Introduction to IPOPT:
A tutorial for downloading, installing, and using IPOPT

Revision number of this document: *Revision* : 2332

July 1, 2013

Abstract

This document is a guide to using IPOPT 3.11. It includes instructions on how to obtain and compile IPOPT, a description of the interface, user options, etc., as well as a tutorial on how to solve a nonlinear optimization problem with IPOPT.

History of this document

The initial version of this document was created by Yoshiaki Kawajir[1] as a course project for *47852 Open Source Software for Optimization*, taught by Prof. François Margot at Tepper School of Business, Carnegie Mellon University. After this, Carl Laird[2] has added significant portions, including the very nice tutorials. The current version is maintained by Stefan Vigerske[3] and Andreas Wächter[4].

Contents

[1] then Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh PA
[2] then Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh PA
[3] GAMS Software GmbH
[4] Department of Industrial Engineering and Management Sciences, Northwestern University

1 Introduction

Ipopt (Interior Point Optimizer, pronounced "Eye–Pea–Opt") is an open source software package for large-scale nonlinear optimization. It can be used to solve general nonlinear programming problems of the form

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

$$\text{s.t.} \quad g^L \leq g(x) \leq g^U \tag{2}$$

$$x^L \leq x \leq x^U, \tag{3}$$

where $x \in \mathbb{R}^n$ are the optimization variables (possibly with lower and upper bounds, $x^L \in (\mathbb{R} \cup \{-\infty\})^n$ and $x^U \in (\mathbb{R} \cup \{+\infty\})^n$), $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ is the objective function, and $g : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ are the general nonlinear constraints. The functions $f(x)$ and $g(x)$ can be linear or nonlinear and convex or non-convex (but should be twice continuously differentiable). The constraints, $g(x)$, have lower and upper bounds, $g^L \in (\mathbb{R} \cup \{-\infty\})^m$ and $g^U \in (\mathbb{R} \cup \{+\infty\})^m$. Note that equality constraints of the form $g_i(x) = \bar{g}_i$ can be specified by setting $g_i^L = g_i^U = \bar{g}_i$.

1.1 Mathematical Background

Ipopt implements an interior point line search filter method that aims to find a local solution of (1)-(3). The mathematical details of the algorithm can be found in several publications [6, 8, 13, 11, 10].

1.2 Availability

The Ipopt package is available from COIN-OR (`http://www.coin-or.org`) under the EPL (Eclipse Public License) open-source license and includes the source code for Ipopt. This means, it is available free of charge, also for commercial purposes. However, if you give away software including Ipopt code (in source code or binary form) and you made changes to the Ipopt source code, you are required to make those changes public and to clearly indicate which modifications you made. After all, the goal of open

source software is the continuous development and improvement of software. For details, please refer to the Eclipse Public License.

Also, if you are using IPOPT to obtain results for a publication, we politely ask you to point out in your paper that you used IPOPT, and to cite the publication [13]. Writing high-quality numerical software takes a lot of time and effort, and does usually not translate into a large number of publications, therefore we believe this request is only fair :). We also have space at the IPOPT project home page where we list publications, projects, etc., in which IPOPT has been used. We would be very happy to hear about your experiences

1.3 Prerequisites

In order to build IPOPT, some third party components are required:

- BLAS (Basic Linear Algebra Subroutines). Many vendors of compilers and operating systems provide precompiled and optimized libraries for these dense linear algebra subroutines. You can also get the source code for a simple reference implementation from `www.netlib.org` and have the IPOPT distribution compile it automatically. However, it is strongly recommended to use some optimized BLAS implemetion, for large problems this can make a runtime difference of an order of magnitude!

  Examples for efficient BLAS implementations are:

  - From hardware vendors:
    - ACML (AMD Core Math Library) by AMD
    - ESSL (Engineering Scientific Subroutine Library) by IBM
    - MKL (Math Kernel Library) by Intel
    - Sun Performance Library by Sun
  - Generic:
    - Atlas (Automatically Tuned Linear Algebra Software)
    - GotoBLAS

  You find more information on the web by googling them.

  Note: BLAS libraries distributed with Linux are usually not optimized.

- LAPACK (Linear Algebra PACKage). Also for LAPACK, some vendors offer precompiled and optimized libraries. But like with BLAS, you can get the source code from `http://www.netlib.org` and have the IPOPT distribution compile it automatically.

  Note that currently LAPACK is only required if you intend to use the quasi-Newton options in IPOPT. You can compile the code without LAPACK, but an error message will then occur if you try to run the code with an option that requires LAPACK. Currently, the LAPACK routines that are used by IPOPT are only `DPOTRF`, `DPOTRS`, and `DSYEV`.

  Note: LAPACK libraries distributed with Linux are usually not optimized.

- A sparse symmetric indefinite linear solver. IPOPT needs to obtain the solution of sparse, symmetric, indefinite linear systems, and for this it relies on third-party code.

  Currently, the following linear solvers can be used:

  - MA27 from the HSL Mathematical Software Library (see `http://www.hsl.rl.ac.uk`).
  - MA57 from the HSL Mathematical Software Library (see `http://www.hsl.rl.ac.uk`).

  - HSL_MA77 from the HSL Mathematical Software Library
    (see `http://www.hsl.rl.ac.uk`).
  - HSL_MA86 from the HSL Mathematical Software Library
    (see `http://www.hsl.rl.ac.uk`).
  - HSL_MA97 from the HSL Mathematical Software Library
    (see `http://www.hsl.rl.ac.uk`).
  - MUMPS (MUltifrontal Massively Parallel sparse direct Solver)
    (see `http://graal.ens-lyon.fr/MUMPS`)
  - The Parallel Sparse Direct Solver (PARDISO)
    (see `http://www.pardiso-project.org`).
    Note: The Pardiso version in Intel's MKL library does not yet support the features necessary for IPOPT.
  - The Watson Sparse Matrix Package (WSMP)
    (see `http://researcher.ibm.com/view_project.php?id=1426`)

  You should include at least one of the linear solvers above in order to run IPOPT, and if you want to be able to switch easily between different alternatives, you can compile IPOPT with all of them.

  The IPOPT library also has mechanisms to load the linear solvers MA27, MA57, HSL_MA77, HSL_MA86, HSL_MA97, and Pardiso from a shared library at runtime, if the library has not been compiled with them (see Section 2.3.6).

  NOTE: The solution of the linear systems is a central ingredient in Ipopt and the optimizer's performance and robustness depends on your choice. The best choice depends on your application, and it makes sense to try different options. Most of the solvers also rely on efficient BLAS code (see above), so you should use a good BLAS library tailored to your system. Please keep this in mind, particularly when you are comparing Ipopt with other optimization codes.

  If you are compiling MA57, HSL_MA77, HSL_MA86, HSL_MA97, or MUMPS within the IPOPT build system, you should also include the METIS linear system ordering package.

  Interfaces to other linear solvers might be added in the future; if you are interested in contributing such an interface please contact us! Note that IPOPT requires that the linear solver is able to provide the inertia (number of positive and negative eigenvalues) of the symmetric matrix that is factorized.

- Furthermore, IPOPT can also use the HSL package MC19 for scaling of the linear systems before they are passed to the linear solver. This may be particularly useful if IPOPT is used with MA27 or MA57. However, it is not required to have MC19 to compile IPOPT; if this routine is missing, the scaling is never performed.

- ASL (AMPL Solver Library). The source code is available at `www.netlib.org`, and the IPOPT makefiles will automatically compile it for you if you put the source code into a designated space. NOTE: This is only required if you want to use IPOPT from AMPL and want to compile the IPOPT AMPL solver executable.

For more information on third-party components and how to obtain them, see Section 2.3.

Since the IPOPT code is written in C++, you will need a C++ compiler to build the IPOPT library. We tried very hard to write the code as platform and compiler independent as possible.

In addition, the configuration script also searches for a Fortran compiler, since some of the dependencies above are written in Fortran. If all third party dependencies are available as self-contained libraries, those compilers are in principle not necessary. Also, it is possible to use the Fortran-to-C compiler `f2c` from `http://www.netlib.org/f2c` to convert Fortran 77 code to C, and compile the resulting C files with a C compiler and create a library containing the required third party dependencies.

When using GNU compilers, we recommend you use the same version numbers for `gcc`, `g++`, and `gfortran`. For `gfortran` specifically, we recommend versions newer than 4.5.2 (versions 4.5.1, 4.5.2, and before 4.2.0 are known to have bugs that caused issues with some of the newer Fortran 90 HSL linear solvers).

1.4 How to use Ipopt

If desired, the IPOPT distribution generates an executable for the modeling environment AMPL. As well, you can link your problem statement with IPOPT using interfaces for C++, C, or Fortran. IPOPT can be used with most Linux/Unix environments, and on Windows using Visual Studio .NET, Cygwin or MSYS/MinGW. In Section 3 this document demonstrates how to solve problems using IPOPT. This includes installation and compilation of IPOPT for use with AMPL as well as linking with your own code.

Additionally, the IPOPT distribution includes interfaces for

- `CUTEr`[5] (for solving problems modeled in SIF),
- `Java`, which allows you to use IPOPT from Java, see the files in the `Ipopt/contrib/JavaInterface` directory,
- `Matlab` (mex interface), which allows you to use IPOPT from Matlab, see `https://projects.coin-or.org/Ipopt/wiki/MatlabInterface`,
- and the `R` project for statistical computing, see the files in the `Ipopt/contrib/RInterface` directory.

There are also interfaces maintained by other people, among them are:

- AIMMS (modeling environment)

  The AIMMSlinks project on COIN-OR, maintained by Marcel Hunting, provides an interface for IPOPT within the AIMMS modeling tool, see `https://projects.coin-or.org/AIMMSlinks`.

- GAMS (modeling environment)

  The GAMSlinks project on COIN-OR, maintained by Stefan Vigerske, includes a GAMS interface for IPOPT, see `https://projects.coin-or.org/GAMSlinks`.

- .NET :

  An interface to the C# language is available here: `http://code.google.com/p/csipopt`

- OPTimization Interface (OPTI) Toolbox

  OPTI is a free `Matlab` toolbox for constructing and solving linear, nonlinear, continuous and discrete optimization problem and comes with IPOPT.

- Optimization Services

  The Optimization Services (OS) project provides a set of standards for representing optimization instances, results, solver options, and communication between clients and solvers, incl. IPOPT, in a distributed environment using Web Services, see `https://projects.coin-or.org/OS`.

- Python:

  An interface to the python language is available here: `https://github.com/xuy/pyipopt`

- Scilab (free Matlab-like environment):

  A Scilab interface is available here: `http://forge.scilab.org/index.php/p/sci-ipopt`

[5] see `http://cuter.rl.ac.uk/cuter-www`

1.5 More Information and Contributions

More and up-to-date information can be found at the IPOPT homepage,

`http://projects.coin-or.org/Ipopt`.

Here, you can find FAQs, some (hopefully useful) hints, a bug report system etc. The website is managed with Wiki, which means that every user can edit the webpages from the regular web browser. **In particular, we encourage Ipopt users to share their experiences and usage hints on the "Success Stories" and "Hints and Tricks" pages, or to list the publications discussing applications of Ipopt in the "Papers related to Ipopt" page**[6]. In particular, if you have trouble getting IPOPT work well for your optimization problem, you might find some ideas here. Also, if you had some difficulties to solve a problem and found a way around it (e.g., by reformulating your problem or by using certain IPOPT options), it would be very nice if you help other users by sharing your experience at the "Hints and Tricks" page.

IPOPT is an open source project, and we encourage people to contribute code (such as interfaces to appropriate linear solvers, modeling environments, or even algorithmic features). If you are interested in contributing code, please have a look at the COIN-OR contributions webpage[7] and contact the IPOPT project leader.

There is also a mailing list for IPOPT, available from the webpage

`http://list.coin-or.org/mailman/listinfo/ipopt`,

where you can subscribe to get notified of updates, to ask general questions regarding installation and usage, or to share your experience with IPOPT. You might want to look at the archives before posting a question. An easy way to search the archive with Google is to specify

"`site:http://list.coin-or.org/pipermail/ipopt`"

in addition to your keywords in the search string.

We try to answer questions posted to the mailing list in a reasonable manner. Please understand that we cannot answer all questions in detail, and because of time constraints, we may not be able to help you model and debug your particular optimization problem.

A short tutorial on getting started with IPOPT is also available [9].

1.6 History of Ipopt

The original IPOPT (Fortran version) was a product of the dissertation research of Andreas Wächter [8], under the supervision of Lorenz T. Biegler at the Chemical Engineering Department at Carnegie Mellon University. The code was made open source and distributed by the COIN-OR initiative, which is now a non-profit corporation. IPOPT has been actively developed under COIN-OR since 2002.

To continue natural extension of the code and allow easy addition of new features, IBM Research decided to invest in an open source re-write of IPOPT in C++. With the help of Carl Laird, who came to the Mathematical Sciences Department at IBM Research as a summer intern in 2004 and 2005 during his PhD studies, the code was re-implemented from scratch.

The new C++ version of the IPOPT optimization code (IPOPT 3.0.0 and beyond) was maintained at IBM Research and remains part of the COIN-OR initiative. The development on the Fortran version has ceased, but the source code can still be downloaded from `http://www.coin-or.org/download/source/Ipopt-Fortran`.

[6] Since we had some malicious hacker attacks destroying the content of the web pages in the past, you are now required to enter a user name and password; simply follow the instructions on top of the main project page.

[7] see `http://www.coin-or.org/contributions.html`

2 Installing Ipopt

The following sections describe the installation procedures on UNIX/Linux systems. For installation instructions on Windows see Section 2.5.

Additional hints on installing IPOPT and its various interfaces is available on the IPOPT and CoinHelp wiki pages, in particular

- IPOPT compilation hints:
  `https://projects.coin-or.org/Ipopt/wiki/CompilationHints`
- Current configuration and installation issues for COIN-OR projects:
  `https://projects.coin-or.org/BuildTools/wiki/current-issues`

2.1 Getting System Packages (Compilers, ...)

Many Linux distributions will come with all necessary tools. All you should need to do is check the compiler versions. On a Debian-based distribution, you can obtain all necessary tools with the following command:

```
sudo apt-get install gcc g++ gfortran subversion patch wget
```

Replace `apt-get` with your relevant package manager, e.g. `yum` for Red Hat-based distributions, `zypper` for SUSE, etc. The `g++` and `gfortran` compilers may need to be specified respectively as `gcc-c++` and `gcc-gfortran` with some package managers.

On Mac OS X, you need either the Xcode Command Line Tools, available at `https://developer.apple.com/downloads` after registering as an Apple Developer, or a community alternative such as `https://github.com/kennethreitz/osx-gcc-installer/downloads` to install the gcc and g++ compilers. If you have a recent version of Xcode installed, the Command Line Tools are available under Preferences, Downloads. In Xcode 3.x, the Command Line Tools are contained in the optional item "UNIX Dev Support" during Xcode installation. These items unfortunately do not come with a Fortran compiler, but you can get `gfortran` from `http://gcc.gnu.org/wiki/GFortranBinaries#MacOS`. We have been able to compile IPOPT using default Xcode versions of `gcc` and `g++` and a newer version of `gfortran` from this link, but consistent version numbers may be an issue in future cases.

2.2 Getting the Ipopt Code

IPOPT is available from the COIN-OR subversion repository. You can either download the code using `svn` (the *subversion* client similar to CVS) or simply retrieve a tarball (compressed archive file). While the tarball is an easy method to retrieve the code, using the *subversion* system allows users the benefits of the version control system, including easy updates and revision control.

2.2.1 Getting the Ipopt code via subversion

Of course, the *subversion* client must be installed on your system if you want to obtain the code this way (the executable is called `svn`); it is already installed by default for many recent Linux distributions. Information about *subversion* and how to download it can be found at `http://subversion.apache.org`.

To obtain the IPOPT source code via subversion, change into the directory in which you want to create a subdirectory `Ipopt` with the IPOPT source code. Then follow the steps below:

1. Download the code from the repository
   `$ svn co https://projects.coin-or.org/svn/Ipopt/stable/3.11 CoinIpopt`
   Note: The `$` indicates the command line prompt, do not type `$`, only the text following it.
2. Change into the root directory of the IPOPT distribution
   `$ cd CoinIpopt`

In the following, "`$IPOPTDIR`" will refer to the directory in which you are right now (output of `pwd`).

2.2.2 Getting the Ipopt code as a tarball

To use the tarball, follow the steps below:

1. Download the desired tarball from `http://www.coin-or.org/download/source/Ipopt`, it has the form `Ipopt-`*x.y.z*`.tgz`, where *x.y.z* is the version number, such as `3.11.0`. There might also be daily snapshot from the stable branch. The number of the latest official release can be found on the IPOPT Trac page.

2. Issue the following commands to unpack the archive file:
   `$ gunzip Ipopt-`*x.y.z*`.tgz`
   `$ tar xvf Ipopt-`*x.y.z*`.tar`
   Note: The `$` indicates the command line prompt, do not type `$`, only the text following it.

3. Rename the directory you just extracted:
   `$ mv Ipopt-`*x.y.z* `CoinIpopt`

4. Change into the root directory of the IPOPT distribution
   `$ cd CoinIpopt`

In the following, "`$IPOPTDIR`" will refer to the directory in which you are right now (output of `pwd`).

2.3 Download External Code

IPOPT uses a few external packages that are not included in the IPOPT source code distribution, namely ASL (the AMPL Solver Library if you want to compile the IPOPT AMPL solver executable), Blas, Lapack.

IPOPT also requires at least one linear solver for sparse symmetric indefinite matrices. There are different possibilities, see Sections 2.3.2–2.3.5. **It is important to keep in mind that usually the largest fraction of computation time in the optimizer is spent for solving the linear system, and that your choice of the linear solver impacts Ipopt's speed and robustness. It might be worthwhile to try different linear solver to experiment with what is best for your application.**

Since this third party software is released under different licenses than IPOPT, we cannot distribute their code together with the IPOPT packages and have to ask you to go through the hassle of obtaining it yourself (even though we tried to make it as easy for you as we could). Keep in mind that it is still your responsibility to ensure that your downloading and usage of the third party components conforms with their licenses.

Note that you only need to obtain the ASL if you intend to use IPOPT from AMPL. It is not required if you want to specify your optimization problem in a programming language (C++, C, or Fortran). Also, currently, Lapack is only required if you intend to use the quasi-Newton options implemented in IPOPT.

2.3.1 Download BLAS, LAPACK and ASL

Note: It is **highly recommended that you obtain an efficient implementation of the BLAS library**, tailored to your hardware; Section 1.3 lists a few options. Assuming that your precompiled efficient BLAS library is `libmyblas.a` in `$HOME/lib`, you need to add the flag `--with-blas="-L$HOME/lib -lmyblas"` when you run `configure` (see Section 2.4). Some of those libraries also include LAPACK.

If you have the download utility `wget` installed on your system (or `ftp` on Mac OS X), retrieving source code for BLAS (the inefficient reference implementation, not required if you have a precompiled library), as well as LAPACK and ASL is straightforward using scripts included with the ipopt distribution. These scripts download the required files from the Netlib Repository (`http://www.netlib.org`).

```
$ cd $IPOPTDIR/ThirdParty/Blas
$ ./get.Blas
$ cd ../Lapack
$ ./get.Lapack
```

```
$ cd ../ASL
$ ./get.ASL
```

If you do not have `wget` (or `ftp` on Mac OS X) installed on your system, please read the `INSTALL.*` files in the `$IPOPTDIR/ThirdParty/Blas`, `$IPOPTDIR/ThirdParty/Lapack` and `$IPOPTDIR/ThirdParty/ASL` directories for alternative instructions.

If you are having firewall issues with `wget`, try opening the `get.<library>` scripts and replace the line `wgetcmd=wget` with `wgetcmd="wget --passive-ftp"`.

If you are getting permissions errors from tar, try opening the `get.<library>` scripts and replace any instances of `tar xf` with `tar --no-same-owner -xf`.

2.3.2 Download HSL Subroutines

There are two versions of HSL available:

HSL Archive contains outdated codes that are freely available for personal commercial or non-commercial usage. Note that you may not redistribute these codes in either source or binary form without purchasing a licence from the authors. This version includes MA27, MA28, and MC19.

HSL 2011 contains more modern codes that are freely available for academic use only. This version includes the codes from the HSL Archive and additionally MA57, HSL_MA77, HSL_MA86, and HSL_MA97. IPOPT supports the HSL 2011 codes from 2012 and 2013, the support for the versions from 2012 may be dropped in a future release.

To obtain the HSL code, you can follow the following steps:

1. Go to `http://hsl.rl.ac.uk/ipopt`.
2. Choose whether to download either the Archive code or the HSL 2011 code. To download, select the relevant "source" link.
3. Follow the instructions on the website, read the license, and submit the registration form.
4. Wait for an email containing a download link (this should take no more than one working day).

You may either:

- Compile the HSL code as part of IPOPT. See the instructions below.
- Compile the HSL code separately either before or after the IPOPT code and use the shared library loading mechanism. See the documentation distributed with the HSL package for information on how to do so.

To compile the HSL code as part of IPOPT, unpack the archive, then move and rename the resulting directory so that it becomes `$IPOPTDIR/ThirdParty/HSL/coinhsl`. IPOPT may then be configured as normal.

Note: Whereas it is essential to have at least one linear solver, the package MC19 could be omitted (with the consequence that you cannot use this method for scaling the linear systems arising inside the IPOPT algorithm). By default, MC19 is only used to scale the linear system when using one of the HSL solvers, but it can also be switched on for other linear solvers (which usually have internal scaling mechanisms). Further, also the package MA28 can be omitted, since it is used only in the experimental dependency detector, which is not used by default.

Note: If you are an academic or a student, we recommend you download the HSL 2011 package as this ensures you have access to the full range of solvers. MA57 can be considerably faster than MA27 on some problems.

Yet another note: If you have a precompiled library containing the HSL codes, you can specify the directory with the header files and the linker flags for this library with the `--with-hsl-incdir` and `--with-hsl-lib` flags for the `configure` script described in Section 2.4.

2.3.3 Obtaining the MUMPS Linear Solver

You can also use the (public domain) sparse linear solver MUMPS. Please visit the MUMPS home page `http://graal.ens-lyon.fr/MUMPS` for more information about the solver. MUMPS is provided as Fortran 90 and C source code. You need to have a Fortran 90 compiler (for example, the GNU compiler `gfortran` is a free one) to be able to use it.

You can obtain the MUMPS code by running the script `$IPOPTDIR/ThirdParty/Mumps/get.Mumps` if you have `wget` (or `ftp` on Mac OS X) installed in your system. Alternatively, you can get the latest version from the MUMPS home page and extract the archive in the directory `$IPOPTDIR/ThirdParty/Mumps`. The extracted directory usually has the MUMPS version number in it, so you need to rename it to `MUMPS` such that you have a file called `$IPOPTDIR/ThirdParty/Mumps/MUMPS/README`.

Once you put the MUMPS source code into the correct place, the IPOPT configuration scripts will automatically detect it and compile MUMPS together with IPOPT, *if your Fortran compiler is able to compile Fortran 90 code.*

Note: MUMPS will perform better with METIS, see Section 2.3.7.

Note: MUMPS uses interally a fake implementation of MPI. If you are using IPOPT within an MPI program together with MUMPS, the code will not run. You will have to modify the MUMPS sources so that the MPI symbols inside the MUMPS code are renamed.

2.3.4 Obtaining the Linear Solver Pardiso

If you would like to compile IPOPT with the Parallel Sparse Direct Linear Solver (Pardiso), you need to obtain the Pardiso library for your operating system. Information about Pardiso can be found at `http://www.pardiso-project.org`.

You can obtain a limited time license of Pardiso for academic or evaluation purposes or buy a non-profit or commercial license. Instructions for this are on the above mentioned website; make sure you read the license agreement before filling out the download form.

Note: Pardiso is included in Intel's MKL library. However, that version does not include the changes done by the Pardiso developers to make the linear solver work smoothly with IPOPT.

Please consult Appendix 2.9 to find out how to configure your IPOPT installation to work with Pardiso.

2.3.5 Obtaining the Linear Solver WSMP

If you would like to compile IPOPT with the Watson Sparse Matrix Package (WSMP), you need to obtain the WSMP library for your operating system. Information about WSMP can be found at `http://www.research.ibm.com/projects/wsmp`.

At this website you can download the library for several operating systems including a trial license key for 90 days that allows you to use WSMP for "educational, research, and benchmarking purposes by non-profit academic institutions" or evaluation purposes by commercial organizations; make sure you read the license agreement before using the library. Once you obtained the library and license, please check if the version number of the library matches the one on the WSMP website.

If a newer version is announced on that website, you can (and probably should) request the current version by sending a message to `wsmp@watson.ibm.com`. Please include the operating system and other details to describe which particular version of WSMP you need.

Note: Only the interface to the shared-memory version of WSMP is currently supported.

Please consult Appendix 2.9 to find out how to configure your IPOPT installation to work with WSMP.

2.3.6 Using the Linear Solver Loader

By default, IPOPT will be compiled with a mechanism, the Linear Solver Loader, which can dynamically load shared libraries with MA27, MA57, HSL_MA77, HSL_MA86, HSL_MA97, or the Pardiso linear solver

at runtime[8]. This means, if you obtain one of those solvers after you compiled IPOPT, you don't need to recompile to use it. Instead, you can just put a shared library called `libhsl.so` or `libpardiso.so` into the shared library search path, `LD_LIBRARY_PATH`. These are the names on most UNIX platforms, including Linux. On Mac OS X, the names are `libhsl.dylib`, `libpardiso.dylib`, and `DYLD_LIBRARY_PATH`. On Windows, the names are `libhsl.dll`, `libpardiso.dll`, and `PATH`.

The Pardiso shared library can be downloaded from the Pardiso website. To create a shared library containing the HSL linear solvers, read the instructions in `$IPOPTDIR/ThirdParty/HSL/INSTALL.HSL`.

2.3.7 Obtaining METIS

The linear solvers MA57, HSL_MA77, HSL_MA86, HSL_MA97, and MUMPS can make use of the matrix ordering algorithms implemented in METIS (see `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`). If you are using one of these linear solvers, you should obtain the METIS source code and put it into `$IPOPTDIR/ThirdParty/Metis`. Read the `INSTALL.Metis` file in that directory, and if you have the `wget` utility (or `ftp` on Mac OS X) installed on your system, you can download the code by running the `./get.Metis` script.

Note, that **only the older METIS 4.x version[9] is supported** by MA57, HSL_MA77, HSL_MA86, HSL_MA97, MUMPS, and the build system. The `./get.Metis` script takes care of downloading the right METIS version.

2.4 Compiling and Installing Ipopt

IPOPT can be easily compiled and installed with the usual `configure`, `make`, `make install` commands. We follow the procedure that is used for most of the COIN-OR projects, based on the GNU autotools. At `https://projects.coin-or.org/CoinHelp` you can find a general description of the tools.

Below are the basic steps for the IPOPT compilation that should work on most systems. For special compilations and for some troubleshooting see Appendix 2.9 and consult the generic COIN-OR help page `https://projects.coin-or.org/CoinHelp` before submitting a ticket or sending a message to the mailing list.

1. Create a directory where you want to compile IPOPT, for example
   `$ mkdir $IPOPTDIR/build`
   and go into this direcrory
   `$ cd $IPOPTDIR/build`

   Note: You can choose any location, including `$IPOPTDIR` itself, as the location of your compilation. However, on COIN-OR we recommend to keep the source and compiled files separate.

2. Run the configure script
   `$ $IPOPTDIR/configure`

   One might have to give options to the configure script, e.g., in order to choose a non-default compiler, or to tell it where some third party code is installed, see Appendix 2.9.

   If the last output line reads "`configure: Main configuration of Ipopt successful`" then everything worked fine. Otherwise, look at the screen output, have a look at the `config.log` output files and/or consult Appendix 2.9.

   The default configure (without any options) is sufficient for most users that downloaded the source code for the linear solver. If you want to see the configure options, consult Appendix 2.9, and also visit the generic COIN-OR configuration instruction page at

   `https://projects.coin-or.org/CoinHelp/wiki/user-configure`

[8] This is not enabled if you compile IPOPT with the MS Visual Studio project files provided in the IPOPT distribution. Further, if you have problems compiling this new feature, you can disable this by specifying `--disable-linear-solver-loader` for the `configure` script

[9] `http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/OLD/metis-4.0.3.tar.gz`

3. Build the code
   `$ make`

   Note: If you are using GNU make, you can also try to speed up the compilation by using the `-jN` flag (e.g., `make -j3`), where `N` is the number of parallel compilation jobs. A good number for `N` is the number of available processors plus one. Under some circumstances, this fails, and you might have to re-issue the command, or omit the `-j` flag.

4. If you want, you can run a short test to verify that the compilation was successful. For this, you just enter
   `$ make test`
   This will test if the AMPL solver executable works (if you got the ASL code) and if the included C++, C, and Fortran examples work.

   Note: The `configure` script is not able to automatically determine the C++ runtime libraries for the C++ compiler. For certain compilers we enabled default values for this, but those might not exist or be wrong for your compiler. In that case, the C and Fortran example in the test will most probably fail to compile. If you don't want to hook up the compiled IPOPT library to some Fortran or C code that you wrote you don't need to worry about this. If you do want to link the IPOPT library with a C or Fortran compiler, you need to find out the C++ runtime libraries (e.g., by running the C++ compiler in verbose mode for a simple example program) and run `configure` again, and this time specify all C++ runtime libraries with the `CXXLIBS` variable (see also Appendix 2.9).

5. Install IPOPT
   `$ make install`
   This installs

   - the IPOPT AMPL solver executable (if ASL source was downloaded) in `$IPOPTDIR/build/bin`,
   - the IPOPT library (`libipopt.so`, `libipopt.a` or similar) and all its dependencies (MUMPS, HSL, Metis libraries) in `$IPOPTDIR/build/lib`,
   - text files `ipopt_addlibs_cpp.txt`, `ipopt_addlibs_c.txt`, and `ipopt_addlibs_f.txt` in `$IPOPTDIR/build/share/coin/doc/Ipopt` each containing a line with linking flags that are required for linking code with the IPOPT library for C++, C, and Fortran main programs, respectively. (This is only for convenience if you want to find out what additional flags are required, for example, to include the Fortran runtime libraries with a C++ compiler.)
   - the necessary header files in `$IPOPTDIR/build/include/coin`.

   You can change the default installation directory (here `$IPOPTDIR/build`) to something else (such as `/usr/local`) by using the `--prefix` switch for `configure`.

6. (Optional) Install IPOPT for use with `CUTEr`
   If you have `CUTEr` already installed on your system and you want to use IPOPT as a solver for problems modeled in `SIF`, type
   `$ make cuter`
   This assumes that you have the environment variable `MYCUTER` defined according to the `CUTEr` instructions. After this, you can use the script `sdipo` as the `CUTEr` script to solve a `SIF` model.

Note: The above procedures show how to compile the code in directories separate from the source files. This comes in handy when you want to compile the code with different compilers, compiler options, or different operating system that share a common file system. To use this feature, change into the directory where you want to compile the code, and then type `$IPOPTDIR/configure` with all the options. For this, the directories with the IPOPT source must not have any configuration and compiled code.

2.5 Installation on Windows

There are several ways to install IPOPT on Windows systems. The first two options, described in Sections 2.5.1 and 2.5.2, are to use Cygwin (see `http://www.cygwin.com`), which offers a comprehensive UNIX-like environment on Windows and in which the installation procedure described earlier in this section can be used. If you want to use the (free) GNU compilers, follow the instructions in Section 2.5.1. If you have the Microsoft C++ compiler and possibly a "native" Fortran compiler (e.g., the Intel Fortran compiler) and want to use those to compile IPOPT, please see Section 2.5.2. If you use MSYS/MinGW (a light-weight UNIX-like environment for Windows), please consider the notes in Section 2.5.3. If you want to compile the IPOPT `mex` interface to MATLAB, then we recommend to use the MSYS/MinGW option.

Note: Some binaries for IPOPT are available on the COIN-OR website at `http://www.coin-or.org/download/binary/Ipopt`. There, also precompiled versions of IPOPT as DLLs (generated from the MSVS solution in IPOPT's subdirectory `$IPOPTDIR/Ipopt/MSVisualStudio/v8-ifort`) are available. Look at the `README` files for details. An example how to use these DLLs from your own MSVS project is in `$IPOPTDIR/Ipopt/MSVisualStudio/BinaryDLL-Link-Example`.

2.5.1 Installation with Cygwin using GNU compilers

Cygwin is a Linux-like environment for Windows; if you don't know what it is you might want to have a look at the Cygwin homepage, `http://www.cygwin.com`.

It is possible to build the IPOPT AMPL solver executable in Cygwin for general use in Windows. You can also hook up IPOPT to your own program if you compile it in the Cygwin environment[10].

If you want to compile IPOPT under Cygwin, you first have to install Cygwin on your Windows system. This is pretty straight forward; you simply download the "setup" program from `http://www.cygwin.com` and start it.

Then you do the following steps (assuming here that you don't have any complications with firewall settings etc - in that case you might have to choose some connection settings differently):

1. Click next
2. Select "install from the internet" (default) and click next
3. Select a directory where Cygwin is to be installed (you can leave the default) and choose all other things to your liking, then click next
4. Select a temp dir for Cygwin setup to store some files (if you put it on your desktop you will later remember to delete it)
5. Select "direct connection" (default) and click next
6. Select some mirror site that seems close by to you and click next
7. OK, now comes the complicated part:
   You need to select the packages that you want to have installed. By default, there are already selections, but the compilers are usually not pre-chosen. You need to make sure that you select the GNU compilers (for Fortran, C, and C++), Subversion, and some additional tools. For this, get the following packages from the associated branches:
   - "Devel": `gcc4`
   - "Devel": `gcc4-fortran`
   - "Devel": `pkg-config`
   - "Devel": `subversion`

[10] It is also possible to build an IPOPT DLL that can be used from non-cygwin compilers, but this is not (yet?) supported.

- "Archive": `unzip`
- "Utils": `patch`
- "Web": `wget`

When a Resolving Dependencies window comes up, be sure to "Select required packages (RECOMMENDED)". This will automatically also select some other packages.

8. Then you click on next, and Cygwin will be installed (follow the rest of the instructions and choose everything else to your liking). At a later point you can easily add/remove packages with the setup program.

9. The version of the GNU Make utility provided by the Cygwin installer will not work. Therefore, you need to download the fixed version from `http://www.cmake.org/files/cygwin/make.exe` and save it to `C:\cygwin\bin`. Double-check this new version by typing `make --version` in a Cygwin terminal (see next point). If you get an error `-bash: /usr/bin/make: Bad address`, then try `http://www.cmake.org/files/cygwin/make.exe-cygwin1.7` instead, rename it to `make.exe` and move it to `C:\cygwin\bin`. (Replace `C:\cygwin` with your installation location if different.)

10. Now that you have Cygwin, you can open a Cygwin window, which is like a UNIX shell window.

11. Now you just follow the instructions in the beginning of Section 2: You download the Ipopt code into your Cygwin home directory (from the Windows explorer that is usually something like `C:\Cygwin\home\your_user_name`). After that you obtain the third party code (as on Linux/UNIX), type

    ```
    ./configure
    ```

    and

    ```
    make install
    ```

    in the correct directories, and hopefully that will work. The Ipopt AMPL solver executable will be in the subdirectory `bin` (called "`ipopt.exe`"). If you want to set the installation, type

    ```
    make test
    ```

2.5.2 Installation with Cygwin using the MSVC++ compiler

This section describes how you can compile Ipopt with the Microsoft Visual C++ compiler under Cygwin. Here you have two options for compiling the Fortran code in the third party dependencies:

- Using a Windows Fortran compiler, e.g., the Intel Fortran compiler, which is also able to compile Fortran 90 code. This would allow you to compile the MUMPS linear solver if you desire to do so.
- Using the `f2c` Fortran to C compiler, available for free at Netlib (see `http://www.netlib.org/f2c`). This can only compile Fortran 77 code (i.e., you won't be able to compile MUMPS). Before doing the following installation steps, you need to follow the instructions in `$IPOPTDIR/BuildTools/compile_f2c/INSTALL`.

Once you have settled on this, do the following:

1. Follow the instructions in Section 2.5.1 until Step 11 and stop after your downloaded the third party code.

2. Now you need to make sure that Cygwin knows about the native compilers. For this you need to edit the file `cygwin.bat` in the Cygwin base directory (usually `C:\cygwin`). Here you need to add a line like the following:

   ```
   call "C:\Program Files\Microsoft Visual Studio 8\VC\vcvarsall.bat"
   ```

On my computer, this sets the environment variables so that I can use the MSVC++ compiler.

If you want to use also a native Fortran compiler, you need to include something like this

```
call "C:\Program Files\Intel\Fortran\compiler80\IA32\BIN\ifortvars.bat"
```

You might have to search around a bit. The important thing is that, after your change, you can type "`cl`" in a newly opened Cygwin windows, and it finds the Microsoft C++ compiler (and if you want to use it, the Fortran compiler, such as the Intel's `ifort`).

3. Run the configuration script, and tell it that you want to use the native compilers:

   ```
   ./configure --enable-doscompile=msvc
   ```

   Make sure the last message is

   ```
   Main Ipopt configuration successful
   ```

4. Now you can compile the code with

   `make`,

   test the installation with

   `make test`,

   and install everything with

   `make install`

2.5.3 Installation with MSYS/MinGW

You can compile IPOPT also under MSYS/MinGW, which is another, more light-weight UNIX-like environment for Windows. It can be obtained from `http://www.mingw.org/`.

If you want to use MSYS/MinGW to compile IPOPT with native Windows compilers (see Section 2.5.2), all you need to install is the basic version[11]. If you also want to use the GNU compilers, you need to install those as well, of course.

A compilation with the GNU compilers works just like with any other UNIX system, as described in Section 2.4. That is, during the installation, select (at least) the C Compiler, C++ Compiler, Fortran Compiler, MSYS Basic System, and the MinGW Developer ToolKit. Additionally, `wget` and `unzip` should be installed with the following command in an MSYS terminal:

```
mingw-get install msys-wget msys-unzip
```

If you want to use the native MSVC++ compiler (with `f2c` or a native Fortran compiler), you essentially follow the steps outlined in Section 2.5.2. Additionally, you need to make sure that the environment variables are set for the compilers (see step 2), this time adding the line to the `msys.bat` file.

For a 64-bit build, you will need to install also a MinGW-64 distribution. We recommend TDM-GCC, which is available from `http://sourceforge.net/projects/tdm-gcc/files/TDM-GCC%20Installer/tdm-gcc-webdl.exe/download`. Install MinGW-64 in a different folder than your existing 32-bit MinGW installation! The components you need are: `core` (under `gcc`), `c++` (under `gcc`), `fortran` (under `gcc`), `openmp` (under `gcc`, necessary if you want to use any multi-threaded linear solvers), `binutils`, and `mingw64-runtime`.

After MinGW-64 is installed, open the file `C:\MinGW\msys\1.0\etc\fstab`, and replace the line

```
C:\MinGW\     /mingw
```

with

```
C:\MinGW64\   /mingw
```

(Replace paths with your installation locations if different.)

[11] a convenient Windows install program is available from `http://sourceforge.net/projects/mingw/files/Installer/mingw-get-inst/`

2.6 Compiling and Installing the Java Interface JIpopt

based on documentation by Rafael de Pelegrini Soares[12]

JIPOPT uses the Java Native Interface (JNI), which is a programming framework that allows Java code running in the Java Virtual Machine (JVM) to call and be called by native applications and libraries written in languages such as C and C++. JIPOPT requires Java 5 or higher.

After building and installing IPOPT, the JIPOPT interface can be build by setting the environment variable `JAVA_HOME` to the directory that contains your JDK, changing to the JIPOPT directory in your IPOPT build, and issuing `make`, e.g.,

```
export JAVA_HOME=/usr/lib/jvm/java-1.5.0
cd $IPOPTDIR/build/Ipopt/contrib/JavaInterface
make
```

This will generate the Java class `org/coinor/Ipopt.class`, which you will need to add into your Java project and the shared object `lib/libjipopt.so` (on Linux/UNIX) or `lib/libjipopt.dylib` (on Mac OS X) or the DLL `lib/jipopt.dll` (on Windows). In order to test your JIPOPT library you can run two example problems by issuing the command `make test` inside the JIPOPT directory.

NOTE: The JIPOPT build procedure currently cannot deal with spaces in the path to the JDK. If you are on Windows and have Java in a path like `C:\Program Files\Java`, try setting `JAVA_HOME` to the DOS equivalent `C:\Progra~1` (or similar).

NOTE: JIPOPT needs to be able to load the IPOPT library dynamically at runtime. Therefore, IPOPT must have been compiled with the `-fPIC` compiler flag. While per default, an Ipopt shared library is compiled with this flag, for a configuration of IPOPT in debug mode (`--enable-debug`) or as static library (`--disable-shared`), the configure flag `--with-pic` need to be used to enable compilation with `-fPIC`.

2.7 Compiling and Installing the R Interface `ipoptr`

The `ipoptr` interface can be build after IPOPT has been build and installed. In the best case, it is sufficient to execute the following command in R:

```
install.packages('$IPOPTDIR/build/Ipopt/contrib/RInterface', repos=NULL, type='source')
```

In certain situations, however, it can be necessary to setup the dynamic library load path to the path where the IPOPT library has been installed, e.g.,

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$IPOPTDIR/build/lib
```

NOTE: R needs to be able to load the IPOPT library dynamically at runtime. Therefore, IPOPT must have been compiled with the `-fPIC` compiler flag. While per default, an Ipopt shared library is compiled with this flag, for a configuration of IPOPT in debug mode (`--enable-debug`) or as static library (`--disable-shared`), the configure flag `--with- pic` need to be used to enable compilation with `-fPIC`.

After installation of the `ipoptr` package, it should be possible to load the package in R and to view the help page:

```
> library('ipoptr')
> ?ipoptr
```

2.8 Compiling and Installing the MATLAB interface

based on documentation by Peter Carbonetto[13]*, Tony Kelman*[14]*, and Ray Zimmerman*

[12] VRTech Industrial Technologies
[13] University of British Columbia
[14] University of California, Berkeley

The MATLAB interface to IPOPT uses the `mex` interface of MATLAB. It has been tested on MATLAB versions 7.2 through 7.7. It might very well work on earlier versions of MATLAB, but there is also a good chance that it will not. It is unlikely that the software will run with versions prior to MATLAB 6.5.

First, note that some binaries of IPOPT `mex` files are available for download at `http://www.coin-or.org/download/binary/Ipopt`. Further, the OPTI Toolbox (`http://www.i2c2.aut.ac.nz/Wiki/OPTI`) comes with a precompiled MATLAB interface for IPOPT on Windows.

2.8.1 Setting up `mex`

To build the interface by yourself, you will need to have MATLAB installed on your system and have it configured to build `mex` files, see `http://www.mathworks.com/support/tech-notes/1600/1605.html` for detail on how to set this up.

Ipopt 3.11 has added Makefile options to automate fixes for commonly-encountered issues with building the MATLAB interface. On Mac OS X or Windows, the file `mexopts.sh` (`mexopts.bat` on Windows) will need to be modified. This is performed automatically by calling `make mexopts` in the `$IPOPTDIR/build/Ipopt/contrib/MatlabInterface/src` directory. No changes will be made if you already have a `mexopts.sh` file in that directory. If you need to make these modifications manually, follow the steps below.

For Mac OS X, the following procedure has been reported: First, one executes a command like

```
/Applications/MATLAB_R2012.app/bin/mex -setup
```

This creates a `mexopts.sh` file in the `/.matlab/R2010` directory. Copy that file to the directory `$IPOPTDIR/build/Ipopt/contrib/MatlabInterface/src` and modify it as follows.

- In the `maci` section (32 bit builds) or the `maci64` section (64 bit builds), change both instances of `libgfortran.dylib` to `libgfortran.a` in the `FC_LIBDIR` line (in case your Fortran compiler only comes with static libraries).
- Remove all occurrences of `-isysroot $SDKROOT` or `-Wl,-syslibroot,$SDKROOT` in case the hard-coded version of the Xcode SDK that Matlab expects does not match what you have installed on your system.
- Remove all occurrences of `-arch $ARCHS` in case you are using a GNU compiler that does not recognize these Apple-specific flags.

On Windows, if you are using the GNU compilers via MinGW, then you will need to use the `gnumex` project. First, execute the script `./get.Gnumex` from the `$IPOPTDIR/Ipopt/contrib/MatlabInterface` directory. Then, after configuring IPOPT, go to `$IPOPTDIR/build/contrib/MatlabInterface/src` and execute `make gnumex`. This will start an instance of MATLAB and open the `gnumex` tool. Check that the options are filled out appropriately for your MinGW installation, click "Make options file," then close this new instance of MATLAB after `gnumex` has created `mexopts.bat`.

Calling `make mexopts` will automatically make the necessary changes to this new `mexopts.bat` file. If you would like to do so manually, the changes are as follows.

- Change `COMPILER=gcc` to `COMPILER=g++`
- Change `GM_MEXLANG=c` to `GM_MEXLANG=cxx`
- Add the contents of the `LIBS=` line from the MATLAB interface Makefile to `GM_ADD_LIBS`
- If you want to statically link the standard libraries into the `mex` file, add `-static` to `GM_ADD_LIBS`

2.8.2 Adjusting configuration and build of Ipopt

The configure script of IPOPT attempts to automatically locate the directory where MATLAB is installed by querying the location of the `matlab` executable. You can also manually specify the MATLAB home directory when calling the configure script with the flag `--with-matlab-home`. You can determine this home directory by the command `matlabroot` within MATLAB.

In practice, it has been found easier to install and use the MATLAB interface by disabling compilation of the shared libraries, and use only static libraries instead. However, these static libraries need to be built in a way that allow using them in a shared library, i.e., they need to build with position-independent code. This is achieved with the configure script flags

```
--disable-shared --with-pic
```

On Mac OS X, it has been reported that additionally the following flags for configure should be used:

```
ADD_CFLAGS="-fno-common -fexceptions -no-cpp-precomp"
ADD_CXXFLAGS="-fno-common -fexceptions -no-cpp-precomp"
ADD_FFLAGS="-fexceptions -fbackslash"
```

With IPOPT 3.11, a *site script for configure* has been added to the MATLAB interface. This script takes care of setting configure options in a way that is appropriate for building an IPOPT `mex` file that is useable via MATLAB. Therefore, instead of setting configure options as described in the previous section, it should be sufficient to create a directory `$IPOPTDIR/build/share`, copy the site file `$IPOPTDIR/contrib/MatlabInterface/MatlabInterface.site` to that directory, and rename it to `config.site` before running configure. Alternatively, you can set an environment variable `CONFIG_SITE` that points to the site file.

This site script sets the configure flags (if not specified by the user) `--disable-shared --with-pic --with-blas=BUILD --with-lapack=BUILD`. The first two flags are discussed above. We also specify that the reference versions of BLAS and LAPACK should be used by default because of a commonly observed issue on 64-bit Linux systems. If IPOPT configure finds BLAS and/or LAPACK libraries already installed then it will use them. However, MATLAB includes its own versions of BLAS and LAPACK, which on 64-bit systems are incompatible with the expected interface used by IPOPT and the BLAS and LAPACK packages available in many Linux distributions. If the IPOPT `mex` file is compiled in such a way that the BLAS and LAPACK libraries are dynamically linked as shared libraries (as found in installed Linux packages), those library dependencies will be overridden by MATLAB's incompatible versions. This can be avoided by statically linking BLAS and LAPACK into the IPOPT `mex` file, which the above combination of configure flags will do. Note, that this issue does not appear to affect Mac OS X versions of MATLAB, so if you would like to use the Apple optimized BLAS and LAPACK libraries you can override these settings and specify `--with-blas='-framework vecLib' --with-lapack='-framework vecLib'`.

The site script also tests whether the compilers on your system are capable of statically linking the standard C++ and Fortran libraries into a shared library. This is possible with GCC versions 4.5.0 or newer on Mac OS X or Windows, 4.7.3 or newer (when GCC itself is built `--with-pic`) on Linux. If this is the case, then the site script will set appropriate configure flags and options in the MATLAB interface `Makefile` to statically link all standard libraries into the IPOPT `mex` file. This should allow a single `mex` file to work with a variety of versions of MATLAB, and on computers that do not have the same compiler versions installed. If this static linking of standard libraries causes any issues, you can disable it with the configure flag `--disable-matlab-static`.

2.8.3 Building the MATLAB interface

After configuring, building, and installing IPOPT itself, it is time to build the MATLAB interface. For that, IPOPT's configure has setup a directory `$IPOPTDIR/build/contrib/MatlabInterface/src` which contains a `Makefile`. You may need to edit this file a little bit to suit for your system setup.

You will find that most of the variables such as `CXX` and `CXXFLAGS` have been automatically (and hopefully, correctly) set according to the flags specified during your initial call to configure script. However, you may need to modify `MATLAB_HOME`, `MEXSUFFIX` and `MEX` as explained in the comments of the Makefile. For example, on Mac OS X, it has been reported that all duplicates of strings like `-L/usr/lib/gcc/i686-apple-darwin11/4.2.1/../../..` should be removed from the `LIBS` line.

Once you think you've set up the `Makefile` properly, type `make install` in the same directory as the `Makefile`. If you didn't get any errors, then you should have ended up with a `mex` file. The `mex` file will be called `ipopt.$MEXEXT`, where `$MEXEXT` is `mexglx` for 32-bit Linux, `mexa64` for 64-bit Linux, `mexw32` for 32-bit Windows, etc.

2.8.4 Making MATLAB aware of the `mex` file

In order to use the `mex` file in MATLAB, you need to tell MATLAB where to find it. The best way to do this is to type

```
addpath sourcedir
```

in the MATLAB command prompt, where `sourcedir` is the location of the `mex` file you created. (For more information, type `help addpath` in MATLAB. You can also achieve the same thing by modifying the `MATLABPATH` environment variable in the UNIX command line, using either the `export` command (in Bash shell) or the `setenv` command (in C-shell).

2.8.5 Additional notes

Starting with version 7.3, MATLAB can handle 64-bit addressing, and the authors of MATLAB have modified the implementation of sparse matrices to reflect this change. However, the row and column indices in the sparse matrix are converted to signed integers, and this could potentially cause problems when dealing with large, sparse matrices on 64-bit platforms with MATLAB version 7.3 or greater.

As MATLAB (version R2008a or newer) includes its own HSL MA57 library, IPOPT's configure can be setup to enable using this library in IPOPT's MA57 solver interface. To enable this, one should specify the configure option `--enable-matlab-ma57`. Note, that using this option is not advisable if one also provides the source for MA57 via ThirdParty/HSL.

2.8.6 Troubleshooting

The installation procedure described above does involve a certain amount of expertise on the part of the user. If you are encountering problems, it is highly recommended that you follow the standard installation of IPOPT first, and then test the installation by running some examples, either in C++ or in AMPL.

What follows are a list of common errors encountered, along with a suggested remedy.

Problem: compilation is successful, but MATLAB crashes

Remedy: Even if you didn't get any errors during compilation, there's still a possibility that you didn't link the `mex` file properly. In this case, executing IPOPT in MATLAB will cause MATLAB to crash. This is a common problem, and usually arises because you did not choose the proper compiler or set the proper compilation flags (e.g. `--with-pic`) when you ran the configure script at the very beginning.

Problem: MATLAB fails to link to IPOPT shared library

Remedy: You might encounter this problem if you try to execute one of the examples in MATLAB, and MATLAB complains that it cannot find the IPOPT shared library. The installation script has been set up so that the `mex` file you are calling knows where to look for the IPOPT shared library. However, if you moved the library then you will run into a problem. One way to fix this problem is to modify the `LDFLAGS` variable in the MATLAB interface `Makefile` (see above) so that the correct path of the IPOPT library is specified. Alternatively, you could modify the `LD_LIBRARY_PATH` environment variable so that

the location of the IPOPT library is included in the path. If none of this is familiar to you, you might want to do a web search to find out more.

Problem: `mwIndex` is not defined

Remedy: You may get a compilation error that says something to the effect that `mwIndex` is not defined. This error will surface on a version of MATLAB prior to 7.3. The solution is to add the flag `-DMWINDEXISINT` to the `CXXFLAGS` variable in the MATLAB interface `Makefile` (see above).

2.9 Expert Installation Options for Ipopt

The configuration script and Makefiles in the IPOPT distribution have been created using GNU's `autoconf` and `automake`. They attempt to automatically adapt the compiler settings etc. to the system they are running on. We tested the provided scripts for a number of different machines, operating systems and compilers, but you might run into a situation where the default setting does not work, or where you need to change the settings to fit your particular environment.

In general, you can see the list of options and variables that can be set for the `configure` script by typing `configure --help`. Also, the generic COIN-OR help pages are a valuable resource of information:

`https://projects.coin-or.org/CoinHelp`

Below a few particular options are discussed:

- The `configure` script tries to determine automatically, if you have BLAS and/or LAPACK already installed on your system (trying a few default libraries), and if it does not find them, it makes sure that you put the source code in the required place.

  However, you can specify a BLAS library (such as your local ATLAS library[15]) explicitly, using the `--with-blas` flag for `configure`. For example,

  ```
  ./configure --with-blas="-L$HOME/lib -lf77blas -lcblas -latlas"
  ```

  To tell the configure script to compile and use the downloaded BLAS source files even if a BLAS library is found on your system, specify `--with-blas=BUILD`.

  Similarly, you can use the `--with-lapack` switch to specify the location of your LAPACK library, or use the keyword `BUILD` to force the IPOPT makefiles to compile LAPACK together with IPOPT.

- Similarly, if you have a precompiled library containing the HSL packages, you can specify the directory with the `CoinHslConfig.h` header file with the `--with-hsl-incdir` flag and the linker flags with the `--with-hsl-lib` flag. Analogously, use `--with-asl-incdir` and `--with-asl-lib` for building against a precompiled AMPL solver library.

- The HSL codes HSL_MA86 and HSL_MA97 can run in parallel if compiled with OpenMP support. By default, this is not enabled by IPOPT's configure so far. To enable OpenMP with GNU compilers, it has been reported that the following configure flags should be used:

  ```
  ADD_CFLAGS=-fopenmp ADD_FFLAGS=-fopenmp ADD_CXXFLAGS=-fopenmp
  ```

- If you want to compile IPOPT with the linear solver Pardiso (see Section 2.3.4), you need to specify the link flags for the library with the `--with-pardiso` flag, including required additional libraries and flags. For example, if you want to compile IPOPT with the parallel version of Pardiso (located in `$HOME/lib`) on an AIX system in 64bit mode, you should add the flag

  ```
  --with-pardiso="-qsmp=omp $HOME/lib/libpardiso_P4AIX51_64_P.so"
  ```

  If you are using the parallel version of Pardiso, you need to specify the number of processors it should run on with the environment variable `OMP_NUM_THREADS`, as described in the Pardiso manual.

[15] see `http://math-atlas.sourceforge.net`

- If you want to compile IPOPT with the linear solver WSMP (see Section 2.3.5), you need to specify the link flags for the library with the `--with-wsmp` flag, including required additional libraries and flags. For example, if you want to compile IPOPT with WSMP (located in `$HOME/lib`) on an Intel IA32 Linux system, you should add the flag

  `--with-wsmp="$HOME/lib/wsmp/wsmp-Linux/lib/IA32/libwsmp.a -lpthread"`

- If you want to compile IPOPT with a precompiled MUMPS library (see Section 2.3.3), you need to specify the directory containing the MUMPS header files with the `--with-mumps-incdir` flag, e.g.,

  `--with-mumps-incdir="$HOME/MUMPS/include"`

  and you also need to provide the link flags for MUMPS with the `--with-mumps-lib` flag.

- If you want to specify that you want to use particular compilers, you can do so by adding the variables definitions for `CXX`, `CC`, and `F77` to the `./configure` command line, to specify the C++, C, and Fortran compiler, respectively. For example,

  `./configure CXX=g++-4.2.0 CC=gcc-4.2.0 F77=gfortran-4.2.0`

  In order to set the compiler flags, you should use the variables `CXXFLAGS`, `CFLAGS`, `FFLAGS`. Note, that the IPOPT code uses "`dynamic_cast`". Therefore it is necessary that the C++ code is compiled including RTTI (Run-Time Type Information). Some compilers need to be given special flags to do that (e.g., "`-qrtti=dyna`" for the AIX `xlC` compiler).

  Please also check the generic COIN-OR help page at
  `https://projects.coin-or.org/CoinHelp/wiki/user-configure#GivingOptions`
  for the description of more variables that can be set for `configure`.

- By default, the IPOPT library is compiled as a shared library, on systems where this is supported. If you want to generate a static library, you need to specify the `--disable-shared` flag. If you want to compile both shared and static libraries, you should specify the `--enable-static` flag.

- If you want to link the IPOPT library with a main program written in C or Fortran, the C and Fortran compiler doing the linking of the executable needs to be told about the C++ runtime libraries. Unfortunately, the current version of `autoconf` does not provide the automatic detection of those libraries. We have hard-coded some default values for some systems and compilers, but this might not work all the time.

  If you have problems linking your Fortran or C code with the IPOPT library `libipopt.a` and the linker complains about missing symbols from C++ (e.g., the standard template library), you should specify the C++ libraries with the `CXXLIBS` variable. To find out what those libraries are, it is probably helpful to link a simple C++ program with verbose compiler output.

  For example, for the Intel compilers on a Linux system, you might need to specify something like

```
./configure CC=icc F77=ifort CXX=icpc \
            CXXLIBS='-L/usr/lib/gcc-lib/i386-redhat-linux/3.2.3 -lstdc++'
```

- Compilation in 64bit mode sometimes requires some special consideration. For example, for compilation of 64bit code on AIX, we recommend the following configuration

```
./configure AR='ar -X64' NM='nm -X64' \
            CC='xlc -q64' F77='xlf -q64' CXX='xlC -q64'\
            CFLAGS='-O3 -bmaxdata:0x3f0000000' \
            FFLAGS='-O3 -bmaxdata:0x3f0000000' \
            CXXFLAGS='-qrtti=dyna -O3 -bmaxdata:0x3f0000000'
```

  (Alternatively, a simpler solution for AIX is to set the environment variable `OBJECT_MODE` to 64.)

- It is possible to compile the IPOPT library in a debug configuration, by specifying `--enable-debug`. Then the compilers will use the debug flags (unless the compilation flag variables are overwritten in the `configure` command line)

  Also, you can tell IPOPT to do some additional runtime sanity checks, by specifying the flag `--with-ipopt-checklevel=1`.

  This usually leads to a significant slowdown of the code, but might be helpful when debugging something.

3 Interfacing your NLP to Ipopt

IPOPT has been designed to be flexible for a wide variety of applications, and there are a number of ways to interface with IPOPT that allow specific data structures and linear solver techniques. Nevertheless, the authors have included a standard representation that should meet the needs of most users.

This tutorial will discuss four interfaces to IPOPT, namely the AMPL modeling language[3] interface, and the C++, C, and Fortran code interfaces. AMPL is a 3rd party modeling language tool that allows users to write their optimization problem in a syntax that resembles the way the problem would be written mathematically. Once the problem has been formulated in AMPL, the problem can be easily solved using the (already compiled) IPOPT AMPL solver executable, `ipopt`. Interfacing your problem by directly linking code requires more effort to write, but can be far more efficient for large problems.

We will illustrate how to use each of the four interfaces using an example problem, number 71 from the Hock-Schittkowsky test suite [5],

$$\min_{x\in\Re^4} \quad x_1x_4(x_1+x_2+x_3)+x_3 \tag{4}$$

$$\text{s.t.} \quad x_1x_2x_3x_4 \geq 25 \tag{5}$$

$$x_1^2+x_2^2+x_3^2+x_4^2 = 40 \tag{6}$$

$$1 \leq x_1, x_2, x_3, x_4 \leq 5, \tag{7}$$

with the starting point

$$x_0 = (1,5,5,1) \tag{8}$$

and the optimal solution

$$x_* = (1.00000000, 4.74299963, 3.82114998, 1.37940829).$$

You can find further, less documented examples for using IPOPT from your own source code in the `Ipopt/examples` subdirectory.

3.1 Using Ipopt through AMPL

Using the AMPL solver executable is by far the easiest way to solve a problem with IPOPT. The user must simply formulate the problem in AMPL syntax, and solve the problem through the AMPL environment. There are drawbacks, however. AMPL is a 3rd party package and, as such, must be appropriately licensed (a free student version for limited problem size is available from the AMPL website, `http://www.ampl.com`). Furthermore, the AMPL environment may be prohibitive for very large problems. Nevertheless, formulating the problem in AMPL is straightforward and even for large problems, it is often used as a prototyping tool before using one of the code interfaces.

This tutorial is not intended as a guide to formulating models in AMPL. If you are not already familiar with AMPL, please consult [3].

The problem presented in equations (4)–(8) can be solved with IPOPT with the following AMPL model.

```
# tell ampl to use the ipopt executable as a solver
# make sure ipopt is in the path!
option solver ipopt;

# declare the variables and their bounds,
# set notation could be used, but this is straightforward
var x1 >= 1, <= 5;
var x2 >= 1, <= 5;
var x3 >= 1, <= 5;
var x4 >= 1, <= 5;

# specify the objective function
minimize obj:
    x1 * x4 * (x1 + x2 + x3) + x3;

# specify the constraints
s.t.
  inequality:
    x1 * x2 * x3 * x4 >= 25;

  equality:
    x1^2 + x2^2 + x3^2 +x4^2 = 40;

# specify the starting point
let x1 := 1;
let x2 := 5;
let x3 := 5;
let x4 := 1;

# solve the problem
solve;

# print the solution
display x1;
display x2;
display x3;
display x4;
```

The line, "`option solver ipopt;`" tells AMPL to use IPOPT as the solver. The IPOPT executable (installed in Section 2.4) must be in the `PATH` for AMPL to find it. The remaining lines specify the problem in AMPL format. The problem can now be solved by starting AMPL and loading the mod file:

```
$ ampl
> model hs071_ampl.mod;
.
.
.
```

The problem will be solved using IPOPT and the solution will be displayed.

At this point, AMPL users may wish to skip the sections about interfacing with code, but should read Section 5 concerning IPOPT options, and Section 6 which explains the output displayed by IPOPT.

3.1.1 Using Ipopt from the command line

It is possible to solve AMPL problems with IPOPT directly from the command line. However, this requires a file in format `.nl` produced by `ampl`. If you have a model and data loaded in `Ampl`, you can create the corresponding `.nl` file with name, say, `myprob.nl` by using the `Ampl` command:

```
write gmyprob
```

There is a small `.nl` file available in the IPOPT distribution. It is located at `Ipopt/test/mytoy.nl`. We use this file in the remainder of this section. We assume that the file `mytoy.nl` is in the current directory and that the command `ipopt` is a shortcut for running the `ipopt` binary available in the `bin` directory of the installation of IPOPT.

We list below commands to perform basic tasks from the Linux prompt.

- To solve `mytoy.nl` from the Linux prompt, use:

  ```
  ipopt mytoy
  ```

- To see all command line options for IPOPT, use:

  ```
  ipopt -=
  ```

- To see more detailed information on all options for IPOPT:

  ```
  ipopt mytoy 'print_options_documentation yes'
  ```

- To run `ipopt`, setting the maximum number of iterations to 2 and print level to 4:

  ```
  ipopt mytoy 'max_iter 2 print_level 4'
  ```

If many options are to be set, they can be collected in a file `ipopt.opt` that is automatically read by IPOPT if present in the current directory, see Section 5.

3.2 Interfacing with Ipopt through code

In order to solve a problem, IPOPT needs more information than just the problem definition (for example, the derivative information). If you are using a modeling language like AMPL, the extra information is provided by the modeling tool and the IPOPT interface. When interfacing with IPOPT through your own code, however, you must provide this additional information. The following information is required by IPOPT:

1. Problem dimensions
   - number of variables
   - number of constraints
2. Problem bounds
   - variable bounds
   - constraint bounds
3. Initial starting point
   - Initial values for the primal x variables
   - Initial values for the multipliers (only required for a warm start option)
4. Problem Structure
   - number of nonzeros in the Jacobian of the constraints
   - number of nonzeros in the Hessian of the Lagrangian function
   - sparsity structure of the Jacobian of the constraints

- sparsity structure of the Hessian of the Lagrangian function

5. Evaluation of Problem Functions
   Information evaluated using a given point (x, λ, σ_f coming from IPOPT)

   - Objective function, $f(x)$
   - Gradient of the objective $\nabla f(x)$
   - Constraint function values, $g(x)$
   - Jacobian of the constraints, $\nabla g(x)^T$
   - Hessian of the Lagrangian function, $\sigma_f \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x)$
     (this is not required if a quasi-Newton options is chosen to approximate the second derivatives)

The problem dimensions and bounds are straightforward and come solely from the problem definition. The initial starting point is used by the algorithm when it begins iterating to solve the problem. If IPOPT has difficulty converging, or if it converges to a locally infeasible point, adjusting the starting point may help. Depending on the starting point, IPOPT may also converge to different local solutions.

Providing the sparsity structure of derivative matrices is a bit more involved. IPOPT is a nonlinear programming solver that is designed for solving large-scale, sparse problems. While IPOPT can be customized for a variety of matrix formats, the triplet format is used for the standard interfaces in this tutorial. For an overview of the triplet format for sparse matrices, see Appendix A. Before solving the problem, IPOPT needs to know the number of nonzero elements and the sparsity structure (row and column indices of each of the nonzero entries) of the constraint Jacobian and the Lagrangian function Hessian. Once defined, this nonzero structure MUST remain constant for the entire optimization procedure. This means that the structure needs to include entries for any element that could ever be nonzero, not only those that are nonzero at the starting point.

As IPOPT iterates, it will need the values for Item 5 in Section 3.2 evaluated at particular points. Before we can begin coding the interface, however, we need to work out the details of these equations symbolically for example problem (4)-(7).

The gradient of the objective $f(x)$ is given by

$$\begin{bmatrix} x_1 x_4 + x_4(x_1 + x_2 + x_3) \\ x_1 x_4 \\ x_1 x_4 + 1 \\ x_1(x_1 + x_2 + x_3) \end{bmatrix}$$

and the Jacobian of the constraints $g(x)$ is

$$\begin{bmatrix} x_2 x_3 x_4 & x_1 x_3 x_4 & x_1 x_2 x_4 & x_1 x_2 x_3 \\ 2x_1 & 2x_2 & 2x_3 & 2x_4 \end{bmatrix}.$$

We also need to determine the Hessian of the Lagrangian[16]. The Lagrangian function for the NLP (4)-(7) is defined as $f(x) + g(x)^T \lambda$ and the Hessian of the Lagrangian function is, technically, $\nabla^2 f(x_k) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x_k)$. However, we introduce a factor (σ_f) in front of the objective term so that IPOPT can ask for the Hessian of the objective or the constraints independently, if required. Thus, for IPOPT the symbolic form of the Hessian of the Lagrangian is

$$\sigma_f \nabla^2 f(x_k) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x_k) \tag{9}$$

[16]If a quasi-Newton option is chosen to approximate the second derivatives, this is not required. However, if second derivatives can be computed, it is often worthwhile to let IPOPT use them, since the algorithm is then usually more robust and converges faster. More on the quasi-Newton approximation in Section 4.2.

and for the example problem this becomes

$$\sigma_f \begin{bmatrix} 2x_4 & x_4 & x_4 & 2x_1+x_2+x_3 \\ x_4 & 0 & 0 & x_1 \\ x_4 & 0 & 0 & x_1 \\ 2x_1+x_2+x_3 & x_1 & x_1 & 0 \end{bmatrix} + \lambda_1 \begin{bmatrix} 0 & x_3x_4 & x_2x_4 & x_2x_3 \\ x_3x_4 & 0 & x_1x_4 & x_1x_3 \\ x_2x_4 & x_1x_4 & 0 & x_1x_2 \\ x_2x_3 & x_1x_3 & x_1x_2 & 0 \end{bmatrix} + \lambda_2 \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

where the first term comes from the Hessian of the objective function, and the second and third term from the Hessian of the constraints (5) and (6), respectively. Therefore, the dual variables λ_1 and λ_2 are the multipliers for constraints (5) and (6), respectively.

The remaining sections of the tutorial will lead you through the coding required to solve example problem (4)–(7) using, first C++, then C, and finally Fortran. Completed versions of these examples can be found in `$IPOPTDIR/Ipopt/examples` under `hs071_cpp`, `hs071_c`, `hs071_f`.

As a user, you are responsible for coding two sections of the program that solves a problem using Ipopt: the main executable (e.g., `main`) and the problem representation. Typically, you will write an executable that prepares the problem, and then passes control over to Ipopt through an `Optimize` or `Solve` call. In this call, you will give Ipopt everything that it requires to call back to your code whenever it needs functions evaluated (like the objective function, the Jacobian of the constraints, etc.). In each of the three sections that follow (C++, C, and Fortran), we will first discuss how to code the problem representation, and then how to code the executable.

3.3 The C++ Interface

This tutorial assumes that you are familiar with the C++ programming language, however, we will lead you through each step of the implementation. For the problem representation, we will create a class that inherits off of the pure virtual base class, `TNLP` (`IpTNLP.hpp`). For the executable (the `main` function) we will make the call to Ipopt through the `IpoptApplication` class (`IpIpoptApplication.hpp`). In addition, we will also be using the `SmartPtr` class (`IpSmartPtr.hpp`) which implements a reference counting pointer that takes care of memory management (object deletion) for you (for details, see Appendix B).

After "`make install`" (see Section 2.4), the header files are installed in `$IPOPTDIR/include/coin` (or in `$PREFIX/include/coin` if the switch `--prefix=$PREFIX` was used for `configure`).

3.3.1 Coding the Problem Representation

We provide the required information by coding the `HS071_NLP` class, a specific implementation of the `TNLP` base class. In the executable, we will create an instance of the `HS071_NLP` class and give this class to Ipopt so it can evaluate the problem functions through the `TNLP` interface. If you have any difficulty as the implementation proceeds, have a look at the completed example in the `Ipopt/examples/hs071_cpp` directory.

Start by creating a new directory `MyExample` under `examples` and create the files `hs071_nlp.hpp` and `hs071_nlp.cpp`. In `hs071_nlp.hpp`, include `IpTNLP.hpp` (the base class), tell the compiler that we are using the Ipopt namespace, and create the declaration of the `HS071_NLP` class, inheriting off of `TNLP`. Have a look at the `TNLP` class in `IpTNLP.hpp`; you will see eight pure virtual methods that we must implement. Declare these methods in the header file. Implement each of the methods in `HS071_NLP.cpp` using the descriptions given below. In `hs071_nlp.cpp`, first include the header file for your class and tell the compiler that you are using the Ipopt namespace. A full version of these files can be found in the `Ipopt/examples/hs071_cpp` directory.

It is very easy to make mistakes in the implementation of the function evaluation methods, in particular regarding the derivatives. Ipopt has a feature that can help you to debug the derivative code, using finite differences, see Section 4.1.

Note that the return value of any `bool`-valued function should be `true`, unless an error occurred, for example, because the value of a problem function could not be evaluated at the required point.

Method `get_nlp_info` with prototype

```
virtual bool get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                          Index& nnz_h_lag, IndexStyleEnum& index_style)
```

Give IPOPT the information about the size of the problem (and hence, the size of the arrays that it needs to allocate).

- `n`: (out), the number of variables in the problem (dimension of x).
- `m`: (out), the number of constraints in the problem (dimension of $g(x)$).
- `nnz_jac_g`: (out), the number of nonzero entries in the Jacobian.
- `nnz_h_lag`: (out), the number of nonzero entries in the Hessian.
- `index_style`: (out), the numbering style used for row/col entries in the sparse matrix format (`C_STYLE`: 0-based, `FORTRAN_STYLE`: 1-based; see also Appendix A).

IPOPT uses this information when allocating the arrays that it will later ask you to fill with values. Be careful in this method since incorrect values will cause memory bugs which may be very difficult to find.

Our example problem has 4 variables (n), and 2 constraints (m). The constraint Jacobian for this small problem is actually dense and has 8 nonzeros (we still need to represent this Jacobian using the sparse matrix triplet format). The Hessian of the Lagrangian has 10 "symmetric" nonzeros (i.e., nonzeros in the lower left triangular part.). Keep in mind that the number of nonzeros is the total number of elements that may *ever* be nonzero, not just those that are nonzero at the starting point. This information is set once for the entire problem.

```
bool HS071_NLP::get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                             Index& nnz_h_lag, IndexStyleEnum& index_style)
{
  // The problem described in HS071_NLP.hpp has 4 variables, x[0] through x[3]
  n = 4;

  // one equality constraint and one inequality constraint
  m = 2;

  // in this example the Jacobian is dense and contains 8 nonzeros
  nnz_jac_g = 8;

  // the Hessian is also dense and has 16 total nonzeros, but we
  // only need the lower left corner (since it is symmetric)
  nnz_h_lag = 10;

  // use the C style indexing (0-based)
  index_style = TNLP::C_STYLE;

  return true;
}
```

Method `get_bounds_info` with prototype

```
virtual bool get_bounds_info(Index n, Number* x_l, Number* x_u,
                             Index m, Number* g_l, Number* g_u)
```

Give IPOPT the value of the bounds on the variables and constraints.

- `n`: (in), the number of variables in the problem (dimension of x).
- `x_l`: (out) the lower bounds x^L for x.
- `x_u`: (out) the upper bounds x^U for x.

- `m`: (in), the number of constraints in the problem (dimension of $g(x)$).
- `g_l`: (out) the lower bounds g^L for $g(x)$.
- `g_u`: (out) the upper bounds g^U for $g(x)$.

The values of `n` and `m` that you specified in `get_nlp_info` are passed to you for debug checking. Setting a lower bound to a value less than or equal to the value of the option `nlp_lower_bound_inf` will cause IPOPT to assume no lower bound. Likewise, specifying the upper bound above or equal to the value of the option `nlp_upper_bound_inf` will cause IPOPT to assume no upper bound. These options, `nlp_lower_bound_inf` and `nlp_upper_bound_inf`, are set to -10^{19} and 10^{19}, respectively, by default, but may be modified by changing the options (see Section 5).

In our example, the first constraint has a lower bound of 25 and no upper bound, so we set the lower bound of constraint `[0]` to 25 and the upper bound to some number greater than 10^{19}. The second constraint is an equality constraint and we set both bounds to 40. IPOPT recognizes this as an equality constraint and does not treat it as two inequalities.

```
bool HS071_NLP::get_bounds_info(Index n, Number* x_l, Number* x_u,
                                Index m, Number* g_l, Number* g_u)
{
  // here, the n and m we gave IPOPT in get_nlp_info are passed back to us.
  // If desired, we could assert to make sure they are what we think they are.
  assert(n == 4);
  assert(m == 2);

  // the variables have lower bounds of 1
  for (Index i=0; i<4; i++)
    x_l[i] = 1.0;

  // the variables have upper bounds of 5
  for (Index i=0; i<4; i++)
    x_u[i] = 5.0;

  // the first constraint g1 has a lower bound of 25
  g_l[0] = 25;
  // the first constraint g1 has NO upper bound, here we set it to 2e19.
  // Ipopt interprets any number greater than nlp_upper_bound_inf as
  // infinity. The default value of nlp_upper_bound_inf and nlp_lower_bound_inf
  // is 1e19 and can be changed through ipopt options.
  g_u[0] = 2e19;

  // the second constraint g2 is an equality constraint, so we set the
  // upper and lower bound to the same value
  g_l[1] = g_u[1] = 40.0;

  return true;
}
```

Method `get_starting_point` with prototype

```
virtual bool get_starting_point(Index n, bool init_x, Number* x,
                                bool init_z, Number* z_L, Number* z_U,
                                Index m, bool init_lambda, Number* lambda)
```

Give IPOPT the starting point before it begins iterating.

- `n`: (in), the number of variables in the problem (dimension of x).
- `init_x`: (in), if true, this method must provide an initial value for x.
- `x`: (out), the initial values for the primal variables, x.

- `init_z`: (in), if true, this method must provide an initial value for the bound multipliers z^L and z^U.
- `z_L`: (out), the initial values for the bound multipliers, z^L.
- `z_U`: (out), the initial values for the bound multipliers, z^U.
- `m`: (in), the number of constraints in the problem (dimension of $g(x)$).
- `init_lambda`: (in), if true, this method must provide an initial value for the constraint multipliers, λ.
- `lambda`: (out), the initial values for the constraint multipliers, λ.

The variables `n` and `m` are passed in for your convenience. These variables will have the same values you specified in `get_nlp_info`.

Depending on the options that have been set, IPOPT may or may not require bounds for the primal variables x, the bound multipliers z^L and z^U, and the constraint multipliers λ. The boolean flags `init_x`, `init_z`, and `init_lambda` tell you whether or not you should provide initial values for x, z^L, z^U, or λ respectively. The default options only require an initial value for the primal variables x. Note, the initial values for bound multiplier components for "infinity" bounds ($x_L^{(i)} = -\infty$ or $x_U^{(i)} = \infty$) are ignored.

In our example, we provide initial values for x as specified in the example problem. We do not provide any initial values for the dual variables, but use an assert to immediately let us know if we are ever asked for them.

```
bool HS071_NLP::get_starting_point(Index n, bool init_x, Number* x,
                                   bool init_z, Number* z_L, Number* z_U,
                                   Index m, bool init_lambda,
                                   Number* lambda)
{
  // Here, we assume we only have starting values for x, if you code
  // your own NLP, you can provide starting values for the dual variables
  // if you wish to use a warmstart option
  assert(init_x == true);
  assert(init_z == false);
  assert(init_lambda == false);

  // initialize to the given starting point
  x[0] = 1.0;
  x[1] = 5.0;
  x[2] = 5.0;
  x[3] = 1.0;

  return true;
}
```

Method `eval_f` with prototype

```
virtual bool eval_f(Index n, const Number* x,
                    bool new_x, Number& obj_value)
```

Return the value of the objective function at the point x.

- `n`: (in), the number of variables in the problem (dimension of x).
- `x`: (in), the values for the primal variables, x, at which $f(x)$ is to be evaluated.
- `new_x`: (in), false if any evaluation method was previously called with the same values in `x`, true otherwise.
- `obj_value`: (out) the value of the objective function ($f(x)$).

The boolean variable `new_x` will be false if the last call to any of the evaluation methods (`eval_*`) used the same x values. This can be helpful when users have efficient implementations that calculate multiple outputs at once. IPOPT internally caches results from the `TNLP` and generally, this flag can be ignored.

The variable `n` is passed in for your convenience. This variable will have the same value you specified in `get_nlp_info`.

For our example, we ignore the `new_x` flag and calculate the objective.

```
bool HS071_NLP::eval_f(Index n, const Number* x, bool new_x, Number& obj_value)
{
  assert(n == 4);

  obj_value = x[0] * x[3] * (x[0] + x[1] + x[2]) + x[2];

  return true;
}
```

Method `eval_grad_f` with prototype

```
virtual bool eval_grad_f(Index n, const Number* x, bool new_x,
                         Number* grad_f)
```

Return the gradient of the objective function at the point x.

- `n`: (in), the number of variables in the problem (dimension of x).
- `x`: (in), the values for the primal variables, x, at which $\nabla f(x)$ is to be evaluated.
- `new_x`: (in), false if any evaluation method was previously called with the same values in `x`, true otherwise.
- `grad_f`: (out) the array of values for the gradient of the objective function ($\nabla f(x)$).

The gradient array is in the same order as the x variables (i.e., the gradient of the objective with respect to `x[2]` should be put in `grad_f[2]`).

The boolean variable `new_x` will be false if the last call to any of the evaluation methods (`eval_*`) used the same x values. This can be helpful when users have efficient implementations that calculate multiple outputs at once. IPOPT internally caches results from the `TNLP` and generally, this flag can be ignored.

The variable `n` is passed in for your convenience. This variable will have the same value you specified in `get_nlp_info`.

In our example, we ignore the `new_x` flag and calculate the values for the gradient of the objective.

```
bool HS071_NLP::eval_grad_f(Index n, const Number* x, bool new_x, Number* grad_f)
{
  assert(n == 4);

  grad_f[0] = x[0] * x[3] + x[3] * (x[0] + x[1] + x[2]);
  grad_f[1] = x[0] * x[3];
  grad_f[2] = x[0] * x[3] + 1;
  grad_f[3] = x[0] * (x[0] + x[1] + x[2]);

  return true;
}
```

Method `eval_g` with prototype

```
virtual bool eval_g(Index n, const Number* x,
                    bool new_x, Index m, Number* g)
```

Return the value of the constraint function at the point x.

- `n`: (in), the number of variables in the problem (dimension of x).
- `x`: (in), the values for the primal variables, x, at which the constraint functions, $g(x)$, are to be evaluated.
- `new_x`: (in), false if any evaluation method was previously called with the same values in `x`, true otherwise.
- `m`: (in), the number of constraints in the problem (dimension of $g(x)$).
- `g`: (out) the array of constraint function values, $g(x)$.

The values returned in `g` should be only the $g(x)$ values, do not add or subtract the bound values g^L or g^U.

The boolean variable `new_x` will be false if the last call to any of the evaluation methods (`eval_*`) used the same x values. This can be helpful when users have efficient implementations that calculate multiple outputs at once. IPOPT internally caches results from the `TNLP` and generally, this flag can be ignored.

The variables `n` and `m` are passed in for your convenience. These variables will have the same values you specified in `get_nlp_info`.

In our example, we ignore the `new_x` flag and calculate the values of constraint functions.

```
bool HS071_NLP::eval_g(Index n, const Number* x, bool new_x, Index m, Number* g)
{
  assert(n == 4);
  assert(m == 2);

  g[0] = x[0] * x[1] * x[2] * x[3];
  g[1] = x[0]*x[0] + x[1]*x[1] + x[2]*x[2] + x[3]*x[3];

  return true;
}
```

Method `eval_jac_g` with prototype

```
virtual bool eval_jac_g(Index n, const Number* x, bool new_x,
                        Index m, Index nele_jac, Index* iRow,
                        Index *jCol, Number* values)
```

Return either the sparsity structure of the Jacobian of the constraints, or the values for the Jacobian of the constraints at the point x.

- `n`: (in), the number of variables in the problem (dimension of x).
- `x`: (in), the values for the primal variables, x, at which the constraint Jacobian, $\nabla g(x)^T$, is to be evaluated.
- `new_x`: (in), false if any evaluation method was previously called with the same values in `x`, true otherwise.
- `m`: (in), the number of constraints in the problem (dimension of $g(x)$).
- `n_ele_jac`: (in), the number of nonzero elements in the Jacobian (dimension of `iRow`, `jCol`, and `values`).
- `iRow`: (out), the row indices of entries in the Jacobian of the constraints.
- `jCol`: (out), the column indices of entries in the Jacobian of the constraints.
- `values`: (out), the values of the entries in the Jacobian of the constraints.

The Jacobian is the matrix of derivatives where the derivative of constraint $g^{(i)}$ with respect to variable $x^{(j)}$ is placed in row i and column j. See Appendix A for a discussion of the sparse matrix format used in this method.

If the `iRow` and `jCol` arguments are not `NULL`, then IPOPT wants you to fill in the sparsity structure of the Jacobian (the row and column indices only). At this time, the `x` argument and the `values` argument will be `NULL`.

If the `x` argument and the `values` argument are not `NULL`, then IPOPT wants you to fill in the values of the Jacobian as calculated from the array `x` (using the same order as you used when specifying the sparsity structure). At this time, the `iRow` and `jCol` arguments will be `NULL`;

The boolean variable `new_x` will be false if the last call to any of the evaluation methods (`eval_*`) used the same x values. This can be helpful when users have efficient implementations that calculate multiple outputs at once. IPOPT internally caches results from the `TNLP` and generally, this flag can be ignored.

The variables `n`, `m`, and `nele_jac` are passed in for your convenience. These arguments will have the same values you specified in `get_nlp_info`.

In our example, the Jacobian is actually dense, but we still specify it using the sparse format.

```
bool HS071_NLP::eval_jac_g(Index n, const Number* x, bool new_x,
                           Index m, Index nele_jac, Index* iRow, Index *jCol,
                           Number* values)
{
  if (values == NULL) {
    // return the structure of the Jacobian

    // this particular Jacobian is dense
    iRow[0] = 0; jCol[0] = 0;
    iRow[1] = 0; jCol[1] = 1;
    iRow[2] = 0; jCol[2] = 2;
    iRow[3] = 0; jCol[3] = 3;
    iRow[4] = 1; jCol[4] = 0;
    iRow[5] = 1; jCol[5] = 1;
    iRow[6] = 1; jCol[6] = 2;
    iRow[7] = 1; jCol[7] = 3;
  }
  else {
    // return the values of the Jacobian of the constraints

    values[0] = x[1]*x[2]*x[3]; // 0,0
    values[1] = x[0]*x[2]*x[3]; // 0,1
    values[2] = x[0]*x[1]*x[3]; // 0,2
    values[3] = x[0]*x[1]*x[2]; // 0,3

    values[4] = 2*x[0]; // 1,0
    values[5] = 2*x[1]; // 1,1
    values[6] = 2*x[2]; // 1,2
    values[7] = 2*x[3]; // 1,3
  }

  return true;
}
```

Method `eval_h` with prototype

```
virtual bool eval_h(Index n, const Number* x, bool new_x,
                    Number obj_factor, Index m, const Number* lambda,
                    bool new_lambda, Index nele_hess, Index* iRow,
                    Index* jCol, Number* values)
```

Return either the sparsity structure of the Hessian of the Lagrangian, or the values of the Hessian of the Lagrangian (9) for the given values for x, σ_f, and λ.

- `n`: (in), the number of variables in the problem (dimension of x).
- `x`: (in), the values for the primal variables, x, at which the Hessian is to be evaluated.
- `new_x`: (in), false if any evaluation method was previously called with the same values in `x`, true otherwise.
- `obj_factor`: (in), factor in front of the objective term in the Hessian, σ_f.
- `m`: (in), the number of constraints in the problem (dimension of $g(x)$).
- `lambda`: (in), the values for the constraint multipliers, λ, at which the Hessian is to be evaluated.
- `new_lambda`: (in), false if any evaluation method was previously called with the same values in `lambda`, true otherwise.
- `nele_hess`: (in), the number of nonzero elements in the Hessian (dimension of `iRow`, `jCol`, and `values`).
- `iRow`: (out), the row indices of entries in the Hessian.
- `jCol`: (out), the column indices of entries in the Hessian.
- `values`: (out), the values of the entries in the Hessian.

The Hessian matrix that IPOPT uses is defined in (9). See Appendix A for a discussion of the sparse symmetric matrix format used in this method.

If the `iRow` and `jCol` arguments are not `NULL`, then IPOPT wants you to fill in the sparsity structure of the Hessian (the row and column indices for the lower or upper triangular part only). In this case, the `x`, `lambda`, and `values` arrays will be `NULL`.

If the `x`, `lambda`, and `values` arrays are not `NULL`, then IPOPT wants you to fill in the values of the Hessian as calculated using `x` and `lambda` (using the same order as you used when specifying the sparsity structure). In this case, the `iRow` and `jCol` arguments will be `NULL`.

The boolean variables `new_x` and `new_lambda` will both be false if the last call to any of the evaluation methods (`eval_*`) used the same values. This can be helpful when users have efficient implementations that calculate multiple outputs at once. IPOPT internally caches results from the `TNLP` and generally, this flag can be ignored.

The variables `n`, `m`, and `nele_hess` are passed in for your convenience. These arguments will have the same values you specified in `get_nlp_info`.

In our example, the Hessian is dense, but we still specify it using the sparse matrix format. Because the Hessian is symmetric, we only need to specify the lower left corner.

```
bool HS071_NLP::eval_h(Index n, const Number* x, bool new_x,
                       Number obj_factor, Index m, const Number* lambda,
                       bool new_lambda, Index nele_hess, Index* iRow,
                       Index* jCol, Number* values)
{
  if (values == NULL) {
    // return the structure. This is a symmetric matrix, fill the lower left
    // triangle only.

    // the Hessian for this problem is actually dense
    Index idx=0;
    for (Index row = 0; row < 4; row++) {
      for (Index col = 0; col <= row; col++) {
        iRow[idx] = row;
        jCol[idx] = col;
        idx++;
      }
    }
```

```
    assert(idx == nele_hess);
  }
  else {
    // return the values. This is a symmetric matrix, fill the lower left
    // triangle only

    // fill the objective portion
    values[0] = obj_factor * (2*x[3]); // 0,0

    values[1] = obj_factor * (x[3]);   // 1,0
    values[2] = 0;                     // 1,1

    values[3] = obj_factor * (x[3]);   // 2,0
    values[4] = 0;                     // 2,1
    values[5] = 0;                     // 2,2

    values[6] = obj_factor * (2*x[0] + x[1] + x[2]); // 3,0
    values[7] = obj_factor * (x[0]);                 // 3,1
    values[8] = obj_factor * (x[0]);                 // 3,2
    values[9] = 0;                                   // 3,3


    // add the portion for the first constraint
    values[1] += lambda[0] * (x[2] * x[3]); // 1,0

    values[3] += lambda[0] * (x[1] * x[3]); // 2,0
    values[4] += lambda[0] * (x[0] * x[3]); // 2,1

    values[6] += lambda[0] * (x[1] * x[2]); // 3,0
    values[7] += lambda[0] * (x[0] * x[2]); // 3,1
    values[8] += lambda[0] * (x[0] * x[1]); // 3,2

    // add the portion for the second constraint
    values[0] += lambda[1] * 2; // 0,0

    values[2] += lambda[1] * 2; // 1,1

    values[5] += lambda[1] * 2; // 2,2

    values[9] += lambda[1] * 2; // 3,3
  }

  return true;
}
```

Method `finalize_solution` with prototype

```
virtual void finalize_solution(SolverReturn status, Index n,
                               const Number* x, const Number* z_L,
                               const Number* z_U, Index m, const Number* g,
                               const Number* lambda, Number obj_value,
                               const IpoptData* ip_data,
                               IpoptCalculatedQuantities* ip_cq)
```

This is the only method that is not mentioned in Section 3.2. This method is called by Ipopt after the algorithm has finished (successfully or even with most errors).

- `status`: (in), gives the status of the algorithm as specified in `IpAlgTypes.hpp`,
  - `SUCCESS`: Algorithm terminated successfully at a locally optimal point, satisfying the convergence tolerances (can be specified by options).

- MAXITER_EXCEEDED: Maximum number of iterations exceeded (can be specified by an option).
- CPUTIME_EXCEEDED: Maximum number of CPU seconds exceeded (can be specified by an option).
- STOP_AT_TINY_STEP: Algorithm proceeds with very little progress.
- STOP_AT_ACCEPTABLE_POINT: Algorithm stopped at a point that was converged, not to "desired" tolerances, but to "acceptable" tolerances (see the `acceptable-...` options).
- LOCAL_INFEASIBILITY: Algorithm converged to a point of local infeasibility. Problem may be infeasible.
- USER_REQUESTED_STOP: The user call-back function `intermediate_callback` (see Section 3.3.4) returned `false`, i.e., the user code requested a premature termination of the optimization.
- DIVERGING_ITERATES: It seems that the iterates diverge.
- RESTORATION_FAILURE: Restoration phase failed, algorithm doesn't know how to proceed.
- ERROR_IN_STEP_COMPUTATION: An unrecoverable error occurred while Ipopt tried to compute the search direction.
- INVALID_NUMBER_DETECTED: Algorithm received an invalid number (such as `NaN` or `Inf`) from the NLP; see also option `check_derivatives_for_naninf`.
- INTERNAL_ERROR: An unknown internal error occurred. Please contact the Ipopt authors through the mailing list.

- `n`: (in), the number of variables in the problem (dimension of x).
- `x`: (in), the final values for the primal variables, x_*.
- `z_L`: (in), the final values for the lower bound multipliers, z_*^L.
- `z_U`: (in), the final values for the upper bound multipliers, z_*^U.
- `m`: (in), the number of constraints in the problem (dimension of $g(x)$).
- `g`: (in), the final value of the constraint function values, $g(x_*)$.
- `lambda`: (in), the final values of the constraint multipliers, λ_*.
- `obj_value`: (in), the final value of the objective, $f(x_*)$.
- `ip_data` and `ip_cq` are provided for expert users.

This method gives you the return status of the algorithm (SolverReturn), and the values of the variables, the objective and constraint function values when the algorithm exited.

In our example, we will print the values of some of the variables to the screen.

```
void HS071_NLP::finalize_solution(SolverReturn status,
                                  Index n, const Number* x, const Number* z_L,
                                  const Number* z_U, Index m, const Number* g,
                                  const Number* lambda, Number obj_value)
{
  // here is where we would store the solution to variables, or write to a file, etc
  // so we could use the solution.

  // For this example, we write the solution to the console
  printf("\n\nSolution of the primal variables, x\n");
  for (Index i=0; i<n; i++) {
    printf("x[%d] = %e\n", i, x[i]);
  }

  printf("\n\nSolution of the bound multipliers, z_L and z_U\n");
```

```
  for (Index i=0; i<n; i++) {
    printf("z_L[%d] = %e\n", i, z_L[i]);
  }
  for (Index i=0; i<n; i++) {
    printf("z_U[%d] = %e\n", i, z_U[i]);
  }

  printf("\n\nObjective value\n");
  printf("f(x*) = %e\n", obj_value);
}
```

This is all that is required for our `HS071_NLP` class and the coding of the problem representation.

3.3.2 Coding the Executable (`main`)

Now that we have a problem representation, the `HS071_NLP` class, we need to code the main function that will call IPOPT and ask IPOPT to find a solution.

Here, we must create an instance of our problem (`HS071_NLP`), create an instance of the IPOPT solver (`IpoptApplication`), initialize it, and ask the solver to find a solution. We always use the `SmartPtr` template class instead of raw C++ pointers when creating and passing IPOPT objects. To find out more information about smart pointers and the `SmartPtr` implementation used in IPOPT, see Appendix B.

Create the file `MyExample.cpp` in the MyExample directory. Include the header files `HS071_NLP.hpp` and `IpIpoptApplication.hpp`, tell the compiler to use the `Ipopt` namespace, and implement the `main` function.

```
#include "IpIpoptApplication.hpp"
#include "hs071_nlp.hpp"

using namespace Ipopt;

int main(int argv, char* argc[])
{
  // Create a new instance of your nlp
  //  (use a SmartPtr, not raw)
  SmartPtr<TNLP> mynlp = new HS071_NLP();

  // Create a new instance of IpoptApplication
  //  (use a SmartPtr, not raw)
  // We are using the factory, since this allows us to compile this
  // example with an Ipopt Windows DLL
  SmartPtr<IpoptApplication> app = IpoptApplicationFactory();

  // Change some options
  // Note: The following choices are only examples, they might not be
  //       suitable for your optimization problem.
  app->Options()->SetNumericValue("tol", 1e-9);
  app->Options()->SetStringValue("mu_strategy", "adaptive");
  app->Options()->SetStringValue("output_file", "ipopt.out");

  // Intialize the IpoptApplication and process the options
  ApplicationReturnStatus status;
  status = app->Initialize();
  if (status != Solve_Succeeded) {
    printf("\n\n*** Error during initialization!\n");
    return (int) status;
  }

  // Ask Ipopt to solve the problem
  status = app->OptimizeTNLP(mynlp);

  if (status == Solve_Succeeded) {
    printf("\n\n*** The problem solved!\n");
```

```
  }
  else {
    printf("\n\n*** The problem FAILED!\n");
  }

  // As the SmartPtrs go out of scope, the reference count
  // will be decremented and the objects will automatically
  // be deleted.

  return (int) status;
}
```

The first line of code in `main` creates an instance of `HS071_NLP`. We then create an instance of the IPOPT solver, `IpoptApplication`. You could use `new` to create a new application object, but if you want to make sure that your code would also work with a Windows DLL, you need to use the factory, as done in the example above. The call to `app->Initialize(...)` will initialize that object and process the options (particularly the output related options). The call to `app->OptimizeTNLP(...)` will run IPOPT and try to solve the problem. By default, IPOPT will write its progress to the console, and return the `SolverReturn` status.

3.3.3 Compiling and Testing the Example

Our next task is to compile and test the code. If you are familiar with the compiler and linker used on your system, you can build the code, telling the linker about the necessary libraries, as listed in the `ipopt_addlibs_cpp.txt` file. If you are using the autotools based build system, then a sample makefile created by configure already exists. Copy `Ipopt/examples/hs071_cpp/Makefile` into your `MyExample` directory. This makefile was created for the `hs071_cpp` code, but it can be easily modified for your example problem. Edit the file, making the following changes,

- change the `EXE` variable
  `EXE = my_example`
- change the `OBJS` variable
  `OBJS = HS071_NLP.o MyExample.o`

and the problem should compile easily with,
`$ make`
Now run the executable,
`$ ./my_example`
and you should see output resembling the following,

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

Number of nonzeros in equality constraint Jacobian...:        4
Number of nonzeros in inequality constraint Jacobian.:        4
Number of nonzeros in Lagrangian Hessian.............:       10

Total number of variables............................:        4
                     variables with only lower bounds:        0
                variables with lower and upper bounds:        4
                     variables with only upper bounds:        0
Total number of equality constraints.................:        1
Total number of inequality constraints...............:        1
        inequality constraints with only lower bounds:        1
   inequality constraints with lower and upper bounds:        0
```

```
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  1.6109693e+01 1.12e+01 5.28e-01   0.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  1.7410406e+01 8.38e-01 2.25e+01  -0.3 7.97e-01    -  3.19e-01 1.00e+00f  1
   2  1.8001613e+01 1.06e-02 4.96e+00  -0.3 5.60e-02   2.0 9.97e-01 1.00e+00h  1
   3  1.7199482e+01 9.04e-02 4.24e-01  -1.0 9.91e-01    -  9.98e-01 1.00e+00f  1
   4  1.6940955e+01 2.09e-01 4.58e-02  -1.4 2.88e-01    -  9.66e-01 1.00e+00h  1
   5  1.7003411e+01 2.29e-02 8.42e-03  -2.9 7.03e-02    -  9.68e-01 1.00e+00h  1
   6  1.7013974e+01 2.59e-04 8.65e-05  -4.5 6.22e-03    -  1.00e+00 1.00e+00h  1
   7  1.7014017e+01 2.26e-07 5.71e-08  -8.0 1.43e-04    -  1.00e-00 1.00e+00h  1
   8  1.7014017e+01 4.62e-14 9.09e-14  -8.0 6.95e-08    -  1.00e+00 1.00e+00h  1

Number of Iterations....: 8

Number of objective function evaluations             = 9
Number of objective gradient evaluations             = 9
Number of equality constraint evaluations            = 9
Number of inequality constraint evaluations          = 9
Number of equality constraint Jacobian evaluations   = 9
Number of inequality constraint Jacobian evaluations = 9
Number of Lagrangian Hessian evaluations             = 8
Total CPU secs in IPOPT (w/o function evaluations)   =      0.220
Total CPU secs in NLP function evaluations           =      0.000

EXIT: Optimal Solution Found.


Solution of the primal variables, x
x[0] = 1.000000e+00
x[1] = 4.743000e+00
x[2] = 3.821150e+00
x[3] = 1.379408e+00


Solution of the bound multipliers, z_L and z_U
z_L[0] = 1.087871e+00
z_L[1] = 2.428776e-09
z_L[2] = 3.222413e-09
z_L[3] = 2.396076e-08
z_U[0] = 2.272727e-09
z_U[1] = 3.537314e-08
z_U[2] = 7.711676e-09
z_U[3] = 2.510890e-09


Objective value
f(x*) = 1.701402e+01


*** The problem solved!
```

This completes the basic C++ tutorial, but see Section 6 which explains the standard console output of Ipopt and Section 5 for information about the use of options to customize the behavior of Ipopt.

The `Ipopt/examples/ScalableProblems` directory contains other NLP problems coded in C++.

3.3.4 Additional methods in `TNLP`

The following methods are available to additional features that are not explained in the example. Default implementations for those methods are provided, so that a user can safely ignore them, unless she wants to make use of those features. From these features, only the intermediate callback is already available in the C and Fortran interfaces.

Method `intermediate_callback` with prototype

```
virtual bool intermediate_callback(AlgorithmMode mode,
                                   Index iter, Number obj_value,
                                   Number inf_pr, Number inf_du,
                                   Number mu, Number d_norm,
                                   Number regularization_size,
                                   Number alpha_du, Number alpha_pr,
                                   Index ls_trials,
                                   const IpoptData* ip_data,
                                   IpoptCalculatedQuantities* ip_cq)
```

It is not required to implement (overload) this method. This method is called once per iteration (during the convergence check), and can be used to obtain information about the optimization status while IPOPT solves the problem, and also to request a premature termination.

The information provided by the entities in the argument list correspond to what IPOPT prints in the iteration summary (see also Section 6). Further information can be obtained from the `ip_data` and `ip_cq` objects (in the C++ interface and for experts only :)).

You you let this method return `false`, IPOPT will terminate with the `User_Requested_Stop` status. If you do not implement this method (as we do in this example), the default implementation always returns `true`.

A frequently asked question is how to access the values of the primal and dual variables in this callback. The values are stored in the `ip_cq` object for the *internal representation* of the problem. To access the values in a form that corresponds to those used in the evaluation routines, the user has to request IPOPT's `TNLPAdapter` object to "resort" the data vectors and to fill in information about possibly filtered out fixed variables. The `TNLPAdapter` can be accessed as follows. First, add the following includes to your `TNLP` implementation:

```
#include "IpIpoptCalculatedQuantities.hpp"
#include "IpIpoptData.hpp"
#include "IpTNLPAdapter.hpp"
#include "IpOrigIpoptNLP.hpp"
```

Next, add the following code to your implementation of the `intermediate_callback`:

```
Ipopt::TNLPAdapter* tnlp_adapter = NULL;
if( ip_cq != NULL )
{
  Ipopt::OrigIpoptNLP* orignlp;
  orignlp = dynamic_cast<OrigIpoptNLP*>(GetRawPtr(ip_cq->GetIpoptNLP()));
  if( orignlp != NULL )
    tnlp_adapter = dynamic_cast<TNLPAdapter*>(GetRawPtr(orignlp->nlp()));
}
```

Note, that retrieving the `TNLPAdapter` will fail (i.e., `orignlp` will be `NULL`) if IPOPT is currently in restoration mode. If, however, `tnlp_adapter` is not `NULL`, then it can be used to obtain primal variable values x and the dual values for the constraints 2 and the variable bounds 3 as follows.

```
double* primals = new double[n];
double* dualeqs = new double[m];
double* duallbs = new double[n];
double* dualubs = new double[n];
tnlp_adapter->ResortX(*ip_data->curr()->x(), primals);
tnlp_adapter->ResortG(*ip_data->curr()->y_c(), *ip_data->curr()->y_d(), dualeqs);
tnlp_adapter->ResortBnds(*ip_data->curr()->z_L(), duallbs,
                         *ip_data->curr()->z_U(), dualubs);
```

Additionally, information about scaled violation of constraint 2 and violation of complementarity constraints can be obtained via

```
tnlp_adapter->ResortG(*ip_data->curr_c(), *ip_data->curr_d_minus_s(), ...)
tnlp_adapter->ResortBnds(*ip_data->curr_compl_x_L(), ...,
                         *ip_data->curr_compl_x_U(), ...)
tnlp_adapter->ResortG(*ip_data->curr_compl_s_L(), ...)
tnlp_adapter->ResortG(*ip_data->curr_compl_s_U(), ...)
```

Method `get_scaling_parameters` with prototype

```
virtual bool get_scaling_parameters(Number& obj_scaling,
                                    bool& use_x_scaling, Index n,
                                    Number* x_scaling,
                                    bool& use_g_scaling, Index m,
                                    Number* g_scaling)
```

This method is called if the `nlp_scaling_method` is chosen as `user-scaling`. The user has to provide scaling factors for the objective function as well as for the optimization variables and/or constraints. The return value should be true, unless an error occurred, and the program is to be aborted.

The value returned in `obj_scaling` determines, how IPOPT should internally scale the objective function. For example, if this number is chosen to be 10, then IPOPT solves internally an optimization problem that has 10 times the value of the original objective function provided by the `TNLP`. In particular, if this value is negative, then IPOPT will maximize the objective function instead of minimizing it.

The scaling factors for the variables can be returned in `x_scaling`, which has the same length as `x` in the other `TNLP` methods, and the factors are ordered like `x`. You need to set `use_x_scaling` to `true`, if you want IPOPT so scale the variables. If it is `false`, no internal scaling of the variables is done. Similarly, the scaling factors for the constraints can be returned in `g_scaling`, and this scaling is activated by setting `use_g_scaling` to `true`.

As a guideline, we suggest to scale the optimization problem (either directly in the original formulation, or after using scaling factors) so that all sensitivities, i.e., all non-zero first partial derivatives, are typically of the order $0.1 - 10$.

Method `get_number_of_nonlinear_variables` with prototype

```
virtual Index get_number_of_nonlinear_variables()
```

This method is only important if the limited-memory quasi-Newton options is used, see Section 4.2. It is used to return the number of variables that appear nonlinearly in the objective function or in at least one constraint function. If a negative number is returned, IPOPT assumes that all variables are nonlinear.

If the user doesn't overload this method in her implementation of the class derived from `TNLP`, the default implementation returns -1, i.e., all variables are assumed to be nonlinear.

Method `get_list_of_nonlinear_variables` with prototype

```
virtual bool get_list_of_nonlinear_variables(Index num_nonlin_vars,
                                             Index* pos_nonlin_vars)
```

This method is called by IPOPT only if the limited-memory quasi-Newton options is used and if the `get_number_of_nonlinear_variables` method returns a positive number; this number is then identical with `num_nonlin_vars` and the length of the array `pos_nonlin_vars`. In this call, you need to list the indices of all nonlinear variables in `pos_nonlin_vars`, where the numbering starts with 0 order 1, depending on the numbering style determined in `get_nlp_info`.

Method `get_variables_linearity` with prototype

```
virtual bool get_variables_linearity(Index n,
                                     LinearityType* var_types)
```

This method is never called by IPOPT, but is used by BONMIN to get information about which variables occur only in linear terms. IPOPT passes a `var_types` array of size `n`, which the user should fill with the appropriate linearity type of the variables (`TNLP::LINEAR` or `TNLP::NON_LINEAR`).

If the user doesn't overload this method in her implementation of the class derived from `TNLP`, the default implementation returns `false`.

Method `get_constraints_linearity` with prototype

```
virtual bool get_constraints_linearity(Index m,
                                       LinearityType* const_types)
```

This method is never called by IPOPT, but is used by BONMIN to get information about which constraints are linear. IPOPT passes a `const_types` array of size `m`, which the user should fill with the appropriate linearity type of the constraints (`TNLP::LINEAR` or `TNLP::NON_LINEAR`).

If the user doesn't overload this method in her implementation of the class derived from `TNLP`, the default implementation returns `false`.

Method `get_var_con_metadata` with prototype

```
virtual bool get_var_con_metadata(Index n,
                                  StringMetaDataMapType& var_string_md,
                                  IntegerMetaDataMapType& var_integer_md,
                                  NumericMetaDataMapType& var_numeric_md,
                                  Index m,
                                  StringMetaDataMapType& con_string_md,
                                  IntegerMetaDataMapType& con_integer_md,
                                  NumericMetaDataMapType& con_numeric_md)
```

This method is used to pass meta data about variables or constraints to IPOPT. The data can be either of integer, numeric, or string type. IPOPT passes this data on to its internal problem representation. The meta data type is a `std::map` with `std::string` as key type and a `std::vector` as value type. So far, IPOPT itself makes only use of string meta data under the key `idx_names`. With this key, variable and constraint names can be passed to IPOPT, which are shown when printing internal vector or matrix data structures if IPOPT is run with a high value for the `print_level` option. This allows a user to identify the original variables and constraints corresponding to IPOPT's internal problem representation.

If the user doesn't overload this method in her implementation of the class derived from `TNLP`, the default implementation does not set any meta data and returns `false`.

Method `finalize_metadata` with prototype

```
virtual void finalize_metadata(Index n,
                               const StringMetaDataMapType& var_string_md,
                               const IntegerMetaDataMapType& var_integer_md,
                               const NumericMetaDataMapType& var_numeric_md,
                               Index m,
                               const StringMetaDataMapType& con_string_md,
                               const IntegerMetaDataMapType& con_integer_md,
                               const NumericMetaDataMapType& con_numeric_md)
```

This method is called just before `finalize_solution` and is used to return any meta data collected during the algorithms run, including the meta data provided by the user with the `get_var_con_metadata` method.

If the user doesn't overload this method in her implementation of the class derived from `TNLP`, the default implementation does nothing.

Method `get_warm_start_iterate` with prototype

```
virtual bool get_warm_start_iterate(IteratesVector& warm_start_iterate)
```

Overload this method to provide an IPOPT iterate which is already in the form IPOPT requires it internally for a warm starts. This method is only for expert users.

If the user doesn't overload this method in her implementation of the class derived from `TNLP`, the default implementation does not provide a warm start iterate and returns `false`.

3.4 The C Interface

The C interface for IPOPT is declared in the header file `IpStdCInterface.h`, which is found in `$IPOPTDIR/include/coin` (or in `$PREFIX/include/coin` if the switch `--prefix=$PREFIX` was used for `configure`); while reading this section, it will be helpful to have a look at this file.

In order to solve an optimization problem with the C interface, one has to create an `IpoptProblem`[17] with the function `CreateIpoptProblem`, which later has to be passed to the `IpoptSolve` function.

The `IpoptProblem` created by `CreateIpoptProblem` contains the problem dimensions, the variable and constraint bounds, and the function pointers for callbacks that will be used to evaluate the NLP problem functions and their derivatives (see also the discussion of the C++ methods `get_nlp_info` and `get_bounds_info` in Section 3.3.1 for information about the arguments of `CreateIpoptProblem`).

The prototypes for the callback functions, `Eval_F_CB`, `Eval_Grad_F_CB`, etc., are defined in the header file `IpStdCInterface.h`. Their arguments correspond one-to-one to the arguments for the C++ methods discussed in Section 3.3.1; for example, for the meaning of `n`, `x`, `new_x`, `obj_value` in the declaration of `Eval_F_CB` see the discussion of "`eval_f`". The callback functions should return `TRUE`, unless there was a problem doing the requested function/derivative evaluation at the given point `x` (then it should return `FALSE`).

Note the additional argument of type `UserDataPtr` in the callback functions. This pointer argument is available for you to communicate information between the main program that calls `IpoptSolve` and any of the callback functions. This pointer is simply passed unmodified by IPOPT among those functions. For example, you can use this to pass constants that define the optimization problem and are computed before the optimization in the main C program to the callback functions.

After an `IpoptProblem` has been created, you can set algorithmic options for IPOPT (see Section 5) using the `AddIpopt...Option` functions. Finally, the IPOPT algorithm is called with `IpoptSolve`, giving IPOPT the `IpoptProblem`, the starting point, and arrays to store the solution values (primal and dual variables), if desired. Finally, after everything is done, you should call `FreeIpoptProblem` to release internal memory that is still allocated inside IPOPT.

In the remainder of this section we discuss how the example problem (4)–(7) can be solved using the C interface. A completed version of this example can be found in `Ipopt/examples/hs071_c`.

In order to implement the example problem on your own, create a new directory `MyCExample` and create a new file, `hs071_c.c`. Here, include the interface header file `IpStdCInterface.h`, along with other necessary header files, such as `stdlib.h` and `assert.h`. Add the prototypes and implementations for the five callback functions. Have a look at the C++ implementation for `eval_f`, `eval_g`, `eval_grad_f`, `eval_jac_g`, and `eval_h` in Section 3.3.1. The C implementations have somewhat different

[17] `IpoptProblem` is a pointer to a C structure; you should not access this structure directly, only through the functions provided in the C interface.

prototypes, but are implemented almost identically to the C++ code. See the completed example in `Ipopt/examples/hs071_c/hs071_c.c` if you are not sure how to do this.

We now need to implement the `main` function, create the `IpoptProblem`, set options, and call `IpoptSolve`. The `CreateIpoptProblem` function requires the problem dimensions, the variable and constraint bounds, and the function pointers to the callback routines. The `IpoptSolve` function requires the `IpoptProblem`, the starting point, and allocated arrays for the solution. The `main` function from the example is shown next and discussed below.

```
int main()
{
  Index n=-1;                           /* number of variables */
  Index m=-1;                           /* number of constraints */
  Number* x_L = NULL;                   /* lower bounds on x */
  Number* x_U = NULL;                   /* upper bounds on x */
  Number* g_L = NULL;                   /* lower bounds on g */
  Number* g_U = NULL;                   /* upper bounds on g */
  IpoptProblem nlp = NULL;              /* IpoptProblem */
  enum ApplicationReturnStatus status; /* Solve return code */
  Number* x = NULL;                     /* starting point and solution vector */
  Number* mult_x_L = NULL;              /* lower bound multipliers at the solution */
  Number* mult_x_U = NULL;              /* upper bound multipliers at the solution */
  Number obj;                           /* objective value */
  Index i;                              /* generic counter */

  /* set the number of variables and allocate space for the bounds */
  n=4;
  x_L = (Number*)malloc(sizeof(Number)*n);
  x_U = (Number*)malloc(sizeof(Number)*n);
  /* set the values for the variable bounds */
  for (i=0; i<n; i++) {
    x_L[i] = 1.0;
    x_U[i] = 5.0;
  }

  /* set the number of constraints and allocate space for the bounds */
  m=2;
  g_L = (Number*)malloc(sizeof(Number)*m);
  g_U = (Number*)malloc(sizeof(Number)*m);
  /* set the values of the constraint bounds */
  g_L[0] = 25; g_U[0] = 2e19;
  g_L[1] = 40; g_U[1] = 40;

  /* create the IpoptProblem */
  nlp = CreateIpoptProblem(n, x_L, x_U, m, g_L, g_U, 8, 10, 0,
   &eval_f, &eval_g, &eval_grad_f,
   &eval_jac_g, &eval_h);

  /* We can free the memory now - the values for the bounds have been
     copied internally in CreateIpoptProblem */
  free(x_L);
  free(x_U);
  free(g_L);
```

```
  free(g_U);

  /* set some options */
  AddIpoptNumOption(nlp, "tol", 1e-9);
  AddIpoptStrOption(nlp, "mu_strategy", "adaptive");

  /* allocate space for the initial point and set the values */
  x = (Number*)malloc(sizeof(Number)*n);
  x[0] = 1.0;
  x[1] = 5.0;
  x[2] = 5.0;
  x[3] = 1.0;

  /* allocate space to store the bound multipliers at the solution */
  mult_x_L = (Number*)malloc(sizeof(Number)*n);
  mult_x_U = (Number*)malloc(sizeof(Number)*n);

  /* solve the problem */
  status = IpoptSolve(nlp, x, NULL, &obj, NULL, mult_x_L, mult_x_U, NULL);

  if (status == Solve_Succeeded) {
    printf("\n\nSolution of the primal variables, x\n");
    for (i=0; i<n; i++)
      printf("x[%d] = %e\n", i, x[i]);

    printf("\n\nSolution of the bound multipliers, z_L and z_U\n");
    for (i=0; i<n; i++)
      printf("z_L[%d] = %e\n", i, mult_x_L[i]);
    for (i=0; i<n; i++)
      printf("z_U[%d] = %e\n", i, mult_x_U[i]);

    printf("\n\nObjective value\nf(x*) = %e\n", obj);
  }

  /* free allocated memory */
  FreeIpoptProblem(nlp);
  free(x);
  free(mult_x_L);
  free(mult_x_U);

  return 0;
}
```

Here, we declare all the necessary variables and set the dimensions of the problem. The problem has 4 variables, so we set `n` and allocate space for the variable bounds (don't forget to call `free` for each of your `malloc` calls before the end of the program). We then set the values for the variable bounds.

The problem has 2 constraints, so we set `m` and allocate space for the constraint bounds. The first constraint has a lower bound of 25 and no upper bound. Here we set the upper bound to `2e19`. IPOPT interprets any number greater than or equal to `nlp_upper_bound_inf` as infinity. The default value of `nlp_lower_bound_inf` and `nlp_upper_bound_inf` is `-1e19` and `1e19`, respectively, and can be changed through IPOPT options. The second constraint is an equality with right hand side 40, so we set both the upper and the lower bound to 40.

We next create an instance of the IpoptProblem by calling CreateIpoptProblem, giving it the problem dimensions and the variable and constraint bounds. The arguments nele_jac and nele_hess are the number of elements in Jacobian and the Hessian, respectively. See Appendix A for a description of the sparse matrix format. The index_style argument specifies whether we want to use C style indexing for the row and column indices of the matrices or Fortran style indexing. Here, we set it to 0 to indicate C style. We also include the references to each of our callback functions. IPOPT uses these function pointers to ask for evaluation of the NLP when required.

After freeing the bound arrays that are no longer required, the next two lines illustrate how you can change the value of options through the interface. IPOPT options can also be changed by creating a ipopt.opt file (see Section 5). We next allocate space for the initial point and set the values as given in the problem definition.

The call to IpoptSolve can provide us with information about the solution, but most of this is optional. Here, we want values for the bound multipliers at the solution and we allocate space for these.

We can now make the call to IpoptSolve and find the solution of the problem. We pass in the IpoptProblem, the starting point x (IPOPT will use this array to return the solution or final point as well). The next 5 arguments are pointers so IPOPT can fill in values at the solution. If these pointers are set to NULL, IPOPT will ignore that entry. For example, here, we do not want the constraint function values at the solution or the constraint multipliers, so we set those entries to NULL. We do want the value of the objective, and the multipliers for the variable bounds. The last argument is a void* for user data. Any pointer you give here will also be passed to you in the callback functions.

The return code is an ApplicationReturnStatus enumeration, see the header file ReturnCodes_inc.h which is installed along IpStdCInterface.h in the IPOPT include directory.

After the optimizer terminates, we check the status and print the solution if successful. Finally, we free the IpoptProblem and the remaining memory and return from main.

3.5 The Fortran Interface

The Fortran interface is essentially a wrapper of the C interface discussed in Section 3.4. The way to hook up IPOPT in a Fortran program is very similar to how it is done for the C interface, and the functions of the Fortran interface correspond one-to-one to the those of the C and C++ interface, including their arguments. You can find an implementation of the example problem (4)–(7) in $IPOPTDIR/Ipopt/examples/hs071_f.

The only special things to consider are:

- The return value of the function IPCREATE is of an INTEGER type that must be large enough to capture a pointer on the particular machine. This means, that you have to declare the "handle" for the IpoptProblem as INTEGER*8 if your program is compiled in 64-bit mode. All other INTEGER-type variables must be of the regular type.

- For the call of IPSOLVE (which is the function that is to be called to run IPOPT), all arrays, including those for the dual variables, must be given (in contrast to the C interface). The return value IERR of this function indicates the outcome of the optimization (see the include file IpReturnCodes.inc in the IPOPT include directory).

- The return IERR value of the remaining functions has to be set to zero, unless there was a problem during execution of the function call.

- The callback functions (EV_* in the example) include the arguments IDAT and DAT, which are INTEGER and DOUBLE PRECISION arrays that are passed unmodified between the main program calling IPSOLVE and the evaluation subroutines EV_* (similarly to UserDataPtr arguments in the C interface). These arrays can be used to pass "private" data between the main program and the user-provided Fortran subroutines.

The last argument of the `EV_*` subroutines, `IERR`, is to be set to 0 by the user on return, unless there was a problem during the evaluation of the optimization problem function/derivative for the given point `X` (then it should return a non-zero value).

3.6 The Java Interface

based on documentation by Rafael de Pelegrini Soares[18]

The Java interface offers an abstract base class `Ipopt` with basic methods to specify an NLP, set a number of IPOPT options, to request IPOPT to solve the NLP, and to retrieve a found solution, if any. A HTML documentation of all available interface methods of the `Ipopt` class can be generated via `javadoc` by executing `make doc` in the JIPOPT build directory.

In the following, we discuss necessary steps to implement the HS071 example with JIPOPT.

First, we create a new directory and therein sub directories `org/coinor/`. Into `org/coinor/` we copy the file `Ipopt.java`, which contains the Java code of the interface, from the corresponding JIPOPT source directory (`$IPOPTDIR/Ipopt/contrib/JavaInterface/org/coinor`). Further, we create a directory `lib` next to the `org` directory and place the previously build JIPOPT library into it (`libjipopt.so` on Linux/UNIX, `libjipopt.dylib` on Mac OS X, `jipopt.dll` on Windows), see also Section 2.6.

Next, we create a new Java source file `HS071.java` and define a class `HS071` that extends the class `Ipopt` of JIPOPT. In the class constructor, we call the `create()` method of JIPOPT, which works analogously to `get_nlp_info()` of the C++ interface. It initializes an `IpoptApplication` object and informs JIPOPT about the problem size (number of variables, constraints, nonzeros in Jacobian and Hessian).

```
/** Initialize the bounds and create the native Ipopt problem. */
public HS071() {
  /* Number of nonzeros in the Jacobian of the constraints */
  int nele_jac = 8;

  /* Number of nonzeros in the Hessian of the Lagrangian (lower or
   * upper triangual part only) */
  int nele_hess = 10;

  /* Number of variables */
  int n = 4;

  /* Number of constraints */
  int m = 2;

  /* Index style for the irow/jcol elements */
  int index_style = Ipopt.C_STYLE;

  /* create the IpoptProblem */
  create(n, m,  nele_jac, nele_hess, index_style);
}
```

Next, we add callback functions that are called by JIPOPT to obtain variable bounds, constraint sides, and a starting point:

```
protected boolean get_bounds_info(int n, double[] x_L, double[] x_U,
                                  int m, double[] g_L, double[] g_U) {
  /* set the values of the variable bounds */
```

[18] VRTech Industrial Technologies

```
  for( int i = 0; i < x_L.length; i++ ) {
    x_L[i] = 1.0;
    x_U[i] = 5.0;
  }

  /* set the values of the constraint bounds */
  g_L[0] = 25.0;
  g_U[0] = 2e19;
  g_L[1] = 40.0;
  g_U[1] = 40.0;

  return true;
}

protected boolean get_starting_point(int n, boolean init_x, double[] x,
                                     boolean init_z, double[] z_L, double[] z_U,
                                     int m, boolean init_lambda,double[] lambda) {
  assert init_z == false;
  assert init_lambda = false;

  if( init_x ) {
    x[0] = 1.0;
    x[1] = 5.0;
    x[2] = 5.0;
    x[3] = 1.0;
  }

  return true;
}
```

In the following, we implement the evaluation methods in a way that is very similar to the C++ interface:

```
protected boolean eval_f(int n, double[] x, boolean new_x, double[] obj_value) {
  obj_value[0] = x[0] * x[3] * (x[0] + x[1] + x[2]) + x[2];

  return true;
}

protected boolean eval_grad_f(int n, double[] x, boolean new_x, double[] grad_f) {
  grad_f[0] = x[0] * x[3] + x[3] * (x[0] + x[1] + x[2]);
  grad_f[1] = x[0] * x[3];
  grad_f[2] = x[0] * x[3] + 1;
  grad_f[3] = x[0] * (x[0] + x[1] + x[2]);

  return true;
}

protected boolean eval_g(int n, double[] x, boolean new_x, int m, double[] g) {
  g[0] = x[0] * x[1] * x[2] * x[3];
  g[1] = x[0] * x[0] + x[1] * x[1] + x[2] * x[2] + x[3] * x[3];

  return true;
}
```

```
protected boolean eval_jac_g(int n, double[] x, boolean new_x, int m, int nele_jac,
                             int[] iRow, int[] jCol, double[] values) {
  if( values == null ) {
    /* return the structure of the jacobian */
    /* this particular jacobian is dense */
    iRow[0] = 0;  jCol[0] = 0;
    iRow[1] = 0;  jCol[1] = 1;
    iRow[2] = 0;  jCol[2] = 2;
    iRow[3] = 0;  jCol[3] = 3;
    iRow[4] = 1;  jCol[4] = 0;
    iRow[5] = 1;  jCol[5] = 1;
    iRow[6] = 1;  jCol[6] = 2;
    iRow[7] = 1;  jCol[7] = 3;
  }
  else {
    /* return the values of the jacobian of the constraints */
    values[0] = x[1]*x[2]*x[3]; /* 0,0 */
    values[1] = x[0]*x[2]*x[3]; /* 0,1 */
    values[2] = x[0]*x[1]*x[3]; /* 0,2 */
    values[3] = x[0]*x[1]*x[2]; /* 0,3 */

    values[4] = 2*x[0];         /* 1,0 */
    values[5] = 2*x[1];         /* 1,1 */
    values[6] = 2*x[2];         /* 1,2 */
    values[7] = 2*x[3];         /* 1,3 */
  }

  return true;
}

protected boolean eval_h(int n, double[] x, boolean new_x, double obj_factor,
                         int m, double[] lambda, boolean new_lambda,
                         int nele_hess, int[] iRow, int[] jCol, double[] values) {
  int idx = 0; /* nonzero element counter */
  int row = 0; /* row counter for loop */
  int col = 0; /* col counter for loop */

  if( values == null ) {
    /* return the structure. This is a symmetric matrix, fill the lower left
     * triangle only. */
    /* the hessian for this problem is actually dense */
    idx = 0;
    for( row = 0; row < 4; row++ ) {
      for( col = 0; col <= row; col++ ) {
        iRow[idx] = row;
        jCol[idx] = col;
        idx++;
      }
    }
  }
```

```
  else {
    /* return the values. This is a symmetric matrix, fill the lower left
     * triangle only */

    /* fill the objective portion */
    values[0] = obj_factor * (2*x[3]);               /* 0,0 */
    values[1] = obj_factor * (x[3]);                 /* 1,0 */
    values[2] = 0;                                   /* 1,1 */
    values[3] = obj_factor * (x[3]);                 /* 2,0 */
    values[4] = 0;                                   /* 2,1 */
    values[5] = 0;                                   /* 2,2 */
    values[6] = obj_factor * (2*x[0] + x[1] + x[2]); /* 3,0 */
    values[7] = obj_factor * (x[0]);                 /* 3,1 */
    values[8] = obj_factor * (x[0]);                 /* 3,2 */
    values[9] = 0;                                   /* 3,3 */

    /* add the portion for the first constraint */
    values[1] += lambda[0] * (x[2] * x[3]);          /* 1,0 */
    values[3] += lambda[0] * (x[1] * x[3]);          /* 2,0 */
    values[4] += lambda[0] * (x[0] * x[3]);          /* 2,1 */
    values[6] += lambda[0] * (x[1] * x[2]);          /* 3,0 */
    values[7] += lambda[0] * (x[0] * x[2]);          /* 3,1 */
    values[8] += lambda[0] * (x[0] * x[1]);          /* 3,2 */

    /* add the portion for the second constraint */
    values[0] += lambda[1] * 2;                      /* 0,0 */
    values[2] += lambda[1] * 2;                      /* 1,1 */
    values[5] += lambda[1] * 2;                      /* 2,2 */
    values[9] += lambda[1] * 2;                      /* 3,3 */
  }
  return true;
}
```

Finally, we add a main routine to run this example. The main routines creates an instance of our object and calls the solve method `OptimizeNLP`:

```
public static void main(String[] args) {
  // Create the problem
  HS071 hs071 = new HS071();

  // solve the problem
  int status = hs071.OptimizeNLP(x);

  // print the status and optimal value
  System.out.println("Status    = " + status);
  System.out.println("Obj Value = " + hs071.getObjectiveValue());
}
```

The `OptimizeNLP` method returns the IPOPT solve status as integer, which indicates whether the problem was solved successfully. Further, the methods `getObjectiveValue()`, `getVariableValues()`, and `getConstraintMultipliers()`, `getLowerBoundMultipliers()`, `getUpperBoundMultipliers()` can be used to obtain the objective value, the primal solution value of the variables, and dual solution values, respectively.

3.7 The R Interface

based on documentation by Jelmer Ypma[19]

The `ipoptr` package (see Section 2.7 for installation instructions) offers a R function `ipoptr` which takes an NLP specification, a starting point, and IPOPT options as input and returns information about a IPOPT run (status, message, ...) and a solution point.

In the following, we discuss necessary steps to implement the HS071 example with `ipoptr`. A more detailed documentation of `ipoptr` is available in `Ipopt/contrib/RInterface/inst/doc/ipoptr.pdf`.

First, we define the objective function and its gradient

```
> eval_f <- function( x ) {
     return( x[1]*x[4]*(x[1] + x[2] + x[3]) + x[3] )
 }
> eval_grad_f <- function( x ) {
     return( c( x[1] * x[4] + x[4] * (x[1] + x[2] + x[3]),
                x[1] * x[4],
                x[1] * x[4] + 1.0,
                x[1] * (x[1] + x[2] + x[3]) ) )
 }
```

Then we define a function that returns the value of the two constraints. We define the bounds of the constraints (in this case the g_L and g_U are 25 and 40) later.

```
> # constraint functions
> eval_g <- function( x ) {
     return( c( x[1] * x[2] * x[3] * x[4],
                x[1]^2 + x[2]^2 + x[3]^2 + x[4]^2 ) )
 }
```

Then we define the structure of the Jacobian, which is a dense matrix in this case, and function to evaluate it

```
> eval_jac_g_structure <- list( c(1,2,3,4), c(1,2,3,4) )
> eval_jac_g <- function( x ) {
     return( c ( x[2]*x[3]*x[4],
                 x[1]*x[3]*x[4],
                 x[1]*x[2]*x[4],
                 x[1]*x[2]*x[3],
                 2.0*x[1],
                 2.0*x[2],
                 2.0*x[3],
                 2.0*x[4] ) )
 }
```

The Hessian is also dense, but it looks slightly more complicated because we have to take into account the Hessian of the objective function and of the constraints at the same time, although you could write a function to calculate them both separately and then return the combined result in `eval_h`.

```
> # The Hessian for this problem is actually dense,
> # This is a symmetric matrix, fill the lower left triangle only.
> eval_h_structure <- list( c(1), c(1,2), c(1,2,3), c(1,2,3,4) )
> eval_h <- function( x, obj_factor, hessian_lambda ) {
```

[19] University College London

```
    values <- numeric(10)
    values[1] = obj_factor * (2*x[4]) # 1,1

    values[2] = obj_factor * (x[4])   # 2,1
    values[3] = 0                     # 2,2

    values[4] = obj_factor * (x[4])   # 3,1
    values[5] = 0                     # 4,2
    values[6] = 0                     # 3,3

    values[7] = obj_factor * (2*x[1] + x[2] + x[3]) # 4,1
    values[8] = obj_factor * (x[1])                 # 4,2
    values[9] = obj_factor * (x[1])                 # 4,3
    values[10] = 0                                  # 4,4


    # add the portion for the first constraint
    values[2] = values[2] + hessian_lambda[1] * (x[3] * x[4]) # 2,1

    values[4] = values[4] + hessian_lambda[1] * (x[2] * x[4]) # 3,1
    values[5] = values[5] + hessian_lambda[1] * (x[1] * x[4]) # 3,2

    values[7] = values[7] + hessian_lambda[1] * (x[2] * x[3]) # 4,1
    values[8] = values[8] + hessian_lambda[1] * (x[1] * x[3]) # 4,2
    values[9] = values[9] + hessian_lambda[1] * (x[1] * x[2]) # 4,3

    # add the portion for the second constraint
    values[1] = values[1] + hessian_lambda[2] * 2 # 1,1
    values[3] = values[3] + hessian_lambda[2] * 2 # 2,2
    values[6] = values[6] + hessian_lambda[2] * 2 # 3,3
    values[10] = values[10] + hessian_lambda[2] * 2 # 4,4

    return ( values )
}
```

After the hard part is done, we only have to define the initial values, the lower and upper bounds of the control variables, and the lower and upper bounds of the constraints. If a variable or a constraint does not have lower or upper bounds, the values `-Inf` or `Inf` can be used. If the upper and lower bounds of a constraint are equal, Ipopt recognizes this as an equality constraint and acts accordingly.

```
> # initial values
> x0 <- c( 1, 5, 5, 1 )
> # lower and upper bounds of control
> lb <- c( 1, 1, 1, 1 )
> ub <- c( 5, 5, 5, 5 )
> # lower and upper bounds of constraints
> constraint_lb <- c(  25, 40 )
> constraint_ub <- c( Inf, 40 )
```

Finally, we can call IPOPT with the `ipoptr` function. In order to redirect the IPOPT output into a file, we use IPOPT's `output_file` and `print_level` options.

```
> opts <- list("print_level" = 0,
```

```
              "file_print_level" = 12,
              "output_file" = "hs071_nlp.out")
> print( ipoptr( x0 = x0,
                 eval_f = eval_f,
                 eval_grad_f = eval_grad_f,
                 lb = lb,
                 ub = ub,
                 eval_g = eval_g,
                 eval_jac_g = eval_jac_g,
                 constraint_lb = constraint_lb,
                 constraint_ub = constraint_ub,
                 eval_jac_g_structure = eval_jac_g_structure,
                 eval_h = eval_h,
                 eval_h_structure = eval_h_structure,
                 opts = opts) )

Call:

ipoptr(x0 = x0, eval_f = eval_f, eval_grad_f = eval_grad_f, lb = lb,
  ub = ub, eval_g = eval_g, eval_jac_g = eval_jac_g,
  eval_jac_g_structure = eval_jac_g_structure, constraint_lb = constraint_lb,
  constraint_ub = constraint_ub, eval_h = eval_h, eval_h_structure = eval_h_structure,
  opts = opts)

Ipopt solver status: 0 ( SUCCESS: Algorithm terminated
successfully at a locally optimal point, satisfying the
convergence tolerances (can be specified by options). )

Number of Iterations....: 8
Optimal value of objective function:  17.0140171451792
Optimal value of controls: 1 4.743 3.82115 1.379408
```

To pass additional data to the evaluation routines, one can either supply additional arguments to the user defined functions and `ipoptr` or define an environment that holds the data and pass this environment to `ipoptr`. Both methods are shown in the file `tests/parameters.R` that comes with `ipoptr`.

As a very simple example, suppose we want to find the minimum of

$$f(x) = a_1 x^2 + a_2 x + a_3$$

for different values of the parameters a_1, a_2 and a_3.

First, we define the objective function and its gradient using, assuming that there is some variable `params` that contains the values of the parameters.

```
> eval_f_ex1 <- function(x, params) {
     return( params[1]*x^2 + params[2]*x + params[3] )
 }
> eval_grad_f_ex1 <- function(x, params) {
     return( 2*params[1]*x + params[2] )
 }
```

Note, that the first parameter should always be the control variable. All of the user-defined functions should contain the same set of additional parameters. You have to supply them as input argument to all functions, even if you're not using them in some of the functions.

Then we can solve the problem for a specific set of parameters, in this case $a_1 = 1$, $a_2 = 2$ and $a_3 = 3$, from initial value $x_0 = 0$, with the following command

```
> # solve using ipoptr with additional parameters
> ipoptr(x0          = 0,
         eval_f      = eval_f_ex1,
         eval_grad_f = eval_grad_f_ex1,
         opts        = list("print_level"=0),
         params      = c(1,2,3) )

Call:

ipoptr(x0 = 0, eval_f = eval_f_ex1, eval_grad_f = eval_grad_f_ex1,
    opts = list(print_level = 0), params = c(1, 2, 3))


Ipopt solver status: 0 ( SUCCESS: Algorithm terminated
successfully at a locally optimal point, satisfying the
convergence tolerances (can be specified by options). )

Number of Iterations....: 1
Optimal value of objective function:  2
Optimal value of controls: -1
```

For the second method, we don't have to supply the parameters as additional arguments to the function.

```
> eval_f_ex2 <- function(x) {
     return( params[1]*x^2 + params[2]*x + params[3] )
 }
> eval_grad_f_ex2 <- function(x) {
     return( 2*params[1]*x + params[2] )
 }
```

Instead, we define an environment that contains specific values of `params`

```
> # define a new environment that contains params
> auxdata        <- new.env()
> auxdata$params <- c(1,2,3)
```

To solve this we supply `auxdata` as an argument to `ipoptr`, which will take care of evaluating the functions in the correct environment, so that the auxiliary data is available.

```
> # pass the environment that should be used to evaluate functions to ipoptr
> ipoptr(x0                 = 0,
         eval_f             = eval_f_ex2,
         eval_grad_f        = eval_grad_f_ex2,
         ipoptr_environment = auxdata,
         opts               = list("print_level"=0) )

Call:

ipoptr(x0 = 0, eval_f = eval_f_ex2, eval_grad_f = eval_grad_f_ex2,
    opts = list(print_level = 0), ipoptr_environment = auxdata)
```

```
Ipopt solver status: 0 ( SUCCESS: Algorithm terminated
successfully at a locally optimal point, satisfying the
convergence tolerances (can be specified by options). )

Number of Iterations....: 1
Optimal value of objective function:  2
Optimal value of controls: -1
```

3.8 The MATLAB Interface

based on documentation by Peter Carbonetto[20]

See Section 2.8 for instructions on how to build a `mex` file of the MATLAB interface for IPOPT and how to make it known to MATLAB.

The `$IPOPTDIR/contrib/MatlabInterface/examples` directory contains several illustrative examples on how to use the MATLAB interface. The best way to understand how to use the interface is to carefully go over these examples.

For more information, type `help ipopt` in the MATLAB prompt.

Further, Jonas Asprion assembled information about the MATLAB `ipopt` function and its arguments: `http://www.idsc.ethz.ch/Downloads/IPOPT_InstallMatlab/IPOPT_MatlabInterface_V0p1.pdf`

Note, that this document refers to IPOPT versions before 3.11. With 3.11, the `auxdata` option has been removed from the `mex` code. The new wrapper function `ipopt_auxdata` implements the same functionality as the previous `ipopt` function, but uses MATLAB function handles to do so.

4 Special Features

4.1 Derivative Checker

When writing code for the evaluation of derivatives it is very easy to make mistakes (much easier than writing it correctly the first time :)). As a convenient feature, IPOPT provides the option to run a simple derivative checker, based on finite differences, before the optimization is started.

To use the derivative checker, you need to use the option `derivative_test`. By default, this option is set to `none`, i.e., no finite difference test is performed, It is set to `first-order`, then the first derivatives of the objective function and the constraints are verified, and for the setting `second-order`, the second derivatives are tested as well.

The verification is done by a simple finite differences approximation, where each component of the user-provided starting point is perturbed one of the other. The relative size of the perturbation is determined by the option `derivative_test_perturbation`. The default value (10^{-8}, about the square root of the machine precision) is probably fine in most cases, but if you believe that you see wrong warnings, you might want to play with this parameter. When the test is performed, IPOPT prints out a line for every partial derivative, for which the user-provided derivative value deviates too much from the finite difference approximation. The relative tolerance for deciding when a warning should be issued, is determined by the option `derivative_test_tol`. If you want to see the user-provided and estimated derivative values with the relative deviation for each single partial derivative, you can switch the `derivative_test_print_all` option to `yes`.

A typical output is:

```
Starting derivative checker.

* grad_f[          2] = -6.5159999999999991e+02    ~ -6.5559997134793468e+02  [ 6.101e-03]
```

[20] University of British Columbia

```
* jac_g [    4,    4] =  0.0000000000000000e+00    ~  2.2160643690464592e-02  [ 2.216e-02]
* jac_g [    4,    5] =  1.3798494268463347e+01 v  ~  1.3776333629422766e+01  [ 1.609e-03]
* jac_g [    6,    7] =  1.4776333636790881e+01 v  ~  1.3776333629422766e+01  [ 7.259e-02]

Derivative checker detected 4 error(s).
```

The star ("*") in the first column indicates that this line corresponds to some partial derivative for which the error tolerance was exceeded. Next, we see which partial derivative is concerned in this output line. For example, in the first line, it is the second component of the objective function gradient (or the third, if the `C_STYLE` numbering is used, i.e., when counting of indices starts with 0 instead of 1). The first floating point number is the value given by the user code, and the second number (after "~") is the finite differences estimation. Finally, the number in square brackets is the relative difference between these two numbers.

For constraints, the first index after `jac_g` is the index of the constraint, and the second one corresponds to the variable index (again, the choice of the numbering style matters).

Since also the sparsity structure of the constraint Jacobian has to be provided by the user, it can be faulty as well. For this, the "v" after a user-provided derivative value indicates that this component of the Jacobian is part of the user provided sparsity structure. If there is no "v", it means that the user did not include this partial derivative in the list of non-zero elements. In the above output, the partial derivative "`jac_g[4,4]`" is non-zero (based on the finite difference approximation), but it is not included in the list of non-zero elements (missing "v"), so that the user probably made a mistake in the sparsity structure. The other two Jacobian entries are provided in the non-zero structure but their values seem to be off.

For second derivatives, the output looks like:

```
*             obj_hess[    1,    1] =  1.8810000000000000e+03 v  ~  1.8820000036612328e+03  [ 5.314e-04]
*     3-th constr_hess[    2,    4] =  1.0000000000000000e+00 v  ~  0.0000000000000000e+00  [ 1.000e+00]
```

There, the first line shows the deviation of the user-provided partial second derivative in the Hessian for the objective function, and the second line show an error in a partial derivative for the Hessian of the third constraint (again, the numbering style matters).

Since the second derivatives are approximates by finite differences of the first derivatives, you should first correct errors for the first derivatives. Also, since the finite difference approximations are quite expensive, you should try to debug a small instance of your problem if you can.

Another useful option is `derivative_test_first_index` which allows your to start the derivative test with variables with a larger index. Finally, it is of course always a good idea to run your code through some memory checker, such as `valgrind` on Linux.

4.2 Quasi-Newton Approximation of Second Derivatives

Ipopt has an option to approximate the Hessian of the Lagrangian by a limited-memory quasi-Newton method (L-BFGS). You can use this feature by setting the `hessian_approximation` option to the value `limited-memory`. In this case, it is not necessary to implement the Hessian computation method `eval_h` in `TNLP`. If you are using the C or Fortran interface, you still need to implement these functions, but they should return `false` or `IERR=1`, respectively, and don't need to do anything else.

In general, when second derivatives can be computed with reasonable computational effort, it is usually a good idea to use them, since then Ipopt normally converges in fewer iterations and is more robust. An exception might be in cases, where your optimization problem has a dense Hessian, i.e., a large percentage of non-zero entries in the Hessian. In such a case, using the quasi-Newton approximation might be better, even if it increases the number of iterations, since with exact second derivatives the computation time per iteration might be significantly higher due to the very large number of non-zero elements in the linear systems that Ipopt solve in order to compute the search direction.

Since the Hessian of the Lagrangian is zero for all variables that appear only linearly in the objective and constraint functions, the Hessian approximation should only take place in the space of

all nonlinear variables. By default, it is assumed that all variables are nonlinear, but you can tell IPOPT explicitly which variables are nonlinear, using the `get_number_of_nonlinear_variables` and `get_list_of_nonlinear_variables` method of the `TNLP` class, see Section 3.3.4. (Those methods have been implemented for the AMPL interface, so you would automatically only approximate the Hessian in the space of the nonlinear variables, if you are using the quasi-Newton option for AMPL models.) Currently, those two methods are not available through the C or Fortran interface.

4.3 Warm-Starting Capabilities via AMPL

based on documentation by Victor M. Zavala[21]

Warm-starting an interior-point algorithm is an important issue. One of the main difficulties arises from the fact that full-space variable information is required to generate the warm-starting point. While IPOPT is currently equipped to retrieve and receive this type of information through the `TNLP` interface, there exist some communication barriers in the AMPL interface. When the user solves the problem (1)–(3), IPOPT will only return the optimal values of the primal variables x and of the constraint multipliers corresponding to the active bounds of $g(x)$ (see (2)). The constraint multiplier values can be accessed through the `.dual` suffix or through the `.sol` file. If this information is used to solve the same problem again, you will notice that IPOPT will take some iterations in finding the same solution. The reason for this is that we are missing the input information of the multipliers z^L and z^U corresponding to the variable bounds (see (3)).

However, IPOPT also passes the values of the bound multipliers z^L and z^U to AMPL. This will be communicated to the AMPL user through the suffixes `ipopt_zL_out` and `ipopt_zU_out`, respectively. The user does not need to declare these suffixes, they will be generated automatically in the AMPL interface. The user can use the suffix values to initialize the bound multipliers for subsequent calls. In order to pass this information to IPOPT, the user will need to declare and assign values to the suffixes `ipopt_zL_in` and `ipopt_zU_in`. For instance, for a given variable `x[i]`, this can be done by setting:

```
let x[i].ipopt_zL_in := x[i].ipopt_zL_out;
let x[i].ipopt_zU_in := x[i].ipopt_zU_out;
```

If the user does not specify some of these values, IPOPT will set these multipliers to 1.0 (as before). In order to make the warm-start effective, the user has control over the following options from AMPL:
`warm_start_init_point`
`warm_start_bound_push`
`warm_start_mult_bound_push`

Note, that the use of this feature is far from solving the complicated issue of warm- starting interior-point algorithms. As a general advice, this feature will be useful if the user observes that the solution of subsequent problems (i.e., for different data instances) preserves the same set of active inequalities and bounds (monitor the values of z^L and z^U for subsequent solutions). In this case, initializing the bound multipliers and setting `warm_start_init_point` to `yes` and setting `warm_start_bound_push`, `warm_start_mult_bound_push` and `mu_init` to a small value (10^{-6} or so) will reduce significantly the number of iterations. This is particularly useful in setting up on-line applications and high-level optimization strategies in AMPL. If active-set changes are observed between subsequent solutions, then this strategy might not decrease the number of iterations (in some cases, it might even tend to increase the number of iterations).

You might also want to try the adaptive barrier update (instead of the default monotone one where above we chose the initial value 10^{-6}) when doing the warm start. This can be activated by setting the `mu_strategy` option to `adaptive`. Also the option `mu_oracle` gives some alternative choices. In general, the adaptive choice often leads to less iterations, but the computational cost per iteration might be higher.

The file `$IPOPTDIR/Ipopt/doc/hs071_warmstart.mod` illustrates the use of the warm-start feature on the HS071 problem, see also Section 3.1.

[21] Department of Chemical Engineering, Carnegie Mellon University

4.4 sIpopt: Optimal Sensitivity Based on Ipopt

based on documentation by Hans Pirnay[22]

The sIPOPT project provides a toolbox that uses NLP sensitivity theory to generate fast approximations to solutions when parameters in the problem change. It has been developed primarily by Hans Pirnay (RWTH-Aachen), Rodrigo López-Negrete (CMU), and Lorenz Biegler (CMU).

Sensitivity of nonlinear programming problems is a key step in any optimization study. Sensitivity provides information on regularity and curvature conditions at KKT points, assesses which variables play dominant roles in the optimization, and provides first order estimates for parametric nonlinear programs. Moreover, for NLP algorithms that use exact second derivatives, sensitivity can be implemented very efficiently within NLP solvers and provide valuable information with very little added computation. This implementation provides IPOPT with the capabilities to calculate sensitivities, and approximate perturbed solutions with them.

The basic sensitivity strategy implemented here is based on the application of the Implicit Function Theorem (IFT) to the KKT conditions of the NLP. As shown by Fiacco (1983), sensitivities can be obtained from a solution with suitable regularity conditions merely by solving a linearization of the KKT conditions. More details can be found in [7]. If you are using sIPOPT for your research, please cite [7].

The sIPOPT project is available in the IPOPT repository under `$IPOPTDIR/Ipopt/contrib/sIPOPT`. After having installed IPOPT successfully, sIPOPT can be build and installed by changing to the directory `$IPOPTDIR/build/Ipopt/contrib/sIPOPT` and executing `make install`. This should copy the generated libraries `libsipopt.*` to `$IPOPTDIR/build/lib` and an AMPL executable `ipopt_sens` to `$IPOPTDIR/build/bin`.

The files `$IPOPTDIR/Ipopt/contrib/sIPOPT/examples/parametric_ampl/parametric.{mod,run}` are an example that shows how to use sIPOPT to solve the NLP

$$\begin{aligned} \min \quad & x_1^2 + x_2^2 + x_3^2, && (10)\\ \text{such that} \quad & 6x_1 + 3x_2 + 2x_3 = p_1, && (11)\\ & p_2 x_1 + x_2 - x_3 = 1, && (12)\\ & x_1, x_2, x_3 \geq 0, && (13) \end{aligned}$$

where we perturb the parameters p_1 and p_2 from $p_a = (p_1, p_2) = (5, 1)$ to $p_b = (4.5, 1)$.

Note, that sIPOPT has been developed under the constraint that it must work with the regular IPOPT code. Due to this constraint, some compromises had to be made. However, there is an ongoing effort to develop sIPOPT 2, which is a fork of the IPOPT code that allows for the explicit definition of parametric NLPs. This code can be found at `https://github.com/athrpf/sipopt2`. If you have questions about sIPOPT 2, please contact Hans Pirnay.

5 Ipopt Options

IPOPT has many (maybe too many) options that can be adjusted for the algorithm. Options are all identified by a string name, and their values can be of one of three types: Number (real), Integer, or String. Number options are used for things like tolerances, integer options are used for things like maximum number of iterations, and string options are used for setting algorithm details, like the NLP scaling method. Options can be set through code, through the AMPL interface if you are using AMPL, or by creating a `ipopt.opt` file in the directory you are executing IPOPT.

The `ipopt.opt` file is read line by line and each line should contain the option name, followed by whitespace, and then the value. Comments can be included with the `#` symbol. For example,

```
# This is a comment
```

[22] RWTH Aachen, `hans.pirnay@avt.rwth-aachen.de`

```
# Turn off the NLP scaling
nlp_scaling_method none

# Change the initial barrier parameter
mu_init 1e-2

# Set the max number of iterations
max_iter 500
```

is a valid `ipopt.opt` file.

Options can also be set in code. Have a look at the examples to see how this is done.

A subset of IPOPT options are available through AMPL. To set options through AMPL, use the internal AMPL command `options`. For example,

```
options ipopt_options "nlp_scaling_method=none mu_init=1e-2 max_iter=500"
```

is a valid options command in AMPL. The most important options are referenced in Appendix C. To see which options are available through AMPL, you can run the AMPL solver executable with the "`-=`" flag from the command prompt. To specify other options when using AMPL, you can always create `ipopt.opt`. Note, the `ipopt.opt` file is given preference when setting options. This way, you can easily override any options set in a particular executable or AMPL model by specifying new values in `ipopt.opt`.

For a list of the most important valid options, see the Appendix C. You can print the documentation for all IPOPT options by using the option

```
print_options_documentation yes
```

and running IPOPT (like the AMPL solver executable, for instance). This will output the documentation of almost all options to the console.

6 Ipopt Output

This section describes the standard IPOPT console output with the default setting for `print_level`. The output is designed to provide a quick summary of each iteration as IPOPT solves the problem.

Before IPOPT starts to solve the problem, it displays the problem statistics (number of nonzero-elements in the matrices, number of variables, etc.). Note that if you have fixed variables (both upper and lower bounds are equal), IPOPT may remove these variables from the problem internally and not include them in the problem statistics.

Following the problem statistics, IPOPT will begin to solve the problem and you will see output resembling the following,

```
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  1.6109693e+01 1.12e+01 5.28e-01   0.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  1.8029749e+01 9.90e-01 6.62e+01   0.1 2.05e+00    -  2.14e-01 1.00e+00f  1
   2  1.8719906e+01 1.25e-02 9.04e+00  -2.2 5.94e-02   2.0 8.04e-01 1.00e+00h  1
```

and the columns of output are defined as,

`iter`: The current iteration count. This includes regular iterations and iterations during the restoration phase. If the algorithm is in the restoration phase, the letter `r'` will be appended to the iteration number.

`objective`: The unscaled objective value at the current point. During the restoration phase, this value remains the unscaled objective value for the original problem.

`inf_pr`: The unscaled constraint violation at the current point. This quantity is the infinity-norm (max) of the (unscaled) constraints (2). During the restoration phase, this value remains the constraint violation of the original problem at the current point. The option `inf_pr_output` can be used to switch to the printing of a different quantity.

Tag	Description
f	f-type iteration in the filter method w/o second order correction
F	f-type iteration in the filter method w/ second order correction
h	h-type iteration in the filter method w/o second order correction
H	h-type iteration in the filter method w/ second order correction
k	penalty value unchanged in merit function method w/o second order correction
K	penalty value unchanged in merit function method w/ second order correction
n	penalty value updated in merit function method w/o second order correction
N	penalty value updated in merit function method w/ second order correction
R	Restoration phase just started
w	in watchdog procedure
s	step accepted in soft restoration phase
t/T	tiny step accepted without line search
r	some previous iterate restored

Table 1: Diagnostic output in `alpha_pr` column.

`inf_du:` The scaled dual infeasibility at the current point. This quantity measure the infinity-norm (max) of the internal dual infeasibility, Eq. (4a) in the implementation paper [13], including inequality constraints reformulated using slack variables and problem scaling. During the restoration phase, this is the value of the dual infeasibility for the restoration phase problem.

`lg(mu):` $\log_{10}$ of the value of the barrier parameter μ.

`||d||:` The infinity norm (max) of the primal step (for the original variables x and the internal slack variables s). During the restoration phase, this value includes the values of additional variables, p and n (see Eq. (30) in [13]).

`lg(rg):` $\log_{10}$ of the value of the regularization term for the Hessian of the Lagrangian in the augmented system (δ_w in Eq. (26) and Section 3.1 in [13]). A dash ("`-`") indicates that no regularization was done.

`alpha_du:` The stepsize for the dual variables (α_k^z in Eq. (14c) in [13]).

`alpha_pr:` The stepsize for the primal variables (α_k in Eq. (14a) in [13]). The number is usually followed by a character for additional diagnostic information regarding the step acceptance criterion, see Table 1.

`ls:` The number of backtracking line search steps (does not include second-order correction steps).

Note that the step acceptance mechanisms in IPOPT consider the barrier objective function (Eq (3a) in [13]) which is usually different from the value reported in the `objective` column. Similarly, for the purposes of the step acceptance, the constraint violation is measured for the internal problem formulation, which includes slack variables for inequality constraints and potentially scaling of the constraint functions. This value, too, is usually different from the value reported in `inf_pr`. As a consequence, a new iterate might have worse values both for the objective function and the constraint violation as reported in the iteration output, seemingly contradicting globalization procedure.

When the algorithm terminates, IPOPT will output a message to the screen based on the return status of the call to `Optimize`. The following is a list of the possible return codes, their corresponding output message to the console, and a brief description.

`Solve_Succeeded:`

Console Message: `EXIT: Optimal Solution Found.`

This message indicates that IPOPT found a (locally) optimal point within the desired tolerances.

`Solved_To_Acceptable_Level`:

Console Message: `EXIT: Solved To Acceptable Level.`

This indicates that the algorithm did not converge to the "desired" tolerances, but that it was able to obtain a point satisfying the "acceptable" tolerance level as specified by the `acceptable_*` options. This may happen if the desired tolerances are too small for the current problem.

`Feasible_Point_Found`:

Console Message: `EXIT: Feasible point for square problem found.`

This message is printed if the problem is "square" (i.e., it has as many equality constraints as free variables) and IPOPT found a feasible point.

`Infeasible_Problem_Detected`:

Console Message: `EXIT: Converged to a point of local infeasibility. Problem may be infeasible.`

The restoration phase converged to a point that is a minimizer for the constraint violation (in the ℓ_1-norm), but is not feasible for the original problem. This indicates that the problem may be infeasible (or at least that the algorithm is stuck at a locally infeasible point). The returned point (the minimizer of the constraint violation) might help you to find which constraint is causing the problem. If you believe that the NLP is feasible, it might help to start the optimization from a different point.

`Search_Direction_Becomes_Too_Small`:

Console Message: `EXIT: Search Direction is becoming Too Small.`

This indicates that IPOPT is calculating very small step sizes and is making very little progress. This could happen if the problem has been solved to the best numerical accuracy possible given the current scaling.

`Diverging_Iterates`:

Console Message: `EXIT: Iterates divering; problem might be unbounded.`

This message is printed if the max-norm of the iterates becomes larger than the value of the option `diverging_iterates_tol`. This can happen if the problem is unbounded below and the iterates are diverging.

`User_Requested_Stop`:

Console Message: `EXIT: Stopping optimization at current point as requested by user.`

This message is printed if the user call-back method `intermediate_callback` returned `false` (see Section 3.3.4).

`Maximum_Iterations_Exceeded`:

Console Message: `EXIT: Maximum Number of Iterations Exceeded.`

This indicates that IPOPT has exceeded the maximum number of iterations as specified by the option `max_iter`.

`Maximum_CpuTime_Exceeded`:

Console Message: `EXIT: Maximum CPU time exceeded.`

This indicates that IPOPT has exceeded the maximum number of CPU seconds as specified by the option `max_cpu_time`.

`Restoration_Failed`:

Console Message: `EXIT: Restoration Failed!`

This indicates that the restoration phase failed to find a feasible point that was acceptable to the filter line search for the original problem. This could happen if the problem is highly degenerate, does not satisfy the constraint qualification, or if your NLP code provides incorrect derivative information.

`Error_In_Step_Computation`:
Console Output: `EXIT: Error in step computation (regularization becomes too large?)!`
This messages is printed if IPOPT is unable to compute a search direction, despite several attempts to modify the iteration matrix. Usually, the value of the regularization parameter then becomes too large. One situation where this can happen is when values in the Hessian are invalid (`NaN` or `Inf`). You can check whether this is true by using the `check_derivatives_for_naninf` option.

`Invalid_Option`:
Console Message: (details about the particular error will be output to the console)
This indicates that there was some problem specifying the options. See the specific message for details.

`Not_Enough_Degrees_Of_Freedom`:
Console Message: `EXIT: Problem has too few degrees of freedom.`
This indicates that your problem, as specified, has too few degrees of freedom. This can happen if you have too many equality constraints, or if you fix too many variables (IPOPT removes fixed variables by default, see also the `fixed_variable_treatment` option).

`Invalid_Problem_Definition`:
Console Message: (no console message, this is a return code for the C and Fortran interfaces only.)
This indicates that there was an exception of some sort when building the `IpoptProblem` structure in the C or Fortran interface. Likely there is an error in your model or the `main` routine.

`Unrecoverable_Exception`:
Console Message: (details about the particular error will be output to the console)
This indicates that IPOPT has thrown an exception that does not have an internal return code. See the specific message for details.

`NonIpopt_Exception_Thrown`:
Console Message: `Unknown Exception caught in Ipopt`
An unknown exception was caught in IPOPT. This exception could have originated from your model or any linked in third party code.

`Insufficient_Memory`:
Console Message: `EXIT: Not enough memory.`
An error occurred while trying to allocate memory. The problem may be too large for your current memory and swap configuration.

`Internal_Error`:
Console: `EXIT: INTERNAL ERROR: Unknown SolverReturn value - Notify IPOPT Authors.`
An unknown internal error has occurred. Please notify the authors of IPOPT via the mailing list.

6.1 Diagnostic Tags for Ipopt

To print additional diagnostic tags for each iteration of IPOPT, set the options `print_info_string` to `yes`. With this, a tag will appear at the end of an iteration line with the following diagnostic meaning that are useful to flag difficulties for a particular IPOPT run. A list of possible strings is given in Table 2.

A Triplet Format for Sparse Matrices

IPOPT was designed for optimizing large sparse nonlinear programs. Because of problem sparsity, the required matrices (like the constraints Jacobian or Lagrangian Hessian) are not stored as dense matrices,

Tag	Description	Reference
!	Tighten resto tolerance if only slightly infeasible	Section 3.3 in [13]
A	Current iteration is acceptable	Alternate termination
a	Perturbation for PD singularity impossible, assume singular	Section 3.1 in [13]
C	Second Order Correction taken	Section 2.4 in [13]
Dh	Hessian degenerate based on multiple iterations	Section 3.1 in [13]
Dhj	Hessian/Jacobian degenerate based on multiple iterations	Section 3.1 in [13]
Dj	Jacobian degenerate based on multiple iterations	Section 3.1 in [13]
dx	δ_x perturbation too large	Section 3.1 in [13]
e	Cutting back α due to evaluation error	in backtracking line search
F-	Filter should be reset, but maximal resets exceeded	Section 2.3 in [13]
F+	Resetting filter due to last few rejections of filter	Section 2.3 in [13]
L	Degenerate Jacobian, δ_c already perturbed	Section 3.1 in [13]
l	Degenerate Jacobian, δ_c perturbed	Section 3.1 in [13]
M	Magic step taken for slack variables	in backtracking line search
Nh	Hessian not yet degenerate	Section 3.1 in [13]
Nhj	Hessian/Jacobian not yet degenerate	Section 3.1 in [13]
Nj	Jacobian not yet degenerate	Section 3.1 in [13]
NW	Warm start initialization failed	in Warm Start Initialization
q	PD system possibly singular, attempt improving sol. quality	Section 3.1 in [13]
R	Solution of restoration phase	Section 3.3 in [13]
S	PD system possibly singular, accept current solution	Section 3.1 in [13]
s	PD system singular	Section 3.1 in [13]
s	Square Problem. Set multipliers to zero	Default initialization routine
Tmax	Trial θ is larger than θ_{max}	filter parameter, see (21) in [13]
W	Watchdog line search procedure successful	Section 3.2 in [13]
w	Watchdog line search procedure unsuccessful, stopped	Section 3.2 in [13]
Wb	Undoing most recent SR1 update	Section 5.4.1 in [1]
We	Skip Limited-Memory Update in restoration phase	Section 5.4.1 in [1]
Wp	Safeguard $B^0 = \sigma I$ for Limited-Memory Update	Section 5.4.1 in [1]
Wr	Resetting Limited-Memory Update	Section 5.4.1 in [1]
Ws	Skip Limited-Memory Update since $s^T y$ is not positive	Section 5.4.1 in [1]
WS	Skip Limited-Memory Update since Δx is too small	Section 5.4.1 in [1]
y	Dual infeasibility, use least square multiplier update	during ipopt algorithm
z	Apply correction to bound multiplier if too large	during ipopt algorithm

Table 2: Diagnostic output appended using `print_info_sting`.

but rather in a sparse matrix format. For the tutorials in this document, we use the triplet format. Consider the matrix

$$\begin{bmatrix} 1.1 & 0 & 0 & 0 & 0 & 0 & 0.5 \\ 0 & 1.9 & 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 2.6 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 7.8 & 0.6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.5 & 2.7 & 0 & 0 \\ 1.6 & 0 & 0 & 0 & 0.4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.9 & 1.7 \end{bmatrix} \tag{14}$$

A standard dense matrix representation would need to store $7 \cdot 7$=49 floating point numbers, where many entries would be zero. In triplet format, however, only the nonzero entries are stored. The triplet format records the row number, the column number, and the value of all nonzero entries in the matrix. For the matrix above, this means storing 14 integers for the rows, 14 integers for the columns, and 14 floating point numbers for the values. While this does not seem like a huge space saving over the 49 floating point numbers stored in the dense representation, for larger matrices, the space savings are very dramatic[23].

The parameter `index_style` in `get_nlp_info` tells IPOPT if you prefer to use C style indexing (0-based, i.e., starting the counting at 0) for the row and column indices or Fortran style (1-based). Tables 3 and 4 below show the triplet format for both indexing styles, using the example matrix (14).

row	col	value
`iRow[0] = 1`	`jCol[0] = 1`	`values[0] = 1.1`
`iRow[1] = 1`	`jCol[1] = 7`	`values[1] = 0.5`
`iRow[2] = 2`	`jCol[2] = 2`	`values[2] = 1.9`
`iRow[3] = 2`	`jCol[3] = 7`	`values[3] = 0.5`
`iRow[4] = 3`	`jCol[4] = 3`	`values[4] = 2.6`
`iRow[5] = 3`	`jCol[5] = 7`	`values[5] = 0.5`
`iRow[6] = 4`	`jCol[6] = 3`	`values[6] = 7.8`
`iRow[7] = 4`	`jCol[7] = 4`	`values[7] = 0.6`
`iRow[8] = 5`	`jCol[8] = 4`	`values[8] = 1.5`
`iRow[9] = 5`	`jCol[9] = 5`	`values[9] = 2.7`
`iRow[10] = 6`	`jCol[10] = 1`	`values[10] = 1.6`
`iRow[11] = 6`	`jCol[11] = 5`	`values[11] = 0.4`
`iRow[12] = 7`	`jCol[12] = 6`	`values[12] = 0.9`
`iRow[13] = 7`	`jCol[13] = 7`	`values[13] = 1.7`

Table 3: Triplet Format of Matrix (14) with `index_style=FORTRAN_STYLE`

The individual elements of the matrix can be listed in any order, and if there are multiple items for the same nonzero position, the values provided for those positions are added.

The Hessian of the Lagrangian is a symmetric matrix. In the case of a symmetric matrix, you only need to specify the lower left triangular part (or, alternatively, the upper right triangular part). For example, given the matrix,

$$\begin{bmatrix} 1.0 & 0 & 3.0 & 0 & 2.0 \\ 0 & 1.1 & 0 & 0 & 5.0 \\ 3.0 & 0 & 1.2 & 6.0 & 0 \\ 0 & 0 & 6.0 & 1.3 & 9.0 \\ 2.0 & 5.0 & 0 & 9.0 & 1.4 \end{bmatrix} \tag{15}$$

the triplet format is shown in Tables 5 and 6.

[23] For an $n \times n$ matrix, the dense representation grows with the the square of n, while the sparse representation grows linearly in the number of nonzeros.

row	col	value
iRow[0] = 0	jCol[0] = 0	values[0] = 1.1
iRow[1] = 0	jCol[1] = 6	values[1] = 0.5
iRow[2] = 1	jCol[2] = 1	values[2] = 1.9
iRow[3] = 1	jCol[3] = 6	values[3] = 0.5
iRow[4] = 2	jCol[4] = 2	values[4] = 2.6
iRow[5] = 2	jCol[5] = 6	values[5] = 0.5
iRow[6] = 3	jCol[6] = 2	values[6] = 7.8
iRow[7] = 3	jCol[7] = 3	values[7] = 0.6
iRow[8] = 4	jCol[8] = 3	values[8] = 1.5
iRow[9] = 4	jCol[9] = 4	values[9] = 2.7
iRow[10] = 5	jCol[10] = 0	values[10] = 1.6
iRow[11] = 5	jCol[11] = 4	values[11] = 0.4
iRow[12] = 6	jCol[12] = 5	values[12] = 0.9
iRow[13] = 6	jCol[13] = 6	values[13] = 1.7

Table 4: Triplet Format of Matrix (14) with `index_style=C_STYLE`

row	col	value
iRow[0] = 1	jCol[0] = 1	values[0] = 1.0
iRow[1] = 2	jCol[1] = 1	values[1] = 1.1
iRow[2] = 3	jCol[2] = 1	values[2] = 3.0
iRow[3] = 3	jCol[3] = 3	values[3] = 1.2
iRow[4] = 4	jCol[4] = 3	values[4] = 6.0
iRow[5] = 4	jCol[5] = 4	values[5] = 1.3
iRow[6] = 5	jCol[6] = 1	values[6] = 2.0
iRow[7] = 5	jCol[7] = 2	values[7] = 5.0
iRow[8] = 5	jCol[8] = 4	values[8] = 9.0
iRow[9] = 5	jCol[9] = 5	values[9] = 1.4

Table 5: Triplet Format of Matrix (15) with `index_style=FORTRAN_STYLE`

B The Smart Pointer Implementation: `SmartPtr<T>`

The `SmartPtr` class is described in `IpSmartPtr.hpp`. It is a template class that takes care of counting references to objects and deleting them when not references anymore. Instead of pointing to an object with a raw C++ pointer (e.g. `HS071_NLP*`), we use a `SmartPtr`. Every time a `SmartPtr` is set to reference an object, it increments a counter in that object (see the `ReferencedObject` base class if you are interested). If a `SmartPtr` is done with the object, either by leaving scope or being set to point to another object, the counter is decremented. When the count of the object goes to zero, the object is automatically deleted. `SmartPtr`'s are very simple, just use them as you would use a standard pointer.

It is very important to use `SmartPtr`'s instead of raw pointers when passing objects to Ipopt. Internally, Ipopt uses smart pointers for referencing objects. If you use a raw pointer in your executable, the object's counter will NOT get incremented. Then, when Ipopt uses smart pointers inside its own code, the counter will get incremented. However, before Ipopt returns control to your code, it will decrement as many times as it incremented earlier, and the counter will return to zero. Therefore, Ipopt will delete the object. When control returns to you, you now have a raw pointer that points to a deleted object.

This might sound difficult to anyone not familiar with the use of smart pointers, but just follow one simple rule; always use a SmartPtr when creating or passing an Ipopt object.

row	col	value
iRow[0] = 0	jCol[0] = 0	values[0] = 1.0
iRow[1] = 1	jCol[1] = 0	values[1] = 1.1
iRow[2] = 2	jCol[2] = 0	values[2] = 3.0
iRow[3] = 2	jCol[3] = 2	values[3] = 1.2
iRow[4] = 3	jCol[4] = 2	values[4] = 6.0
iRow[5] = 3	jCol[5] = 3	values[5] = 1.3
iRow[6] = 4	jCol[6] = 0	values[6] = 2.0
iRow[7] = 4	jCol[7] = 1	values[7] = 5.0
iRow[8] = 4	jCol[8] = 3	values[8] = 9.0
iRow[9] = 4	jCol[9] = 4	values[9] = 1.4

Table 6: Triplet Format of Matrix (15) with `index_style=C_STYLE`

C Options Reference

Options can be set using `ipopt.opt`, through your own code, or through the AMPL `ipopt_options` command. See Section 5 for an explanation of how to use these commands. Shown here is a list of the most important options for IPOPT. To view the full list of options, you can set the option `print_options_documentation` to `yes` or simply run the IPOPT AMPL solver executable as

```
ipopt --print-options
```

Usually, option values are identical for the regular mode of IPOPT and the restoration phase. However, to set an option value specifically for the restoration phase, the prefix "`resto.`" should be appended. For example, to set the acceptable tolerance for the restoration phase, use the keyword "`resto.acceptable_tol`".

The most common options are:

C.1 Output

print_level: Output verbosity level.
Sets the default verbosity level for console output. The larger this value the more detailed is the output. The valid range for this integer option is $0 \leq$ `print_level` ≤ 12 and its default value is 5.

print_user_options: Print all options set by the user.
If selected, the algorithm will print the list of all options set by the user including their values and whether they have been used. In some cases this information might be incorrect, due to the internal program flow. The default value for this string option is "no".
Possible values:

- no: don't print options
- yes: print options

print_options_documentation: Switch to print all algorithmic options.
If selected, the algorithm will print the list of all available algorithmic options with some documentation before solving the optimization problem. The default value for this string option is "no".
Possible values:

- no: don't print list
- yes: print list

print_frequency_iter: Determines at which iteration frequency the summarizing iteration output line should be printed.
Summarizing iteration output is printed every print_frequency_iter iterations, if at least print_frequency_time seconds have passed since last output. The valid range for this integer option is $1 \leq$ `print_frequency_iter` $<$ $+$`inf` and its default value is 1.

print_frequency_time: Determines at which time frequency the summarizing iteration output line should be printed.
Summarizing iteration output is printed if at least print_frequency_time seconds have passed since last output and the iteration number is a multiple of print_frequency_iter. The valid range for this real option is $0 \leq$ `print_frequency_time` $<$ $+$`inf` and its default value is 0.

output_file: File name of desired output file (leave unset for no file output).
NOTE: This option only works when read from the ipopt.opt options file! An output file with this name will be written (leave unset for no file output). The verbosity level is by default set to "print_level", but can be overridden with "file_print_level". The file name is changed to use only small letters. The default value for this string option is "".
Possible values:

- *: Any acceptable standard file name

file_print_level: Verbosity level for output file.
NOTE: This option only works when read from the ipopt.opt options file! Determines the verbosity level for the file specified by "output_file". By default it is the same as "print_level". The valid range for this integer option is $0 \leq$ `file_print_level` ≤ 12 and its default value is 5.

option_file_name: File name of options file.
By default, the name of the Ipopt options file is "ipopt.opt" - or something else if specified in the IpoptApplication::Initialize call. If this option is set by SetStringValue BEFORE the options file is read, it specifies the name of the options file. It does not make any sense to specify this option within the options file. Setting this option to an empty string disables reading of an options file. The default value for this string option is "ipopt.opt".
Possible values:

- *: Any acceptable standard file name

print_info_string: Enables printing of additional info string at end of iteration output.
This string contains some insider information about the current iteration. For details, look for "Diagnostic Tags" in the Ipopt documentation. The default value for this string option is "no".
Possible values:

- no: don't print string
- yes: print string at end of each iteration output

inf_pr_output: Determines what value is printed in the "inf_pr" output column.
Ipopt works with a reformulation of the original problem, where slacks are introduced and the problem might have been scaled. The choice "internal" prints out the constraint violation of this formulation. With "original" the true constraint violation in the original NLP is printed. The default value for this string option is "original".
Possible values:

- internal: max-norm of violation of internal equality constraints
- original: maximal constraint violation in original NLP

print_timing_statistics: Switch to print timing statistics.
If selected, the program will print the CPU usage (user time) for selected tasks. The default value for this string option is "no".
Possible values:

- no: don't print statistics
- yes: print all timing statistics

C.2 Termination

tol: Desired convergence tolerance (relative).
Determines the convergence tolerance for the algorithm. The algorithm terminates successfully, if the (scaled) NLP error becomes smaller than this value, and if the (absolute) criteria according to "dual_inf_tol", "primal_inf_tol", and "compl_inf_tol" are met. (This is epsilon_tol in Eqn. (6) in implementation paper). See also "acceptable_tol" as a second termination criterion. Note, some other algorithmic features also use this quantity to determine thresholds etc. The valid range for this real option is $0 <$ `tol` $<$ `+inf` and its default value is $1 \cdot 10^{-08}$.

max_iter: Maximum number of iterations.
The algorithm terminates with an error message if the number of iterations exceeded this number. The valid range for this integer option is $0 \leq$ `max_iter` $<$ `+inf` and its default value is 3000.

max_cpu_time: Maximum number of CPU seconds.
A limit on CPU seconds that Ipopt can use to solve one problem. If during the convergence check this limit is exceeded, Ipopt will terminate with a corresponding error message. The valid range for this real option is $0 <$ `max_cpu_time` $<$ `+inf` and its default value is $1 \cdot 10^{+06}$.

dual_inf_tol: Desired threshold for the dual infeasibility.
Absolute tolerance on the dual infeasibility. Successful termination requires that the max-norm of the (unscaled) dual infeasibility is less than this threshold. The valid range for this real option is $0 <$ `dual_inf_tol` $<$ `+inf` and its default value is 1.

constr_viol_tol: Desired threshold for the constraint violation.
Absolute tolerance on the constraint violation. Successful termination requires that the max-norm of the (unscaled) constraint violation is less than this threshold. The valid range for this real option is $0 <$ `constr_viol_tol` $<$ `+inf` and its default value is 0.0001.

compl_inf_tol: Desired threshold for the complementarity conditions.
Absolute tolerance on the complementarity. Successful termination requires that the max-norm of the (unscaled) complementarity is less than this threshold. The valid range for this real option is $0 <$ `compl_inf_tol` $<$ `+inf` and its default value is 0.0001.

acceptable_tol: "Acceptable" convergence tolerance (relative).
Determines which (scaled) overall optimality error is considered to be "acceptable." There are two levels of termination criteria. If the usual "desired" tolerances (see tol, dual_inf_tol etc) are satisfied at an iteration, the algorithm immediately terminates with a success message. On the other hand, if the algorithm encounters "acceptable_iter" many iterations in a row that are considered "acceptable", it will terminate before the desired convergence tolerance is met. This is useful in cases where the algorithm might not be able to achieve the "desired" level of accuracy. The valid range for this real option is $0 <$ `acceptable_tol` $<$ `+inf` and its default value is $1 \cdot 10^{-06}$.

acceptable_iter: Number of "acceptable" iterates before triggering termination.
If the algorithm encounters this many successive "acceptable" iterates (see "acceptable_tol"), it terminates, assuming that the problem has been solved to best possible accuracy given round-off. If it is set to zero, this heuristic is disabled. The valid range for this integer option is $0 \leq$ `acceptable_iter` $<$ `+inf` and its default value is 15.

acceptable_constr_viol_tol: "Acceptance" threshold for the constraint violation.
Absolute tolerance on the constraint violation. "Acceptable" termination requires that the max-norm of the (unscaled) constraint violation is less than this threshold; see also acceptable_tol. The valid range for this real option is $0 <$ `acceptable_constr_viol_tol` $<$ `+inf` and its default value is 0.01.

acceptable_dual_inf_tol: "Acceptance" threshold for the dual infeasibility.
Absolute tolerance on the dual infeasibility. "Acceptable" termination requires that the (max-norm of the unscaled) dual infeasibility is less than this threshold; see also acceptable_tol. The valid range for this real option is $0 <$ `acceptable_dual_inf_tol` $<$ `+inf` and its default value is $1 \cdot 10^{+10}$.

acceptable_compl_inf_tol: "Acceptance" threshold for the complementarity conditions.
Absolute tolerance on the complementarity. "Acceptable" termination requires that the max-norm of the (unscaled) complementarity is less than this threshold; see also acceptable_tol. The valid range for this real option is $0 <$ `acceptable_compl_inf_tol` $<$ `+inf` and its default value is 0.01.

acceptable_obj_change_tol: "Acceptance" stopping criterion based on objective function change.
If the relative change of the objective function (scaled by Max(1,—f(x)—)) is less than this value, this part of the acceptable tolerance termination is satisfied; see also acceptable_tol. This is useful for the quasi-Newton option, which has trouble to bring down the dual infeasibility. The valid range for this real option is $0 \leq$ `acceptable_obj_change_tol` $<$ `+inf` and its default value is $1 \cdot 10^{+20}$.

diverging_iterates_tol: Threshold for maximal value of primal iterates.
If any component of the primal iterates exceeded this value (in absolute terms), the optimization is aborted with the exit message that the iterates seem to be diverging. The valid range for this real option is $0 <$ `diverging_iterates_tol` $<$ `+inf` and its default value is $1 \cdot 10^{+20}$.

C.3 NLP Scaling

obj_scaling_factor: Scaling factor for the objective function.
This option sets a scaling factor for the objective function. The scaling is seen internally by Ipopt but the unscaled objective is reported in the console output. If additional scaling parameters are computed (e.g. user-scaling or gradient-based), both factors are multiplied. If this value is chosen to be negative, Ipopt will maximize the objective function instead of minimizing it. The valid range for this real option is $-$`inf` $<$ `obj_scaling_factor` $<$ `+inf` and its default value is 1.

nlp_scaling_method: Select the technique used for scaling the NLP.
Selects the technique used for scaling the problem internally before it is solved. For user-scaling, the parameters come from the NLP. If you are using AMPL, they can be specified through suffixes ("scaling_factor") The default value for this string option is "gradient-based".
Possible values:

- none: no problem scaling will be performed
- user-scaling: scaling parameters will come from the user
- gradient-based: scale the problem so the maximum gradient at the starting point is scaling_max_gradient

- equilibration-based: scale the problem so that first derivatives are of order 1 at random points (only available with MC19)

nlp_scaling_max_gradient: Maximum gradient after NLP scaling.
This is the gradient scaling cut-off. If the maximum gradient is above this value, then gradient based scaling will be performed. Scaling parameters are calculated to scale the maximum gradient back to this value. (This is g_max in Section 3.8 of the implementation paper.) Note: This option is only used if "nlp_scaling_method" is chosen as "gradient-based". The valid range for this real option is $0 <$ `nlp_scaling_max_gradient` $< +$`inf` and its default value is 100.

nlp_scaling_min_value: Minimum value of gradient-based scaling values.
This is the lower bound for the scaling factors computed by gradient-based scaling method. If some derivatives of some functions are huge, the scaling factors will otherwise become very small, and the (unscaled) final constraint violation, for example, might then be significant. Note: This option is only used if "nlp_scaling_method" is chosen as "gradient-based". The valid range for this real option is $0 \leq$ `nlp_scaling_min_value` $< +$`inf` and its default value is $1 \cdot 10^{-08}$.

C.4 NLP

bound_relax_factor: Factor for initial relaxation of the bounds.
Before start of the optimization, the bounds given by the user are relaxed. This option sets the factor for this relaxation. If it is set to zero, then then bounds relaxation is disabled. (See Eqn.(35) in implementation paper.) The valid range for this real option is $0 \leq$ `bound_relax_factor` $< +$`inf` and its default value is $1 \cdot 10^{-08}$.

honor_original_bounds: Indicates whether final points should be projected into original bounds.
Ipopt might relax the bounds during the optimization (see, e.g., option "bound_relax_factor"). This option determines whether the final point should be projected back into the user-provide original bounds after the optimization. The default value for this string option is "yes".
Possible values:

- no: Leave final point unchanged
- yes: Project final point back into original bounds

check_derivatives_for_naninf: Indicates whether it is desired to check for Nan/Inf in derivative matrices
Activating this option will cause an error if an invalid number is detected in the constraint Jacobians or the Lagrangian Hessian. If this is not activated, the test is skipped, and the algorithm might proceed with invalid numbers and fail. If test is activated and an invalid number is detected, the matrix is written to output with print_level corresponding to J_MORE_DETAILED; so beware of large output! The default value for this string option is "no".
Possible values:

- no: Don't check (faster).
- yes: Check Jacobians and Hessian for Nan and Inf.

nlp_lower_bound_inf: any bound less or equal this value will be considered -inf (i.e. not lower bounded).
The valid range for this real option is $-$`inf` $<$ `nlp_lower_bound_inf` $< +$`inf` and its default value is $-1 \cdot 10^{+19}$.

nlp_upper_bound_inf: any bound greater or this value will be considered +inf (i.e. not upper bounded). The valid range for this real option is $-\texttt{inf} < \texttt{nlp\_upper\_bound\_inf} < +\texttt{inf}$ and its default value is $1 \cdot 10^{+19}$.

fixed_variable_treatment: Determines how fixed variables should be handled.
The main difference between those options is that the starting point in the "make_constraint" case still has the fixed variables at their given values, whereas in the case "make_parameter" the functions are always evaluated with the fixed values for those variables. Also, for "relax_bounds", the fixing bound constraints are relaxed (according to" bound_relax_factor"). For both "make_constraints" and "relax_bounds", bound multipliers are computed for the fixed variables. The default value for this string option is "make_parameter".
Possible values:

- make_parameter: Remove fixed variable from optimization variables
- make_constraint: Add equality constraints fixing variables
- relax_bounds: Relax fixing bound constraints

jac_c_constant: Indicates whether all equality constraints are linear
Activating this option will cause Ipopt to ask for the Jacobian of the equality constraints only once from the NLP and reuse this information later. The default value for this string option is "no".
Possible values:

- no: Don't assume that all equality constraints are linear
- yes: Assume that equality constraints Jacobian are constant

jac_d_constant: Indicates whether all inequality constraints are linear
Activating this option will cause Ipopt to ask for the Jacobian of the inequality constraints only once from the NLP and reuse this information later. The default value for this string option is "no".
Possible values:

- no: Don't assume that all inequality constraints are linear
- yes: Assume that equality constraints Jacobian are constant

hessian_constant: Indicates whether the problem is a quadratic problem
Activating this option will cause Ipopt to ask for the Hessian of the Lagrangian function only once from the NLP and reuse this information later. The default value for this string option is "no".
Possible values:

- no: Assume that Hessian changes
- yes: Assume that Hessian is constant

C.5 Initialization

bound_frac: Desired minimum relative distance from the initial point to bound.
Determines how much the initial point might have to be modified in order to be sufficiently inside the bounds (together with "bound_push"). (This is kappa_2 in Section 3.6 of implementation paper.) The valid range for this real option is $0 < \texttt{bound\_frac} \leq 0.5$ and its default value is 0.01.

bound_push: Desired minimum absolute distance from the initial point to bound.
Determines how much the initial point might have to be modified in order to be sufficiently inside the bounds (together with "bound_frac"). (This is kappa_1 in Section 3.6 of implementation paper.) The valid range for this real option is $0 <$ `bound_push` $< +$`inf` and its default value is 0.01.

slack_bound_frac: Desired minimum relative distance from the initial slack to bound.
Determines how much the initial slack variables might have to be modified in order to be sufficiently inside the inequality bounds (together with "slack_bound_push"). (This is kappa_2 in Section 3.6 of implementation paper.) The valid range for this real option is $0 <$ `slack_bound_frac` ≤ 0.5 and its default value is 0.01.

slack_bound_push: Desired minimum absolute distance from the initial slack to bound.
Determines how much the initial slack variables might have to be modified in order to be sufficiently inside the inequality bounds (together with "slack_bound_frac"). (This is kappa_1 in Section 3.6 of implementation paper.) The valid range for this real option is $0 <$ `slack_bound_push` $< +$`inf` and its default value is 0.01.

bound_mult_init_val: Initial value for the bound multipliers.
All dual variables corresponding to bound constraints are initialized to this value. The valid range for this real option is $0 <$ `bound_mult_init_val` $< +$`inf` and its default value is 1.

constr_mult_init_max: Maximum allowed least-square guess of constraint multipliers.
Determines how large the initial least-square guesses of the constraint multipliers are allowed to be (in max-norm). If the guess is larger than this value, it is discarded and all constraint multipliers are set to zero. This options is also used when initializing the restoration phase. By default, "resto.constr_mult_init_max" (the one used in RestoIterateInitializer) is set to zero. The valid range for this real option is $0 \leq$ `constr_mult_init_max` $< +$`inf` and its default value is 1000.

bound_mult_init_method: Initialization method for bound multipliers
This option defines how the iterates for the bound multipliers are initialized. If "constant" is chosen, then all bound multipliers are initialized to the value of "bound_mult_init_val". If "mu-based" is chosen, the each value is initialized to the the value of "mu_init" divided by the corresponding slack variable. This latter option might be useful if the starting point is close to the optimal solution. The default value for this string option is "constant".
Possible values:

- constant: set all bound multipliers to the value of bound_mult_init_val
- mu-based: initialize to mu_init/x_slack

C.6 Barrier Parameter

mehrotra_algorithm: Indicates if we want to do Mehrotra's algorithm.
If set to yes, Ipopt runs as Mehrotra's predictor-corrector algorithm. This works usually very well for LPs and convex QPs. This automatically disables the line search, and chooses the (unglobalized) adaptive mu strategy with the "probing" oracle, and uses "corrector_type=affine" without any safeguards; you should not set any of those options explicitly in addition. Also, unless otherwise specified, the values of "bound_push", "bound_frac", and "bound_mult_init_val" are set more aggressive, and sets "alpha_for_y=bound_mult". The default value for this string option is "no".
Possible values:

- no: Do the usual Ipopt algorithm.
- yes: Do Mehrotra's predictor-corrector algorithm.

mu_strategy: Update strategy for barrier parameter.
Determines which barrier parameter update strategy is to be used. The default value for this string option is "monotone".
Possible values:

- monotone: use the monotone (Fiacco-McCormick) strategy
- adaptive: use the adaptive update strategy

mu_oracle: Oracle for a new barrier parameter in the adaptive strategy.
Determines how a new barrier parameter is computed in each "free-mode" iteration of the adaptive barrier parameter strategy. (Only considered if "adaptive" is selected for option "mu_strategy"). The default value for this string option is "quality-function".
Possible values:

- probing: Mehrotra's probing heuristic
- loqo: LOQO's centrality rule
- quality-function: minimize a quality function

quality_function_max_section_steps: Maximum number of search steps during direct search procedure determining the optimal centering parameter.
The golden section search is performed for the quality function based mu oracle. (Only used if option "mu_oracle" is set to "quality-function".) The valid range for this integer option is $0 \leq$ `quality_function_max_section_step` $+$`inf` and its default value is 8.

fixed_mu_oracle: Oracle for the barrier parameter when switching to fixed mode.
Determines how the first value of the barrier parameter should be computed when switching to the "monotone mode" in the adaptive strategy. (Only considered if "adaptive" is selected for option "mu_strategy".) The default value for this string option is "average_compl".
Possible values:

- probing: Mehrotra's probing heuristic
- loqo: LOQO's centrality rule
- quality-function: minimize a quality function
- average_compl: base on current average complementarity

adaptive_mu_globalization: Globalization strategy for the adaptive mu selection mode.
To achieve global convergence of the adaptive version, the algorithm has to switch to the monotone mode (Fiacco-McCormick approach) when convergence does not seem to appear. This option sets the criterion used to decide when to do this switch. (Only used if option "mu_strategy" is chosen as "adaptive".) The default value for this string option is "obj-constr-filter".
Possible values:

- kkt-error: nonmonotone decrease of kkt-error
- obj-constr-filter: 2-dim filter for objective and constraint violation
- never-monotone-mode: disables globalization

mu_init: Initial value for the barrier parameter.
This option determines the initial value for the barrier parameter (mu). It is only relevant in the monotone, Fiacco-McCormick version of the algorithm. (i.e., if "mu_strategy" is chosen as "monotone") The valid range for this real option is $0 <$ `mu_init` $< +$`inf` and its default value is 0.1.

mu_max_fact: Factor for initialization of maximum value for barrier parameter.
This option determines the upper bound on the barrier parameter. This upper bound is computed as the average complementarity at the initial point times the value of this option. (Only used if option "mu_strategy" is chosen as "adaptive".) The valid range for this real option is $0 <$ `mu_max_fact` $< +$`inf` and its default value is 1000.

mu_max: Maximum value for barrier parameter.
This option specifies an upper bound on the barrier parameter in the adaptive mu selection mode. If this option is set, it overwrites the effect of mu_max_fact. (Only used if option "mu_strategy" is chosen as "adaptive".) The valid range for this real option is $0 <$ `mu_max` $< +$`inf` and its default value is 100000.

mu_min: Minimum value for barrier parameter.
This option specifies the lower bound on the barrier parameter in the adaptive mu selection mode. By default, it is set to the minimum of 1e-11 and min("tol","compl_inf_tol")/("barrier_tol_factor"+1), which should be a reasonable value. (Only used if option "mu_strategy" is chosen as "adaptive".) The valid range for this real option is $0 <$ `mu_min` $< +$`inf` and its default value is $1 \cdot 10^{-11}$.

mu_target: Desired value of complementarity.
Usually, the barrier parameter is driven to zero and the termination test for complementarity is measured with respect to zero complementarity. However, in some cases it might be desired to have Ipopt solve barrier problem for strictly positive value of the barrier parameter. In this case, the value of "mu_target" specifies the final value of the barrier parameter, and the termination tests are then defined with respect to the barrier problem for this value of the barrier parameter. The valid range for this real option is $0 \leq$ `mu_target` $< +$`inf` and its default value is 0.

barrier_tol_factor: Factor for mu in barrier stop test.
The convergence tolerance for each barrier problem in the monotone mode is the value of the barrier parameter times "barrier_tol_factor". This option is also used in the adaptive mu strategy during the monotone mode. (This is kappa_epsilon in implementation paper). The valid range for this real option is $0 <$ `barrier_tol_factor` $< +$`inf` and its default value is 10.

mu_linear_decrease_factor: Determines linear decrease rate of barrier parameter.
For the Fiacco-McCormick update procedure the new barrier parameter mu is obtained by taking the minimum of mu*"mu_linear_decrease_factor" and mu$\hat{}$"superlinear_decrease_power". (This is kappa_mu in implementation paper.) This option is also used in the adaptive mu strategy during the monotone mode. The valid range for this real option is $0 <$ `mu_linear_decrease_factor` < 1 and its default value is 0.2.

mu_superlinear_decrease_power: Determines superlinear decrease rate of barrier parameter.
For the Fiacco-McCormick update procedure the new barrier parameter mu is obtained by taking the minimum of mu*"mu_linear_decrease_factor" and mu$\hat{}$"superlinear_decrease_power". (This is theta_mu in implementation paper.) This option is also used in the adaptive mu strategy during the monotone mode. The valid range for this real option is $1 <$ `mu_superlinear_decrease_power` < 2 and its default value is 1.5.

C.7 Multiplier Updates

alpha_for_y: Method to determine the step size for constraint multipliers.
This option determines how the step size (alpha_y) will be calculated when updating the constraint multipliers. The default value for this string option is "primal".
Possible values:

- primal: use primal step size
- bound-mult: use step size for the bound multipliers (good for LPs)
- min: use the min of primal and bound multipliers
- max: use the max of primal and bound multipliers
- full: take a full step of size one
- min-dual-infeas: choose step size minimizing new dual infeasibility
- safer-min-dual-infeas: like "min_dual_infeas", but safeguarded by "min" and "max"
- primal-and-full: use the primal step size, and full step if delta_x ¡= alpha_for_y_tol
- dual-and-full: use the dual step size, and full step if delta_x ¡= alpha_for_y_tol
- acceptor: Call LSAcceptor to get step size for y

alpha_for_y_tol: Tolerance for switching to full equality multiplier steps.
This is only relevant if "alpha_for_y" is chosen "primal-and-full" or "dual-and-full". The step size for the equality constraint multipliers is taken to be one if the max-norm of the primal step is less than this tolerance. The valid range for this real option is $0 \leq$ `alpha_for_y_tol` $<$ `+inf` and its default value is 10.

recalc_y: Tells the algorithm to recalculate the equality and inequality multipliers as least square estimates.
This asks the algorithm to recompute the multipliers, whenever the current infeasibility is less than recalc_y_feas_tol. Choosing yes might be helpful in the quasi-Newton option. However, each recalculation requires an extra factorization of the linear system. If a limited memory quasi-Newton option is chosen, this is used by default. The default value for this string option is "no".
Possible values:

- no: use the Newton step to update the multipliers
- yes: use least-square multiplier estimates

recalc_y_feas_tol: Feasibility threshold for recomputation of multipliers.
If recalc_y is chosen and the current infeasibility is less than this value, then the multipliers are recomputed. The valid range for this real option is $0 <$ `recalc_y_feas_tol` $<$ `+inf` and its default value is $1 \cdot 10^{-06}$.

C.8 Line Search

max_soc: Maximum number of second order correction trial steps at each iteration.
Choosing 0 disables the second order corrections. (This is p$\hat{\text{m}}$ax of Step A-5.9 of Algorithm A in the implementation paper.) The valid range for this integer option is $0 \leq$ `max_soc` $<$ `+inf` and its default value is 4.

watchdog_shortened_iter_trigger: Number of shortened iterations that trigger the watchdog.
If the number of successive iterations in which the backtracking line search did not accept the first trial point exceeds this number, the watchdog procedure is activated. Choosing "0" here disables the watchdog procedure. The valid range for this integer option is $0 \leq$ `watchdog_shortened_iter_trigger` $<$ `+inf` and its default value is 10.

watchdog_trial_iter_max: Maximum number of watchdog iterations.
This option determines the number of trial iterations allowed before the watchdog procedure is aborted and the algorithm returns to the stored point. The valid range for this integer option is $1 \leq$ `watchdog_trial_iter_max` $<$ `+inf` and its default value is 3.

accept_every_trial_step: Always accept the first trial step.
Setting this option to "yes" essentially disables the line search and makes the algorithm take aggressive steps, without global convergence guarantees. The default value for this string option is "no".
Possible values:

- no: don't arbitrarily accept the full step
- yes: always accept the full step

corrector_type: The type of corrector steps that should be taken (unsupported!).
If "mu_strategy" is "adaptive", this option determines what kind of corrector steps should be tried. The default value for this string option is "none".
Possible values:

- none: no corrector
- affine: corrector step towards mu=0
- primal-dual: corrector step towards current mu

C.9 Warm Start

warm_start_init_point: Warm-start for initial point
Indicates whether this optimization should use a warm start initialization, where values of primal and dual variables are given (e.g., from a previous optimization of a related problem.) The default value for this string option is "no".
Possible values:

- no: do not use the warm start initialization
- yes: use the warm start initialization

warm_start_bound_push: same as bound_push for the regular initializer.
The valid range for this real option is $0 <$ `warm_start_bound_push` $<$ `+inf` and its default value is 0.001.

warm_start_bound_frac: same as bound_frac for the regular initializer.
The valid range for this real option is $0 <$ `warm_start_bound_frac` ≤ 0.5 and its default value is 0.001.

warm_start_slack_bound_frac: same as slack_bound_frac for the regular initializer.
The valid range for this real option is $0 <$ `warm_start_slack_bound_frac` ≤ 0.5 and its default value is 0.001.

warm_start_slack_bound_push: same as slack_bound_push for the regular initializer.
The valid range for this real option is $0 <$ `warm_start_slack_bound_push` $<$ `+inf` and its default value is 0.001.

warm_start_mult_bound_push: same as mult_bound_push for the regular initializer.
The valid range for this real option is $0 <$ `warm_start_mult_bound_push` $<$ `+inf` and its default value is 0.001.

warm_start_mult_init_max: Maximum initial value for the equality multipliers.
The valid range for this real option is $-$`inf` $<$ `warm_start_mult_init_max` $<$ `+inf` and its default value is $1 \cdot 10^{+06}$.

C.10 Restoration Phase

expect_infeasible_problem: Enable heuristics to quickly detect an infeasible problem.
This options is meant to activate heuristics that may speed up the infeasibility determination if you expect that there is a good chance for the problem to be infeasible. In the filter line search procedure, the restoration phase is called more quickly than usually, and more reduction in the constraint violation is enforced before the restoration phase is left. If the problem is square, this option is enabled automatically. The default value for this string option is "no".
Possible values:

- no: the problem probably be feasible
- yes: the problem has a good chance to be infeasible

expect_infeasible_problem_ctol: Threshold for disabling "expect_infeasible_problem" option.
If the constraint violation becomes smaller than this threshold, the "expect_infeasible_problem" heuristics in the filter line search are disabled. If the problem is square, this options is set to 0. The valid range for this real option is $0 \leq$ `expect_infeasible_problem_ctol` $<$ `+inf` and its default value is 0.001.

expect_infeasible_problem_ytol: Multiplier threshold for activating "expect_infeasible_problem" option.
If the max norm of the constraint multipliers becomes larger than this value and "expect_infeasible_problem" is chosen, then the restoration phase is entered. The valid range for this real option is $0 <$ `expect_infeasible_problem_ytol` $<$ `+inf` and its default value is $1 \cdot 10^{+08}$.

start_with_resto: Tells algorithm to switch to restoration phase in first iteration.
Setting this option to "yes" forces the algorithm to switch to the feasibility restoration phase in the first iteration. If the initial point is feasible, the algorithm will abort with a failure. The default value for this string option is "no".
Possible values:

- no: don't force start in restoration phase
- yes: force start in restoration phase

soft_resto_pderror_reduction_factor: Required reduction in primal-dual error in the soft restoration phase.
The soft restoration phase attempts to reduce the primal-dual error with regular steps. If the damped primal-dual step (damped only to satisfy the fraction-to-the-boundary rule) is not decreasing the primal-dual error by at least this factor, then the regular restoration phase is called. Choosing "0" here disables the soft restoration phase. The valid range for this real option is $0 \leq$ `soft_resto_pderror_reduction_factor` $<$ `+inf` and its default value is 0.9999.

required_infeasibility_reduction: Required reduction of infeasibility before leaving restoration phase. The restoration phase algorithm is performed, until a point is found that is acceptable to the filter and the infeasibility has been reduced by at least the fraction given by this option. The valid range for this real option is $0 \leq$ `required_infeasibility_reduction` < 1 and its default value is 0.9.

bound_mult_reset_threshold: Threshold for resetting bound multipliers after the restoration phase. After returning from the restoration phase, the bound multipliers are updated with a Newton step for complementarity. Here, the change in the primal variables during the entire restoration phase is taken to be the corresponding primal Newton step. However, if after the update the largest bound multiplier exceeds the threshold specified by this option, the multipliers are all reset to 1. The valid range for this real option is $0 \leq$ `bound_mult_reset_threshold` $< +$`inf` and its default value is 1000.

constr_mult_reset_threshold: Threshold for resetting equality and inequality multipliers after restoration phase.
After returning from the restoration phase, the constraint multipliers are recomputed by a least square estimate. This option triggers when those least-square estimates should be ignored. The valid range for this real option is $0 \leq$ `constr_mult_reset_threshold` $< +$`inf` and its default value is 0.

evaluate_orig_obj_at_resto_trial: Determines if the original objective function should be evaluated at restoration phase trial points.
Setting this option to "yes" makes the restoration phase algorithm evaluate the objective function of the original problem at every trial point encountered during the restoration phase, even if this value is not required. In this way, it is guaranteed that the original objective function can be evaluated without error at all accepted iterates; otherwise the algorithm might fail at a point where the restoration phase accepts an iterate that is good for the restoration phase problem, but not the original problem. On the other hand, if the evaluation of the original objective is expensive, this might be costly. The default value for this string option is "yes".
Possible values:

- no: skip evaluation
- yes: evaluate at every trial point

C.11 Linear Solver

linear_solver: Linear solver used for step computations.
Determines which linear algebra package is to be used for the solution of the augmented linear system (for obtaining the search directions). Note, the code must have been compiled with the linear solver you want to choose. Depending on your Ipopt installation, not all options are available. The default value for this string option is "ma27".
Possible values:

- ma27: use the Harwell routine MA27
- ma57: use the Harwell routine MA57
- ma77: use the Harwell routine HSL_MA77
- ma86: use the Harwell routine HSL_MA86
- ma97: use the Harwell routine HSL_MA97
- pardiso: use the Pardiso package
- wsmp: use WSMP package

- mumps: use MUMPS package
- custom: use custom linear solver

linear_system_scaling: Method for scaling the linear system.
Determines the method used to compute symmetric scaling factors for the augmented system (see also the "linear_scaling_on_demand" option). This scaling is independent of the NLP problem scaling. By default, MC19 is only used if MA27 or MA57 are selected as linear solvers. This value is only available if Ipopt has been compiled with MC19. The default value for this string option is "mc19".
Possible values:

- none: no scaling will be performed
- mc19: use the Harwell routine MC19
- slack-based: use the slack values

linear_scaling_on_demand: Flag indicating that linear scaling is only done if it seems required.
This option is only important if a linear scaling method (e.g., mc19) is used. If you choose "no", then the scaling factors are computed for every linear system from the start. This can be quite expensive. Choosing "yes" means that the algorithm will start the scaling method only when the solutions to the linear system seem not good, and then use it until the end. The default value for this string option is "yes".
Possible values:

- no: Always scale the linear system.
- yes: Start using linear system scaling if solutions seem not good.

max_refinement_steps: Maximum number of iterative refinement steps per linear system solve.
Iterative refinement (on the full unsymmetric system) is performed for each right hand side. This option determines the maximum number of iterative refinement steps. The valid range for this integer option is $0 \leq$ `max_refinement_steps` $< +$`inf` and its default value is 10.

min_refinement_steps: Minimum number of iterative refinement steps per linear system solve.
Iterative refinement (on the full unsymmetric system) is performed for each right hand side. This option determines the minimum number of iterative refinements (i.e. at least "min_refinement_steps" iterative refinement steps are enforced per right hand side.) The valid range for this integer option is $0 \leq$ `min_refinement_steps` $< +$`inf` and its default value is 1.

C.12 Hessian Perturbation

max_hessian_perturbation: Maximum value of regularization parameter for handling negative curvature.
In order to guarantee that the search directions are indeed proper descent directions, Ipopt requires that the inertia of the (augmented) linear system for the step computation has the correct number of negative and positive eigenvalues. The idea is that this guides the algorithm away from maximizers and makes Ipopt more likely converge to first order optimal points that are minimizers. If the inertia is not correct, a multiple of the identity matrix is added to the Hessian of the Lagrangian in the augmented system. This parameter gives the maximum value of the regularization parameter. If a regularization of that size is not enough, the algorithm skips this iteration and goes to the restoration phase. (This is delta_wmax in the implementation paper.) The valid range for this real option is $0 <$ `max_hessian_perturbation` $< +$`inf` and its default value is $1 \cdot 10^{+20}$.

min_hessian_perturbation: Smallest perturbation of the Hessian block.
The size of the perturbation of the Hessian block is never selected smaller than this value, unless no perturbation is necessary. (This is delta_wmin in implementation paper.) The valid range for this real option is $0 \leq$ `min_hessian_perturbation` $<$ `+inf` and its default value is $1 \cdot 10^{-20}$.

first_hessian_perturbation: Size of first x-s perturbation tried.
The first value tried for the x-s perturbation in the inertia correction scheme.(This is delta_0 in the implementation paper.) The valid range for this real option is $0 <$ `first_hessian_perturbation` $<$ `+inf` and its default value is 0.0001.

perturb_inc_fact_first: Increase factor for x-s perturbation for very first perturbation.
The factor by which the perturbation is increased when a trial value was not sufficient - this value is used for the computation of the very first perturbation and allows a different value for for the first perturbation than that used for the remaining perturbations. (This is bar_kappa_w+ in the implementation paper.) The valid range for this real option is $1 <$ `perturb_inc_fact_first` $<$ `+inf` and its default value is 100.

perturb_inc_fact: Increase factor for x-s perturbation.
The factor by which the perturbation is increased when a trial value was not sufficient - this value is used for the computation of all perturbations except for the first. (This is kappa_w+ in the implementation paper.) The valid range for this real option is $1 <$ `perturb_inc_fact` $<$ `+inf` and its default value is 8.

perturb_dec_fact: Decrease factor for x-s perturbation.
The factor by which the perturbation is decreased when a trial value is deduced from the size of the most recent successful perturbation. (This is kappa_w- in the implementation paper.) The valid range for this real option is $0 <$ `perturb_dec_fact` < 1 and its default value is 0.333333.

jacobian_regularization_value: Size of the regularization for rank-deficient constraint Jacobians.
(This is bar delta_c in the implementation paper.) The valid range for this real option is $0 \leq$ `jacobian_regularization_valu`
`+inf` and its default value is $1 \cdot 10^{-08}$.

C.13 Quasi-Newton

hessian_approximation: Indicates what Hessian information is to be used.
This determines which kind of information for the Hessian of the Lagrangian function is used by the algorithm. The default value for this string option is "exact".
Possible values:

- exact: Use second derivatives provided by the NLP.
- limited-memory: Perform a limited-memory quasi-Newton approximation

limited_memory_update_type: Quasi-Newton update formula for the limited memory approximation.
Determines which update formula is to be used for the limited-memory quasi-Newton approximation. The default value for this string option is "bfgs".
Possible values:

- bfgs: BFGS update (with skipping)
- sr1: SR1 (not working well)

limited_memory_max_history: Maximum size of the history for the limited quasi-Newton Hessian approximation.
This option determines the number of most recent iterations that are taken into account for the limited-memory quasi-Newton approximation. The valid range for this integer option is $0 \leq$ `limited_memory_max_history` $<$ $+$`inf` and its default value is 6.

limited_memory_max_skipping: Threshold for successive iterations where update is skipped.
If the update is skipped more than this number of successive iterations, we quasi-Newton approximation is reset. The valid range for this integer option is $1 \leq$ `limited_memory_max_skipping` $<$ $+$`inf` and its default value is 2.

limited_memory_initialization: Initialization strategy for the limited memory quasi-Newton approximation.
Determines how the diagonal Matrix B_0 as the first term in the limited memory approximation should be computed. The default value for this string option is "scalar1".
Possible values:

- scalar1: sigma = $s\hat{T}y/s\hat{T}s$
- scalar2: sigma = $y\hat{T}y/s\hat{T}y$
- scalar3: arithmetic average of scalar1 and scalar2
- scalar4: geometric average of scalar1 and scalar2
- constant: sigma = limited_memory_init_val

limited_memory_init_val: Value for B0 in low-rank update.
The starting matrix in the low rank update, B0, is chosen to be this multiple of the identity in the first iteration (when no updates have been performed yet), and is constantly chosen as this value, if "limited_memory_initialization" is "constant". The valid range for this real option is $0 <$ `limited_memory_init_val` $<$ $+$`inf` and its default value is 1.

limited_memory_init_val_max: Upper bound on value for B0 in low-rank update.
The starting matrix in the low rank update, B0, is chosen to be this multiple of the identity in the first iteration (when no updates have been performed yet), and is constantly chosen as this value, if "limited_memory_initialization" is "constant". The valid range for this real option is $0 <$ `limited_memory_init_val_max` $<$ $+$`inf` and its default value is $1 \cdot 10^{+08}$.

limited_memory_init_val_min: Lower bound on value for B0 in low-rank update.
The starting matrix in the low rank update, B0, is chosen to be this multiple of the identity in the first iteration (when no updates have been performed yet), and is constantly chosen as this value, if "limited_memory_initialization" is "constant". The valid range for this real option is $0 <$ `limited_memory_init_val_min` $<$ $+$`inf` and its default value is $1 \cdot 10^{-08}$.

limited_memory_special_for_resto: Determines if the quasi-Newton updates should be special during the restoration phase.
Until Nov 2010, Ipopt used a special update during the restoration phase, but it turned out that this does not work well. The new default uses the regular update procedure and it improves results. If for some reason you want to get back to the original update, set this option to "yes". The default value for this string option is "no".
Possible values:

- no: use the same update as in regular iterations

- yes: use the a special update during restoration phase

C.14 Derivative Test

derivative_test: Enable derivative checker
If this option is enabled, a (slow!) derivative test will be performed before the optimization. The test is performed at the user provided starting point and marks derivative values that seem suspicious The default value for this string option is "none".
Possible values:

- none: do not perform derivative test
- first-order: perform test of first derivatives at starting point
- second-order: perform test of first and second derivatives at starting point
- only-second-order: perform test of second derivatives at starting point

derivative_test_perturbation: Size of the finite difference perturbation in derivative test.
This determines the relative perturbation of the variable entries. The valid range for this real option is $0 <$ `derivative_test_perturbation` $<$ `+inf` and its default value is $1 \cdot 10^{-08}$.

derivative_test_tol: Threshold for indicating wrong derivative.
If the relative deviation of the estimated derivative from the given one is larger than this value, the corresponding derivative is marked as wrong. The valid range for this real option is $0 <$ `derivative_test_tol` $<$ `+inf` and its default value is 0.0001.

derivative_test_print_all: Indicates whether information for all estimated derivatives should be printed.
Determines verbosity of derivative checker. The default value for this string option is "no".
Possible values:

- no: Print only suspect derivatives
- yes: Print all derivatives

derivative_test_first_index: Index of first quantity to be checked by derivative checker
If this is set to -2, then all derivatives are checked. Otherwise, for the first derivative test it specifies the first variable for which the test is done (counting starts at 0). For second derivatives, it specifies the first constraint for which the test is done; counting of constraint indices starts at 0, and -1 refers to the objective function Hessian. The valid range for this integer option is $-2 \leq$ `derivative_test_first_index` $<$ `+inf` and its default value is -2.

point_perturbation_radius: Maximal perturbation of an evaluation point.
If a random perturbation of a points is required, this number indicates the maximal perturbation. This is for example used when determining the center point at which the finite difference derivative test is executed. The valid range for this real option is $0 \leq$ `point_perturbation_radius` $<$ `+inf` and its default value is 10.

C.15 MA27 Linear Solver

ma27_pivtol: Pivot tolerance for the linear solver MA27.
A smaller number pivots for sparsity, a larger number pivots for stability. This option is only available if Ipopt has been compiled with MA27. The valid range for this real option is $0 <$ `ma27_pivtol` < 1 and its default value is $1 \cdot 10^{-08}$.

ma27_pivtolmax: Maximum pivot tolerance for the linear solver MA27.
Ipopt may increase pivtol as high as pivtolmax to get a more accurate solution to the linear system. This option is only available if Ipopt has been compiled with MA27. The valid range for this real option is $0 <$ `ma27_pivtolmax` < 1 and its default value is 0.0001.

ma27_liw_init_factor: Integer workspace memory for MA27.
The initial integer workspace memory = liw_init_factor * memory required by unfactored system. Ipopt will increase the workspace size by meminc_factor if required. This option is only available if Ipopt has been compiled with MA27. The valid range for this real option is $1 \leq$ `ma27_liw_init_factor` $<$ `+inf` and its default value is 5.

ma27_la_init_factor: Real workspace memory for MA27.
The initial real workspace memory = la_init_factor * memory required by unfactored system. Ipopt will increase the workspace size by meminc_factor if required. This option is only available if Ipopt has been compiled with MA27. The valid range for this real option is $1 \leq$ `ma27_la_init_factor` $<$ `+inf` and its default value is 5.

ma27_meminc_factor: Increment factor for workspace size for MA27.
If the integer or real workspace is not large enough, Ipopt will increase its size by this factor. This option is only available if Ipopt has been compiled with MA27. The valid range for this real option is $1 \leq$ `ma27_meminc_factor` $<$ `+inf` and its default value is 2.

C.16 MA57 Linear Solver

ma57_pivtol: Pivot tolerance for the linear solver MA57.
A smaller number pivots for sparsity, a larger number pivots for stability. This option is only available if Ipopt has been compiled with MA57. The valid range for this real option is $0 <$ `ma57_pivtol` < 1 and its default value is $1 \cdot 10^{-08}$.

ma57_pivtolmax: Maximum pivot tolerance for the linear solver MA57.
Ipopt may increase pivtol as high as ma57_pivtolmax to get a more accurate solution to the linear system. This option is only available if Ipopt has been compiled with MA57. The valid range for this real option is $0 <$ `ma57_pivtolmax` < 1 and its default value is 0.0001.

ma57_pre_alloc: Safety factor for work space memory allocation for the linear solver MA57.
If 1 is chosen, the suggested amount of work space is used. However, choosing a larger number might avoid reallocation if the suggest values do not suffice. This option is only available if Ipopt has been compiled with MA57. The valid range for this real option is $1 \leq$ `ma57_pre_alloc` $<$ `+inf` and its default value is 1.05.

ma57_pivot_order: Controls pivot order in MA57
This is ICNTL(6) in MA57. The valid range for this integer option is $0 \leq$ `ma57_pivot_order` ≤ 5 and its default value is 5.

ma57_automatic_scaling: Controls MA57 automatic scaling
This option controls the internal scaling option of MA57. For higher reliability of the MA57 solver, you may want to set this option to yes. This is ICNTL(15) in MA57. The default value for this string option is "no".
Possible values:

- no: Do not scale the linear system matrix
- yes: Scale the linear system matrix

ma57_block_size: Controls block size used by Level 3 BLAS in MA57BD
This is ICNTL(11) in MA57. The valid range for this integer option is $1 \leq$ `ma57_block_size` $<$ `+inf` and its default value is 16.

ma57_node_amalgamation: Node amalgamation parameter
This is ICNTL(12) in MA57. The valid range for this integer option is $1 \leq$ `ma57_node_amalgamation` $<$ `+inf` and its default value is 16.

ma57_small_pivot_flag: If set to 1, then when small entries defined by CNTL(2) are detected they are removed and the corresponding pivots placed at the end of the factorization. This can be particularly efficient if the matrix is highly rank deficient.
This is ICNTL(16) in MA57. The valid range for this integer option is $0 \leq$ `ma57_small_pivot_flag` ≤ 1 and its default value is 0.

C.17 MA77 Linear Solver

ma77_print_level: Debug printing level for the linear solver MA77
The valid range for this integer option is $-$`inf` $<$ `ma77_print_level` $<$ `+inf` and its default value is -1.

ma77_buffer_lpage: Number of scalars per MA77 buffer page
Number of scalars per an in-core buffer in the out-of-core solver MA77. Must be at most ma77_file_size. The valid range for this integer option is $1 \leq$ `ma77_buffer_lpage` $<$ `+inf` and its default value is 4096.

ma77_buffer_npage: Number of pages that make up MA77 buffer
Number of pages of size buffer_lpage that exist in-core for the out-of-core solver MA77. The valid range for this integer option is $1 \leq$ `ma77_buffer_npage` $<$ `+inf` and its default value is 1600.

ma77_file_size: Target size of each temporary file for MA77, scalars per type
MA77 uses many temporary files, this option controls the size of each one. It is measured in the number of entries (int or double), NOT bytes. The valid range for this integer option is $1 \leq$ `ma77_file_size` $<$ `+inf` and its default value is 2097152.

ma77_maxstore: Maximum storage size for MA77 in-core mode
If greater than zero, the maximum size of factors stored in core before out-of-core mode is invoked. The valid range for this integer option is $0 \leq$ `ma77_maxstore` $<$ `+inf` and its default value is 0.

ma77_nemin: Node Amalgamation parameter
Two nodes in elimination tree are merged if result has fewer than ma77_nemin variables. The valid range for this integer option is $1 \leq$ `ma77_nemin` $<$ `+inf` and its default value is 8.

ma77_order: Controls type of ordering used by HSL_MA77
This option controls ordering for the solver HSL_MA77. The default value for this string option is "metis". Possible values:

- amd: Use the HSL_MC68 approximate minimum degree algorithm
- metis: Use the MeTiS nested dissection algorithm (if available)

ma77_small: Zero Pivot Threshold
Any pivot less than ma77_small is treated as zero. The valid range for this real option is $0 \leq$ `ma77_small` $<$ `+inf` and its default value is $1 \cdot 10^{-20}$.

ma77_static: Static Pivoting Threshold
See MA77 documentation. Either ma77_static=0.0 or ma77_static¿ma77_small. ma77_static=0.0 disables static pivoting. The valid range for this real option is $0 \leq$ `ma77_static` $<$ `+inf` and its default value is 0.

ma77_u: Pivoting Threshold
See MA77 documentation. The valid range for this real option is $0 \leq$ `ma77_u` ≤ 0.5 and its default value is $1 \cdot 10^{-08}$.

ma77_umax: Maximum Pivoting Threshold
Maximum value to which u will be increased to improve quality. The valid range for this real option is $0 \leq$ `ma77_umax` ≤ 0.5 and its default value is 0.0001.

C.18 MA86 Linear Solver

ma86_print_level: Debug printing level for the linear solver MA86
The valid range for this integer option is $-$`inf` $<$ `ma86_print_level` $<$ `+inf` and its default value is -1.

ma86_nemin: Node Amalgamation parameter
Two nodes in elimination tree are merged if result has fewer than ma86_nemin variables. The valid range for this integer option is $1 \leq$ `ma86_nemin` $<$ `+inf` and its default value is 32.

ma86_order: Controls type of ordering used by HSL_MA86
This option controls ordering for the solver HSL_MA86. The default value for this string option is "auto". Possible values:

- auto: Try both AMD and MeTiS, pick best
- amd: Use the HSL_MC68 approximate minimum degree algorithm
- metis: Use the MeTiS nested dissection algorithm (if available)

ma86_scaling: Controls scaling of matrix
This option controls scaling for the solver HSL_MA86. The default value for this string option is "mc64". Possible values:

- none: Do not scale the linear system matrix
- mc64: Scale linear system matrix using MC64
- mc77: Scale linear system matrix using MC77 [1,3,0]

ma86_small: Zero Pivot Threshold
Any pivot less than ma86_small is treated as zero. The valid range for this real option is $0 \leq$ `ma86_small` $<$ `+inf` and its default value is $1 \cdot 10^{-20}$.

ma86_static: Static Pivoting Threshold
See MA86 documentation. Either ma86_static=0.0 or ma86_static¿ma86_small. ma86_static=0.0 disables static pivoting. The valid range for this real option is $0 \leq$ `ma86_static` $<$ `+inf` and its default value is 0.

ma86_u: Pivoting Threshold
See MA86 documentation. The valid range for this real option is $0 \leq$ `ma86_u` ≤ 0.5 and its default value is $1 \cdot 10^{-08}$.

ma86_umax: Maximum Pivoting Threshold
Maximum value to which u will be increased to improve quality. The valid range for this real option is $0 \leq$ `ma86_umax` ≤ 0.5 and its default value is 0.0001.

C.19 MA97 Linear Solver

ma97_print_level: Debug printing level for the linear solver MA97
The valid range for this integer option is $-$`inf` $<$ `ma97_print_level` $<$ $+$`inf` and its default value is 0.

ma97_nemin: Node Amalgamation parameter
Two nodes in elimination tree are merged if result has fewer than ma97_nemin variables. The valid range for this integer option is $1 \leq$ `ma97_nemin` $<$ $+$`inf` and its default value is 8.

ma97_order: Controls type of ordering used by HSL_MA97
The default value for this string option is "auto".
Possible values:

- auto: Use HSL_MA97 heuristic to guess best of AMD and METIS
- best: Try both AMD and MeTiS, pick best
- amd: Use the HSL_MC68 approximate minimum degree algorithm
- metis: Use the MeTiS nested dissection algorithm
- matched-auto: Use the HSL_MC80 matching with heuristic choice of AMD or METIS
- matched-metis: Use the HSL_MC80 matching based ordering with METIS
- matched-amd: Use the HSL_MC80 matching based ordering with AMD

ma97_scaling: Specifies strategy for scaling in HSL_MA97 linear solver
The default value for this string option is "dynamic".
Possible values:

- none: Do not scale the linear system matrix
- mc30: Scale all linear system matrices using MC30
- mc64: Scale all linear system matrices using MC64
- mc77: Scale all linear system matrices using MC77 [1,3,0]
- dynamic: Dynamically select scaling according to rules specified by ma97_scalingX and ma97_switchX options.

ma97_scaling1: First scaling.
If ma97_scaling=dynamic, this scaling is used according to the trigger ma97_switch1. If ma97_switch2 is triggered it is disabled. The default value for this string option is "mc64".
Possible values:

- none: No scaling
- mc30: Scale linear system matrix using MC30
- mc64: Scale linear system matrix using MC64
- mc77: Scale linear system matrix using MC77 [1,3,0]

ma97_scaling2: Second scaling.
If ma97_scaling=dynamic, this scaling is used according to the trigger ma97_switch2. If ma97_switch3 is triggered it is disabled. The default value for this string option is "mc64".
Possible values:

- none: No scaling
- mc30: Scale linear system matrix using MC30
- mc64: Scale linear system matrix using MC64
- mc77: Scale linear system matrix using MC77 [1,3,0]

ma97_scaling3: Third scaling.
If ma97_scaling=dynamic, this scaling is used according to the trigger ma97_switch3. The default value for this string option is "mc64".
Possible values:

- none: No scaling
- mc30: Scale linear system matrix using MC30
- mc64: Scale linear system matrix using MC64
- mc77: Scale linear system matrix using MC77 [1,3,0]

ma97_small: Zero Pivot Threshold
Any pivot less than ma97_small is treated as zero. The valid range for this real option is $0 \leq$ `ma97_small` $<$ `+inf` and its default value is $1 \cdot 10^{-20}$.

ma97_solve_blas3: Controls if blas2 or blas3 routines are used for solve
The default value for this string option is "no".
Possible values:

- no: Use BLAS2 (faster, some implementations bit incompatible)
- yes: Use BLAS3 (slower)

ma97_switch1: First switch, determine when ma97_scaling1 is enabled.
If ma97_scaling=dynamic, ma97_scaling1 is enabled according to this condition. If ma97_switch2 occurs this option is henceforth ignored. The default value for this string option is "od_hd_reuse".
Possible values:

- never: Scaling is never enabled.
- at_start: Scaling to be used from the very start.
- at_start_reuse: Scaling to be used on first iteration, then reused thereafter.
- on_demand: Scaling to be used after Ipopt request improved solution (i.e. iterative refinement has failed).
- on_demand_reuse: As on_demand, but reuse scaling from previous itr
- high_delay: Scaling to be used after more than 0.05*n delays are present
- high_delay_reuse: Scaling to be used only when previous itr created more that 0.05*n additional delays, otherwise reuse scaling from previous itr
- od_hd: Combination of on_demand and high_delay
- od_hd_reuse: Combination of on_demand_reuse and high_delay_reuse

ma97_switch2: Second switch, determine when ma97_scaling2 is enabled.
If ma97_scaling=dynamic, ma97_scaling2 is enabled according to this condition. If ma97_switch3 occurs this option is henceforth ignored. The default value for this string option is "never".
Possible values:

- never: Scaling is never enabled.
- at_start: Scaling to be used from the very start.
- at_start_reuse: Scaling to be used on first iteration, then reused thereafter.
- on_demand: Scaling to be used after Ipopt request improved solution (i.e. iterative refinement has failed).
- on_demand_reuse: As on_demand, but reuse scaling from previous itr
- high_delay: Scaling to be used after more than 0.05*n delays are present
- high_delay_reuse: Scaling to be used only when previous itr created more that 0.05*n additional delays, otherwise reuse scaling from previous itr
- od_hd: Combination of on_demand and high_delay
- od_hd_reuse: Combination of on_demand_reuse and high_delay_reuse

ma97_switch3: Third switch, determine when ma97_scaling3 is enabled.
If ma97_scaling=dynamic, ma97_scaling3 is enabled according to this condition. The default value for this string option is "never".
Possible values:

- never: Scaling is never enabled.
- at_start: Scaling to be used from the very start.
- at_start_reuse: Scaling to be used on first iteration, then reused thereafter.
- on_demand: Scaling to be used after Ipopt request improved solution (i.e. iterative refinement has failed).
- on_demand_reuse: As on_demand, but reuse scaling from previous itr
- high_delay: Scaling to be used after more than 0.05*n delays are present
- high_delay_reuse: Scaling to be used only when previous itr created more that 0.05*n additional delays, otherwise reuse scaling from previous itr
- od_hd: Combination of on_demand and high_delay
- od_hd_reuse: Combination of on_demand_reuse and high_delay_reuse

ma97_u: Pivoting Threshold
See MA97 documentation. The valid range for this real option is $0 \leq$ `ma97_u` ≤ 0.5 and its default value is $1 \cdot 10^{-08}$.

ma97_umax: Maximum Pivoting Threshold
See MA97 documentation. The valid range for this real option is $0 \leq$ `ma97_umax` ≤ 0.5 and its default value is 0.0001.

C.20 MUMPS Linear Solver

mumps_pivtol: Pivot tolerance for the linear solver MUMPS.
A smaller number pivots for sparsity, a larger number pivots for stability. This option is only available if Ipopt has been compiled with MUMPS. The valid range for this real option is $0 \leq$ `mumps_pivtol` ≤ 1 and its default value is $1 \cdot 10^{-06}$.

mumps_pivtolmax: Maximum pivot tolerance for the linear solver MUMPS.
Ipopt may increase pivtol as high as pivtolmax to get a more accurate solution to the linear system. This option is only available if Ipopt has been compiled with MUMPS. The valid range for this real option is $0 \leq$ `mumps_pivtolmax` ≤ 1 and its default value is 0.1.

mumps_mem_percent: Percentage increase in the estimated working space for MUMPS.
In MUMPS when significant extra fill-in is caused by numerical pivoting, larger values of mumps_mem_percent may help use the workspace more efficiently. On the other hand, if memory requirement are too large at the very beginning of the optimization, choosing a much smaller value for this option, such as 5, might reduce memory requirements. The valid range for this integer option is $0 \leq$ `mumps_mem_percent` $<$ `+inf` and its default value is 1000.

mumps_permuting_scaling: Controls permuting and scaling in MUMPS
This is ICNTL(6) in MUMPS. The valid range for this integer option is $0 \leq$ `mumps_permuting_scaling` $\leq$ 7 and its default value is 7.

mumps_pivot_order: Controls pivot order in MUMPS
This is ICNTL(7) in MUMPS. The valid range for this integer option is $0 \leq$ `mumps_pivot_order` ≤ 7 and its default value is 7.

mumps_scaling: Controls scaling in MUMPS
This is ICNTL(8) in MUMPS. The valid range for this integer option is $-2 \leq$ `mumps_scaling` ≤ 77 and its default value is 77.

C.21 Pardiso Linear Solver

pardiso_msglvl: Pardiso message level
This determines the amount of analysis output from the Pardiso solver. This is MSGLVL in the Pardiso manual. The valid range for this integer option is $0 \leq$ `pardiso_msglvl` $<$ `+inf` and its default value is 0.

pardiso_matching_strategy: Matching strategy to be used by Pardiso
This is IPAR(13) in Pardiso manual. This option is only available if Ipopt has been compiled with Pardiso. The default value for this string option is "complete+2x2".
Possible values:

- complete: Match complete (IPAR(13)=1)
- complete+2x2: Match complete+2x2 (IPAR(13)=2)
- constraints: Match constraints (IPAR(13)=3)

pardiso_out_of_core_power: Enables out-of-core variant of Pardiso
Setting this option to a positive integer k makes Pardiso work in the out-of-core variant where the factor is split in $2\hat{k}$ subdomains. This is IPARM(50) in the Pardiso manual. This option is only available if Ipopt has been compiled with Pardiso. The valid range for this integer option is $0 \leq$ `pardiso_out_of_core_power` $<$ `+inf` and its default value is 0.

C.22 WSMP Linear Solver

wsmp_num_threads: Number of threads to be used in WSMP
This determines on how many processors WSMP is running on. This option is only available if Ipopt has been compiled with WSMP. The valid range for this integer option is $-\texttt{inf} < \texttt{wsmp\_num\_threads} < +\texttt{inf}$ and its default value is 1.

wsmp_ordering_option: Determines how ordering is done in WSMP (IPARM(16)
This corresponds to the value of WSSMP's IPARM(16). This option is only available if Ipopt has been compiled with WSMP. The valid range for this integer option is $-2 \leq \texttt{wsmp\_ordering\_option} \leq 3$ and its default value is 1.

wsmp_pivtol: Pivot tolerance for the linear solver WSMP.
A smaller number pivots for sparsity, a larger number pivots for stability. This option is only available if Ipopt has been compiled with WSMP. The valid range for this real option is $0 < \texttt{wsmp\_pivtol} < 1$ and its default value is 0.0001.

wsmp_pivtolmax: Maximum pivot tolerance for the linear solver WSMP.
Ipopt may increase pivtol as high as pivtolmax to get a more accurate solution to the linear system. This option is only available if Ipopt has been compiled with WSMP. The valid range for this real option is $0 < \texttt{wsmp\_pivtolmax} < 1$ and its default value is 0.1.

wsmp_scaling: Determines how the matrix is scaled by WSMP.
This corresponds to the value of WSSMP's IPARM(10). This option is only available if Ipopt has been compiled with WSMP. The valid range for this integer option is $0 \leq \texttt{wsmp\_scaling} \leq 3$ and its default value is 0.

wsmp_singularity_threshold: WSMP's singularity threshold.
WSMP's DPARM(10) parameter. The smaller this value the less likely a matrix is declared singular. This option is only available if Ipopt has been compiled with WSMP. The valid range for this real option is $0 < \texttt{wsmp\_singularity\_threshold} < 1$ and its default value is $1 \cdot 10^{-18}$.

D Options available via the AMPL Interface

The following is a list of options that is available via AMPL:

acceptable_compl_inf_tol Acceptance threshold for the complementarity conditions

acceptable_constr_viol_tol Acceptance threshold for the constraint violation

acceptable_dual_inf_tol Acceptance threshold for the dual infeasibility

acceptable_tol Acceptable convergence tolerance (relative)

alpha_for_y Step size for constraint multipliers

bound_frac Desired minimal relative distance of initial point to bound

bound_mult_init_val Initial value for the bound multipliers

bound_push Desired minimal absolute distance of initial point to bound

bound_relax_factor Factor for initial relaxation of the bounds

compl_inf_tol Acceptance threshold for the complementarity conditions

constr_mult_init_max Maximal allowed least-square guess of constraint multipliers

constr_viol_tol Desired threshold for the constraint violation

diverging_iterates_tol Threshold for maximal value of primal iterates

dual_inf_tol Desired threshold for the dual infeasibility

expect_infeasible_problem Enable heuristics to quickly detect an infeasible problem

file_print_level Verbosity level for output file

halt_on_ampl_error Exit with message on evaluation error

hessian_approximation Can enable Quasi-Newton approximation of hessian

honor_original_bounds If no, solution might slightly violate bounds

linear_scaling_on_demand Enables heuristic for scaling only when seems required

linear_solver Linear solver to be used for step calculation

linear_system_scaling Method for scaling the linear systems

ma27_pivtol Pivot tolerance for the linear solver MA27

ma27_pivtolmax Maximal pivot tolerance for the linear solver MA27

ma57_pivot_order Controls pivot order in MA57

ma57_pivtol Pivot tolerance for the linear solver MA57

ma57_pivtolmax Maximal pivot tolerance for the linear solver MA57

max_cpu_time CPU time limit

max_iter Maximum number of iterations

max_refinement_steps Maximal number of iterative refinement steps per linear system solve

max_soc Maximal number of second order correction trial steps

maxit (Ipopt name: max_iter) Maximum number of iterations

min_refinement_steps Minimum number of iterative refinement steps per linear system solve

mu_init Initial value for the barrier parameter

mu_max Maximal value for barrier parameter for adaptive strategy

mu_oracle Oracle for a new barrier parameter in the adaptive strategy

mu_strategy Update strategy for barrier parameter

nlp_scaling_max_gradient Maximum gradient after scaling

nlp_scaling_method Select the technique used for scaling the NLP

obj_scaling_factor Scaling factor for the objective function

option_file_name File name of options file (default: ipopt.opt)

outlev (Ipopt name: print_level) Verbosity level

output_file File name of an output file (leave unset for no file output)

pardiso_matching_strategy Matching strategy for linear solver Pardiso

pardiso_out_of_core_power Enables out-of-core version of linear solver Pardiso

print_level Verbosity level

print_options_documentation Print all available options (for ipopt.opt)

print_user_options Toggle printing of user options

required_infeasibility_reduction Required infeasibility reduction in restoration phase

slack_bound_frac Desired minimal relative distance of initial slack to bound

slack_bound_push Desired minimal absolute distance of initial slack to bound

tol Desired convergence tolerance (relative)

wantsol solution report without -AMPL: sum of, 1 == write .sol file, 2 == print primal variable values, 4 == print dual variable values, 8 == do not print solution message

warm_start_bound_push Enables to specify how much should variables should be pushed inside the feasible region

warm_start_init_point Enables to specify bound multiplier values

warm_start_mult_bound_push Enables to specify how much should bound multipliers should be pushed inside the feasible region

watchdog_shortened_iter_trigger Trigger counter for watchdog procedure

References

[1] L. T. Biegler. *Nonlinear Programming: Concepts, Algorithms and Applications to Chemical Processes* SIAM, Philadelphia (2010)

[2] F. E. Curtis, J. Huber, O. Schenk, A. Wächter. A note on the implementation of an interior-point algorithm for nonlinear optimization with inexact step computations. *Mathematical Programming*, 136(1):209–227, 2012 doi: 10.1007/s10107-012-0557-4. preprint at `http://www.optimization-online.org/DB_HTML/2011/04/2992.html`

[3] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language For Mathematical Programming*. Thomson Publishing Company, Danvers, MA, USA, 1993.

[4] O. Schenk, A. Wächter, and M. Hagemann. Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization. *Computational Optimization Applications*, 36(2-3):321–341, 2007 doi: 10.1007/s10589-006-9003-y.

[5] W. Hock and K. Schittkowski. Test examples for nonlinear programming codes. *Lecture Notes in Economics and Mathematical Systems*, 187, 1981. doi: 10.1007/978-3-642-48320-2.

[6] J. Nocedal, A. Wächter, and R. A. Waltz. Adaptive barrier strategies for nonlinear interior methods. *SIAM Journal on Optimization*, 19(4):1674–1693, 2008. doi: 10.1137/060649513. preprint at `http://www.optimization-online.org/DB_HTML/2005/03/1089.html`

[7] H. Pirnay, R. Lopez-Negrete, and L. T. Biegler. Optimal Sensitivity based on IPOPT. *Mathematical Programming Computations*, 4(4):307–331, 2012. doi: 10.1007/s12532-012-0043-2. preprint at `http://www.optimization-online.org/DB_HTML/2011/04/3008.html`

[8] A. Wächter. *An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, January 2002. available at `http://researcher.watson.ibm.com/researcher/files/us-andreasw/thesis.pdf`

[9] A. Wächter. Short Tutorial: Getting Started With Ipopt in 90 Minutes. In *Combinatorial Scientific Computing* (U. Naumann, O. Schenk, H. D. Simon, eds.), 2009. urn: nbn:de:0030-drops-20890

[10] A. Wächter and L. T. Biegler. Line search filter methods for nonlinear programming: Local convergence. *SIAM Journal on Optimization*, 16(1):32–48, 2005. doi: 10.1137/S1052623403426544

[11] 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. doi: 10.1137/S1052623403426556

[12] A. Wächter and L. T. Biegler. Global and Local Convergence of Line Search Filter Methods for Nonlinear Programming. Optimization Online, 2001. `http://www.optimization-online.org/DB_HTML/2001/08/367.html`

[13] A. Wächter and L. T. Biegler. On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006. doi: 10.1007/s10107-004-0559-y. preprint at `http://www.optimization-online.org/DB_HTML/2004/03/836.html`