HiOp – User Guide

version 1.1.0

by

Cosmin G. Petra, **Nai-Yuan Chiang**, and **Jingyi Wang**

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

7000 East Avenue,
Livermore, CA 94550, USA.

Oct 15, 2017
Updated Sept 22, 2024

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). Four main classes of optimization problems are supported by `HiOp`.

- `HiOp-Dense` supports NLPs with billions of variables with or without bounds but only limited number of constraints. This solver is a memory-distributed, MPI-based quasi-Newton interior-point solver using limited-memory approximations for the Hessians.
- `HiOp-Sparse` supports general sparse and large-scale NLPs sparse second-order derivatives. This functionality is similar to that of the state-of-the-art Ipopt [6], but with additional features such as the inertia-free approach [2]. The solver offers GPU acceleration via Nvidia CUDA Toolkit or AMD HIP Toolkit, and requires RAJA portability abstraction layer when GPU acceleration is enabled.
- `HiOp-MDS` supports NLPs that have dense and sparse blocks, for which a "Newton" interior-point solver is available together with a specialized, so-called mixed dense-sparse (MDS) linear algebra capable of achieving good performance on GPUs via Magma dense linear solver.
- `HiOp-PriDec` is an asynchronous memory-distributed optimization solver for two-stage stochastic programming problems. It implements a master-worker asynchronous scheduler based on MPI to improve load balancing. GPU acceleration can be achieved in solving each subproblem by `HiOp-MDS` or `HiOp-Sparse`.

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` [3].

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. z 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 [**?**]. 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 [**?**]. 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++11. At minimum, HiOp requires BLAS and LAPACK, however, the more advanced solvers require additional dependencies (MPI, RAJA and Umpire, CUDA, HIP, MAGMA, CoinHSL, PARDISO, STRUMPACK, etc.). The CMake-based build system of HiOp generally detects these prerequisites automatically and warns the user when such prerequisites are missing.

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/AMD)GPU hardware

Starting version 0.3, HiOp offers support for offloading computations to NVIDIA GPU accelerators when solving NLPs in the mixed dense-sparse (MDS) form. Support for CUDA should be enabled during the build by using cmake options -DHIOP_USE_GPU and -DHIOP_USE_CUDA, which will result in using the CUDA accelerators for the internal linear solves; in addition, the options -DHIOP_USE_RAJA will employ RAJA portability abstraction to perform the remaining linear algebra computations on the GPU device or on the host (with OpenMP acceleration). When RAJA is enabled, HiOp can be instructed to use Umpire as memory manager (see option mem_space). As of v0.5, the combination of RAJA and Umpire enables HiOp to perform iterations of the Newton IPM solver solely on the device by setting option mem_space to device and option compute_mode to gpu.

Starting version 0.6, HiOp offers support for offloading computations to AMD GPU accelerators when solving NLPs in the mixed dense-sparse (MDS) form. Support for HIP should be enabled during the build by using cmake options -DHIOP_USE_GPU and -DHIOP_USE_HIP, which will result in using the HIP accelerators for the internal linear solves.

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, rocm library version 4.5.0 or later and a recent Magma linear solver library (as well as a physical NVIDIA/AMD GPU device). HiOp offers an extensive build support for using customized NVIDIA libraries, AMD libraries, and/or Magma solver as well as for advanced troubleshooting. The user is referred to the following CMake scripts for more details:

- cmake/FindHiopCudaLibraries.cmake
- cmake/FindHiopHipLibraries.cmake
- cmake/FindHiOpMagma.cmake scripts.

⚠ **Note:** Installing NVIDIA CUDA, AMD HIP, 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`.

⚠ **Note:** The C++ interface uses the integer types `size_type` and `index_type`. The type `hiop::size_type` is used for container (*e.g.,* NLPs, vectors, matrices, etc.) sizes and generally holds a nonnegative integer. The `hiop::index_type` type should be used for indexes within containers and is generally holding a nonnegative integer. These two types are defined within `HiOp` namespace (see `hiop_defs.h`) and currently set to `int`. This choice allows a streamlined integration (that is, type conversions are not needed and arrays of indexes can be reused) with the low level linear algebra libraries, such as sparse and dense linear solver libraries, which generally use `int`.

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.

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

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

```
bool get_vars_info(const size_type& 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 -1^{20} or/and larger than 1^{20}, respectively. The last argument is not used and can set to any value of the enum `hiop::hiopInterfaceDenseConstraints::NonlinearityType`.

```
bool get_cons_info(const size_type& 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 size_type& 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 size_type& n,
                 const double* x, bool new_x,
                 double* gradf);
```

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

```
bool eval_cons(const size_type& n, const size_type& m,
               const size_type& num_cons,
               const index_type* 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 size_type& n, const size_type& m,
              const size_type& num_cons, const index_type* 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.

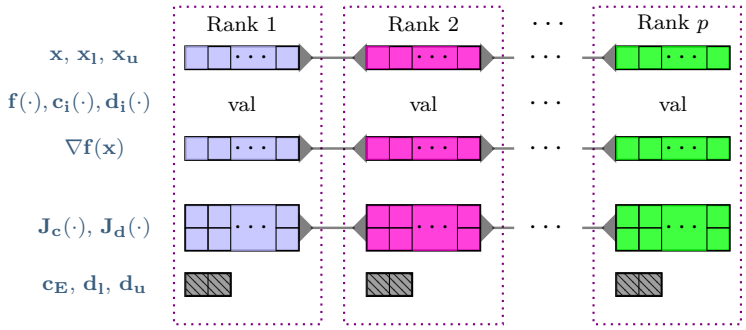

Figure 1: Depiction of the distribution of the data of the optimization problem (1)-(4) across MPI ranks. The vectors and matrices with storage dependent on the number of optimization variables are distributed. Other data, *i.e.*, scalar function values or vectors of small size (shown in dashed dark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

```
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`.

```
bool get_vecdistrib_info(size_type global_n, size_type* 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 `DenseConsEx1` (deriving `hiop::hiopInterfaceDenseConstraints`), the aforementioned sequence of steps can be performed by:

```
#include "NlpDenseConsEx1.hpp"               //the NLP representation class
#include "hiopInterface.hpp"   //HiOP encapsulation of the NLP
#include "hiopAlgFilterIPM.hpp"     //solver class
using namespace hiop;
...
DenseConsEx1 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 `NlpDenseConsEx1`, `NlpDenseConsEx2`, and `NlpDenseConsEx3` inside directory `src/Drivers/` under the `HiOp`'s root directory contain more detailed examples of the use of `HiOp`.

3.1.5 Checkpointing

File checkpointing is available for `HiOp`'s quasi-Newton IPM solver, which is used exclusively to solve `hiopInterfaceDenseConstraints` formulation. This can be helpful when running a job on a cluster that enforces limits on the jobs running time. Later, this feature will also be provided for other solvers, such as the Newton IPM (used exclusively with sparse NLP) and HiOp-PriDec.

The checkpointing I/O is based on Axom's scalable Sidre data manager (see `https://axom.readthedocs.io/en/develop/axom/sidre/docs/sphinx/index.html` for more information) and, thus, requires an Axom-enabled build (use "-DHIOP_USE_AXOM=ON" with cmake).

There are two ways to use `HiOp`'s checkpointing. The first is via the quasi-Newton solver's API, namely, the methods

```
void load_state_from_sidre_group(const ::axom::sidre::Group& group);
void save_state_to_sidre_group(::axom::sidre::Group& group);
```

of `hiopAlgFilterIPMQuasiNewton` solver class. New Sidre views will be created (or reused) within the group passed as argument to load / save state variables of the quasi-Newton solver. Alternatively, `hiopAlgFilterIPMQuasiNewton` solver class offers similar methods to work directly with a file, namely,

```
bool load_state_from_file(const ::std::string& path) noexcept;
bool save_state_to_file(const ::std::string& path) noexcept;
```

These two methods will create the Sidre group internally and checkpoint to/from it using the first two methods.

A second avenue to checkpoint is via user options. This is detailed in Section 4.1.10.

⚠ **Note:** A couple of particularities stemming from the use of Sidre must be acknowledged. First, a checkpoint file should be loaded using HiOp with the same number of MPI ranks as when it was saved. Second, checkpointing is not available for non-MPI builds due to Axom having MPI as a dependency. Finally, when loading from or saving to a checkpoint file, the sizes of the file's variables (Sidre views) must match the sizes of the HiOp variables to which the data is loaded or saved, meaning `HiOp` will throw an exception if an existing file is (re)used to load or save a algorithm state for a problem that changed sizes since the file was created.

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.

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

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

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

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

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 (7) 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 (8) 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. Internally, a slack variable s is introduced and the inequality constraints (7) are replaced by additional equality constraints and boundary constraints:

$$d(x) = s \qquad [y_d] \tag{9}$$

$$[v_l] \qquad d_l \le s \le d_u \qquad [v_u] \tag{10}$$

As a result, `HiOp` requires the user to provide the following quantities:

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

$$\nabla^2 L(x) = \nabla^2 f(x) + \sum_{i=1}^{m_E} y_{c,i} \nabla^2 c_i(x) + \sum_{i=1}^{m_I} y_{d,i} \nabla^2 d_i(x). \tag{11}$$

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.

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

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

```
bool get_vars_info(const size_type& 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 -1^{20} or/and larger than 1^{20}, respectively. The last argument is not used and can set to any value of the enum `hiop::hiopInterface::NonlinearityType`.

```
bool get_cons_info(const size_type& 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 size_type& 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 size_type& n,
                 const double* x, bool new_x,
                 double* gradf);
```

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

```
 bool eval_cons(const size_type& n, const size_type& m,
                const size_type& num_cons,
                const index_type* 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 size_type& n, const size_type& m,
              const size_type& num_cons, const index_type* idx_cons,
              const double* x, bool new_x,
              const size_type& nnzJacS, index_type* iJacS, index_type* 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 size_type& n, const size_type& m,
              const double* x, bool new_x,
              const size_type& nnzJacS, index_type* iJacS, index_type* 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 size_type& n, const size_type& m,
               const double* x, bool new_x, const double& obj_factor,
               const double* lambda, bool new_lambda,
               const size_type& nsparse, const size_type& ndense,
               const size_type& nnzHSS, index_type* iHSS, index_type* 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 "NlpSparseEx1.hpp"        //the NLP representation class
#include "hiopInterface.hpp"       //HiOP encapsulation of the NLP
#include "hiopAlgFilterIPM.hpp"    //solver class
using namespace hiop;
...
NlpSparseEx1 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 `NlpSparseEx1` and `NlpSparseEx2` inside directory `src/Drivers/` under the `HiOp`'s root directory contain more detailed examples of the use of the sparse NLP interface of `HiOp`.

3.2.4 Solvers options for `hiopInterfaceSparse` NLP formulations

The optimization solver and linear algebra strategy within is controlled via the option **KKTLinsys**. For sparse NLPs, the default value (under "auto") is "xdycyd". Individual linear solvers can be selected via the option **linear_solver_sparse**. GPU-capable linear solvers are available, namely, cuSOLVER sparse LU and Ginkgo, when option "compute_mode" is set to "hybrid". We recommend setting "KKTLinsys" to "auto", "linear_solver_sparse" to ''auto", and choosing CPU or GPU linear algebra backend by setting "compute_mode" to "cpu" or "hybrid", respectively.

A so-called condensed sparse optimization solver is currently under development with the goal of increasing adoption of GPUs. It uses a so-called condensed linear algebra KKT formulation (see Section A.0.1), specialized sparse matrix device and host kernels, Cholesky-based linear solves on the device, and a variation of the filter line-search interior-point algorithm currently implemented by HiOp. The variation of the algorithm is choosen to improve the numerical conditioning of the linear systems. As a result, the following options need to be used with the condensed sparse optimization solver:

```
KKTLinsys condensed
compute_mode hybrid
linsol_mode speculative
fixed_var relax

tau_min 0.9
theta_mu 1.1
kappa_mu 0.8

elastic_mode correct_it_adjust_bound
elastic_mode_bound_relax_final 1e-10
elastic_mode_bound_relax_initial 0.01
elastic_bound_strategy  mu_projected

fact_acceptor inertia_free
```

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

$$\min_{x_d \in \mathbb{R}^{n_d}, x_s \in \mathbb{R}^{n_s}} \quad f(x_d, x_s) \tag{12}$$

$$\text{s.t.} \quad c(x_d, x_s) = c_E, \tag{13}$$

$$d^l \le d(x_d, x_s) \le d^u, \tag{14}$$

$$x_d^l \le x_d \le x_d^u,\ x_s^l \le x_s \le x_s^u. \tag{15}$$

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 (14) are assumed to be $d^l \in \mathbb{R}^{m_I} \cup \{-\infty\}$, $d^u \in \mathbb{R}^{m_I} \cup \{+\infty\}$, $d_i^l < d_i^u$, and at least of one of d_i^l and d_i^u are finite for each $i \in \{1, \ldots, m_I\}$. The vector bounds x_d^l, x_d^u, x_s^l, and x_s^u in (15) 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 (15).

The salient idea behind mixed dense-sparse problems of the form (12)-(15) 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 (12)–(15) 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.

`HiOp` offers support for NVIDIA GPU 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 and Magma linear solver library. The Newton interior-point solver for MDS problems offers the possibility to perform the linear algebra and the great majority of the optimization computations on the device; this can be achieved by setting option `compute_mode` to `gpu` and the option `mem_space` to `device`. This combination of the two options will require the problem evaluation functions implemented by the user (see Section 3.3.1 below) to run on the device. If the user code does not support this, then `HiOp` should be used with `compute_mode` set to `hybrid` and the option `mem_space` set to `default`; this combination will offload the majority of linear algebra and optimization computations to the device. The `HiOp`'s RAJA version of Example 1 (see `src/Drivers/NlpMdsEx1RajaDriver.cpp`) provides an example of implementing a MDS NLP so that it that can be solved by running `HiOp`'s Newton solver on the device (*i.e.*, `compute_mode` set to `gpu` and with `mem_space` set to `device`).

We note that MDS NLPs have no support for coarse grain (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

$$Jc(x_d, x_s) = \begin{bmatrix} J_{x_d} c(x_d, x_s) & J_{x_s} c(x_d, x_s) \end{bmatrix} \tag{16}$$

and

$$Jd(x_d, x_s) = \begin{bmatrix} J_{x_d} d(x_d, x_s) & J_{x_s} d(x_d, x_s) \end{bmatrix}. \tag{17}$$

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

$$\nabla^2 L(x_d, x_s) = \lambda_0 \nabla^2 f(x_d, x_s) + \sum_{i=1}^{m_E} \lambda_i^E \nabla^2 c_i(x_d, x_s) + \sum_{i=1}^{m_I} \lambda_i^I \nabla^2 d_i(x_d, x_s). \tag{18}$$

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 (12)–(15) 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`.

⚠ **Note:** Regarding the implementation of hiop::hiopInterfaceMDS on the device, all pointers marked as "managed by Umpire" are allocated by HiOp using the Umpire's API. They all are addressed in the same memory space; however, the memory space can be host (typically CPU), device (typically GPU), or unified memory (um) spaces as per Umpire specification. The selection of the memory space is done via the option "mem_space" of HiOp. It is the responsibility of the implementers of the HiOp's interfaces to work with the "managed by Umpire" pointers in the same memory space as the one specified by the "mem_space" option.

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

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

```
1 bool get_vars_info(const size_type& 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 -1^{20} or/and larger than 1^{20}, respectively. The last argument is not used and can set to any value of the enum `hiop::hiopInterfaceDenseConstraints::NonlinearityType`. While array `type` is allocated on host, arrays `xlow` and `xupp` are managed by Umpire.

```
1 bool get_cons_info(const size_type& m, double* clow, double* cupp,
2                    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). While array `type` is allocated on host, arrays `clow` and `cupp` are managed by Umpire.

```
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 (17) and (18), respectively. The last parameter `nnz_sparse_Hess_Lagr_SD` is not used momentarily and should be set to zero.

```
bool eval_f(const size_type& 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. Array `x` is managed by Umpire.

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

Same as above but for $\nabla f(x)$. Arrays x and *gradf* are managed by Umpire.

```
bool eval_cons(const size_type& n, const size_type& m,
               const size_type& num_cons,
     const index_type* 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`. Arrays `idx_cons`, `x` and `cons` are managed by Umpire.

```
eval_Jac_cons(const size_type& n, const size_type& m,
              const size_type& num_cons, const index_type* idx_cons,
              const double* x, bool new_x,
              const size_type& nsparse, const size_type& ndense,
              const size_type& nnzJacS,
              index_type* iJacS, index_type* 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:** Arrays `idx_cons`, `x`, `iJacS`, `jJacS`, `MJacS` and `JacD` are managed by Umpire.

⚠ **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 size_type& n, const size_type& m,
                   const double* x, bool new_x,
                   const size_type& nsparse, const size_type& ndense,
                   const size_type& nnzJacS,
                   index_type* iJacS, index_type* 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:** Arrays `x`, `iJacS`, `jJacS`, `MJacS` and `JacD` are managed by Umpire.

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

```
bool eval_Hess_Lagr(const size_type& n, const size_type& m,
                    const double* x, bool new_x, const double& obj_factor,
                    const double* lambda, bool new_lambda,
                    const size_type& nsparse, const size_type& ndense,
                    const size_type& nnzHSS,
                    index_type* iHSS, index_type* jHSS, double* MHSS,
                    double* HDD,
                    size_type& nnzHSD, index_type* iHSD, index_type* 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 (18). The argument `HDD` stores $\nabla^2 L(x_d, x_d)$ from (18).

⚠ **Note:** The last four arguments, which are supposed to store the cross-Hessian $\nabla^2 L(x_s, x_d)$ from (18), 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).

⚠ **Note:** Arrays `x`, `lambda`, `iHSS`, `jHSS`, `MHSS`, `HDD`, `iHSD`, `jHSD` and `MHSD` are managed by Umpire.

⚠ **Device computations:** `HiOp` supports full device/GPU acceleration for MDS NLPs. To achieve this, the user can use option `compute_mode` set to `gpu` and option `mem_space` set to `device`. However, the user needs to be able to evaluate the model on the device. The rule of thumb is that all the *pointer* arguments of the callback methods of this section will be on the device (with a few exceptions) so that the user can populate the arrays on the device. This is illustrated and discussed in detail in `src/Drivers/NlpMdsRajaEx1.hpp`, which is part of the RAJA Example 1 (see `src/Drivers/NlpMdsEx1RajaDriver.cpp`) that is capable of running completely in the device memory space with minimal host-device transfer.

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 **NlpMdsEx1Driver.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 **MdsEx1** (implemented in **NlpMdsFormEx1.hpp**) derived from **hiop::hiopInterfaceMDS** is as follows:

```
#include "NlpMdsFormEx1.hpp"                //the NLP representation class
#include "hiopInterface.hpp"    //HiOP encapsulation of the NLP
#include "hiopAlgFilterIPM.hpp"      //solver class
using namespace hiop;
...
MdsEx1* my_nlp = new MdsEx1(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 Structured NLPs suitable to primal decomposition (PriDec) schemes

Starting v0.5, **HiOp** also offers parallel computing capabilities via the PriDec solver for NLPs with separable objective terms in the form of:

$$\min_{x\in\mathbb{R}^n} \quad f(x) + \sum_{i=1}^{K} r_i(x) \tag{19}$$

$$\text{s.t.} \quad c(x) = c_E, \qquad [y_c] \tag{20}$$

$$[v_l] \qquad d_l \le d(x) \le d_u, \qquad [v_u] \tag{21}$$

$$[z_u] \qquad x_l \le x \le x_u. \qquad [z_u] \tag{22}$$

Mathematically, the above problem is identical (and has the same specification) to the NLP (1)-(4), with the exception of the so-called "recourse" terms $r_i(x)$ appearing in the objective. Each of these functions are real-valued, $r_i : \mathbb{R}^n \to \mathbb{R}$, for all $i \in \{1, 2, \ldots, K\}$, and can be of various order of differentiability. As of now, the recourse functions $r_i(x)$ need to be Lipschitz continuous and continuously differentiable. It is also possible for $r_i(x)$ to be Lipschitz and only weakly concave (with convergence guarantees). The users are encouraged to contact **HiOp** developers for the latest developements in this area. A compact description of the mathematical algorithm implemented by PriDec can be found in [8], while the parallelization strategy can be found in [7].

The input in which **HiOp** expects for this class of problems is a bit different than for NLPs of the form (1)-(4) and MDS NLPs introduced in the previous sections. This is mainly caused by the specifics of the primal decomposition algorithm/solver that was purposedly developed

to solve (19)-(22) for large K (*e.g.*, $K = O(10^6)$) efficiently on a massively parallel computing platform. Nevertheless, for smaller K, problems of form (19)-(22) can be solved with `HiOp` using the sparse and MDS input interfaces.

The primal decomposition algorithm requires a separation or breakdown of the evaluation of (19)-(22) into the following computational "units".

1. solving the so-called "master problem" of the form

$$\begin{aligned} & \min_{x\in\mathbb{R}^n} && f(x) + q(x) && && (23)\\ & \text{s.t.} && c(x) = c_E && [y_c] && (24)\\ [v_l] \quad & && d_l \le d(x) \le d_u && [v_u] && (25)\\ [z_u] \quad & && x_l \le x \le x_u && [z_u] && (26) \end{aligned}$$

   for a real function $q(x)$ constructed by `HiOp` PriDec solver, which serves as an approximation to $\sum_{i=1}^{K} r_i(x)$. The evaluation of $q(x)$, its gradient and sparse Hessian are provided by `HiOp` PriDec solver based on the function values and graidents of $r_i(x)$; The master problem is implemented based on the basecase problem

$$\begin{aligned} & \min_{x\in\mathbb{R}^n} && f(x) && && (27)\\ & \text{s.t.} && c(x) = c_E && [y_c] && (28)\\ [v_l] \quad & && d_l \le d(x) \le d_u && [v_u] && (29)\\ [z_u] \quad & && x_l \le x \le x_u && [z_u] && (30) \end{aligned}$$

   where no recourse functions exist. To determine whether $q(x)$ is included in the objective, a boolean variable is used. The basecase problem class also contains a hiopInterfacePriDecProblem::RecourseApproxEvaluator object, that stores and updates the function $q(x)$. The PriDec solver constructs $q(x)$ at each iteration and then passes it on to the basecase problem so that the full problem (23)-(26) can be solved. In other words, the user does not need to provide $q(x)$ in their objective, but needs to write the basecase problem (27)-(30) such that its objective (or potentially constraint in the future) can be extended.

2. evaluating the recourse functions $r_i(x)$ and their (sub)gradients $\nabla r_i(x)$, for all $i \in \{1, 2, \ldots, K\}$. If there is no analytical form for $r_i(x)$, as in the case of two-stage problems, the user might need to implement and solve a second-stage optimization problem. Nevertheless, `HiOp` PriDec solver expects to be returned a function value and a (sub)gradient at a given x.

To streamline steps 1 and 2 above, the master problem is implemented with the class hiopInterfacePriDecProblem, which has methods for solving the master problem and evaluating recourse functions. We stress that it is the user's responsibility to implement steps 1 and 2 above. In regards to 1, the function $q(x)$ is an approximation to the recourse $R(x) := \sum_{i=1}^{K} r_i(x)$ from (23)-(26), which is built based on the function and gradient evaluations of $r_i(x)$, computed at step 2.

The user can safely assume that $q(x)$ is a strictly convex quadratic function (however the function may be only convex and nonquadratic in a future version of `HiOp`). `HiOp` assumes that the user can solve the master problem (23)-(26) in some efficient way and that the user can return the optimal solution vector. In the examples given, the master problem is setup and solved with `HiOp`.

Self-contained examples of the use of `HiOp`'s PriDec solver are present in `NlpPriDecEx1` and `NlpPriDecEx2` examples under the `Drivers` directory.

3.5 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 size_type& n, const size_type& m,
                        double* x0,
                        bool& duals_avail,
                        double* z_bndL0, double* z_bndU0,
                        double* lambda0,
                        bool& slacks_avail,
                        double* ineq\_slack);
```

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 size_type& 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.
- `slacks_avail` boolean flag expressing whether the initial values for the inequality slacks (added by HiOp internally) are given by the user.
- `ineq_slack` is an array containing the starting point for the slacks added by `HiOp` to transfer inequalities to equalities internally.

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:** Arrays `x0`, `z_bndL0`, `z_bndU0`, `lambda0` and `ineq_slack` are managed by Umpire.

⚠ **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.

A third method to initialize the point is offered to advanced users, as it will skip all the safeguards in `HiOp`, e.g., checking if it is 'nullptr' or project x into variable bounds.

```
bool get_warmstart_point(const size_type& n, const size_type& m,
                         double* x0,
                         double* z_bndL0, double* z_bndU0,
                         double* lambda0,
                         double* ineq\_slack,
                         double* vl0, double* vu0);
```

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.
- `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.
- `ineq_slack` is an array containing the starting point for the slacks added by `HiOp` to transfer inequalities to equalities internally.
- `vl0` and `vu0` starting points for the duals of the (inequality) constraints lower and upper bounds.

This method should only be implemented when user wants to use a warmstart point and should be used with caution.

⚠ **Note:** Arrays `x0`, `z_bndL0`, `z_bndU0`, `lambda0`, `ineq_slack`, `vl0` and `vu0` are managed by Umpire.

3.6 Obtain information from `HiOp`

`HiOp` provides two callback functions for the user to obtain information about the optimization status.

```
void solution_callback(hiopSolveStatus status,
                       size_type n,
                       const double* x,
                       const double* z_L,
                       const double* z_U,
                       size_type m,
                       const double* g,
                       const double* lambda,
                       double obj_value);
```

Callback method called by HiOp when the optimal solution is reached. User can use it to retrieve primal-dual optimal solution.

Parameters:

- `status` status of the solution process.
- `n` global number of variables.
- `x` array of (local) entries of the primal variables at solution.
- `z_L` array of (local) entries of the dual variables for lower bounds at solution.
- `z_U` array of (local) entries of the dual variables for upper bounds at solution.
- `g` array of the values of the constraints body at solution.
- `lambda` array of (local) entries of the dual variables for constraints at solution.
- `obj_value` objective value at solution

⚠ **Note:** Arrays `x`, `z_L`, `z_U`, `g` and `lambda` are managed by Umpire.

```
bool iterate_callback(int iter,
                      double obj_value,
                      double logbar_obj_value,
                      int n,
                      const double* x,
                      const double* z_L,
                      const double* z_U,
                      int m_ineq,
                      const double* s,
                      int m,
                      const double* g,
                      const double* lambda,
                      double inf_pr,
                      double inf_du,
                      double onenorm_pr,
                      double mu,
                      double alpha_du,
                      double alpha_pr,
                      int ls_trials);
```

Intermediate callback method called by HiOp at the end of each iteration. User can obtain information about the optimization status while `HiOp` solves the problem. If the user (implementer) of this methods returns false, HiOp will stop the optimization with `hiop::hiopSolveStatus::User_Stopped` return code. Parameters:

- `iter` the current iteration number
- `obj_value` objective value
- `logbar_obj_value` log barrier objective value
- `n` global number of variables
- `x` array of (local) entries of the primal variables (managed by Umpire, see note below)
- `z_L` array of (local) entries of the dual variables for lower bounds (managed by Umpire, see note below)

- `z_U` array of (local) entries of the dual variables for upper bounds (managed by Umpire, see note below)
- `m_ineq` the number of inequality constraints
- `s` array of the slacks added to transfer inequalities to equalities (managed by Umpire, see note below)
- `m` the number of constraints
- `g` array of the values of the constraints body (managed by Umpire, see note below)
- `lambda` array of (local) entries of the dual variables for constraints (managed by Umpire, see note below)
- `inf_pr` inf norm of the primal infeasibilities
- `inf_du` inf norm of the dual infeasibilities
- `onenorm_pr` one norm of the primal infeasibilities
- `mu` the log barrier parameter
- `alpha_du` dual step size
- `alpha_pr` primal step size
- `ls_trials` the number of line search iterations

⚠ **Note:** Arrays `x`, `z_L`, `z_U`, `s`, `g` and `lambda` are managed by Umpire.

⚠ **Note:** HiOp's option `callback_mem_space` can be used to change the memory location of array parameters managaged by Umpire. More specifically, when `callback_mem_space` is set to 'host' (and `mem_space` is 'device'), HiOp transfers the arrays from device to host first, and then returns pointers on host whose data is managed by Umpire. These pointers can be then used in host memory space (without the need to rely on or use Umpire).

3.7 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

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

- via the options file(s) that should be placed in the same directory where the application driver using `HiOp` is executed. The format of an option file is very basic, each of its lines should contain a single pair `option_name option_value`. Lines that begin with '#' or consist of only white characters 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 various NLP formulation and PriDec solver classes. The options 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);
```

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

The NLP solvers load options from the file `hiop.options`. The PriDec solver will look for and load options from up to three files:

- `hiop_pridec.options` specifies options for the **PriDec** algorithm/solver
- `hiop_pridec_master.options` specifies options for the NLP solver used to solve the **master** problem. This master NLP solver does not necessarily have to be one of `HiOp`'s NLP solvers. The name of this file can be controlled via the string option `options_file_master_prob` of the PriDec solver, in `hiop_pridec.options`.
- `hiop.options` specifies the options for the **worker** NLP solver. This applies only when the worker NLP solver is one of the `HiOp`'s solvers. This file will not be used by worker solvers other than `HiOp`; they will use their default option files.

For example, when the PriDec solver is used with `HiOp`'s NLP solvers for both the master and the worker subproblems, the user should create the three options files above to customize the PriDec, master, and worker solvers. As another example, when Ipopt is used for both master and worker subproblems, the user should use the default "ipopt.opt" file for the worker and use "hiop_pridec_master.options" for Ipopt options for the master subproblem (or, if another file needs to be used, change the name of the master options file via `options_file_master_prob` in PriDec's `hiop_pridec.options` option file).

If HiOp needs to solve a feasibility problem internally, it treats the feasibility problem as a new optimization problem and launchs a standalone internal process to solve the problem. The file, `hiop_fr.options`, can be used to control the options for solving the feasibility problem by HiOp. The name of this option file can be tuned by parameter 'options_file_fr_prob'.

⚠ **Note:** If an option file is not present, `HiOp` will use default values (unless the user changes the options at runtime via the API).

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

4.1 Options for NLP solvers

4.1.1 Termination criteria and output

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: 10.

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: 3000.

rel_tolerance: error tolerance for the NLP relative to errors at the initial point. A null value disables this option. Double values in $[0, 0.1]$. Default value: 0.

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

max_soc_iter: maximum number of iterations in second order correction. Integer values between 1 to 10^6. Default value: 4.

4.1.2 Filter-IPM algorithm selection and parameters

mu0: initial log-barrier parameter mu. Double values in $[10^{-16}, 10^3]$. Default value: 1.0.

kappa_eps: μ is reduced when when log-barrier error is below kappa_eps $\times$ μ. Double values in $[10^{-6}, 1000]$. Default value: 10.

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

kappa1: sufficiently-away-from-the-boundary projection parameter used in the shift of the user-provided initial point. Double values in $[10^{-16}, 0.1]$. Default value: 0.01.

kappa2: shift projection parameter used in initialization for doubly bounded variables. Double values in $[10^{-15}, 0.49999]$. Default value: 0.01.

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

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

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.

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.

sigma_update_strategy: string option specifying the updating strategy for the multiplier of the identity in the secant approximation. Possible values are "sigma0", "sty", " sty_inv", "snrm_ynrm" and "sty_srnm_ynrm". Default value: is "sty".

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.

kappa_soc: factor to decrease the constraint violation in second order correction. Double values in $[0, 10^{20}]$. Default value: 0.99.

warm_start: string option with "yes" or "no" values deciding whether `HiOp` uses warm start from the user provided primal-dual point. Note that all the primal, dual and slack variables must be provided. Default value: "no".

4.1.3 Line search and step computation

fact_acceptor: the criteria used to accept a factorization:

- "inertia_correction" (default): the most stable approch which requires inertia information provided by the given linear solvers (see parameter **linear_solver_sparse**);
- "inertia_free": apply inertia free method. This approch is typically used when the given linear solver cannot provide inertia information.

neg_curv_test_fact: apply curvature test to check if a factorization is acceptable. This is the scaling factor used to determines if a direction is considered to have sufficiently positive curvature. Only valid when parameter `fact_acceptor` is set to `inertia_free`. Double values in $[0, 10^{20}]$. Default value: 10^{-11}.

min_step_size: minimum step size allowed in line-search. If step size is less than this number, feasibility restoration problem is activated. Double values in $[0, 10^6]$. Default value: 10^{-16}.

theta_max_fact: maximum constraint violation (θ_{max}) is scaled by this fact before using in the fileter line-search algorithm. (eqn (21) in [6]). Double values in $[0, 10^7]$. Default value: 10^4.

theta_min_fact: minimum constraint violation (θ_{min}) is scaled by this fact before using in the fileter line-search algorithm. (eqn (21) in [6]). Double values in $[0, 10^7]$. Default value: 10^{-4}.

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.

accept_every_trial_step: Disables the filter line-search and take the close-to-boundary step. This step can be further controlled (while the filter line-search remains disabled) using options *moving_lim_abs* and *moving_lim_abs*. This option takes the string values: "no" (default) and "yes".

moving_lim_abs: Controls the maximum allowed update of the x primal variables during the line-search. Primal step-length may be reduced to ensure that the inf norm of the "Δx" update/direction is less than or equal with the option's value. It can be used with *accept_every_trial_step*. Numerical range is $[0, 10^8]$. A zero value will disable this option. Default value: 0.

moving_lim_rel: Controls the maximum allowed update of the primal variables *relative* to close-to-boundary step. With this option the line-search will be started using the close-to-boundary step *scaled* by the option's value. Default value: 0 (disabled). It can be used with *accept_every_trial_step*. Numerical range is $[0, 1]$. A zero value will disable this option. Default value: 0.

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: 1000.

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}.

4.1.4 Feasibility restoration

force_resto: string option with "yes" or "no" values deciding whether `HiOp` forces applying feasibility restoration. Default value: "no".

options_file_fr_prob: string option indicates the name of the option file for the feasibility restoration problem. Default value: "hiop_fr_ci.options".

kappa_resto: factor to decrease the constraint violation in feasibility restoration. Double values in $[0, 1]$. Default value: 0.9.

4.1.5 Elastic mode

elastic_mode: type of elastic mode used within `HiOp`:

- "none" (default): does not apply elastic mode;
- "tighten_bound ": tightens the bounds when μ changes.
- "correct_it": tightens the bounds, and corrects the slacks and slack duals when μ changes.
- "correct_it_adjust_bound": tightens the bounds, corrects the slacks and slack duals, and adjusts the bounds again from the modified iterate when μ changes.

elastic_bound_strategy: Strategy used to tighen the bounds, when μ changes:

- "mu_projected" (default): sets the new bound relax factor to $(\mu - \mu_{target})/(\mu_{init} - \mu_{target}) * (bound\_relax\_perturb\_initial - bound\_relax\_perturb\_final) + bound\_relax\_perturb\_final$
- "mu_scaled ": sets the new bound relax factor to $0.995 * \mu$

elastic_mode_bound_relax_final: the final/minimum bound relaxation factor in the elastic mode. This value must be less or equal to elastic_mode_bound_relax_initial. If user provides elastic_mode_bound_relax_final > elastic_mode_bound_relax_initial, HiOp will use the default values for both parameters. Double values in $[10^{-16}, 0.1]$. Default value: 10^{-12}.

elastic_mode_bound_relax_initial: the initial bound relaxation factor in the elastic mode. This value must be greater or equal to elastic_mode_bound_relax_final. If user provides elastic_mode_bound_relax_final > elastic_mode_bound_relax_initial, HiOp will use the default values for both parameters. Double values in $[10^{-16}, 0.1]$. Default value: 10^{-2}.

4.1.6 Regularization

delta_0_bar: first perturbation of the Hessian block for inertia correction. Double values in $[0, 10^{40}]$. Default value: 10^{-4}.

delta_c_bar: factor for regularization for potentially rank-deficient Jacobian ($\delta_c = \texttt{delta\_c\_bar} * \mu_c^{\kappa}$). Double values in $[10^{-20}, 10^{40}]$. Default value: 10^{-8}.

delta_w_max_bar: largest perturbation of the Hessian block for inertia correction. Double values in $[10^{-40}, 10^{40}]$. Default value: 10^{20}.

delta_w_min_bar: smallest perturbation of the Hessian block for inertia correction. Double values in $[0, 1000]$. Default value: 10^{-20}.

kappa_c: exponent of μ when computing regularization for potentially rank-deficient Jacobian ($\delta_c = \texttt{delta\_c\_bar} * \mu_c^{\kappa}$). Double values in $[0, 10^{40}]$. Default value: 0.25.

kappa_w_minus : factor to decrease the most recent successful perturbation for inertia correction. Double values in $[10^{-20}, 1]$. Default value: 0.3333.

kappa_w_plus: factor to increase perturbation when it did not provide correct inertia correction (not first iteration). Double values in $[1, 10^{40}]$. Default value: 8.

kappa_w_plus_bar: factor to increase perturbation when it did not provide correct inertia correction (first iteration when scale not known). Double values in $[1, 10^{40}]$. Default value: 100.

regularization_method: whether randomized method is used to compute regularizations.

- "standard" (default) - no randomized method is used. Regularization is computed as a scalar times an identiy matrix, i.e., δI.
- "randomized" - use randomized regularizations.

normaleqn_regularization_priority : when normal equation is used and the iterate matrix is not p.d., updating dual regularization is more efficient than updating the primal ones. Only valid when option `KKTLinsys` is set to `normaleqn`

- "primal_first" - update primal regularizations to correct positive definiteness. If primal regularization is larger than the value provided by option **delta_w_max_bar**, `HiOp` will try to increase dual regularitions.
- "dual_first" (default) - update dual regularizations to correct positive definiteness. If dual regularization is larger than the value provided by option **delta_w_max_bar**, `HiOp` will try to increase primal regularitions.

4.1.7 Solving internal linear systems

duals_init_linear_solver_sparse: string option specifying the sparse linear solver used to solve the least-square problem in dual initialization (see **duals_init**). Possible values are 'auto", "ma57", "pardiso", "resolve", "strumpack" or "ginkgo". Default value: is "auto".

linear_solver_sparse: string option specifying the sparse linear solver used to solve the sparse KKT system. Possible values are "auto", "ma57", "pardiso", "resolve", "strumpack" or "ginkgo". Default value: is "auto".

⚠ **When KKTLinsys is 'full' (unsymmetric)** : only resolve, strumpack, and pardiso are available (and will be selected in this order under 'auto' or incompatible/unsupported value for 'linear_solver_sparse').

⚠ **For KKTLinsys 'xycyd' and 'xdycyd' (symmetric indefinite)** :

- 'cpu' compute mode: ma57, pardiso, strumpack, and ginko are available and will be selected in this order under 'auto' or incompatible/unsupported value for 'linear_solver_sparse'.
- 'hybrid' compute mode: resolve, strumpack, ma57, and pardiso and will be selected in this order under 'auto' or incompatible/unsupported value for 'linear_solver_sparse'.
- 'gpu' compute mode: not supported with the above values for 'KKTLinsys'.
- For KKTLinsys 'condensed' and 'normal' (symmetric positive definite system): under 'cpu' compute mode only ma57 is supported (not efficient, use only for debugging); 'hybrid' compute mode, cusolve-chol is supported and will be selected under 'auto' or incompatible/unsupported value for 'linear_solver_sparse'; 'gpu' compute mode: work in progress.

⚠ **For KKTLinsys 'condensed' and 'normal' (symmetric positive definite system)** :

- 'cpu' compute mode: only ma57 is supported (not efficient, use only for debugging).
- 'hybrid' compute mode: cusolve-chol is supported and will be selected under 'auto' or incompatible/unsupported value for 'linear_solver_sparse'.
- 'gpu' compute mode: work in progress.

ir_inner_cusolver_maxit: FGMRES maximum number of iterations. Integer values in $[0, 1000]$. Default value: 50.

ir_inner_cusolver_restart: FGMRES restart value. Integer values in $[0, 100]$. Default value: 20.

ir_inner_cusolver_tol : FGMRES tolerance. Double values in $[10^{-16}, 0.1]$. Default value: 10^{-12}.

ir_outer_maxit: max number of outer iterative refinement iterations. Setting this to 0 deactivates the outer iterative refinement. Integer values in $[0, 100]$. Default value: 8.

ir_outer_tol_factor: iterative refinement (IR) is applied if the inf-norm of the full KKT residual is larger than $\min(\mu * ir\_outer\_tol\_factor, ir\_outer\_tol\_min)$. Double values in $[10^{-20}, 1]$. Default value: 0.01.

ir_outer_tol_min: iterative refinement (IR) is applied if the inf-norm of the full KKT residual is larger than $\min(\mu * ir\_outer\_tol\_factor, ir\_outer\_tol\_min)$. Double values in $[10^{-20}, 10^{20}]$. Default value: 10^{-6}.

ir_inner_cusolver_gs_scheme: Gram-Schmidt orthogonalization version for FMGRES:

- "mgs " (default): modified Gram-Schmidt
- "cgs2": reorthogonalized classical Gram-Schmidt (three synchs)
- "mgs_two_synch": two synch (stable) MGS
- "mgs_pm": post-modern MGS, two synchs

ginkgo_exec : string option with "cuda", "hip" or "reference" values selecting the hardware architecture to run the Ginkgo linear solver on. Only valid when parameter `linear_solver_sparse` is set to `ginkgo`. Default value: "reference".

cusolver_lu_factorization : so far, only 'klu' option is available.

cusolver_lu_refactorization: numerical refactorization function after sparsity pattern of factors is computed. 'glu' is and 'rf' is

- "glu " (default): experimental approach
- "rf": NVIDIA's stable refactorization

linear_solver_sparse_ordering: permutation to promote sparsity in the (Chol) factorization:

- "metis ": based on a wrapper of METIS_NodeND
- "symamd-eigen" (default): based on EIGEN implementation of approx. min. degree (AMD) orderings in its symmetric form
- "symamd-cuda ": based on CUDA implementation of AMD orderings in its symmetric form
- "symrcm ": based on CUDA implementation of reverse Cuthill-McKee orderings in its symmetric form
- "amd-ssparse ": symmetric approximate minimum degree (AMD) from Suite Sparse library
- "colamd-ssparse ": column approximate minimum degree (COLAMD) from Suite Sparse library

4.1.8 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": unsymmetric suitable for LU solvers (experimental).
- "condensed": symmetric condensed linear system that is suitable for sparse Cholesky solvers (available when no eq. constraints are present). See Section A.0.1 for more information
- "normaleqn": symmetric normal equation system that is suitable for sparse Cholesky solvers (available when problem is LP or separable convex QP). See Section A.0.2 for more information

The last five options are available only with option `Hessian` setting to `analyticalExact`.

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 the 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. It is fully tested with MDS NLPs; for other NLPs this option is experimental, should be used only by advanced users (as of v0.5). This option requires Umpire to be used as the memory manager with `mem_space` option being set to `device` or `um`.

mem_space: determines the memory space in which future internal linear algebra objects will be created. When `HiOp` is built with RAJA/Umpire, user can set this option to either 'default', 'host', 'device' or 'um', and internally the data of HiOp vectors/matrices will be managed by Umpire. If HiOp was built without RAJA/Umpire support, only 'default' is available for this option.:

- "default" (default): allocations are done by `HiOp` in the cpu's memory space;
- "host": allocations via Umpire in Umpire's "HOST" memory space, typically CPU memory;
- "device": allocations via Umpire in Umpire's "DEVICE" device memory space; the option is supported only for MDS NLPs and requires the user's model evaluation on the device;
- "um": allocations via Umpire's unified memory model, known as "UM".

callback_mem_space: determines the memory space to which `HiOp` will return the solutions. When `HiOp` is built with RAJA/Umpire and option **mem_space** is set to 'device', user can set this option to either 'default', 'host' or 'device'. If HiOp was built without RAJA/Umpire support, only 'default' is available for this option.:

- "default" (default): returns the solutions pointers on the cpu's memory space;
- "host": returns the solutions pointers allocated by Umpire in Umpire's "HOST" memory space, typically CPU memory;
- "device": returns the solutions pointers allocated by Umpire in Umpire's "DEVICE" device memory space;
- "um": returns the solutions pointers allocated by Umpire's unified memory, known as "UM". Only available when **mem_space** is set to 'um'.

4.1.9 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. After rescaling, the inf-norm of the gradient will equal the value of this option. Double values in $[10^{-20}, 10^{20}]$. Default value: 100.

scaling_max_obj_grad: if a positive value is given, the objective of user's NLP will be scaled so that the inf-norm of its gradient is equal to the value of this option. This option takes precedence over "scaling_max_grad". Double values in $[0, 10^{20}]$. Default value: 0.

scaling_max_con_grad: if a positive value is given, each constraint of user's NLP will be scaled so that the inf-norm of its gradient is equal to the value of this. This option takes precedence over "scaling_max_grad". Double values in $[0, 10^{20}]$. Default value: 0.

scaling_min_grad: a positive value for this option will be used as a lower bound for (and will overwrite) the scaling factors computed as instructed by options "scaling_max_grad", "scaling_max_obj_grad" and "scaling_max_con_grad". Double values in $[0, 10^{20}]$. Default value: 10^{-8}.

eq_relax_factor: perturbation of the equalities to allow posing them as inequalities. This factor is relative to the maximum between the magnitude of the equalities rhs and 1.0. Used only by 'hiopNlpSparseIneq' formulation class. Double values in $[10^{-15}, 1]$. Default value: 10^{-8}.

4.1.10 Checkpointing of the solver state and restarting

As detailed in Section 3.1.5, `HiOp` can save/load its internal state to/from disk. All the options in this section require an Axom-enabled build (use "-DHIOP_USE_AXOM=ON" with cmake) and are supported only by the quasi-Newton IPM solver (`hiopAlgFilterIPMQuasiNewton` class) for the `hiopInterfaceDenseConstraints` NLP formulation/interface.
checkpoint_save: Save state of NLP solver to file indicated by value of option "checkpoint_file". String values "yes" or "no", default "no".
checkpoint_load_on_start On (re)start the NLP solver will load checkpoint file specified by "checkpoint_file" option. String values "yes" or "no", default "no".
checkpoint_file Path to checkpoint file to load from or save to. If present, the character "#" is replaced with the iteration number at which the checkpointing is saved (but *not* when loaded). `HiOp` adds a ".root" extension internally if the value of the option is a directory. If this option is not specified and loading or saving checkpoints is enabled, `HiOp` will use a file named "hiop_state_chk".
checkpoint_save_every_N_iter Iteration frequency of saving checkpoints to disk if "checkpoint_save" is "yes". Takes positive integer values with a default value 10.

4.1.11 Miscellaneous options

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. Those larger values are explained in hiopLogger.hpp. The higher the value the more verbose the output will be.

print_options: string option with "yes", "no" or "short" values deciding whether the options should be printed on the output before solver (re)starts. Setting this option to 'yes' prints all the parameter names, values and descriptions, while 'short' only prints the parameter names and values. Default value: "no".

write_kkt: string option with "yes" or "no" values deciding whether `HiOp` writes internal KKT linear system (matrix, rhs, sol) to external files. Default value: "no".

time_kkt: string option with "on" or "off" values deciding whether `HiOp` turns on/off performance timers and reporting of the computational constituents of the KKT solve process. Default value: "off".

4.2 Options for PriDec solver

Here we list the options that are recognized by the `HiOp`'s PriDec solver.

4.2.1 Termination criteria and output

tolerance: maximum (absolute) error allowed. This value is compared against the decrease of the objective predicted by the solution to the subproblem with the approximation model ($q(x)$ in (23)). Double values in $[10^{-14}, 0.1]$. Default value: 10^{-5}.

max_iter: maximum number of iterations. Integer values between 1 to 10^6. Default value: 3000.

acceptable_tolerance: PriDec solver will terminate if the inf-norm of the decrease in objective value is below this value for `acceptable_iterations` many consecutive iterations. Double values in $[10^{-14}, 0.1]$. Default value: 10^{-3}.

acceptable_iterations: number of iterations passing the acceptable tolerance (see **acceptable_tolerance**) after which PriDec solver terminates. Integer values between 1 and 10^6. Default value: 25.

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 outputs warnings. The value of 2 is reserved for future use. A value of 3 will also output a table with PriDec solver's convergence metrics and trust-region type of measure of the quality of the approximation model 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.

4.2.2 Algorithm selection and parameters

alpha_min: lower bound for the scalar quadratic coefficient in the approximation model of the objective. It is a global value and has higher priority than the update rule of alpha. Double values in $[10^{-8}, 10^{3}]$. Default value: 10^{-5}.

alpha_max: upper bound for the scalar quadratic coefficient in the approximation model of the objective. It is a global value and has higher priority than the update rule of alpha. Double values in $[1, 10^{14}]$. Default value: 10^{6}. An assert error will be reported if **alpha_min** is bigger than **alpha_max**.

4.2.3 Miscellaneous options

mem_space: specifies the primary memory space in which PriDec solver's internal linear algebra objects will be created:

- "default" (default): allocations are done by `HiOp` in the cpu's memory space;
- "host": allocations via Umpire in Umpire's "HOST" memory space, typically CPU memory;
- "device": allocations via Umpire in Umpire's "DEVICE" device memory space;
- "um": allocations via Umpire's unified memory model, know as "UM".

⚠ The memory space for PriDec solver must match the memory space used by the master NLP solver, otherwise undefined behaviour will occur. This consistency is not checked by HiOp since it is impossible to do so when black-box NLP solvers are used for the master problem. It is the user's responsibility to ensure that the memory spaces match. When HiOp is used a master solver, the PriDec solver's `mem_space` option must match the master HiOp's option `mem_space`. When a CPU master solver is used with PriDec solver, the PriDec's `mem_space` option must be set to "default".

print_options: string option with "yes" or "no" values deciding whether the options should be printed on the output before solver (re)starts. Default value: "no".

5 Licensing and copyright

HiOp is free software; you can modify it and/or redistribute it under the terms of the following modified BSD 3-clause license:

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] N.-Y. Chiang and V. M. Zavala. An inertia-free filter line-search algorithm for large-scale nonlinear programming. *Computational Optimization and Applications*, 64(2):327–354, 2016.

[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.

[7] J. Wang, N. Chiang, and C. G. Petra. An asynchronous distributed-memory optimization solver for two-stage stochastic programming problems. In *2021 20th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 33–40, 2021.

[8] J. Wang and C. G. Petra. A sequential quadratic programming algorithm for nonsmooth problems with upper-$\mathcal{C}^2$ objective. *SIAM Journal on Optimization*, 33(3):2379–2405, 2023.

A Appendix

A.0.1 Condensed Linear System

The condensed approach supports sparse NLPs with no equality constraints of the form

$$\min_{x \in \mathbb{R}^n} \quad f(x) \tag{31}$$

$$[v_l] \qquad d_l \leq d(x) \leq d_u \qquad [v_u] \tag{32}$$

$$[z_l] \qquad x_l \leq x \leq x_u \qquad [z_u] \tag{33}$$

Here $f : \mathbb{R}^n \to \mathbb{R}$, $d : \mathbb{R}^n \to \mathbb{R}^{m_I}$. The bounds appearing in the inequality constraints (32) are assumed to be $d^l \in \mathbb{R}^{m_I} \cup \{-\infty\}$, $d^u \in \mathbb{R}^{m_I} \cup \{+\infty\}$, $d_i^l < d_i^u$, and at least of one of d_i^l and d_i^u are finite for all $i \in \{1, \ldots, m_I\}$. The bounds in (33) are such that $x^l \in \mathbb{R}^n \cup \{-\infty\}$, $x^u \in \mathbb{R}^n \cup \{+\infty\}$, and $x_i^l < x_i^u$, $i \in \{1, \ldots, n\}$. The quantities insides brackets are the Lagrange multipliers of the constraints. Whenever a bound is infinite, the corresponding multiplier is by convention zero. Internally, a slack variable s is introduced and the inequality constraints (32) are replaced by additional equality constraints and bound constraints:

$$d(x) = s \qquad [y_d] \tag{34}$$

$$[v_l] \qquad d_l \leq s \leq d_u \qquad [v_u] \tag{35}$$

⚠ **Note:** If equality constraints $c(x) = c_E$ are present, they will be slightly relaxed to inequalities $c_E - C_1 \leq c(x) \leq c_E + C_1$, where C_1 is a small positive perturbation that will be updated by `HiOp` internally. Consequently, with the condensed linear algebra, `HiOp` solves problems with equality constraints as inequality-only problems in the form of (31)-(33).

Using the notations from [**?**], the condensed linear system solves the most stable "xdycyd" KKT linear system

$$\begin{bmatrix} H + D_x + \delta_w I & 0 & J_d^T \\ 0 & D_d + \delta_w I & -I \\ J_d & -I & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta d \\ \Delta y_d \end{bmatrix} = \begin{bmatrix} r_x \\ r_d \\ r_{y_d} \end{bmatrix} \tag{36}$$

by solving the following sequence of linear systems

$$Q := H + D_x + \delta_w I + J_d^T (D_d + \delta_w I) J_d \tag{37}$$

$$Q \Delta x = r_x + J_d^T (D_d + \delta_w I) r_{y_d} + J_d^T r_d \tag{38}$$

$$\Delta d = J_d \Delta x - r_{y_d} \tag{39}$$

$$\Delta y_d = D_d \Delta d - r_d \tag{40}$$

Equation (38) is referred to as the condensed linear system. `HiOp` ensures that the matrix Q is positive definite by using a combination of dual and primal regularizations. Using the condensed linear algebra is therefore capable of using sparse Cholesky solvers. This is particularly relevant for GPU computations efficient and robust Cholesky solvers are currently more mature than an indefinite linear solvers (required by the "xdycyd" linear system). Currently, `HiOp` has GPU acceleration using cuSolverSP "cusolverSpDcsrlsvchol" from the NVIDIA's CUDA Toolkit.

A.0.2 Normal Equation

The normal equation approach supports sparse LPs or QPs in the form of (5)-(8), where $f : \mathbb{R}^n \to \mathbb{R}$ is a linear or a convex quadratic function with diagonal Hessian and $c : \mathbb{R}^n \to \mathbb{R}^{m_E}$ and $d : \mathbb{R}^n \to \mathbb{R}^{m_I}$ are affine functions.

⚠ **Note:** If equality constraints $c(x) = c_E$ are presented, they will be slightly relaxed to inequalities $c_E - C_1 \leq c(x) \leq c_E + C_1$, where C_1 is a small positive perturbation that will be updated by `HiOp` internally. Consequently, with the condensed linear algebra, `HiOp` solves problems with equality constraints as inequality-only problems in the form of (31)-(33).

Internally, normal equation solves the most stable 'xdycyd' KKT linear system

$$\begin{bmatrix} H + D_x + \delta_w I & 0 & J_c^T & J_d^T \\ 0 & D_d + \delta_w I & 0 & -I \\ J_c & 0 & 0 & 0 \\ J_d & -I & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta d \\ \Delta y_c \\ \Delta y_d \end{bmatrix} = \begin{bmatrix} r_x \\ r_d \\ r_{y_c} \\ r_{y_d} \end{bmatrix} \tag{41}$$

by solving the following linear system:

$$K \begin{bmatrix} \Delta y_c \\ \Delta y_d \end{bmatrix} = \begin{bmatrix} \tilde{r}_{y_c} \\ \tilde{r}_{y_d} \end{bmatrix}. \tag{42}$$

Above

$$K = \begin{bmatrix} J_c & 0 \\ J_d & -I \end{bmatrix} \begin{bmatrix} H + D_x + \delta_w I & 0 \\ 0 & D_d + \delta_w I \end{bmatrix}^{-1} \begin{bmatrix} J_c & 0 \\ J_d & -I \end{bmatrix}^T \tag{43}$$

and

$$\begin{bmatrix} \tilde{r}_{y_c} \\ \tilde{r}_{y_d} \end{bmatrix} = \begin{bmatrix} J_c & 0 \\ J_d & -I \end{bmatrix} \begin{bmatrix} H + D_x + \delta_w I & 0 \\ 0 & D_d + \delta_w I \end{bmatrix}^{-1} \begin{bmatrix} r_x \\ r_d \end{bmatrix} - \begin{bmatrix} r_{y_c} \\ r_{y_d} \end{bmatrix}. \tag{44}$$

Since matrix K (43) is forced to be positive definite by the algorithmic mechanism, the normal equation system (42) can be solved using Cholesky solvers. In particular, GPU acceleration is achieved by using cuSolverSP "cusolverSpDcsrlsvchol" solver from the NVIDIA's CUDA Toolkit.

Once Δy_c and Δy_d have been calculated, `HiOp` computes Δx and Δd from

$$\begin{bmatrix} \Delta x \\ \Delta d \end{bmatrix} = \begin{bmatrix} H + D_x + \delta_w I & 0 \\ 0 & D_d + \delta_w I \end{bmatrix}^{-1} \left(\begin{bmatrix} r_x \\ r_d \end{bmatrix} - \begin{bmatrix} J_c^T & J_d T \\ 0 & -I \end{bmatrix} \begin{bmatrix} \Delta y_c \\ \Delta y_d \end{bmatrix} \right). \tag{45}$$