Short Tutorial: Getting Started With Ipopt in 90 Minutes

Andreas Wächter[1]

IBM T.J. Watson Research Center
Department of Business Analytics and Mathematical Sciences
1101 Kitchawan Road, Yorktown Heights, NY 10598, USA
andreasw@us.ibm.com

Abstract. IPOPT is an open-source software package for large-scale nonlinear optimization. This tutorial gives a short introduction that should allow the reader to install and test the package on a UNIX-like system, and to run simple examples in a short period of time.

Keywords. Nonlinear Optimization, Tutorial, Ipopt, COIN-OR

1 Introduction

IPOPT is an open-source software package for large-scale nonlinear optimization. It can be used to address general nonlinear programming problems of the form

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

$$\text{s.t.} \quad g^L \le g(x) \le g^U \tag{1b}$$

$$x^L \le x \le x^U, \tag{1c}$$

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

Such optimization problems arise in a number of important engineering, financial, scientific, and medical applications, ranging from the optimal control of industrial processes (e.g., [1]) and the design of digital circuits (e.g., [2]) to portfolio optimization (e.g., [3]), from parameter identification in systems biology (e.g., [4]) to hyperthermia cancer treatment planning (e.g., [5]).

IPOPT implements an interior-point line-search filter method; the mathematical details of the algorithm can be found in several publications [6,7,8,9,10]. This approach makes IPOPT particularly suitable for large problems with up

Dagstuhl Seminar Proceedings 09061
Combinatorial Scientific Computing
http://drops.dagstuhl.de/opus/volltexte/2009/2089

to millions of variables and constraints, assuming that the Jacobian matrix of constraint function is sparse, but also small and dense problems can be solved efficiently. It is important to keep in mind that the algorithm is only trying to find a *local* minimizer of the problem; if the problem is nonconvex, many stationary points with different objective function values might exist, and it depends on the starting point and algorithmic choices which particular one the method converges to.

In general, the computational effort during the optimization with IPOPT is typically concentrated in the solution of linear systems, or in the computation of the problem functions and their derivatives, depending on the particular application. With respect to both these tasks, research in Combinatorial Scientific Computing is of central importance: The KKT system is a saddle point problem; see [11] for a survey of the recent developments in this area, as well as progress in weighted graph matchings [12,13,14] and parallel partitioning tools, such as ParMetis[15], Scotch[16], and Zoltan[17]. Furthermore, the computation of derivatives can be facilitated by automatic differentiation tools, such as ADIFOR[18], ADOL-C[19] and OPENAD[20].

Optimization problems can be given to IPOPT either by using a modeling language, such AMPL[1] or GAMS[2], which allow one to specify the mathematical problem formulation in an easily readable text format, or by writing programming code (C++, C, Fortran 77, Matlab) that computes the problem functions $f(x)$ and $g(x)$ and their derivatives.

This document provides only a short introduction to the IPOPT package. Detailed information can be found on the IPOPT home page www.coin-or.org/Ipopt and from the IPOPT documentation available with the source code[3]. The instruction here are for Linux (or UNIX-like systems, including Cygwin); if you want to use IPOPT with the Windows Developer Studio instructions see the IPOPT documentation.

The remainder of this tutorial is structured as follows: After presenting a motivating challenging example application in Section 2, easy-to-follow installation instruction are provided in Section 3, which allow the reader to start experimenting with the code in Section 4. Section 5 gives some mathematical background of the underlying algorithm, providing the basis to explain the output (Section 6) and algorithmic options (Section 7). Some advise regarding good modeling practices is given in Section 8. Finally, Section 9 discusses how IPOPT can be used from programming code, and a coding exercise is provided for the interested reader.

2 Example Application: PDE-Constrained Optimization

One class of optimization problems that give rise to very large and sparse nonlinear optimization problems is the optimization of models described by partial-

[1] www.ampl.com
[2] www.gams.com
[3] Also available online: www.coin-or.org/Ipopt/documentation

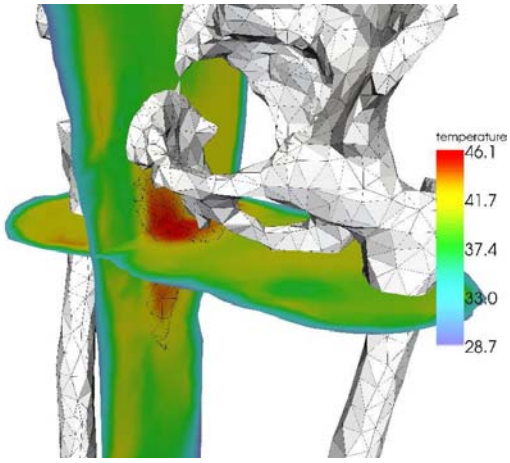

Fig. 1. Heated Tumor (heated in red) In Hyperthermia Treatment Planning.

differential equations (PDEs). Here, the "unknowns" are functions defined on a subset of $\mathbb{R}^2$ or $\mathbb{R}^3$ that are required to satisfy one or more PDEs, such as a temperature profile obeying a heat transfer equation. The degrees of freedom stem from a finite set of control parameters (e.g., intensities of a small number of microwave antennas) or from another function defined over in the full domain of the PDE or its boundary (e.g., material properties within the body conducting the heat, or a controllable temperature profile at the boundary).

There are a number of ways to tackle such problems. Ignoring many subtle details, we consider here the reformulation of the originally infinite-dimensional problem into a discretized finite-dimensional version. In this process, the domain of the PDE is discretized into a finite number of well-distributed points, and the new optimization variables are the values of the original functions at those points. Furthermore, the original PDE is then expressed as a set of constraints for each such point, where the differential operators are approximated based on the function values at neighboring points, e.g., by means of finite differences. In this way, each of those constraints involves only a small number of variables, leading to sparse derivative matrices. At the same time, the approximation error made by discretizing the original problem is reduced by increasing the number of discretization points, and it is therefore very desirable to solve large instances of the reformulation. This makes this application very suitable for an interior-point algorithm such as Ipopt.

A specific medical example application is hyperthermia treatment planning: Here, a patient is exposed to microwave radiation, emitted by several antennas, in order to heat a tumor; see Figure 1. The purpose of this procedure is to increase the tumor's susceptibility to regular radiation or chemotherapy.

Mathematically, the problem can be formulated as this PDE-constrained optimization problem:

$$\min \int_{x\in\Omega_t} (\boldsymbol{T}-\boldsymbol{T}_{ther})^2 d\Omega + \int_{x\notin\Omega_t} (\boldsymbol{T}-\boldsymbol{T}_{health})^2 d\Omega \tag{2a}$$

$$s.t. -\nabla\cdot(\kappa\nabla\boldsymbol{T}) + c_b\omega(\boldsymbol{T}-\boldsymbol{T}_b) = \frac{\sigma}{2}\left|\sum_i u_i E_i\right|^2 \quad \text{in } \Omega \tag{2b}$$

$$\kappa\partial_n\boldsymbol{T} = q_{const} \quad \text{on } \partial\Omega \tag{2c}$$

$$\boldsymbol{T}|_{\Omega/\Omega_t} \le \boldsymbol{T}_{lim}. \tag{2d}$$

Here, Ω is the considered region of the patient's body, $\Omega_t \subset \Omega$ the domain of tumor tissue, and $\boldsymbol{T}$ the temperature profile. The constant κ is the heat diffusion coefficient, c_b the specific heat of blood, $w(T)$ the temperature-dependent perfusion, $\boldsymbol{T}_b$ the arterial blood temperature, q_{const} the human vessel blood flux, σ the electrical conductivity, u_i the complex-valued control of antenna i, and E_i the corresponding electrical field. The PDE is given in (2b) with a Neumann boundary condition (2c). The goal of the optimization is to find optimal controls u_i in order to minimize the deviation from the desired temperature inside and outside the tumor ($\boldsymbol{T}_{ther}$ and $\boldsymbol{T}_{health}$, respectively), as expressed in the objective function (2a). To avoid excessively high temperature in the healthy tissue, the bound constraint (2d) is explicitly included.

Christen and Schenk[21] recently solved an instance of this problem with real patient data and 12 antennas, resulting in a discretized NLP with $n = 854,499$ variables. The KKT matrix (see (6) below), which is factorized to compute the optimization steps, had more than 27 million nonzero elements. This is a considerably large nonlinear optimization problem, which required 182 Ipopt iterations and took 48 hours on a 8-core Xeon Linux workstation, using the parallel linear solver Pardiso[4].

3 Installation

The Ipopt project is hosted by the COIN-OR Foundation[5], which provides a repository with a number of different operations-research related open-source software packages. The Ipopt source code is distributed under the Common Public License (CPL) and can be used for commercial purposes (check the license for details).

In order to compile the Ipopt package, you will need to obtain the Ipopt source code from the COIN-OR repository. This can be done by downloading a compressed archive[6] or preferably by using the subversion[7] repository management tool, `svn`; the latter option allows for a convenient upgrade of the source

[4] www.pardiso-project.org
[5] www.coin-or.org
[6] look for the latest version in www.coin-or.org/download/source/Ipopt
[7] subversion.tigris.org

```
1  > cd $MY_IPOPT_DIR
2  > svn co https://project.coin-or.org/svn/Ipopt/stable/X.Y Ipopt-source
3  > cd Ipopt-source/ThirdParty/Blas
4  > ./get.Blas
5  > cd ../Lapack
6  > ./get.Lapack
7  > cd ../Mumps
8  > ./get.Mumps
9  > cd ../ASL
10 > ./get.ASL
11
12 > cd $MY_IPOPT_DIR
13 > mkdir build
14 > cd build
15 > ../Ipopt-source/configure
16 > make
17 > make test
18 > make install
```

Fig. 2. Installing IPOPT (Basic Version)

code to newer versions. In addition, IPOPT requires some third-party packages, namely

- BLAS (Basic Linear Algebra Subroutines)
- LAPACK (Linear Algebra PACKage)
- A symmetric indefinite linear solver (currently, interfaces are available to the solvers MA27, MA57, MUMPS, Pardiso, WSMP)
- ASL (AMPL Solver Library)

Refer to the IPOPT documentation for details of the different components.

Assuming that the shell variable `MY_IPOPT_DIR` contains the name of a directory that you just created for your new IPOPT installation, Figure 2 lists the commands to obtain a basic version of all the required source code and to compile the code; you will need to replace the string "`X.Y`" with the current stable release number which you find on the IPOPT homepage (e.g., "`3.6`"). The command in the second line downloads the IPOPT source files, including documentation and examples, to a new subdirectory, `Ipopt-source`. The commands in lines 3–10 visit several subdirectories and run provided scripts for downloading third-party source code components[8]. The commands in lines 12–15 run the configuration script, in a directory separate from the source code (this allows you to compile different versions, such as in optimized and debug mode, and to start over easily); make sure it says "`configure: Main configuration of`

[8] Note that it is your responsibility to make sure you have the legal right to use this code; read the `INSTALL.*` files in the `ThirdParty` subdirectories.

`Ipopt successful`" at the end. Finally, the last three lines compile the IPOPT code, try to execute it (check the test output for complaints!), and finally install it. The AMPL solver executable "`ipopt`" will be in the `bin` subdirectory, the library in `lib` and the header files under `include/ipopt`.

Note that this will give you only a basic version of IPOPT, sufficient for small to medium-size problems. For better performance, you should consider getting optimized BLAS and LAPACK libraries, and choose a linear solver that fits your application. Also, there are several options for the `configure` script that allow you to choose different compilers, compiler options, provide precompiled third-party libraries, etc.; check the documentation for details.

4 First Experiments

Now that you have compiled and installed IPOPT, you might want to play with it for a little while before we go into some algorithmic details and the different ways to use the software.

The easiest way to run IPOPT with example problems is in connection with a modeling language, such as AMPL. For this, you need to have a version of the AMPL language interpreter (an executable called "`ampl`"). If you do not have one already, you can download a free student version from the AMPL website[9]; this version has a limit on the problem size (max. 300 variables and constraints), but it is sufficient for getting familiar with IPOPT and solving small problems. Make sure that both the `ampl` executable, as well as the `ipopt` solver executable that you just compiled (located in the `bin` subdirectory) are in your shell's `PATH`.

Now you can go to the subdirectory `Ipopt/tutorial/AmplExperiments` of `$MY_IPOPT_DIR/build`. Here you find a few simple AMPL model files (with the extension `.mod`). Have a look at such a file; for example, `hs71.mod` formulates Problem 71 of the standard Hock-Schittkowski collection[22]:

$$\begin{aligned} \min_{x\in\mathbb{R}^4} \quad & x^{(1)}x^{(4)}\left(x^{(1)}+x^{(2)}+x^{(3)}\right) \\ s.t. \quad & x^{(1)}x^{(2)}x^{(3)}x^{(4)} \geq 25 \\ & (x^{(1)})^2+(x^{(2)})^2+(x^{(3)})^2+(x^{(4)})^2 = 40 \\ & 1 \leq x \leq 5 \end{aligned}$$

Even though explaining the AMPL language[23] is beyond the scope of this tutorial, you will see that it is not difficult to understand such a model file since the syntax is quite intuitive.

In order to solve an AMPL model with IPOPT, start the AMPL interpreter by typing "`ampl`" in the directory where the model files are located. Figure 3 depicts a typical AMPL session: The first line, which has to be entered only once per AMPL session, tells AMPL to use IPOPT as the solver. Lines 2–3 load a model file (replace the string "`FILE`" with the correct file name, such as `hs71`). The

[9] www.ampl.com/DOWNLOADS

```
1 ampl: option solver ipopt;
2 ampl: reset;
3 ampl: model FILE.mod;
4 ampl: solve;
5         [...IPOPT output...]
6 ampl: display x;
```

Fig. 3. A typical AMPL session

fourth line runs IPOPT, and you will see the output of the optimizer, including the `EXIT` message (hopefully "`Optimal Solution Found`"). Finally, you can use the `display` command to examine the final values of the variables (replace "`x`" by the name of a variable in the model file). You can repeat steps 2–6 for different or modified model files. If AMPL complains about syntax error and shows you the prompt "`ampl?`", you need to enter a semicolon (`;`) and start over from line 2. Now you have all the information to find out which of the components in the final solution for `hs71.mod` is not correct in $x_* = (1, 5, 3.82115, 1.37941)\ldots$?

You can continue exploring on your own, using and modifying the example model files; AMPL knows the standard intrinsic functions (such as `sin`, `log`, `exp`). If you are looking for more AMPL examples, have a look at the `MoreAmplModels.txt` file in this directory which has a few URLs for website with more NLP AMPL models.

5 The Algorithm

In this section we present informally some details of the algorithm implemented in IPOPT. The main goal is to convey enough information to explain the output of the software and some of the algorithmic options available to a user. Rigorous mathematical details can be found in the publications cited in the Introduction.

Internally, IPOPT replaces any inequality constraint in (1b) by an equality constraint and a new bounded slack variable (e.g., $g_i(x) - s_i = 0$ with $g_i^L \leq s_i \leq g_i^U$), so that bound constraints are the only inequality constraints. To further simplify the notation in this section, we assume here that all variables have only lower bounds of zero, so that the problem is given as

$$\min_{x \in \mathbb{R}^n} \quad f(x) \tag{3a}$$

$$\text{s.t.} \quad c(x) = 0 \tag{3b}$$

$$x \geq 0. \tag{3c}$$

As an interior point (or "barrier") method, IPOPT considers the auxiliary barrier problem formulation

$$\min_{x\in\mathbb{R}^n} \quad \varphi_\mu(x) = f(x) - \mu\sum_{i=1}^{n}\ln(x_i) \tag{4a}$$

$$\text{s.t.} \quad c(x) = 0, \tag{4b}$$

where the bound constraints (3c) are replaced by a logarithmic barrier term which is added to the objective function. Given a value for the barrier parameter $\mu > 0$, the barrier objective function $\varphi_\mu(x)$ goes to infinity if any of the variables x_i approach their bound zero. Therefore, an optimal solution of (4) will be in the interior of the region defined by (3c). The amount of influence of the barrier term depends on the size of the barrier parameter μ, and one can show, under certain standard conditions, that the optimal solutions $x_*(\mu)$ of (4) converge to an optimal solution of the original problem (3) as $\mu \to 0$. Therefore, the overall solution strategy is to solve a sequence of barrier problems (4): Starting with a moderate value of μ (e.g., 0.1) and a user-supplied starting point, the corresponding barrier problem (4) is solved to a relaxed accuracy; then μ is decreased and the next problem is solved to a somewhat tighter accuracy, using the previous approximate solution as a starting point. This is repeated until a solution for (3), or at least a point satisfying the first-order optimality conditions up to user tolerances, has been found. These first-order optimality conditions for the barrier problem (4) are given as

$$\nabla f(x) + \nabla c(x)y - z = 0 \tag{5a}$$

$$c(x) = 0 \tag{5b}$$

$$XZe - \mu e = 0 \tag{5c}$$

$$x, z \geq 0 \tag{5d}$$

with $\mu = 0$, where $y \in \mathbb{R}^m$ and $z \in \mathbb{R}^n$ are the Lagrangian multipliers for the equality and bound constraints, respectively. Furthermore, we introduced the notation $X = \mathrm{Diag}(x)$, $Z = \mathrm{Diag}(z)$ and $e = (1, \ldots, 1)^T$.

It is important to note that not only minimizers for (3), but also some maximizers and saddle points satisfy (5), and that there is no guarantee that IPOPT will always converge to a local minimizer of (3). However, the Hessian regularization described below aims to encourage the algorithm to avoid maximizers and saddle points.

It remains to describe how the approximate solution of (4) for a fixed $\bar{\mu}$ is computed. The first-order optimality conditions for (4) are given by (5) with $\mu = \bar{\mu}$, and a Newton-type algorithm is applied to the nonlinear system of equations (5a)–(5c) to generate a converging sequence of iterates that always strictly satisfy (5d). Given an iterate (x_k, y_k, z_k) with $x_k, z_k > 0$, the Newton step $(\Delta x_k, \Delta y_k, \Delta z_k)$ for (5a)–(5c) is computed from

$$\begin{bmatrix} W_k + X_k^{-1}Z_k + \delta_x I & \nabla c(x_k) \\ \nabla c(x_k)^T & 0 \end{bmatrix} \begin{pmatrix} \Delta x_k \\ \Delta y_k \end{pmatrix} = -\begin{pmatrix} \nabla\varphi_\mu(x_k) + \nabla c(x_k)y_k \\ c(x_k) \end{pmatrix} \tag{6}$$

with $\delta_x = 0$ and $\Delta z_k = \mu X_k^{-1} e - z_k - X_k^{-1} Z_k \Delta x_k$. Here, W_k is the Hessian of the Lagrangian function, i.e.,

$$W_k = \nabla^2 f(x_k) + \sum_{j=1}^{m} y_k^{(j)} \nabla^2 c^{(j)}(x_k). \tag{7}$$

After the Newton step has been computed, the algorithm first computes maximum step sizes as the largest $\alpha_k^{x,\max}, \alpha_k^{z,\max} \in (0, 1]$ satisfying

$$x_k + \alpha_k^{x,\max} \Delta x_k \geq (1-\tau) x_k, \qquad z_k + \alpha_k^{z,\max} \Delta z_k \geq (1-\tau) z_k$$

with $\tau = \min\{0.99, \mu\}$; this fraction-to-the-boundary rule ensures that the new iterate will again strictly satisfy (5d). Then a line search with trial step sizes $\alpha_{k,l}^x = 2^{-l} \alpha_k^{x,\max}$, $l = 0, 1, \ldots$ is performed, until a step size $\alpha_{k,l}^x$ is found that results in "sufficient progress" (see below) toward solving the barrier problem (4) compared to the current iterate. Finally, the new iterate is obtained by setting

$$x_{k+1} = x_k + \alpha_{k,l}^x \Delta x_k, \quad y_{k+1} = y_k + \alpha_{k,l}^x \Delta y_k, \quad z_{k+1} = z_k + \alpha_k^{z,\max} \Delta z_k$$

and the next iteration is started. Once the optimality conditions (5) for the barrier problem are sufficiently satisfied, μ is decreased.

In order to decide if a trial point is acceptable as new iterate, Ipopt uses a "filter" method to decide if progress has been made toward the solution of the barrier problem (4). Here, a trial point is deemed better than the current iterate, if it (sufficiently) decreases either the objective function $\varphi_\mu(x)$ or the norm of the constraint violation $\|c(x)\|_1$. In addition, similar progress has to be made also with respect to a list of some previous iterates (the "filter") in order to avoid cycling. Overall, under standard assumptions this procedure can be shown to make the algorithm globally convergent (i.e., at least one limit point is a stationary point for (4)).

Two crucial points are important in connection with the line search: First, in order to guarantee that the direction Δx_k obtained from (6) is indeed a descent direction (e.g., resulting in a degrees of $\varphi_\mu(x)$ at a feasible iterate), the projection of the upper left block of the matrix in (6) onto the null space of $\nabla c(x_k)^T$ should be positive definite, or, equivalently, the matrix in (6) should have exactly n positive and m negative eigenvalues. The latter condition can easily be verified after a symmetric indefinite factorization, and if the number of negative eigenvalues is too large, Ipopt perturbs the matrix by choosing a $\delta_x > 0$ by a trial-and-error heuristic, until the inertia of this matrix is as desired.

Secondly, it may happen that no acceptable step size can be found. In this case, the algorithm switches to a "feasibility restoration phase" which temporarily ignores the objective function, and instead solves a different optimization problem to minimize the constraint violation $\|c(x)\|_1$ (in a way that tries to find the feasible point closest to the point at which the restoration phase was started). The outcome of this is either that the a point is found that allows the return to the regular Ipopt algorithm solving (4), or a local minimizer of the ℓ_1-norm constraint violation is obtained, indicating to a user that the problem is (locally) infeasible.

Table 1. IPOPT iteration output

col #	header	meaning
1	`iter`	iteration counter k (`r`: restoration phase)
2	`objective`	current value of objective function, $f(x_k)$
3	`inf_pr`	current primal infeasibility (max-norm), $\|c(x_k)\|_\infty$
4	`inf_du`	current dual infeasibility (max-norm), $\|$Eqn. (5*a*)$\|_\infty$
5	`lg(mu)`	$\log_{10}$ of current barrier parameter μ
6	`||d||`	max-norm of the primal search direction, $\|\Delta x_k\|_\infty$
7	`lg(rg)`	$\log_{10}$ of Hessian perturbation δ_x (`--`: none, $\delta_x = 0$)
8	`alpha_du`	dual step size $\alpha_k^{z,\max}$
9	`alpha_pr`	primal step size α_k^x
10	`ls`	number of backtracking steps $l + 1$

6 IPOPT Output

When you ran the AMPL examples in Section 4, you already saw the IPOPT output: After a short notice about the IPOPT project, self-explanatory information is printed regarding the size of the problem that is solved. Then IPOPT displays some intermediate information for each iteration of the optimization procedure, and closes with some statistics concerning the computational effort and finally the `EXIT` message, indicating whether the optimization terminated successfully.

Table 1 lists the columns of the iteration output. The first column is simply the iteration counter, whereby an appended "`r`" indicates that the algorithm is currently in the restoration phase; the iteration counter is not reset after an update of the barrier parameter or when the algorithm switched between the restoration phase and the regular algorithm. The next two columns indicate the value of the objective function (not the barrier function, $\varphi_\mu(x_k)$) and constraint violation. Note that these are not quite the quantities used in the filter line search, and that therefore you might sometimes see an increase in both from one iteration to the next. The fourth column is a measure of optimality; remember that IPOPT aims to find a point satisfying the optimality conditions (5). The last column gives an indication how many trial points had to be evaluated.

In a typical optimization run, you would want to see that the objective function is going to the optimal value, and the constraint violation and the dual infeasibility, as well as the size of the primal search direction are going to zero in the end. Also, as the value of the barrier parameter is going to zero, you will see a decrease in the number listed in column 5; if the algorithm switches to the restoration phase a different value of μ might be chosen. Furthermore, the larger the step sizes in columns 8 and 9, the better is the usually the progress. Finally, a nonzero value of δ_x indicates that the iterates seem to be in a region where the original problem is not strictly convex. If you see nonzero values even at the very end of the optimization, it might indicate that the algorithm terminated at a critical point that is not a minimizer (but still satisfies (5) up to some tolerance).

7 IPOPT Options

There is a large number of options that can be set by a user to change the algorithmic behavior and other aspects of IPOPT. Most of them are described in the "IPOPT Options" section of the IPOPT documentation. It is also possible to see a list of the options by running the AMPL solver executable as "`ipopt --print-options | more`." Options can be set in an options file (called `ipopt.opt`) residing in the directory where IPOPT is run. The format is simply one option name per line, followed by the chosen value; anything in a line after a hash ("`#`") is treated as a comment and ignored. A subset of options can also be set directly from AMPL, using

```
option ipopt_options "option1=value1 option2=value2 ...";
```

in the AMPL script before running the solver. To see which particular IPOPT options can be set in that way, type "`ipopt -=`" in the command line.

An effort has been made to choose robust and efficient default values for all options, but if the algorithm fails to converge or speed is important, it is worthwhile to experiment with different choices. In this section, we mention only a few specific options that might be helpful:

- *Termination:* In general, IPOPT terminates successfully if the optimality error, a scaled variant of (5), is below the value of `tol`. The precise definition of the termination test is given in [8, Eqn. (5)]. Note that it is possible to control the components of the optimality error individually (using `dual_inf_tol`, `constr_viol_tol`, and `compl_inf_tol`). Furthermore, in some cases it might be difficult to satisfy this "desired" termination tolerance (due to numerical issues), and IPOPT might terminate then at a point satisfying looser criteria that can be controlled with the "`acceptable_*_tol`" options. Finally, IPOPT will stop if the maximum number of iterations (default 3000) is exceeded; you can change this using `max_iter`.
- *Output:* The amount of output written to standard output is determined by the value of `print_level` with default value 5. To switch off all output, choose 0; to see even the individual values in the KKT matrix, choose 12. If you want look at detailed output, it is best to create an output file with the `output_file` option together with `file_print_level`.
- *Initialization:* IPOPT will begin the optimization by setting the x components of the iterates to the user-supplied starting point (AMPL will assume that 0 is the initial point of a variable unless it is explicitly set!). However, as an interior point method, IPOPT requires that all variables lie strictly within their bounds, and therefore it modifies the user-supplied point if necessary to make sure none of the components are violating or are too close to their bounds. The options `bound_frac` and `bound_push` determine how much distance to the bounds is enforced, depending on the difference of the upper and lower bounds or the distance to just one of the bounds, respectively (see [8, Section 3.6]).
  The z values are all set to the value of `bound_mult_init_val`, and the initial y values are computed as those that minimize the 2-norm of Eqn. (5a) (see

also `constr_mult_init_max`). Furthermore, `mu_init` determines the initial value of the barrier parameter μ.

- *Problem modification:* IPOPT seems to perform better if the feasible set of the problem has a nonempty relative interior. Therefore, by default, IPOPT relaxes all bounds (including bounds on inequality constraints) by a very small amount (on the order of 10^{-8}) before the optimization is started. In some cases, this can lead to problems, and this features can be disabled by setting `bound_relax_factor` to 0.
  Furthermore, internally IPOPT might look at a problem in a scaled way: At the beginning, scaling factors are computed for all objective and constraint functions to ensure that none of their gradients is larger than `nlp_scaling_max_gradient`, unless a non-default option is chosen for `nlp_scaling_method`. The objective function can be handled separately, using `obj_scaling_factor`; it often it worthwhile experimenting with this last option.
- *Further options:* IPOPT's options are tailored to solving general nonlinear optimization problems. However, switching `mehrotra_algorithm` to `yes` might make IPOPT perform much better for linear programs (LPs) or convex quadratic programs (QPs) by choosing some default option differently. One such setting, `mu_strategy=adaptive`, might also work well in nonlinear circumstances; for many problems, this adaptive barrier strategy seems to reduce the number of iterations, but at a somewhat higher computational costs per iteration.
  If you are using AMPL, the modeling software computes first and second derivatives of the problem function $f(x)$ and $g(x)$ for you. However, if IPOPT is used within programming code, second derivatives (for the W_k matrix in (7)) might not be available. For this purpose, IPOPT includes the possibility to approximate this matrix by means of a limited-memory quasi-Newton method (`hessian_approximation=limited-memory`). This option makes the code usually less robust than if exact second derivatives are used. In addition, IPOPT's termination tests are not very good at determining in this situation when a solution is found, and you might have to experiment with the `acceptable_*_tol` options.

8 Modeling Issues

For good performance, an optimization problem should be scaled well. While it is difficult to say what that means in general, it seems advisable that the absolute values of the non-zero derivatives typically encountered are on the same order of magnitude. In contrast to linear programming where all derivative information is known at the start of the optimization and does not change, it is difficult for a nonlinear optimization algorithm to automatically determine good scaling factors, and the modeler should try to avoid formulations where some non-zero entries in the gradients are typically very small or very large. A rescaling can be done by multiplying an individual constraint function by some factor, and by replacing a variable x_i by $\tilde{x}_i = c \cdot x_i$ for a constant $c \neq 0$.

Linear problems are easier to solve than nonlinear problems, and convex problems are easier to solve than non-convex ones. Therefore, it makes sense to explore different, equivalent formulations to make it easier for the optimization method to find a solution. In some cases it is worth introducing extra variables and constraints if that leads to "fewer nonlinearities" or sparser derivative matrices.

An obvious example of better modeling is to use "$y = c \cdot x$" instead of "$y/x = c$" if c is a constant. But a number of other tricks are possible. As a demonstration consider the optimization problem

$$\begin{aligned} \min_{x,p\in\mathbb{R}^n} \quad & \sum_{i=1}^{n} x_i \\ \text{s.t.} \quad & \prod_{i=1}^{n} p_i \geq 0.1; \qquad \frac{x_i}{p_i} \geq \frac{i}{10n} \quad \forall_{i=1}^{n}; \qquad x \geq 0, \quad 0 \leq p \leq 1. \end{aligned}$$

Can you find a better formulation, leading to a convex problem with much sparser derivative matrices? (See `Ipopt/tutorial/Modeling/bad1*` for the solution)

Ipopt needs to evaluate the function values and their derivatives at all iterates and trial points. If a trial point is encountered where an expression cannot be calculated (e.g., the argument of a log is a non-positive number), the step is further reduced. But it is better to avoid such points in the model: As an interior point method, Ipopt keeps all variables within their bounds (possibly somewhat relaxed, see the `bound_relax_factor` option). Therefore, it can be useful to replace the argument of a function with a limited range of definition by a variable with appropriate bounds. For example, instead of "$\log(h(x))$", use "$\log(y)$" with a new variable $y \geq \epsilon$ (with a small constant $\epsilon > 0$) and a new constraint $h(x) - y = 0$.

9 Ipopt With Program Code

Rather than using a modeling language tool such as AMPL or GAMS, Ipopt can also be employed for optimization problem formulations that are implemented in a programming language, such as C, C++, Fortran or Matlab. For this purposes, a user must implement a number of functions/methods that provide to Ipopt the required information:

- Problem size [`get_nlp_info`] and bounds [`get_bounds_info`];
- Starting point [`get_starting_point`];
- Function values $f(x_k)$ [`eval_f`] and $g(x_k)$ [`eval_g`];
- First derivatives $\nabla f(x_k)$ [`eval_grad_f`] and $\nabla c(x_k)$ [`eval_jac_g`];
- Second derivatives $\sigma_f \nabla^2 f(x_k) + \sum_j \lambda_k^{(j)} \nabla^2 c^{(j)}(x_k)$ [`eval_h`];
- Do something with the solution [`finalize_solution`].

In square brackets are the names of the corresponding methods of the `TNLP` base class for the C++ interface, which is described in detail in the IPOPT documentation; the other interfaces are similar. Example code is provided in the IPOPT distribution. If you used the installation procedures described in Figure 2, you will find it in the subdirectories of `$MY_IPOPT_DIR/build/Ipopt/examples`, together with Makefiles tailored to your system.

Here is some quick-start information:

- For C++, the header file `include/ipopt/IpTNLP.hpp` contains the base class of a new class that you must implement. Examples are in the `Cpp_example`, `hs071_cpp` and `ScalableProblems` subdirectories.
- For C, the header file `include/ipopt/IpStdCInterface.h` has the prototypes for the IPOPT functions and the callback functions. An example is in the `hs071_c` subdirectory.
- For Fortran, the function calls are very similar to the C interface, and example code is in `hs071_f`.
- For Matlab, you first have to compile and install the Matlab interface. To do this, go into the directory
  `$MY_IPOPT_DIR/build/Ipopt/contrib/MatlabInterface/src`
  and type "`make clean`" (there is no dependency check in that Makefile) followed by "`make install`". This will install the required Matlab files into `$MY_IPOPT_DIR/build/lib`, so you need to add that directory into your Matlab path. Example code is in
  `$MY_IPOPT_DIR/build/Ipopt/contrib/MatlabInterface/examples`
  together with a `startup.m` file that sets the correct Matlab path.

9.1 Programming Exercise

After you had a look at the example programs, you can try to solve the following coding exercise included in the IPOPT distribution. In `$MY_IPOPT_DIR/build/Ipopt/tutorial/CodingExercise` you find an AMPL model file for the scalable example problem

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & \sum_{i=1}^{n}(x_i-1)^2 \\ s.t. \quad & (x_i^2+1.5x_i-a_i)\cos(x_{i+1})-x_{i-1}=0 \qquad \text{for} \quad i=2,\ldots,n-1 \\ & -1.5\le x\le 0 \end{aligned}$$

and subdirectories for each modeling language. In each case, you find directories

- `1-skeleton`: This has the main code structure, but no code for function evaluation, etc. This code will not compile or run;
- `2-mistake`: This has a running version of the code with all functions implemented. However, mistakes are purposely included;
- `3-solution`: This has a correct implementation of the exercise.

To do the exercise, copy the content of the `1-skeleton` directory to a new directory as a starting point for your implementation.

This is a scalable example with $n \geq 3$; it will be easier to start debugging and checking the results with a small instance. The AMPL model can help to determine the correct solution for a given n, so that it is easy to verify if your code is correct. A very useful tool is also IPOPT's derivative checker; see the `derivative_test` option and the corresponding section in the IPOPT documentation. Also, for C++ and C, a memory checker (such as `valgrind` for Linux) comes in very handy.

The following is a suggested procedure to tackle the exercise; it refers to method names in the C++ interface, but also applies to the other programming languages.

1. Work out derivatives, including the sparsity structure.
2. Implement
   - problem information (size, bounds, starting point): `get_nlp_info`, `get_bounds_info`, `get_starting_point`;
   - code for the objective function value and its gradient: `eval_f`, `eval_grad_f`;
   - code for the constraint values and the (dense) Jacobian structure: `eval_g`, `eval_jac_g` (structure only - in this first step, pretend it is dense to make it easier).
3. Run a small instance (e.g., $n = 5$) with the following options and check the solution:
   - `jacobian_approximation=finite-difference-values`
   - `hessian_approximation=limited-memory`
   - `tol=1e-5`.
4. Implement
   - code for constraint derivatives: `eval_jac_g` (now with correct sparsity pattern and values).
5. Check the first derivatives and verify the solution:
   - `derivative_test=first-order`
   - `hessian_approximation=limited-memory`
6. Implement
   - code for the Hessian of Lagrangian: `eval_h`
7. Check the second derivatives and verify the solution:
   - `derivative_test=second-order`

If you want to take a short-cut, you may start with the code in `2-mistake` and look for the mistakes. Here, using the derivative checker will be very helpful. Also, until the derivatives are correct, there is no point in running IPOPT for many iterations, so you want to set `max_iter` to 1.

10 Not Covered

There are a number of features and options not covered in this short tutorial. The IPOPT documentation describes some of them, and there is also additional information on the IPOPT home page. Furthermore, a mailing list is available to pose question regarding to the use of IPOPT, as well as the bug tracking system. Links to those resources are available at the IPOPT home page.

Acknowledgments

The author thanks Olaf Schenk and Jon Lee for their comments that improved the presentation of this tutorial.

References

1. Biegler, L.T., Zavala, V.M.: Large-scale nonlinear programming using IPOPT: An integrating framework for enterprise-wide dynamic optimization. Computers and Chemical Engineering **33** (2009) 575–582
2. Wächter, A., Visweswariah, C., Conn, A.R.: Large-scale nonlinear optimization in circuit tuning. Future Generation Computer Systems **21** (2005) 1251–1262
3. Gondzio, J., Grothey, A.: Solving non-linear portfolio optimization problems with the primal-dual interior point method. European Journal of Operational Research **181** (2007) 1019–1029
4. Lu, J., Muller, S., Machne, R., Flamm, C.: SBML ODE solver library: Extensions for inverse analysis. In: Proceedings of the Fifth International Workshop on Computational Systems Biology, WCSB. (2008)
5. Schenk, O., Wächter, A., Weiser, M.: Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. SIAM Journal on Scientific Computing **31** (2008) 939–960
6. Nocedal, J., Wächter, A., Waltz, R.A.: Adaptive barrier strategies for nonlinear interior methods. SIAM Journal on Optimization **19** (2009) 1674–1693
7. Wächter, A.: An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2002)
8. Wächter, A., Biegler, L.T.: On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. Mathematical Programming **106** (2006) 25–57
9. Wächter, A., Biegler, L.T.: Line search filter methods for nonlinear programming: Motivation and global convergence. SIAM Journal on Optimization **16** (2005) 1–31
10. Wächter, A., Biegler, L.T.: Line search filter methods for nonlinear programming: Local convergence. SIAM Journal on Optimization **16** (2005) 32–48
11. Benzi, M., Golub, G.H., Liesen, J.: Numerical solution of saddle point problems. Acta Numerica **14** (2005) 1–137
12. Olschowka, M., Neumaier, A.: A new pivoting strategy for Gaussian elimination. Linear Algebra and Its Applications **240** (1996) 131–151
13. Duff, I.S., Pralet, S.: Strategies for scaling and pivoting for sparse symmetric indefinite problems. SIAM Journal on Matrix Analysis and Applications **27** (2005) 313–340

14. Schenk, O., Gartner, K.: On fast factorization pivoting methods for sparse symmetric indefinite systems. Electronic Transactions on Numerical Analysis **23** (2006) 158–179
15. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning for irregular graphs. SIAM Review **41** (1999) 278–300
16. Pellegrini, F.: SCOTCH 5.0 Uses's guide. Technical report, LaBRI, Universite Bordeaux I (2007) Available at http://www.labri.fr/pelegrin/scotch.
17. Devine, K., Boman, E., Heaphy, R., Hendrickson, B., Vaughan, C.: Zoltan data management services for parallel dynamic applications. Computing in Science and Engineering **4** (2002) 90–96
18. Bischof, C., Carle, A., Hovland, P., Khademi, P., Mauer, A.: ADIFOR 2.0 User's Guide. Technical Report ANL/MCS-TM-192, Argonne National Laboratory, Argonne, IL, USA (1995) Available at http://www.mcs.anl.gov/research/projects/adifor.
19. Griewank, A., Juedes, D., Utke, J.: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. ACM Transactions on Mathematical Software **22** (1996) 131–167
20. Utke, J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., Hill, C., Wunsch, C.: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes. Technical Report AIB-2007-14, (Department of Computer Science, RWTH Aachen) Available at http://www.mcs.anl.gov/OpenAD.
21. Christen, M., Schenk, O.: (2009) Personal communication.
22. Hock, W., Schittkowski, K.: Test examples for nonlinear programming codes. Lecture Notes in Economics and Mathematical Systems **187** (1981)
23. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Modeling Language For Mathematical Programming. Thomson Publishing Company, Danvers, MA, USA (1993)