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

Revision number of this document: *Revision* : 608

February 23, 2006

Abstract

This document is a guide to using IPOPT 3.1 (the new C++ version of IPOPT). 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.

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. The current version is maintained by Carl Laird[2] and Andreas Wächter[3].

Contents

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

1 Introduction

IPOPT (Interior Point Optimizer, pronounced "I–P–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, $g^L \in (\mathbb{R} \cup \{-\infty\})^n$ 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 :).

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. But you can also get the source code from `www.netlib.org` and have the IPOPT distribution compile it automatically.

- 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`.

- A sparse symmetric indefinite linear solver. The 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/`).
  - The Watson Sparse Matrix Package (WSMP)
    (see `http://www-users.cs.umn.edu/ãgupta/wsmp.html`)
  - The Parallel Sparse Direct Linear Solver (PARDISO)
    (see `http://www.computational.unibas.ch/cs/scicomp/software/pardiso/`).

  You need to include at least one of the linear solvers above in order to run IPOPT.

  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.

- 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` 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. But so far we have not tested this ourselves, and currently the configuration script for IPOPT looks for a Fortran compiler.

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 or Cygwin. 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.

Finally, the IPOPT distribution includes an interface for `CUTEr`[4], if you want to use IPOPT to solve problems modeled in SIF.

The old (Fortran 2.x) version of IPOPT has been interface with Matlab, and is also available from NEOS, and the new version will be available through similar means in the future. Please check the IPOPT homepage for updates.

1.5 More Information and Contributions

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

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

Here, you can find FAQs, some (hopefully useful) hints, a bug report system etc. The website is managed with Wiki, which means that every user can edit the webpages from the regular web browser. In particular, we encourage IPOPT users to share their experiences and usage hints on the "Success Stories" and "Hints and Tricks" pages[5]

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 constributions webpage[6], and contact the IPOPT project leader.

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

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

where you can subscribe to get notified of updates, and to ask general questions regarding installation and usage. (You might want to look at the archives before posting a question.)

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.

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

[5] 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 in the last paragraph of the Documentation section on the main project page.

[6] 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 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++. The new C++ version of the IPOPT optimization code (IPOPT 3.0 and beyond) is currently developed at IBM Research and remains part of the COIN-OR initiative. Future development on the Fortran version will cease with the exception of occasional bug fix releases.

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*[7] 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://www.coin-or.org/svn/Ipopt/trunk Ipopt`
   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 Ipopt`

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 latest tarball from `http://www.coin-or.org/Tarballs`. The file you should look for has the form `ipopt-3.x.x.tar.gz` (where "`3.x.x.`" is the version number). Put this file in a directory under which you want to put the IPOPT installation.
2. Issue the following commands to unpack the archive file:
   `$ gunzip ipopt-3.x.x.tar.gz`
   `$ tar xvf ipopt-3.x.x.tar`
   Note: The `$` indicates the command line prompt, do not type `$`, only the text following it.

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

3. Change into the root directory of the IPOPT distribution
   `$ cd ipopt-3.x.x`

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), BLAS, LAPACK. It also requires a sparse symmetric linear solver.

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

If you have the download utility `wget` installed on your system, retrieving BLAS, 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/Extern/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 `$IPOPTDIR/Extern/blas`, `$IPOPTDIR/Extern/lapack` and `$IPOPTDIR/Extern/ASL` directories for alternative instructions.

2.2.2 Download HSL Subroutines

IPOPT requires a sparse symmetric linear solver. There are different possibilities. In this section we describe how to obtain the source code for MA27 (and MC19) from the Harwell Subroutine Library (HSL). Those routines are freely available for non-commercial, academic use, but it is your responsibility to investigate the licensing of all third party code.

The use of alternative linear solvers is described in Appendix D. 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/Extern/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/Extern/HSL/`
    Note: Some browsers append a file extension (`.txt`) when you save the file, so you may have to rename it.

Note: Whereas currently obtaining MA27 is essential for using IPOPT, MC19 could be omitted (with the consequence that you cannot use this method for scaling the linear systems arising inside the IPOPT algorithm).

Note: If you have the source code for the linear solver MA57 (successor of MA27) in a file called ma57ad.f (including all dependencies), you can simply put it into the `$IPOPTDIR/Extern/HSL/` directory. The IPOPT configuration script will then find this file and compile it into the IPOPT library (just as is would compile MA27).

2.3 Compiling and Installing IPOPT

IPOPT can be easily compiled and installed with the usual `configure`, `make`, `make install` commands. Below are the basic steps that should work on most systems. For special compilations and for some troubleshooting see Appendix D and consult the IPOPT homepage before submitting a ticket or sending a message to the mailing list.

1. Go to the main directory of IPOPT:
   `$ cd $IPOPTDIR`

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

   If the last output line of the script reads "`configure: Configuration successful`" then everything worked fine. Otherwise, look at the screen output, have a look at the `config.log` output file and/or consult Appendix D.

   The default configure (without any options) is sufficient for most users. If you want to see the configure options, consult Appendix D.

3. Build the code
   `$ make`

4. Install IPOPT
   `$ make install`
   This installs

   - the IPOPT AMPL solver executable (if ASL source was downloaded) in `$IPOPTDIR/bin`,
   - the IPOPT library (`libipopt.a`) in `$IPOPTDIR/lib`,
   - text files `ipopt_addlibs_cpp.txt` and `ipopt_addlibs_f.txt` in `$IPOPTDIR/lib` 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/include/ipopt`.

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

5. 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.

2.4 Installation on Windows

There are two ways to install IPOPT on Windows systems. The first option, described in Section 2.4.1, is to use Cygwin (see `www.cygwin.com`), which offers a UNIX-like environment on Windows and in which the installation procedure described earlier in this section can be used. The IPOPT distribution also includes projects files for the Microsoft Visual Studio (see Section 2.4.2).

2.4.1 Installation with Cygwin

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[8].

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

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

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

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`").

2.4.2 Using Visual Studio

The IPOPT distribution includes project files that can be used to compile the IPOPT library and a Fortran and C++ example within the Microsoft Visual Studio. The project files have been created with Microsoft Visual C++ .NET 2003 Standard, and the Intel Visual Fortran Compiler 8.1.

In order to use those project files, download the IPOPT source code, as well as the required third party code (put it into the `Extern/blas`, `Extern/lapack`, and `Extern/HSL` directories—ASL is not required for the Fortran and C examples). Then open the solution file

```
$IPOPTDIR\Windows\VisualStudio_dotNET\Ipopt\Ipopt.sln
```

Note: Since the project files were created only with the Standard edition of the C++ compiler, code optimization might be disabled; for fast performance make sure you enable code optimization.

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

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

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.

3.2 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

```
# 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

1. Problem dimensions
   - number of variables
   - number of constraints
2. Problem bounds
   - variable bounds
   - constraint bounds
3. Initial starting point
   - Initial values for the primal x variables
   - Initial values for the multipliers (only required for a warm start option)
4. Problem Structure
   - number of nonzeros in the Jacobian of the constraints
   - number of nonzeros in the Hessian of the Lagrangian function
   - sparsity structure of the Jacobian of the constraints
   - sparsity structure of the Hessian of the Lagrangian function
5. Evaluation of Problem Functions
   Information evaluated using a given point (x, λ, σ_f coming from IPOPT)
   - Objective function, $f(x)$
   - Gradient of the objective $\nabla f(x)$
   - Constraint function values, $g(x)$
   - Jacobian of the constraints, $\nabla g(x)^T$
   - Hessian of the Lagrangian function, $\sigma_f \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x)$
     (this is not required if a quasi-Newton options is chosen to approximate the second derivatives)

Figure 2: Information required by IPOPT

algorithm when it begins iterating to solve the problem. If IPOPT has difficulty converging, or if it converges to a locally infeasible point, adjusting the starting point may help. Depending on the starting point, IPOPT may also converge to different local solutions.

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

As IPOPT iterates, it will need the values for Item 5. in 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

$$\begin{bmatrix} x_1x_4 + x_4(x_1 + x_2 + x_3) \\ x_1x_4 \\ x_1x_4 + 1 \\ x_1(x_1 + x_2 + x_3) \end{bmatrix},$$

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

$$\begin{bmatrix} x_2x_3x_4 & x_1x_3x_4 & x_1x_2x_4 & x_1x_2x_3 \\ 2x_1 & 2x_2 & 2x_3 & 2x_4 \end{bmatrix}.$$

We also need to determine the Hessian of the Lagrangian[9]. 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

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

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

[9] 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.

executable that prepares the problem, and then passes control over to IPOPT through an `Optimize` or `Solve` call. In this call, you will give IPOPT everything that it requires to call back to your code whenever it needs functions evaluated (like the objective function, the Jacobian of the constraints, etc.). In each of the three sections that follow (C++, C, and Fortran), we will first discuss how to code the problem representation, and then how to code the executable.

3.3 The C++ Interface

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

After "`make install`" (see Section 2.3), the header files are installed in `$IPOPTDIR/include/ipopt` (or in `$PREFIX/include/ipopt` 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 `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 `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 occured, for example, because the value of a problem function could not be evaluated at the required point.

Method `get_nlp_info` with prototype

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

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

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

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

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

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

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

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

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

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

  return true;
}
```

Method `get_bounds_info` with prototype

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

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

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

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

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

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

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

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

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

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

  return true;
}
```

Method `get_starting_point` with prototype

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

Give IPOPT the starting point before it begins iterating.

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

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

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

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

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

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

  return true;
}
```

Method `eval_f` with prototype

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

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

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

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

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

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

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

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

  return true;
}
```

Method `eval_grad_f` with prototype

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

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

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

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

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

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

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

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

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

  return true;
}
```

Method `eval_g` with prototype

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

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

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

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

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

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

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

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

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

  return true;
}
```

Method `eval_jac_g` with prototype

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

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

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

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

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

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

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

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

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

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

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

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

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

  return true;
}
```

Method `eval_h` with prototype

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

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

- `n`: (in), the number of variables in the problem (dimension of x).
- `x`: (in), the values for the primal variables, x, at which the Hessian is to be evaluated.
- `new_x`: (in), false if any evaluation method was previously called with the same values in `x`, true otherwise.
- `obj_factor`: (in), factor in front of the objective term in the Hessian, $sigma_f$.
- `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. **??**eq:IpoptLAG). 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;
}
```

TODO: User STOP method here?

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)
```

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.
  - `DIVERGING_ITERATES`: It seems that the iterates diverge.
  - `RESTORATION_FAILURE`: Restoration phase failed, algorithm doesn't know how to proceed.
  - `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, $f(x_*)$.

TODO: Should be provide IpStatistics here?

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`), 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.

```
#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)
  SmartPtr<IpoptApplication> app = new IpoptApplication();

  // Change some options
  app->Options()->SetNumericValue("tol", 1e-9);
  app->Options()->SetStringValue("mu_strategy", "adaptive");

  // Ask Ipopt to solve the problem
  ApplicationReturnStatus 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`. 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, including the IPOPT library `libipopt.a` (and 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 `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,

```
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   1.7159878e+01 2.01e-02 5.20e-01  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0 y
    1   1.7146308e+01 1.63e-01 1.47e-01  -1.0 1.15e-01    -  9.86e-01 1.00e+00f  1
    2   1.7065508e+01 3.10e-02 8.47e-02  -1.7 1.99e-01    -  9.54e-01 1.00e+00h  1 Nhj
    3   1.7002626e+01 4.10e-02 4.81e-03  -2.5 5.52e-02    -  1.00e+00 1.00e+00h  1
    4   1.7019082e+01 1.20e-03 1.81e-04  -2.5 1.10e-02    -  1.00e+00 1.00e+00h  1
    5   1.7014253e+01 1.80e-04 4.87e-05  -3.8 4.86e-03    -  1.00e+00 1.00e+00h  1
    6   1.7014020e+01 9.25e-07 2.15e-07  -5.7 2.76e-04    -  1.00e+00 1.00e+00h  1
    7   1.7014017e+01 1.01e-10 2.60e-11  -8.6 3.32e-06    -  1.00e+00 1.00e+00h  1

Number of Iterations....: 7

                                   (scaled)                 (unscaled)
Objective...............:   1.7014017145177885e+01    1.7014017145177885e+01
Dual infeasibility......:   2.5980210027546616e-11    2.5980210027546616e-11
Constraint violation....:   1.8175683180743363e-11    1.8175683180743363e-11
Complementarity.........:   2.5282956951655172e-09    2.5282956951655172e-09
Overall NLP error.......:   2.5282956951655172e-09    2.5282956951655172e-09


Number of objective function evaluations             = 8
Number of objective gradient evaluations             = 8
Number of equality constraint evaluations            = 8
Number of inequality constraint evaluations          = 8
Number of equality constraint Jacobian evaluations   = 8
Number of inequality constraint Jacobian evaluations = 8
Number of Lagrangian Hessian evaluations             = 9

EXIT: Optimal Solution Found.


Solution of the primal variables, x
x[0] = 1
x[1] = 4.743
x[2] = 3.82115
x[3] = 1.37941


Solution of the bound multipliers, z_L and z_U
z_L[0] = 1.08787
z_L[1] = 6.69317e-10
z_L[2] = 8.8877e-10
z_L[3] = 6.57011e-09
z_U[0] = 6.26262e-10
z_U[1] = 9.78906e-09
z_U[2] = 2.12283e-09
z_U[3] = 6.92528e-10


Objective value
f(x*) = 17.014


*** 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 `Examples/ScalableProblems` directory contains another set of NLP problems coded in C++.

3.4 The C Interface

The C interface for IPOPT is declared in the header file `IpStdCInterface.h`, which is found in `$IPOPTDIR/include/ipopt` (or in `$PREFIX/include/ipopt` 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`[10] with the function `CreateIpoptProblem`, which later has to be passed to the `IpoptSolve` function.

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

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

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

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

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

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

```
enum ApplicationReturnStatus status; /* Solve return code */
Number* x = NULL;                    /* starting point and solution vector */
Number* mult_x_L = NULL;             /* lower bound multipliers
at the solution */
Number* mult_x_U = NULL;             /* upper bound multipliers
at the solution */
Number obj;                          /* objective value */
Index i;                             /* generic counter */

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

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

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

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

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

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

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

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

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

  printf("\n\nSolution of the bound multipliers, z_L and z_U\n");
  for (i=0; i<n; i++) {
```

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

    printf("\n\nObjective value\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 `PARAMS.DAT` 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/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

4.2 Quasi-Newton Approximation of Second Derivatives

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 `PARAMS.DAT` file in the directory you are executing IPOPT.

The `PARAMS.DAT` 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 `PARAMS.DAT` 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 "nlp_scaling_method=none mu_init=1e-2 max_iter=500"
```

is a valid options command in AMPL. The most common options are referenced in Appendix C. These are also the options that are available through AMPL using the `options` command **TODO: CHECK IF THAT IS CORRECT**. To specify other options when using AMPL, you can always create `PARAMS.DAT`. Note, the `PARAMS.DAT` 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 `PARAMS.DAT`.

For a short list of the valid options, see the Appendix C. You can print the documentation for all IPOPT options by adding 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 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.

`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.

`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.

`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.

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[11].

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 triangual part (or, alternatively, the upper right triangular part). For

[11] 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 `PARAMS.DAT`, through your own code, or through the AMPL `options` command. See Section 5 for an explanation of how to use these commands. Shown here is a short list of the most common options for Ipopt. To view the full list of options, run the ipopt executable with the option,

```
print_options_documentation yes
```

The most common options are:

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` ≤ 10 and its default value is 3.

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 "cmpl_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. The valid range for this real option is $0 <$ `tol` $<$ `+inf` and its default value is $1 \cdot 10^{-08}$.

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

dual_inf_tol: Desired threshold for the dual infeasibility.
Absolute tolerance on the dual infesaibility. Successful termination requires that the (unscaled) dual infeasibility is less than this threshold. The valid range for this real option is $0 <$ `dual_inf_tol` $<$ `+inf` and its default value is 0.0001.

constr_mult_init_max: Maximum allowed least-square guess of constraint multipliers.
Determines how large the initial least-square guesses of the contraint 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.

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

pivtol: Pivot tolerance for the linear solver MA27.
A smaller number pivots for sparsity, a larger number pivots for stability. The valid range for this real option is $0 <$ `pivtol` < 1 and its default value is $1 \cdot 10^{-08}$.

pivtolmax: Maximum pivot tolerance.
Ipopt may increase pivtol as high as pivtolmax to get a more accurate solution to the linear system. The valid range for this real option is $0 <$ `pivtolmax` < 1 and its default value is 0.0001.

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_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_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 "probing".
Possible values:

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

corrector_type: The type of corrector steps that should be taken.
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

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 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

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.

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 $0 <$ `bound_frac` ≤ 0.5 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.

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.

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 implmentation paper.) The valid range for this real option is $0 \leq$ `bound_relax_factor` $<$ `+inf` and its default value is $1 \cdot 10^{-08}$.

acceptable_compl_inf_tol: Acceptance threshold for the complementarity conditions.
Absolute tolerance on the complementarity. Acceptable termination requires that 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_constr_viol_tol: Acceptance threshold for the constraint violation.
Absolute tolerance on the constraint violation. Acceptable termination requires that 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 infesaibility. Acceptable termination requires that 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 0.01.

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

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
- 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
- safe_min_dual_infeas: like "min_dual_infeas", but safeguarded by "min" and "max"

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 qucikly than usually, and more reduction in the constraint violation is enforced. 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

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

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.

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

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.

output_file: File name of desired output file (leave unset for no file output).
NOTE: This option only works when read from the PARAMS.DAT 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", or but can be overridden with "file_print_level". 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 PARAMS.DAT 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` ≤ 10 and its default value is 3.

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`. 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[12]) explicitly, using the `--with-blas` flag for `configure`. For example,

  ```
  ./configure --with=blas="-L$HOME/lib -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 its location with the `--with-hsl` flag. And the location of the AMPL solver library (with the ASL header files) can be specified with `--with-asldir`. **TODO Other linear solvers**

- 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++ CC=gcc F77=g77
  ```

  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).

- 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

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

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

- To build library/archive files (with the ending `.a`) including C++ code in some environments, it is necessary to use the C++ compiler instead of `ar` to build the archive. This is for example the case for some older compilers on SGI and SUN. For this, the `configure` variables `AR`, `ARFLAGS`, and `AR_X` are provided. Here, `AR` specifies the command for the archiver for creating an archive, and `ARFLAGS` specifies additional flags. `AR_X` contains the command for extracting all files from an archive. For example, the default setting for SUN compilers for our configure script is

  `AR='CC -xar' ARFLAGS='-o' AR_X='ar x'`

- 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), and additional debug checks are compiled into the code (see `IpDebug.hpp`). 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/Ipopt/configure (or $HOME/ipopt-3.1.0/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.