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

Revision number of this document: *Revision* : 1812

November 26, 2010

Abstract

This document is a guide to using IPOPT 3.9.1. 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 Andreas Wächter[3].

Contents

---

[1] then Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh PA
[2] then Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh PA
[3] Department of Mathematical Sciences, IBM T.J. Watson Research Center, Yorktown Heights, NY

The following names used in this document are trademarks or registered trademarks: AMPL, IBM, Intel, Matlab, Microsoft, MKL, Visual Studio C++, Visual Studio C++ .NET

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 [3, 4, 7, 6, 5].

1.2   Availability

The IPOPT package is available from COIN-OR (`www.coin-or.org`) under the CPL (Common 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 Common 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 [7]. 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 `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 Harwell Subroutine Library
    (see `http://www.cse.clrc.ac.uk/nag/hsl/`).

  - MA57 from the Harwell Subroutine Library
    (see `http://www.cse.clrc.ac.uk/nag/hsl/`).

  - MUMPS (MUltifrontal Massively Parallel sparse direct Solver)
    (see `http://graal.ens-lyon.fr/MUMPS/`)

  - The Parallel Sparse Direct Solver (PARDISO)
    (see `http://www.computational.unibas.ch/cs/scicomp/software/pardiso/`).
    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://www-users.cs.umn.edu/~agupta/wsmp.html`)

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 MA27, MA57, and Pardiso linear solvers from a shared library at runtime, if the library has not been compiled with them (see Section 2.2.7).

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 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 Harwell Subroutine 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[4].

- 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 2.2.

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 currently also searches for a Fortran, 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 `www.netlib.org/f2c` to convert Fortran code to C, and compile the resulting C files with a C compiler and create a library containing the required third party dependencies. We have tested and used this in connection with the Microsoft Visual C++ compiler, and instructions on how to use it in this context are given below.

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. Below 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.

The IPOPT distribution includes an interface for `CUTEr`[5] (for solving problems modeled in SIF), and also a Matlab (mex) interface which allows you to use Ipopt from Matlab, see

> `https://projects.coin-or.org/Ipopt/wiki/MatlabInterface.`

The distribution also includes an interface to the R project for statistical computing, see the files in the `Ipopt/contrib/RInterface` directory.

There are also interfaces maintained by other people:

- 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`.

---

[4]There are more recent scaling routines in the HSL, but they have not (yet) been integrated. Contributions are welcome!
[5]see `http://cuter.rl.ac.uk/cuter-www/`

- 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`.

- Scilab (free Matlab-like envirnoment):

  Edson Cordeiro do Valle has written an interface to use Ipopt from Scilab,
  see `http://www.scilab.org/contrib/displayContribution.php?fileID=839`

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

- Python:

  An interface to the python language is available here: `http://code.google.com/p/pyipopt`

1.5   More Information and Contributions

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

$$\texttt{http://projects.coin-or.org/Ipopt}.$$

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 contributions webpage[7], and contact the IPOPT project leader.

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

$$\texttt{http://list.coin-or.org/mailman/listinfo/ipopt},$$

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. However, if you have a challenging optimization problem and are interested in consulting services by IBM Research, please contact the IPOPT project leader, Andreas Wächter.

---

[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`

1.6 History of Ipopt

The original IPOPT (Fortran version) was a product of the dissertation research of Andreas Wächter [4], 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) is 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/`.

2 Installing Ipopt

The following sections describe the installation procedures on UNIX/Linux systems. For installation instructions on Windows see Section 2.4.

2.1 Getting the Ipopt Code

IPOPT is available from the COIN-OR subversion repository. You can either download the code using `svn` (the *subversion*[8] 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.

2.1.1 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.tigris.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/3.9 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`).

2.1.2 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 `3.9.1`. 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.

---

[8]see `http://subversion.tigris.org/`

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`).

2.2 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. It also requires at least one linear solver for symmetric indefinite matrices.

Since this third party software released under different licenses than IPOPT, we cannot distribute that 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 if 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.

2.2.1 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 2.3). Some of those libraries also include LAPACK.

If you have the download utility `wget` installed on your system, 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 (`www.netlib.org`).

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

If you do not have `wget` 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.

2.2.2 Download HSL Subroutines

IPOPT requires a sparse symmetric linear solver. There are different possibilities. **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.

In this section we describe how to obtain the source code for MA27 (and MC19) from the Harwell Subroutine Library (HSL) Archive. Those routines are freely available for non-commercial, academic use, but it is your responsibility to investigate the licensing of all third party code. (Since recently, also MA57 (the successor of MA27) is available for free for academic institutions, via the "HSL 2007 for Researchers" library, see below.)

The use of alternative linear solvers is described in Sections 2.2.3-2.2.6. You do not necessarily have to use MA27 as described in this section, but at least one linear solver is required for IPOPT to function.

1. Go to `http://hsl.rl.ac.uk/archive/hslarchive.html`

2. Follow the instruction on the website, read the license, and submit the registration form.

3. Go to *HSL Archive Programs*, and find the package list.

4. In your browser window, click on *MA27*.

5. Make sure that *Double precision:* is checked. Click *Download package (comments removed)*

6. Save the file as `ma27ad.f` in `$IPOPTDIR/ThirdParty/HSL/`
   Note: Some browsers append a file extension (`.txt`) when you save the file, in which case you have to rename it.

7. Go back to the package list using the back button of your browser.

8. In your browser window, click on *MC19*.

9. Make sure *Double precision:* is checked. Click *Download package (comments removed)*

10. Save the file as `mc19ad.f` in `$IPOPTDIR/ThirdParty/HSL/`
    Note: Some browsers append a file extension (`.txt`) when you save the file, so you may have to rename it.

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 Harwell solvers, but it can also be switched on for other linear solvers (which usually have internal scaling mechanisms).

Note: If you satisfy the conditions to use the HSL 2007 for Researchers library (see `http://hsl.rl.ac.uk/hsl2007/hsl20074researchers.html`), you can download the MA57 in a way very similar to the procedure outlined above. To have Ipopt compile it for you, you need to put the source code into a *single* file called ma57ad.f (*including all dependencies!*) that is in the `$IPOPTDIR/ThirdParty/HSL/` directory. The IPOPT configuration script will then find this file and compile it into the IPOPT library (just as is would compile MA27). MA57 will perform better with METIS, see Section 2.2.4.

Yet another note: If you have a precompiled library containing the Harwell codes, you can specify the location of this library with the `--with-hsl` flag for the `configure` script described in Section 2.3.

2.2.3   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 source code by requesting the latest version from the MUMPS home page, and then extract the source code in the directory `$IPOPTDIR/ThirdParty/Mumps`. The extracted MUMPS directory usually has the MUMPS version number in it, and you need to rename it to `MUMPS` so that you have a file called `$IPOPTDIR/ThirdParty/Mumps/MUMPS/Conditions_of_Use`.

You can also try to run the script `get.Mumps` in `$IPOPTDIR/ThirdParty/Mumps` if you have `wget` installed in your system. The MUMPS version number in that file might be out of date; in that case edit the value assigned to the `mumps_ver` variable at the beginning of the script.

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 2.2.4.

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 modified the MUMPS sources so that the MPI symbols inside the MUMPS code are renamed.

2.2.4 Obtaining METIS

The linear solvers MA57 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 those 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 installed on your system, you can download the code by running the `./get.Metis` script.

2.2.5 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 free download for Pardiso if you want to use it for "non-commercial and non-profit internal research purposes" and are an "academic, non-profit, or government agency" (taken from the license agreement). 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 D to find out how to configure your IPOPT installation to work with Pardiso.

2.2.6 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.alphaworks.ibm.com/tech/wsmp`

At this website you can obtain a "complimentary 90-day evaluation license" and download the library for several operating systems; make sure you read the license agreement before downloading the code. Once you obtained the code and the license, please check if the version number of the library matches the one on the WSMP website at

`http://www-users.cs.umn.edu/~agupta/wsmp`

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.

You can use the bugfix releases with the license you obtained from alphaWorks.

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

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

2.2.7 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, or the Pardiso linear solver at runtime[9]. This means, if you obtain one of those solvers after you already had compiled IPOPT, you don't need to recompile IPOPT to now use them. 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 Darwin, the names are `libhsl.dylib`, `libpardiso.dylib`, and `DYLD_LIBRARY_PATH`. On Windows (where IPOPT cannot be used with Pardiso), the names are `libhsl.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 Compiling and Installing Ipopt

IPOPT can be easily compiled and installed with the usual `configure`, `make`, `make install` commands. We follow the precedure 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 D 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 D.

   If the last output line of the script 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 D.

   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 D, and also visit the generic COIN-OR configuration instruction page at
   `https://projects.coin-or.org/CoinHelp/wiki/user-configure`

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.

---

[9]This is not enabled if you compile Ipopt with the Developer 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

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 D).

5. Install IPOPT

   `$ make install`

   This installs

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

   You can change the default installation directory (here `$IPOPTDIR`) 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.

2.4 Installation on Windows

There are several ways to install IPOPT on Windows systems. The first two option, described in Sections 2.4.1 and 2.4.2, is to use Cygwin (see `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 2.4.1. 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 2.4.2. If you use MSYS/MinGW (a light-weight UNIX-like environment for Windows), please consider the notes in Section 2.4.3. The IPOPT distribution also includes projects files for the Microsoft Visual Studio (see Section 2.4.4).

NEW: Some binaries for IPOPT are available on the COIN website at

`http://www.coin-or.org/download/binary/Ipopt/`

There are also precompiled versions of Ipopt as DLLs (generated from the MSVC solution in IPOPT's subdirecotry `$IPOPTDIR/Ipopt/MSVisualStudio/v8-ifort`. Look at the `README` files for details. An example how to use these DLLs from your own MSVC project is in `$IPOPTDIR/Ipopt/MSVisualStudio/BinaryDLL-Link-Example`.

2.4.1 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, `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 `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++ — together with the MinGW options), the GNU Make, and Subversion. For this, click on the "Devel" branch (which opens a subtree) and select:

   - gcc
   - gcc-core
   - gcc-g77
   - gcc-g++
   - gcc-mingw
   - gcc-mingw-core
   - gcc-mingw-g77
   - gcc-mingw-g++
   - make
   - 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.

Then, in the "Web" branch, please select "wget" (which will make the installation of third party dependencies for IPOPT easier)

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

10. Now you just follow the instructions in the beginning of Sections 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 (like 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`

   NOTE: By default, the compiled binaries (library and executables) will be "Cygwin-native", i.e., in order to run executables using this, the `Cygwin1.dll` has to be present (e.g., in a Cygwin window). If you want to compile things in a way that allow your executables to run outside of Cygwin, e.g., in a regular DOS prompt, you need to specify the option "`--enable-doscompile`" when you run `configure`.

2.4.2   Installation with Cygwin using the MSCV++ 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 2.4.1 until Step 10 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 set the environment variables to 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. If want to compile the AMPL solver executable, you need to compile the ASL library from a script. For this you need to change into the ASL compilation directory, execute the script `compile_MS_ASL`, and go back to the directory where you were:

   `cd ThirdParty/ASL`

   `./compile_MS_ASL`

   `cd -`

5. Now you can compile the code with

   `make`,

   test the installation with

   `make test`,

   and install everything with

   `make install`

2.4.3   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 compiler IPOPT with native Windows compilers (see Section 2.4.2), 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 2.3. If you want to use the native MSVC++ compiler (with `f2c` or a native Fortran compiler), you essentially follow the steps outlined in Section 2.4.2. In the latter case, you again 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. Also, you need to run the `compile_MS_ASL` script in the `ThirdParty/ASL` immediately after you run the configuration script.

2.4.4   Using Microsoft Visual Studio

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

The IPOPT distribution includes project files that can be used to compile the IPOPT library, the AMPL solver executable, and a C++ example within the Microsoft Visual Studio. The project files have

---

[11]which used to come as a convenient Windows install program, but as I'm writing this, I can't find it on the web...

been created with Microsoft Visual 8 Express. Fortran files in the third party dependencies need to be converted to C code using the `f2c` Fortran to C compiler[12].

In order to use those project files, download the IPOPT source code, as well as the required third party code (put it into the `ThirdParty\Blas`, `ThirdParty\Lapack`, `ThirdParty\HSL`, `ThirdParty\ASL` directories. Detailed step-by-step instructions on how to install `f2c`, translate the Fortran code to C files, and further details are described in the file

`$IPOPTDIR\Ipopt\MSVisualStudio\v8\README.TXT`

After that, you can open the solution file

`$IPOPTDIR\Ipopt\MSVisualStudio\v8\Ipopt.sln`

If you are compiling IPOPT with different linear solvers, you need to edit the configuration header file `Ipopt\src\Common\IpoptConfig.h`, in the section after

```
/****************************************************************************/
/*              HERE DEFINE THE CONFIGURATION SPECIFIC MACROS          */
/****************************************************************************/
```

and include the corresponding source files in `Ipopt\src\Algorithm\LinearSolvers` and add the corresponding libraries to your project.

3 Interfacing your NLP to Ipopt: A tutorial example.

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[1] 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 [2],

$$\min_{x \in \Re^4} \quad x_1 x_4 (x_1 + x_2 + x_3) + x_3 \tag{4}$$

$$\text{s.t.} \quad x_1 x_2 x_3 x_4 \geq 25 \tag{5}$$

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

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

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.

---

[12]Projects files for a previous version of IPOPT that used the Intel Fortran compiler are in $IPOPTDIR\Ipopt\NoLongerMaintainedWindows, but they are probably outdated, and you will have to correct them.

```
# 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;
```

Figure 1: AMPL model file hs071_ampl.mod

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, 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 [1].

The problem presented in equations (4)–(8) can be solved with IPOPT with the AMPL model file given in Figure 1.

The line, "option solver ipopt;" tells AMPL to use IPOPT as the solver. The IPOPT executable (installed in Section 2.3) 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 information required by IPOPT is shown in Figure 2. 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

18

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 $(x, \lambda, \sigma_f$ 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)

Figure 2: Information required by IPOPT

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 Figure 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[13]. 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, so that IPOPT can ask for the Hessian of the objective or the constraints independently if required, we introduce a factor (σ_f) in front of the objective term. For IPOPT then, the symbolic form of the Hessian of the Lagrangian is

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

(with the σ_f parameter), and for the example problem this becomes

$$
\sigma_f
\begin{bmatrix}
2x_4 & x_4 & x_4 & 2x_1 + x_2 + x_3 \\
x_4 & 0 & 0 & x_1 \\
x_4 & 0 & 0 & x_1 \\
2x_1 + x_2 + x_3 & x_1 & x_1 & 0
\end{bmatrix}
+ \lambda_1
\begin{bmatrix}
0 & x_3 x_4 & x_2 x_4 & x_2 x_3 \\
x_3 x_4 & 0 & x_1 x_4 & x_1 x_3 \\
x_2 x_4 & x_1 x_4 & 0 & x_1 x_2 \\
x_2 x_3 & x_1 x_3 & x_1 x_2 & 0
\end{bmatrix}
+ \lambda_2
\begin{bmatrix}
2 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 2
\end{bmatrix}
$$

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 then 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.

---

[13]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.

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 2.3), 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 information required in Figure 2 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 under `examples`, called `MyExample` 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.

```cpp
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

```cpp
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.

```cpp
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 HSO71_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, $sigma_f$.

- 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 Eq. 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 Figure 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).

    – 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 `HS071_NLP.hpp` and `IpIpoptApplication.hpp`, tell the compiler to use the `Ipopt` namespace, and implement the `main` function.

```cpp
#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, process this options (particularly the output related options), and the call to `app->OptimizeTNLP(...)` will run IPOPT and try to solve the problem. By default, IPOPT will write to 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 IPOPT library `libipopt.so` (or `libipopt.a`), as well as other necessary libraries, as listed in the `ipopt_addlibs_cpp.txt` and `ipopt_addlibs_f.txt`

files. If you are using Linux/UNIX, then a sample makefile exists already that was created by configure. 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 Common Public License (CPL).
         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 IPOPTand 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. These features is not yet(?) available from C or Fortran.

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 requires a premature termination.

The information provided by the entities in the argument list corresponds 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 (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`.

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`. Then the user is 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 to be 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., then 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`.

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`[14] with the function `CreateIpoptProblem`, which later has to be passed to the `IpoptSolve` function.

---

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

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 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;
```

```c
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\n");
  printf("f(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).

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]
* 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 start ("`*`") 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 those two numbers.

For constraints, the first index after `jac_g` is the number of the constraint, and the second one corresponds to the variable number (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 lines look 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 using the `hessian_approximation=limited-memory` option. 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 here might be the case, where your optimization problem has a dense Hessian or a large percentage of non-zero entries in the Hessian, and then using the quasi-Newton approximation might be better, even if it the number of iterations increases, since 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 solves in order to compute the search direction, if exact second derivatives are used.

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.

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. Don't forget to ensure you have a newline at the end of the file. For example,

```
# This is a comment

# 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 all of the options documentation 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 while in 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 scaled primal infeasibility at the current point. During the restoration phase, this value is the primal infeasibility of the original problem at the current point.

inf_du: The scaled dual infeasibility at the current point. 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 mu.

||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 [7]).

lg(rg): $\log_{10}$ of the value of the regularization term for the Hessian of the Lagrangian in the augmented system.

alpha_du: The stepsize for the dual variables.

alpha_pr: The stepsize for the primal variables.

ls: The number of backtracking line search steps.

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 `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 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`.

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 Message: `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).

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 Message: `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.

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

$$
\begin{bmatrix}
1.1 & 0 & 0 & 0 & 0 & 0 & 0.5 \\
0 & 1.9 & 0 & 0 & 0 & 0 & 0.5 \\
0 & 0 & 2.6 & 0 & 0 & 0 & 0.5 \\
0 & 0 & 7.8 & 0.6 & 0 & 0 & 0 \\
0 & 0 & 0 & 1.5 & 2.7 & 0 & 0 \\
1.6 & 0 & 0 & 0 & 0.4 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.9 & 1.7
\end{bmatrix}
\tag{10}
$$

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 savings over the 49 floating point numbers stored in the dense representation, for larger matrices, the space savings are very dramatic[15].

The option `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 1 and 2 below show the triplet format for both indexing styles, using the example matrix (10).

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 1: Triplet Format of Matrix (10) 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

---

[15]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 2: Triplet Format of Matrix (10) with `index_style=C_STYLE`

example, given the matrix,

$$\begin{bmatrix} 1.0 & 0 & 3.0 & 0 & 2.0 \\ 0 & 1.1 & 0 & 0 & 5.0 \\ 3.0 & 0 & 1.2 & 6.0 & 0 \\ 0 & 0 & 6.0 & 1.3 & 9.0 \\ 2.0 & 5.0 & 0 & 9.0 & 1.4 \end{bmatrix} \tag{11}$$

the triplet format is shown in Tables 3 and 4.

Table 3: Triplet Format of Matrix (10) with `index_style=FORTRAN_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 4: Triplet Format of Matrix (10) with `index_style=C_STYLE`

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

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

The `SmartPtr` class is described in `IpSmartPtr.hpp`. It is a template class that takes care of deleting objects for us so we need not be concerned about memory leaks. 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 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.

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 yes
```

or simply run the IPOPT AMPL solver executable as

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

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

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 (to overwrite default).
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. The default value for this string option is "".
Possible values:

- *: Any acceptable standard file name

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 $0 < \texttt{tol} < +\texttt{inf}$ 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 $0 \leq \texttt{max\_iter} < +\texttt{inf}$ 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 $0 < \texttt{max\_cpu\_time} < +\texttt{inf}$ 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 $0 < \texttt{dual\_inf\_tol} < +\texttt{inf}$ 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 $0 < \texttt{constr\_viol\_tol} < +\texttt{inf}$ 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 $0 < \texttt{compl\_inf\_tol} < +\texttt{inf}$ 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 $0 < \texttt{acceptable\_tol} < +\texttt{inf}$ 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 $0 < \texttt{bound\_frac} \le 0.5$ 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_s` $+$`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

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_fact- or"+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 $0 < \mathtt{mu\_min} < +\mathtt{inf}$ 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 $0 \le \mathtt{mu\_target} < +\mathtt{inf}$ 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 $0 < \mathtt{barrier\_tol\_factor} < +\mathtt{inf}$ 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^"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 $0 < \mathtt{mu\_linear\_decrease\_factor} < 1$ 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^"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 $1 < \mathtt{mu\_superlinear\_decrease\_power} < 2$ 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 p̂max 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 $0 < \texttt{warm\_start\_bound\_push} < +\texttt{inf}$ 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 $0 < \texttt{warm\_start\_bound\_frac} \leq 0.5$ 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 $0 < \texttt{warm\_start\_slack\_bound\_frac} \leq 0.5$ 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 $0 < \texttt{warm\_start\_slack\_bound\_push} < +\texttt{inf}$ 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 $0 < \texttt{warm\_start\_mult\_bound\_push} < +\texttt{inf}$ 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 $-\texttt{inf} < \texttt{warm\_start\_mult\_init\_max} < +\texttt{inf}$ 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 $0 \le$ `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_y` $+$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 $0 \le$ `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 $0 \le$ `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 $0 \le$ `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 $0 \le$ `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 MA77

- 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_w$\hat{}$max 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_w$\hat{}$min 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$\hat{+}$ 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$\hat{+}$ 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_v` $+$`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 10.

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` $< +\texttt{inf}$ 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.This is ICNTL(15) in MA57. The default value for this string option is "yes".
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` $< +\texttt{inf}$ 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` $< +\texttt{inf}$ 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 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` $< +\mathtt{inf}$ 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.18   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` $< +\mathtt{inf}$ 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` $< +\mathtt{inf}$ and its default value is 0.

C.19   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 $-\mathtt{inf} <$ `wsmp_num_threads` $< +\mathtt{inf}$ 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 $-2 \leq$ `wsmp_ordering_option` ≤ 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 $0 <$ `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 $0 <$ `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 $0 \leq$ `wsmp_scaling` ≤ 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 $0 <$ `wsmp_singularity_threshold` < 1 and its default value is $1 \cdot 10^{-18}$.

D   Detailed Installation Information

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:

<div align="center">https://projects.coin-or.org/CoinHelp</div>

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[16]) 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 Harwell Subroutines, you can specify the link flags with the `--with-hsl` flag. And the *location of the directory* with the AMPL solver library `amplsolver.a` and the ASL header files can be specified with `--with-asldir`.

- If you want to compile IPOPT with the linear solver Pardiso (see Section 2.2.5), 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.

- If you want to compile IPOPT with the linear solver WSMP (see Section 2.2.6), 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 2.2.3), 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`

---

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

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.

- It is not necessary to produce the binary files in the directories where the source files are. If you want to compile the code on different systems or with different compilers/options on a shared file system, you can keep one single copy of the source files in one directory, and the binary files for each configuration in separate directories. For this, simply run the configure script in the directory where you want the base directory for the IPOPT binary files. For example:

  ```
  $ mkdir $HOME/Ipopt-objects
  $ cd $HOME/Ipopt-objects
  $ $HOME/CoinIpopt/configure
  ```

References

[1] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language For Mathematical Programming.* Thomson Publishing Company, Danvers, MA, USA, 1993.

[2] W. Hock and K. Schittkowski. Test examples for nonlinear programming codes. *Lecture Notes in Economics and Mathematical Systems*, 187, 1981.

[3] J. Nocedal, A. Wächter, and R. A. Waltz. Adaptive barrier strategies for nonlinear interior methods. Technical Report RC 23563, IBM T.J. Watson Research Center, Yorktown Heights, USA, March 2005.

[4] 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.

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

[6] 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.

[7] 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.