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

Yoshiaki Kawajiri*, Carl D. Laird†

August 23, 2005

Contents

*Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA, 15213, Email: kawajiri@cmu.edu
†Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA, 15213, Email: kawajiri@cmu.edu

1 Overview

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

$$\min_x \quad f(x) \tag{1}$$

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

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

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 are assumed to 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 $g_i^L = g_i^U = \bar{g}_i$.

Ipopt implements an interior point line search filter method. The mathematical details of the algorithm can be found in the reports [2] [3].

The Ipopt package is available from COIN-OR[1] under the open-source CPL license and includes the complete source code for Ipopt. The Ipopt distribution generates an executable for different modeling environments, including AMPL. As well, you can link your problem statement with Ipopt using interfaces for Fortran, C, or C++. Ipopt can be used with most Linux/Unix environments, and on Windows using Visual Studio .NET or Cygwin. The purpose of this document is to demonstrate 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. General questions related to Ipopt should be addressed to Ipopt mailing list (`http://list.coin-or.org/mailman/listinfo/coin-ipopt`). You might want to look at the archives before posting a question.

2 History of Ipopt

The original Ipopt (Fortran version) was a product of the dissertation research of Andreas Wächter [3], 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.

This document is a guide to using Ipopt 3.0 (the new C++ version of Ipopt).

3 Getting the Code

Ipopt is available from the COIN-OR subversion repository. You can either download the code using svn (the subversion client similar to cvs) or simply retrieve the tarball (at the writing of this document, the tarballs were not available, therefore, directions only include use of svn). 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.

To use subversion, follow the steps below:

1. Create a directory to store the code
   `$ mkdir Ipopt`
   Note the `$` indicates the command line prompt, do not type `$`, only the text following it.

2. Download the code to the new directory
   `$ cd Ipopt; svn co https://www.coin-or.org/svn/ipopt-devel/trunk`

3.1 Download External Code

Ipopt uses a few external packages that are not included in the distribution, namely ASL (the Ampl Solver Library), BLAS, and some routines from the Harwell Subroutine Library.

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 yout optimization problem in a programming language (C++, C, or Fortran).

3.1.1 Download BLAS and ASL

If you have the download utility `wget` installed on your system, retrieving ASL, and BLAS is straightforward using scripts included with the ipopt distribution. These scripts download the required files from the Netlib Repository (http://www.netlib.org).

```
$ cd trunk/ipopt/Extern/blas; ./get.blas
$ cd ../ASL; ./get.asl
```

If you don't have `wget` installed on your system, please read the `INSTALL.*` files in the `trunk/ipopt/Extern/blas` and `trunk/ipopt/Extern/ASL` directories for alternative instructions.

3.1.2 Download HSL Subroutines

In addition to the IPOPT source code, two additional subroutines have to be downloaded from the Harwell Subroutine Library (HSL). The required routines are freely available for non-commercial, academic use, but it is your responsibility to investigate the licensing of all third party code.

1. Go to `http://hsl.rl.ac.uk/archive/hslarchive.html`
2. Follow the instruction on the website, 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 `ipopt/trunk/Extern/HSL/`
   Note: Some browsers append a file extension (`.txt`) when you save the file, so you may 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 `ipopt/trunk/Extern/HSL/`
    Note: Some browsers append a file extension (`.txt`) when you save the file, so you may have to rename it.

4 Compiling and Installing Ipopt

Ipopt can be easily compiled and installed with the usual `configure`, `make`, `make install` commands.

1. Go to the main directory of Ipopt:
   `$ cd; cd ipopt/trunk`
2. Run the configure script
   `$ ./configure`
   The default configure (without any options) is sufficient for most users. If you want to see the configure options, run `./configure --help`.
3. Build the code
   `$ make`
4. Install Ipopt
   `$ make install`
   This installs the the ipopt library and necessary header files to ?...?
5. Test the installation
   `$ make test`
   This should ?...?

5 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 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) executable. 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,

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

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

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

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

with the starting point,

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

and the optimal solution,

$$x^{\star}=(1.00000000, 4.74299963, 3.82114998, 1.37940829).$$

6 Tutorial Example: Using the AMPL interface

Interfacing through the AMPL interface 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 [**?**].

The problem presented in equations (4-8) can be solved with Ipopt with the following AMPL mod file.

6.0.3 hs071_ampl.mod

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

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

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

# specify the constraints
s.t.

        inequality:
                x1 * x2 * x3 * x4 >= 25;

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

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

# solve the problem
solve;

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

The line, "`option solver ipopt;`" tells ampl to use ipopt as the solver. The ipopt executable (installed in Section 4) must be in the path for AMPL to find it. The remaining lines specify the problem in AMPL format. The problem can now be solved by starting ampl and loading the mod file.

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

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

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

7 Tutorial Example: 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 1. The problem dimensions and bounds are straightforward and come solely from the problem definition. The initial starting point is used by the algorithm when it begins iterating to solve the problem. If Ipopt has difficulty converging, or if it converges to a locally infeasible point, adjusting the starting point may help.

Providing the problem structure 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, a 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 nonzeros and the structure (row and column indices of each of the nonzeros) of the Jacobian and the Hessian. Once defined, this nonzero structure MUST remain constant for the entire problem. 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.

Figure 1: Information Required By Ipopt

1. Problem dimensions
   - number of variables
   - number of constraints
2. Problem bounds
   - variable bounds
   - constraint bounds
3. Initial starting point
   - Initial values for the primal x variables
   - Initial values for the multipliers (only required for a warm start option)
4. Problem Structure
   - number of nonzeros in the Jacobian of the constraints
   - number of nonzeros in the Hessian of the Lagrangian
   - Structure of the Jacobian of the constraints
   - Structure of the Hessian of the Lagrangian
5. Evaluation of Problem Functions
   Information evaluated using a given point (x_k, λ_k, σ_f coming from Ipopt)
   - Objective function, $f(x_k)$
   - Gradient of the objective $\nabla f(x_k)$
   - Constraint residuals, $g(x_k)$
   - Jacobian of the constraints, $\nabla g(x_k)$
   - Hessian of the Lagrangian, $\sigma_f \nabla^2 f(x_k) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x_k)$

As Ipopt iterates, it will need the values for the items in (5) 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 is given by

$$\begin{bmatrix} x_1x_4 + x_4(x_1 + x_2 + x_3) \\ x_1x_4 \\ x_1x_4 + 1 \\ x_1(x_1 + x_2 + x_3) \end{bmatrix}, \tag{9}$$

the Jacobian of the constraints is,

$$\begin{bmatrix} x_2x_3x_4 & x_1x_3x_4 & x_1x_2x_4 & x_1x_2x_3 \\ 2x_1 & 2x_2 & 2x_3 & 2x_4 \end{bmatrix}. \tag{10}$$

We need to determine the Hessian of the Lagrangian. The Lagrangian is given by $f(x) + g(x)^T\lambda$ and the Hessian of the Lagrangian 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. The value for σ_f is generally 1, although it may include scaling factors or even be set to zero to retrieve the Hessian of the constraints alone.

For our implementation then, the symbolic form of the Hessian of the Lagrangian (with the σ_f parameter) is,

$$\sigma_f \begin{bmatrix} 2x_4 & x_4 & x_4 & 2x_1 + x_2 + x_3 \\ x_4 & 0 & 0 & x_1 \\ x_4 & 0 & 0 & x_1 \\ 2x_1 + x_2 + x_3 & x_1 & x_1 & 0 \end{bmatrix} + \lambda_1 \begin{bmatrix} 0 & x_3x_4 & x_2x_4 & x_2x_3 \\ x_3x_4 & 0 & x_1x_4 & x_1x_3 \\ x_2x_4 & x_1x_4 & 0 & x_1x_2 \\ x_2x_3 & x_1x_3 & x_1x_2 & 0 \end{bmatrix} + \lambda_2 \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \tag{11}$$

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

The remainder of this section 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 `Ipopt/trunk/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 executable (`main`) and the problem representation. Generally, you will write an executable that prepares the problem, and then passes control over to Ipopt through an `Optimize` call. In this `Optimize` call, you will give Ipopt everything that it requires to call back to your code whenever it needs functions evaluated (like the objective, the Jacobian, 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.

7.1 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.

7.1.1 Coding the Problem Representation

We provide the information required in Figure 1 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 `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.

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`).
- `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 style used for row/col entries in the sparse matrix format (C_STYLE: 0-based, FORTRAN_STYLE: 1-based).

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 two constraints (m). The Jacobian for this small problem is actually dense and has 8 nonzeros (we will still represent this Jacobian using the sparse matrix triplet format). The Hessian of the Lagrangian has 10 "symmetric" nonzeros. 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;
}
```

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

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` (data member of `TNLP`) 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 **??**).

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;
}
```

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`).
- `init_lambda`: (in), if true, this method must provide an initial value for the constraint multipliers, `lambda`.
- `lambda`: (out), the initial values for the constraint multipliers, `lambda`.

The index variables n, and m are passed in only to help your debugging. 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, and the equality 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, or λ respectively. The default options only require an initial value for the primal variables `x`.

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;
}
```

virtual bool eval_f(Index n, const Number* x, bool new_x, Number& obj_value)
Return the value of the objective function as calculated using `x`.

- `n`: (in), the number of variables in the problem (dimension of `x`).
- `x`: (in), the current values for the primal variables, `x`.
- `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 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 index variable n is passed in only to help your debugging. 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;
}
```

virtual bool eval_grad_f(Index n, const Number* x, bool new_x, Number* grad_f)
Return the gradient of the objective to Ipopt, as calculated by the values in `x`.

- `n`: (in), the number of variables in the problem (dimension of `x`).
- `x`: (in), the current values for the primal variables, `x`.
- `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 used the same x values. This can be helpful when users have efficient implementations that caclulate multiple outputs at once. Ipopt internally caches results from the `TNLP` and generally, this flag can be ignored.

The index variable n is passed in only to help your debugging. 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;
}
```

virtual bool eval_g(Index n, const Number* x, bool new_x, Index m, Number* g)
Give Ipopt the value of the constraints as calculated by the values in `x`.

- `n`: (in), the number of variables in the problem (dimension of `x`).
- `x`: (in), the current values for the primal variables, `x`.
- `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`).

- g: (out) the array of constraint residuals.

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 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 index variables n, and m are passed in only to help your debugging. 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 for the gradient of the objective.

```
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;
}
```

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 structure of the Jacobian of the constraints, or the values for the Jacobian of the constraints as calculated by the values in x.

- n: (in), the number of variables in the problem (dimension of x).
- x: (in), the current values for the primal variables, x.
- 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).
- n_elel_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 i with respect to variable 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 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 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 used the same x values. This can be helpful when users have efficient implementations that caclulate multiple outputs at once. Ipopt internally caches results from the `TNLP` and generally, this flag can be ignored.

The index variables `n`, `m`, and `nele_jac` are passed in only to help your debugging. 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;
}
```

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 the structure of the Hessian of the Lagrangian or the values of the Hessian of the Lagrangian as calculated by the values in `obj_factor`, `x`, and `lambda`.

- `n`: (in), the number of variables in the problem (dimension of `x`).
- `x`: (in), the current values for the primal variables, `x`.
- `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.
- `m`: (in), the number of constraints in the problem (dimension of `g`).
- `lambda`: (in), the current values of the equality multipliers to use for each constraint in the evaluation of the Hessian.
- `n_elel_jac`: (in), the number of nonzero elements in the Jacobian (dimension of `iRow`, `jCol`, and `values`).
- `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.

If the `iRow` and `jCol` arguments are not NULL, then Ipopt wants you to fill in the structure of the Hessian (the row and column indices 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 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 used the same values. This can be helpful when users have efficient implementations that caclulate multiple outputs at once. Ipopt internally caches results from the `TNLP` and generally, this flag can be ignored.

The index variables `n`, `m`, and `nele_hess` are passed in only to help your debugging. 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;
}
```

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 1. 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.
  - `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 ??? options).
  - `LOCAL_INFEASIBILITY`: Algorithm converged to a point of local infeasibility. Problem may be infeasible.
  - `RESTORATION_FAILURE`: Restoration phase was called, but failed to find a more feasible point.
  - `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 current values for the primal variables, `x`.
- `z_L`: (in), the current values for the lower bound multipliers.
- `z_U`: (in), the current values for the upper bound multipliers.
- `m`: (in), the number of constraints in the problem (dimension of `g`).
- `g`: (in), the current value of the constraint residuals.
- `lambda`: (in), the current values of the equality multipliers.
- `obj_value`: (in), the current value of the objective.

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

7.1.2 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.

7.1.3 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 (and other necessary libraries). 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 **??** which explains the standard console output of Ipopt, Section **??** to learn about adjusting the output produced from Ipopt through the use of the Journalist, and Section **??** for information about the use of options to customize the behavior of Ipopt.

7.2 The C Interface

Have a look at `IpStdCInterface.h` to see the declarations for the C interface. For the C++ interface, all problem information is provided through the NLP class. In the C interface, we instead create an IpoptProblem (C structure) and pass that structure to the IpoptSolve call.

The IpoptProblem structure contains the problem dimensions, the variable and constraint bounds, and the function pointers for callbacks that will be used to evaluate the NLP. We then make the call to IpoptSolve, giving Ipopt the IpoptProblem structure, the starting point, and arrays to store the solution values, if desired.

A completed version of this example can be found in `Examples/hs071_c`. We first create the necessary callback functions for evaluating the NLP. We require callbacks to evaluate the objective value, constraints, gradient of the objective, Jacobian of the constraints, and the Hessian of the Lagrangian. These callbacks are implemented using function pointers (see `IpStdCInterface.h`). Have a look at the documentation for `eval_f`, `eval_g`, `eval_grad_f`, `eval_jac_g`, and `eval_h` in Section 7.1.1. The C implementations will have different prototypes, but will be implemented almost identically to the C++ code.

Create a new directory `MyCExample` and create a new file, `hs071_c.c`. Here, include the interface header file `IpStdCInterface.h`, along with `malloc.h` and `assert.h`. Add the prototypes and implementations for the five callback functions. See the completed example in `Examples/hs071_c/hs071_c.c`.

We now need to implement the `main` function, create the `IpoptProblem`, 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 below.

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

  n=4;
  x_L = (Number*)malloc(sizeof(Number)*n);
  x_U = (Number*)malloc(sizeof(Number)*n);
  for (i=0; i<n; i++) {
    x_L[i] = 1.0;
    x_U[i] = 5.0;
  }

  m=2;
  g_L = (Number*)malloc(sizeof(Number)*m);
  g_U = (Number*)malloc(sizeof(Number)*m);
  g_L[0] = 25; g_U[0] = 2e19;
  g_L[1] = 40; g_U[1] = 40;

  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);


  x = (Number*)malloc(sizeof(Number)*n);
  x[0] = 1.0;
  x[1] = 5.0;
  x[2] = 5.0;
  x[3] = 1.0;

  AddIpoptNumOption(nlp, "tol", 1e-9);
  AddIpoptStrOption(nlp, "mu_strategy", "adaptive");

  status = IpoptSolve(nlp, x, NULL, &obj, NULL, NULL, NULL, NULL);

  FreeIpoptProblem(nlp);
  free(x_L);
```

```
  free(x_U);
  free(g_L);
  free(g_U);
  free(x);

  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). 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 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. The second constraint is an equality, 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 can use these function pointers to ask for evaluation of the NLP when 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. 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 variable to return the solution 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 residuals at the solution or the equality 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 `Interfaces/ReturnCodes_inc.h`.

After the problem has solved, we check the status and print the solution if successful. Finally, we free the memory and return from `main`.

8 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. Note, the PARAMS.DAT file is given preference when setting options. This way, you can easily override any options set in a particular executable by creating 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 executable, for instance). This will output all of the options documentation to the console.

Acknowledgement

The initial version of this document was created by Yoshiaki Kawajir 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 authors of Ipopt greatly thank Yoshi for his efforts.

References

[1] http://www.coin-or.org

[2] Wächter, A. and Biegler, L.T.:"On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming", Research Report, IBM T. J. Watson Research Center, Yorktown, USA (2004)

[3] Wächter, A.:"An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering", Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, USA (2002)

[4] Margot, F.: Course material for *47852 Open Source Software for Optimization*, Carnegie Mellon University (2005)

A Triplet Format for Sparse Matrices

Ipopt was designed for optimizing large sparse nonlinear programs. Because of problem sparsity, the required matrices (like the Jacobian or Hessian) are not stored as traditional dense matrices, but rather in a sparse matrix format. For the tutorials in this document, we use the triplet format. Consider the matrix,

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

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[1]

In triplet format used in these Ipopt interfaces, the row and column numbers are 1-based **have table for both 0- and 1-based**, and the above matrix is represented by

row	*col*	*value*
1	1	1.1
1	7	0.5
2	2	1.9
2	7	0.5
3	3	2.6
3	7	0.5
4	3	7.8
4	4	0.6
5	4	1.5
5	5	2.7
6	1	1.6
6	5	0.4
7	6	0.9
7	7	1.7

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.

MISSING: Symmetric matrices

[1] 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.

B The Smart Pointer Implementation: SmartPtr

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 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

```
* print_options_documentation   ("no")
   Switch to print list all algorithmic options
     If selected, the algorithm will print the list of all available
     algorithmic options with some documentation before solving the
     optimization problem.
   Possible values:
    - no                      [don't print list]
    - yes                     [print list]


### Convergence ###

*tol                                    0 <  (      1e-08) <  +inf
   Desired convergence tolerance (relative).
     Determines the convergence tolerance for the algorthim.  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).  [Some other algorithmic features also
     use this quantity.]

* max_iter                                0 <= (       3000) <  +inf
   Maximum number of iterations.
     The algorithm terminates with an error message if the number of
     iterations exceeded this number. [Also used in RestoFilterConvCheck]
```

```
* dual_inf_tol                          0 <  (     0.0001) <  +inf
   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.

* constr_viol_tol                       0 <  (     0.0001) <  +inf
   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.

* compl_inf_tol                         0 <  (     0.0001) <  +inf
   Desired threshold for the complementarity conditions.
     Absolute tolerance on the complementarity. Successful termination
     requires that the (unscaled) complementarity is less than this threshold.

* acceptable_tol                        0 <  (      1e-06) <  +inf
   Acceptable convergence tolerance (relative).
     Determines which (scaled) overall optimality error is considered to be
     acceptable. If the algorithm encounters "acceptable_iter" iterations in a
     row that are considered "acceptable", it will terminate before the
     desired convergence tolerance ("tol", "dual_inf_tol", etc) is met.

* acceptable_dual_inf_tol               0 <  (       0.01) <  +inf
   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.

* acceptable_constr_viol_tol            0 <  (       0.01) <  +inf
   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.

* acceptable_compl_inf_tol              0 <  (       0.01) <  +inf
   Acceptance threshold for the complementarity conditions.
     Absolute tolerance on the complementarity. Acceptable termination
     requires that the (unscaled) complementarity is less than this threshold.

### NLP Scaling ###

* nlp_scaling_method            ("gradient_based")
   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)
   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]

* obj_scaling_factor                  -inf <  (          1) <  +inf
   Scaling factor for the objective function.
     This option allows to set a scaling factor for the objective function
     that is used to scale the problem that is seen internally by Ipopt. If
     additional scaling parameters are computed (e.g. user-scaling or
     gradient-based), this factor is multiplied in addition. If this value is
     chosen to be negative, Ipopt will maximize the objective function.

* nlp_scaling_max_gradient               0 <  (        100) <  +inf
   maximum gradient after 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 will scale the maximum gradient back to this value. Note: This
     option is only used if "nlp_scaling_method" is chosen as "gradient_based".

### Mu Update ###

* mu_strategy                    ("monotone")
   Update strategy for barrier parameter.
     Determines which barrier parameter strategy is to be used.
   Possible values:
    - monotone                [use the monotone (Fiacco-McCormick) strategy]
    - adaptive                [use the adaptive update strategy]

* mu_oracle                      ("probing")
   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".
   Possible values:
    - probing                 [Mehrotra's probing heuristic]
    - loqo                    [LOQO's centrality rule]
    - quality_function        [minimize a quality function]

* mu_init                                0 <  (        0.1) <  +inf
   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"

### Line Search ###

* max_soc                                 0 <= (          4) <  +inf
   Maximal number of second order correction trial steps.
     Determines the maximal number of second order correction trial steps that
     should be performed.  Choosing 0 disables the second order corrections.
     (This is p^{max} of Step A-5.9 of Algorithm A in implementation paper.)

* corrector_type                (("none")
   Type of corrector steps.
     Determines what kind of corrector steps should be tried.
   Possible values:
    - none                     [no corrector]
    - affine                   [corrector step towards mu=0]
    - primal-dual              [corrector step towards current mu]

* alpha_for_y                    (("primal")
   Step size for constraint multipliers.
     Determines which step size (alpha_y) should be used to update the
     constraint multipliers.
   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     (("no")
   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 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 is enabled automatically.
   Possible values:
    - no                       [the problem probably be feasible]
    - yes                      [the problem has a good chance to be infeasible]

### Initialization ###

* bound_push                              0 <  (       0.01) <  +inf
```

```
   Desired minimal absolute distance of initial point to bound
     Determines (together with "bound_frac") by how much the initial point
     might have to be modified in order to be sufficiently inside the bounds.

* bound_frac                             0 <  (       0.01) <= 0.5
   Desired minimal relative distance of initial point to bound
     Determines (together with "bound_push") by how much the initial point
     might have to be modified in order to be sufficiently inside the bounds.

* constr_mult_init_max                   0 <= (       1000) <  +inf
   Maximal allowed least-square guess of constraint multipliers.
     Determines how large the initial least-square guesses of the contraint
     multipliers (in max-norm) are allowed to be. If the guess is larger than
     this value, it is discarded and all constraint multipliers are set to
     zero.  This options is also used in the classes
     "RestoIterateInitializer".  By default, "resto.constr_mult_init_max" (the
     one used in RestoIterateInitializer) is set to zero.

* bound_mult_init_val                    0 <  (          1) <  +inf
   Initial value for the bound multipliers
     All dual variables corresponding to bound constraints are initialized to
     this value.

### Step Calculation ###

* min_refinement_steps                   0 <= (          1) <  +inf
   Minimum number of iterative refinement steps per linear system solve.
     Iterative refinement (on the unsymmetric full system) is performed for
     each right hand side.  This option determines the minimal number of
     iterative refinements, i.e., by setting it to 1 at least one iterative
     refinement step per right hand side is enforce.

* max_refinement_steps                   0 <= (         10) <  +inf
   Maximal number of iterative refinement steps per linear system solve.
     Iterative refinement (on the unsymmetric full system) is performed for
     each right hand side.  This option determines the maximal number of
     iterative refinements.


### NLP ###

* bound_relax_factor                     0 <= (      1e-08) <  +inf
   Factor for initial relaxation of the bounds.
     Before start of the optimization, the bounds given by the user are
     relaxed.  This option determines the factor by how much.  If it is set to
     zero, then this option is disabled.  (See Eqn.(35) in implmentation
```

paper.)

MA27 Linear Solver

* pivtol 0 < (1e-08) < 1
  pivot tolerance for the linear solver. smaller number - pivot for sparsity,
  larger number - pivot for stability

* pivtolmax 0 < (0.0001) < 1
  maximum pivot tolerance. IPOPT may increase pivtol as high as pivtolmax to
  get a more accurate solution to the linear system