not desired, the user can switch it off and perform only CPU (host) computations by setting `HiOp`'s option `compute_mode` to `cpu`. The Newton interior-point solver for MDS problems offers ample device/GPU (and limited CPU/multicore) fine-grain parallelism, however it does not offer support of interprocess/internode parallelism.

The following quantities are required by `HiOp`:

D1 objective and constraint functions $f(x_d, x_s)$, $c(x_d, x_s)$, $d(x_d, x_s)$;

D2 the first-order derivatives: $\nabla f(x_d, x_s)$, $Jc(x_d, x_s)$, $Jd(x_d, x_s)$; the two Jacobians will have a MDS structure in the sense that the left blocks will be dense while the right blocks will be sparse in their expressions

$$Jc(x_d, x_s) = \begin{bmatrix} J_{x_d} c(x_d, x_s) & J_{x_s} c(x_d, x_s) \end{bmatrix} \tag{14}$$

and

$$Jd(x_d, x_s) = \begin{bmatrix} J_{x_d} d(x_d, x_s) & J_{x_s} d(x_d, x_s) \end{bmatrix}. \tag{15}$$

`HiOp` does not track MDS structure within the gradient $\nabla f(x_d, x_s)$ and treats it as an unstructured vector.

D3 the second-order derivatives in the form of the Hessian of the Lagrangian

$$\nabla^2 L(x_d, x_s) = \lambda_0 \nabla^2 f(x_d, x_s) + \sum_{i=1}^{m_E} \lambda_i^E \nabla^2 c_i(x_d, x_s) + \sum_{i=1}^{m_I} \lambda_i^I \nabla^2 d_i(x_d, x_s). \tag{16}$$

We remark that $\nabla^2 L(x_d, x_s)$ has a so-called MDS structure in the sense that $\nabla^2_{x_d^2} L(x_d, x_s)$ is dense, $\nabla^2_{x_s^2} L(x_d, x_s)$ is sparse, and $\nabla^2_{x_d x_s} L(x_d, x_s)$ and $\nabla^2_{x_s x_d} L(x_d, x_s)$ are zero; this is a consequence of the assumptions A1 and A2 above,

D4 the simple bounds x_l and x_u, the inequalities bounds: d_l and d_u, and the right-hand size of the equality constraints c_E.

3.3.1 The C++ interface

The above optimization problem (10)–(13) can be specified by using the C++ interface, namely by deriving and providing an implementation for the `hiop::hiopInterfaceMDS` abstract class.

We present next the methods of this abstract class that needs to be implemented in order to specify the parts D1-D4 of the optimization problem. All the methods of this section are "pure" virtual in `hiop::hiopInterfaceMDS` abstract class and need to be provided by the user implementation.

⚠ **Note:** Unless stated otherwise, all the functions that return `bool` should return `false` when an error occurs, otherwise should return `true`.

```
bool get_prob_sizes(long long& n, long long& m);
```

Provides the number of decision variables and the number of constraints ($m = m_E + m_I$).

```
bool get_vars_info(const long long& n, double *xlow, double* xupp,
                   NonlinearityType* type);
```

Provides the lower and upper bounds x_l and x_u on the decision variables. When a variable (let us say the i^{th}) has no lower or/and upper bounds, the i^{th} entry of xlow and/or xupp should be less than $-1e20$ or/and larger than $1e20$, respectively. The last argument is not used and can set to any value of the enum `hiop::hiopInterfaceDenseConstraints::NonlinearityType`.

```
bool get_cons_info(const long long& m, double* clow, double* cupp,
                   NonlinearityType* type);
```

Similar to the above, but for the inequality bounds d_l and d_u. For equalities, set the corresponding entries in clow and cupp equal to the desired value (from c_E).

```
bool get_sparse_dense_blocks_info(int& nx_sparse, int& nx_dense,
                                  int& nnz_sparse_Jaceq,
                                  int& nnz_sparse_Jacineq,
                                  int& nnz_sparse_Hess_Lagr_SS,
                                  int& nnz_sparse_Hess_Lagr_SD);
```

Specifies the number of nonzero elements in the *sparse blocks* of the Jacobians of the constraints and of the Hessian of the Lagrangian, see (15) and (16), respectively. The last parameter `nnz_sparse_Hess_Lagr_SD` is not used momentarily and should be set to zero.

```
bool eval_f(const long long& n,
            const double* x, bool new_x,
            double& obj_value);
```

Implement this method to compute the function value $f(x)$ in `obj_value` for the provided decision variables x. The input argument `new_x` specifies whether the variables x have been changed since the previous call of one of the `eval_` methods. Use this argument to "buffer" the objective and gradients function and derivative evaluations when this is possible.

```
bool eval_grad_f(const long long& n,
                 const double* x, bool new_x,
                 double* gradf);
```

Same as above but for $\nabla f(x)$.

```
bool eval_cons(const long long& n, const long long& m,
               const long long& num_cons,
     const long long* idx_cons, const double* x,
     bool new_x, double* cons);
```

Implement this method to provide the value of the constraints $c(x)$ and/or $d(x)$. The input parameter `num_cons` specifies how many constraints (out of `m`) needs to evaluated; `idx_cons` array specifies the indexes, which are zero-based, of the constraints and is of size `num_cons`. These values should be provided in `cons`, which is also an array of size `num_cons`.

```
eval_Jac_cons(const long long& n, const long long& m,
              const long long& num_cons, const long long* idx_cons,
              const double* x, bool new_x,
              const long long& nsparse, const long long& ndense,
              const int& nnzJacS, int* iJacS, int* jJacS, double* MJacS,
              double* JacD);
```

Evaluates the Jacobian of constraints split in the sparse (triplet format) and dense submatrices (row-wise contiguous memory storage). The methods is called by `HiOp` twice once for equalities and once for inequalities and passes during each of these calls the `idx_cons` array of the indexes of equalities and inequalities in the whole body of constraints.

It is advantageous to provide this method when the underlying NLP's constraints come naturally split in equalities and inequalities. When this is not convenient to do so, use `eval_Jac_cons` below.

Parameters:

- first six: see `eval_cons`.
- `nnzJacS`, `iJacS`, `jJacS`, `MJacS` are for number of nonzeros, (i, j) indexes, and nonzero values of the sparse Jacobian.
- `JacD` should contain the Jacobian with respect to the dense variables of the MDS problem. The array should store this Jacobian submatrix row-wise, meaning that the each row of the Jacobian is contiguous in memory and starts right after the previous row.

⚠ **Note:** When implementing this method one should be aware that:

1. `JacD` parameter will be always non-null
2. When `iJacS` and `jJacS` are non-null, the implementer should provide the (i, j) indexes in these arrays.
3. When `MJacS` is non-null, the implementer should provide the values corresponding to entries specified by `iJacS` and `jJacS`.
4. `iJacS` and `jJacS` are both either non-null or null during a call.
5. The pair (`iJacS`, `jJacS`) and `MJacS` can be both non-null during the same call or only one of them non-null; but they will not be both null.

```
bool eval_Jac_cons(const long long& n, const long long& m,
                   const double* x, bool new_x,
                   const long long& nsparse, const long long& ndense,
                   const int& nnzJacS, int* iJacS, int* jJacS, double* MJacS,
                   double* JacD);
```

Evaluates the Jacobian of equality and inequality constraints *in one call*. This Jacobian is mixed dense-sparse (MDS), which means is structurally split in the sparse (triplet format) and dense matrices (contiguous rows storage)

⚠ **Note:** HiOp will call this method whenever the implementer/user returns `false` from the previous, two-calls `eval_Jac_cons`; we remark that this method should return `false` during both calls (for equalities and inequalities) made to it by `HiOp`.

The main difference from the above `eval_Jac_cons` is that the implementer/user of this method does not have to split the constraints into equalities and inequalities; instead, HiOp does this internally.

Parameters:

- first four: number of variables, number of constraints, (primal) variables at which the Jacobian should be evaluated, and boolean flag indicating whether the variables `x` have changed since a previous call to any of the function and derivative evaluations.
- `nsparse` and `ndense`: number of sparse and dense variables, respectively, adding up to `n`.
- `nnzJacS`, `iJacS`, `jJacS`, `MJacS`: number of nonzeros, (i, j) indexes, and nonzero values of the sparse Jacobian block; these indexes are within the sparse Jacobian block (not within the entire Jacobian).
- `JacD`: dense Jacobian block as a contiguous array storing the matrix by rows.

⚠ **Note:** Notes 1-5 from the previous, two-call `eval_Jac_cons` applies here as well.

```
bool eval_Hess_Lagr(const long long& n, const long long& m,
                    const double* x, bool new_x, const double& obj_factor,
                    const double* lambda, bool new_lambda,
                    const long long& nsparse, const long long& ndense,
                    const int& nnzHSS, int* iHSS, int* jHSS, double* MHSS,
                    double* HDD,
                    int& nnzHSD, int* iHSD, int* jHSD, double* MHSD);
```

Evaluates the Hessian of the Lagrangian function in three structural blocks given by the MDS structure of the problem. The arguments `nnzHSS`, `iHSS`, `jHSS`, and `MHSS` hold $\nabla^2 L(x_s, x_s)$ from (16). The argument `HDD` stores $\nabla^2 L(x_d, x_d)$ from (16).

⚠ **Note:** The last four arguments, which are supposed to store the cross-Hessian $\nabla^2 L(x_s, x_d)$ from (16), are for now assumed to hold a zero matrix. The implementer should return `nnzHSD=0` during the first call to `eval_Hess_Lagr`. On subsequent calls, `HiOp` will pass the sparse triplet `HSD` arrays set to `NULL` and the implementer (obviously) should not use them.

⚠ **Note:** Notes 1-5 from `eval_Jac_cons` apply to arrays `iHSS`, `jHSS`, and `MHSS` storing the sparse part of the Hessian as well as to the `HDD` array storing the dense block of the Hessian.

⚠ **Note:** The rule of thumb is that when specifying *symmetric* matrices to `HiOp`, only the *upper triangle elements* should be specified by the user. The rule applies both to sparse and dense matrices. More info on `HiOp`'s conventions on matrices storage can be found at `https://github.com/LLNL/hiop/tree/develop/src/LinAlg`.

⚠ **Note:** The array `lambda` contains the multipliers of constraints. These multipliers come have the same order as the constraints in `eval_cons` (this is a new behavior introduced in `HiOp` v0.4).

3.3.2 Calling `HiOp` for a `hiopInterfaceMDS` formulation

Once an implementation of the `hiop::hiopInterfaceMDS` abstract interface class containing the user's NLP representation is available, the user code needs to create a `HiOp` problem formulation that encapsulate the NLP representation, instantiate an optimization algorithm class, and start the numerical optimization process.

A detailed, self-contained example can be found in `src/Drivers/` directory in `nlpMDS_ex4_driver.cpp` files for an illustration of aforementioned sequence of steps. A synposis of `HiOp` code that solves and MDS NLP implemented presumably in a class `Ex4` (implemented in `nlpMDSForm_ex4.hpp`) derived from `hiop::hiopInterfaceMDS` is as follows:

```
#include "nlpMDSForm_ex4.hpp"                //the NLP representation class
#include "hiopInterface.hpp"    //HiOP encapsulation of the NLP
#include "hiopAlgFilterIPM.hpp"      //solver class
using namespace hiop;
...
Ex4* my_nlp = new Ex4(n_sp, n_de); //instantiate your NLP representation class
hiopNlpMDS nlp(*my_nlp); //create HiOP encapsulation
nlp.options->SetStringValue("Hessian", "analytical_exact");
nlp.options->SetNumericValue("mu0", 0.01);  //set initial value for  barrier ↩
    parameter
hiopAlgFilterIPMNewton solver(&nlp);          //create a solver object
hiopSolveStatus status = solver.run();      //numerical optimization
double obj_value = solver.getObjective();  //get objective
...
```

3.4 Specifying a starting point for the optimization process

The user can provide an initial primal or primal-dual point implementing the method `get_starting_point` of the NLP specification interfaces `hiopInterfaceDenseConstraints` or `hiopInterfaceMDS`.

```
bool get_starting_point(const long long& n, const long long& m,
```

```
                          double* x0,
                          bool& duals_avail,
                          double* z_bndL0, double* z_bndU0,
                          double* lambda0);
```

A second method is offered to user to provide an initial primal starting point. This method will be soon deprecated as its functionality is a subset of the method above and should be avoided.

```
bool get_starting_point(const long long& n, double* x0);
```

Parameters:

- `n` and `m` are the number of variables and the number of constraints.
- `x0` array of values for the initial primal variables/starting point.
- `duals_avail` boolean flag expressing whether the user wishes to specifiy the a starting point for dual variables.
- `z_bndL0` and `z_bndU0` starting points for the duals of the lower and upper bounds.
- `lambda0` is an array containing the starting point for the duals of the constraints. It is allocated to have the dimension of the constraints body and the entries in `lambda0` should have the same order as the constraints body (that is equalities may be mixed with inequalities), see `eval_cons` methods; `HiOp` keeps track internally whether each value in `lambda0` is a multiplier for an equality or for an inequlity constraint.

These methods should return `true` if the user successfully provided starting values for the primal or for the primal and dual variables. If the first method above returns `false`, then `HiOp` will attempt calling the second method above. This behavior is for backward compatibility. If a starting point cannot be set by the user, both methods should return `false`. Also, we remark that the methods do not need to be implemented since default implementations returning `false` are provided by the base class; in this case, `HiOp` will use a starting point of all zeros (which is subjected to internal adjustments, see below).

⚠ **Note:** The starting point returned by the user in `x0` using the methods above is subject to internal adjustments in `HiOp` and may differ from `x0` with which the methods of the previous section are first called.

3.5 Compiling and linking your project with the `HiOp` library

`HiOp`'s build system offers `HiOp` as a static library. For a straightforward integration of `HiOp` in the user's project, one needs to

- append to the compiler's include path the location of the HiOP's headers:

  ```
  -Ihiop-dir/include
  ```

- specify `libhiop.a` to the linker, possibly adding the HiOP's library directory to the linker's libraries paths:

  ```
  -Lhiop-dir/lib -lhiop
  ```

Here, `hiop-dir` is the `HiOp`'s distribution directory (created using `HiOp`'s build system, in particular by using `make install` command).

In addition, a shared dynamic load library can be also built by using `HIOP_BUILD_SHARED` option with cmake.

4 Solver options

`HiOp` prints all the available user options and their values on the standard output.

⚠ Each option i. should be of one of types numeric/double, integer, and string; ii. has a value associated; iii. possess a range of values; and, iv. has a default value.

The user can set `HiOp`'s options in two ways:

- via `hiop.options` file that should be placed in the same directory where the application driver using `HiOp` is executed. The format of the `hiop.options` is very basic, each of its lines should contain a single pair `option_name option_value`. Lines that begin with '#' are discarded. The option value is checked to have the correct type (numeric, integer, or string) and to be in the expected range. If the checks fail, then the option is set to the default value and a warning message is displayed.
- at runtime via the `HiOp`'s API using the `options` member of the `hiop::hiopInterfaceDenseConstraints` or `hiop::hiopInterfaceMDS` classes. This object has three methods that allows the user to set options based on their types:

```
1 bool SetNumericValue(const char* name, const double& value);
2 bool SetIntegerValue(const char* name, const int& value);
3 bool SetStringValue (const char* name,  const char* value);
```

⚠ **Note:** Options set in `hiop.options` file at runtime overwrite options set at runtime via the above API.

4.1 Termination criteria and output

tolerance: maximum (absolute) NLP optimality error allowed at the optimal solution. Double values in $[10^{-14}, 0.1]$. Default value: 10^{-8}.

acceptable_tolerance: HiOp will terminate if the inf-norm of the NLP optimality residuals is below this value for `acceptable_iterations` many consecutive iterations. Double values in $[10^{-14}, 0.1]$. Default value 10^{-6}.

max_iter: maximum number of iterations. Integer values between 1 to 10^6. Default value: 3 000.

acceptable_iterations: number of iterations passing the acceptable tolerance (see **acceptable_tolerance**) after which HiOp terminates. Integer values between 1 and 10^6. Default value 15.

verbosity_level: integer between 0 and 12 specifying the verbosity of `HiOp`'s output. A value of 0 disables any output (but still outputs fatal errors). A value of 1 also outputs warnings. The value of 2 is reserved for future use. A value of 3 will also output a table with `HiOp`'s convergence metrics at each iteration. A value of 4 and higher will display additional info related to the internals of the algorithm and is generally used only for debugging/development purposes. The higher the value the more verbose the output will be.

4.2 Algorithm selection and parameters

mu0: initial log-barrier parameter mu. Double values in $[10^{-6}, 10^{3}]$. Default value: 1.0.

kappa_eps: mu is reduced when when log-bar error is below kappa_eps $\times$ mu. Double values in $[10^{-14}, 0.1]$. Default value: 10^{-8}.

kappa_mu: linear reduction coefficient for mu (eqn. (7) in [6]). Double values in $[10^{-8}, 0.999]$. Default value: 0.2.

theta_mu: exponential reduction coefficient for mu (eqn. (7) in [6]). Double values in $[1, 2]$. Default value: 1.5.

tau_min: fraction-to-the-boundary parameter used in the line-search to back-off from the boundary (eqn. (8) in [6]). Double values in $[0.9, 0.99999]$. Default value: 0.99.

eta_phi: parameter of (suff. decrease) in Armijo Rule. Double values in $[0, 0.01]$. Default value 10^{-8}.

kappa1: sufficiently-away-from-the-boundary projection parameter used in the shift of the user-provided initial point. Double values in $[10^{-8}, 0.1]$. Default value: 0.01.

kappa2: shift projection parameter used in initialization for doubly bounded variables. Double values in $[10^{-8}, 0.49999]$. Default value: 0.01.

smax: the primal-dual IPM equations are rescaled when the average value of the is larger than this threshold value. Double values in $[1, 10^{7}]$. Default value: 100.

duals_init: type of the update for the initialization of Lagrange multipliers corresponding to the equality constraints. Possible values one of the the strings "lsq" (least-square (LSQ) solve initialization) and "zero" (multipliers are set identically to zero). Default value is "lsq".

duals_lsq_ini_max: max inf-norm allowed for initial duals when computed with LSQ (see **duals_init**); if norm is greater, the duals for the equality constraints will be set to zero. Double values between 10^{-16} and 10^{10}. Default value: 1 000.

duals_update_type: string option specifying the type of update of the multipliers of the eq. constraints after each iteration. Possible values are "lsq" (update based on a LSQ solve) and "linear" (Newton update based on the dual steplength. When "Hessian" is "quasinewton_approx" the default value for this options is "lsq". When "Hessian" is "analytical_exact" the default value is "linear".

recalc_lsq_duals_tol: threshold for inf-norm under which the LSQ computation of duals is used. If the inf-norm of the duals of the equality constraints is larger than the value of this options, these duals are set to zero. This options requires "duals_update_type" to be "lsq" (the option is ignored otherwise). Double values in $[0, 10^{10}]$. Default value 10^{-6}.

accept_every_trial_stepduals: disable the line-search and take the close-to-boundary step. String values: "no" (default) and "yes".

Hessian: type of Hessian used with the filter IPM.

- "quasinewton_approx" (default) - HiOp will build secant BFGS approximation for the Hessian and use a quasi-Newton filter IPM
- "analytical_exact" - Hessian provided by the user and a Newton filter IPM algorithm will be used

sigma0: initial value of the initial multiplier of the identity in the secant approximation. Numeric values in $[0, 10^7]$. Default value 1.

secant_memory_len: size of the memory (number of (s, y) pairs) of the Hessian secant approximation. Integer values between 0 and 256. Default value 6.

4.3 Linear algebra computational kernels

KKTLinsys: type of KKT linear system *formulation* used internally

- "auto" (default): decided by `HiOp` based on the type of interface/NLP solved and "compute_mode" and "Hessian" options
- "xycyd": symmetric indefinite (less stable but smaller size)
- "xdycyd": symmetric indefinite (more stable but larger size)
- "full": unsymetric suitable for LU solvers (experimental)

linsol_mode: for some problem classes and KKT linearizations, one can instruct `HiOp` to switch between strategies for solving the IPM linear systems.

- "stable" (default): the most stable factorization is used
- "speculative": switch to faster linear solvers when is detected to be safe to do so. This is available for MDS problems and can offer considerable speed-up for these problems. The option is experimental and should be used only by advanced users
- "forcequick" rely on fast solvers (experimental, avoid)

compute_mode: offloading of computations to GPUs

- "auto" (default): identical to "hybrid"
- "cpu": run everything on cpu
- "hybrid": `HiOp` will decide internally based on the type of NLP problem solved and other options which computational kernels will be offloaded to GPU. It usually runs the expensive linear solves on GPU but the remaining computations on the host/CPU.
- "gpu": run the all the computational kernels on the device; some computations (*e.g.*, logic and control loop) will run on CPU. As of v0.4, this compute mode is experimental, should be used only by advanced users.

mem_space: specifies the primary memory space in which the RAJA-based optimization and linear algebra objects will be created

- "default" (default): the default memory space as per RAJA specification
- "host": cpu memory space
- "device": device memory space
- "um": unified memory space

4.4 Problem preprocessing

fixed_var: treatment of variables that are detected to be fixed (according to the tolerance specified by "fixed_var_tolerance")

- "none" (default): will not handle fixed variable and will exit with an error message if such variable is encountered
- "relax": relax the fixed variables accordingly to "fixed_var_perturb" option below
- "'remove": remove variables from the (internal) NLP formulation

fixed_var_tolerance: a variable (say the ith) is considered fixed if

$$|(x_u)_i - (x_l)_i| < \text{fixed\_var\_tolerance} \times \max(|(x_u)_i|, 1).$$

This option takes double values in $[10^{-30}, 10^{-2}]$ and has a default value 10^{-15}.

fixed_var_perturb: fixed variable perturbation of the lower and upper bounds for fixed variables relative their magnitude. A variable (say the ith) (that is detected to be fixed) is "relaxed" accordingly to

$$(x_l)_i = (x_l)_i - \max(|(x_u)_i|, 1) \times \text{fixed\_var\_perturb}$$
$$(x_u)_i = (x_u)_i + \max(|(x_u)_i|, 1) \times \text{fixed\_var\_perturb}.$$

This option takes double values in $[10^{-14}, 0.1]$ and has a default value 10^{-8}.
bound_relax_perturb: perturbation of the lower and upper bounds for all variables and all constraints relative to their magnitude. A variable or constraint (say the ith) with lower and upper bounds $(x_l)_i$ and $(x_u)_i$, respectively, is "relaxed" accordingly to

$$(x_l)_i = (x_l)_i - \max(|(x_l)_i|, 1) \times \text{bound\_relax\_perturb}$$
$$(x_u)_i = (x_u)_i + \max(|(x_u)_i|, 1) \times \text{bound\_relax\_perturb}.$$

This option takes double values in $[0, 10^{20}]$ and has a default value 10^{-8}.
scaling_type: scaling method for the user's NLP

- "none" (default): perform no problem scaling
- "gradient": will scale the problem such that the inf-norm of gradient at the initial point is less or equal to the value of "scaling_max_grad" option.

scaling_max_grad: The user's NLP will be rescaled if the inf-norm of the gradient at the starting point is larger than the value of this option. Double values in $[10^{-20}, 10^{20}]$. Default value 100.

5 Licensing and copyright

6 Acknowledgments

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. The author also acknowledges the support from the LDRD Program of Lawrence Livermore National Laboratory under the projects 16-ERD-025 and 17-SI-005.

References

[1] R. H. Byrd, J. Nocedal, and R. B. Schnabel. Representations of quasi-newton matrices and their use in limited memory methods. *Mathematical Programming*, 63(1):129–156, 1994.

[2] C. G. Petra. A memory-distributed quasi-Newton solver for nonlinear programming problems with a small number of general constraints. Technical Report LLNL-JRNL-739001, Lawrence Livermore National Laboratory, October 2017.

[3] C. G. Petra. A memory-distributed quasi-newton solver for nonlinear programming problems with a small number of general constraints. *Journal of Parallel and Distributed Computing*, 133:337–348, 2019.

[4] A. Wächter and L. T. Biegler. Line search filter methods for nonlinear programming: Local convergence. *SIAM Journal on Optimization*, 16(1):32–48, 2005.

[5] A. Wächter and L. T. Biegler. Line search filter methods for nonlinear programming: Motivation and global convergence. *SIAM Journal on Optimization*, 16(1):1–31, 2005.

[6] A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.