sIPOPT Reference Manual

Hans Pirnay, Rodrigo López-Negrete, and Lorenz T. Biegler
Chemical Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213

April 8, 2011

Contents

1 Introduction

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

The basic sensitivity strategy implemented here is based on the application of the Implicit Function Theorem (IFT) to the KKT conditions of the NLP. As shown in Fiacco [1], sensitivities can be obtained

from a solution with suitable regularity conditions merely by solving a linearization of the KKT conditions. In Pirnay *et al.* [2] we have extended these results to the barrier penalty method implemented in IPOPT. In the following subsections we have summarized the main concepts in the paper.

1.1 Barrier Sensitivity

Consider the parametric nonlinear program of the form:

$$\min_x \quad f(x;p) \tag{1a}$$
$$\text{s.t.} \quad c(x;p) = 0, x \geq 0 \tag{1b}$$

with the vectors $x \in \mathbb{R}^{n_x}$, $p \in \mathbb{R}^{n_p}$, and $c(x;p) : \mathbb{R}^{n_x+n_p} \to \mathbb{R}^m$. Without loss of generality, only the variables x have been assumed zero or positive. However, the following derivations can be extended to the case where there are both upper and lower bounds.

The IPOPT NLP algorithm substitutes a barrier function for the inequality constraints and solves the following sequence of problems with $\mu \to 0$:

$$\min_x \quad B(x;p,\mu) = f(x;p) - \mu_\ell \sum_{i=1}^{n_x} ln(x_i) \tag{2a}$$
$$\text{s.t.} \quad c(x;p) = 0 \tag{2b}$$

At a solution with $p = p_0$ (the nominal value) we compute the sensitivities $\frac{dx^*(p_0)}{dp}$ and $\frac{df(x^*;p_0)}{dp} = \frac{\partial f(x^*;p_0)}{\partial p} + \frac{dx(p_0)}{dp}\frac{\partial f(x^*;p_0)}{\partial x}$. To calculate these sensitivities, we first consider properties of the solutions of (1) obtained by IPOPT when $p = p_0$ [1, 3].

For NLP (1), the Karush-Kuhn-Tucker (KKT) conditions are defined as:

$$\nabla_x L(x^*, \lambda^*, \nu^*; p_0) = \nabla_x f(x^*; p_0) + \nabla_x c(x^*; p_0)\lambda^* - \nu^* = 0 \tag{3a}$$
$$c(x^*; p_0) = 0 \tag{3b}$$
$$0 \leq \nu^* \perp x^* \geq 0 \tag{3c}$$

For the KKT conditions to serve as necessary conditions for a local minimum of (1), constraint qualifications are needed, such as Linear Independence Constraint Qualification (LICQ) or Mangasarian-Fromowitz Constraint Qualification (MFCQ). Definitions of these regularity conditions may be found in Biegler [4], Nocedal and Wright [5], or Fiacco [1].

Calculation of the sensitivity of the primal and dual variables with respect to p now proceeds from the implicit function theorem (IFT) applied to the optimality conditions of (2) at p_0. Defining the quantities:

$$M(s(\mu;p_0)) = \begin{bmatrix} W\left(s\left(\mu;p_0\right)\right) & A\left(x\left(\mu;p_0\right)\right) & -I \\ A\left(x\left(\mu;p_0\right)\right)^T & 0 & 0 \\ V\left(\mu;p_0\right) & 0 & X(\mu;p_0) \end{bmatrix} \tag{4}$$

and

$$N_p(s(\mu;p_0)) = \begin{bmatrix} \nabla_{xp}L(s(\mu;p_0)) \\ \nabla_p c(x(\mu;p_0)) \\ 0 \end{bmatrix}, \quad N_\mu = \begin{bmatrix} 0 \\ 0 \\ -\mu e \end{bmatrix} \tag{5}$$

where $W(s(\mu;p_0))$ denotes the Hessian $\nabla_{xx}L(x,\lambda,\nu)$ of the Lagrangian function evaluated at $s(\mu;p_0)$, $A(x(\mu;p_0)) = \nabla_x c(x)$ evaluated at $x(\mu;p_0)$, $X = diag\{x\}$ and $V = diag\{\nu\}$, application of IFT leads to:

$$M(s(\mu;p_0))\frac{ds(\mu;p_0)}{dp}^T + N_p(s(\mu;p_0)) = 0. \tag{6}$$

Because LICQ, Strict Complementarity (SC), and SSOSC hold, $M(s(\mu;p_0))$ is nonsingular and the sensitivities can be calculated from:

$$\frac{ds(\mu;p_0)}{dp}^T = -M\left(s\left(\mu;p_0\right)\right)^{-1} N_p\left(s\left(\mu;p_0\right)\right). \tag{7}$$

Note, also, that $M(s(\mu;p_0))$ is directly available in factored form from the solution of (2), so the sensitivity can be calculated through a simple backsolve. For small values of μ and $\|p - p_0\|$ it can be shown from the above properties [1] that

$$s(\mu;p) = s(\mu;p_0) - M(s(\mu;p_0))^{-1}N_p(s(\mu;p_0))(p-p_0) + o\|p-p_0\|. \tag{8}$$

Finally, in the way IPOPT is implemented, it cannot distinguish between variables and parameters. Thus we can make this distinction apparent by adding some artificial variables and constraints. In this way we write:

$$\begin{aligned} \min_{x,w} \quad & f(x,w) && \text{(9a)} \\ \text{s.t.} \quad & c(x,w) = 0, x \geq 0 && \text{(9b)} \\ & w - p_0 = 0 && \text{(9c)} \end{aligned}$$

Note that the NLP solution is equivalent to (1), and it is easy to see that the NLP sensitivity is equivalent as well. Writing the KKT conditions for (9) leads to:

$$\nabla_x f(x,w) + \nabla_x c^T(x,w)\lambda - \nu = 0 \tag{10a}$$
$$\nabla_w f(x,w) + \nabla_w c^T(x,w)\lambda + \bar{\lambda} = 0 \tag{10b}$$
$$c(x) = 0 \tag{10c}$$
$$XVe = 0 \tag{10d}$$
$$w - p_0 = 0 \tag{10e}$$

In this definition $\bar{\lambda}$ represents the Lagrange multiplier corresponding to the equation $w - p = 0$. For the Newton step we write:

$$\begin{bmatrix} W & \nabla_{xw}L(x,w,\lambda,\nu) & A & -I & 0 \\ \nabla_{wx}L(x,w,\lambda,\nu) & \nabla_{ww}L(x,w,\lambda,\nu) & \nabla_w c(x,w) & 0 & I \\ A^T & \nabla_w c(x,w)^T & 0 & 0 & 0 \\ V & 0 & 0 & X & 0 \\ 0 & I & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta w \\ \Delta\lambda \\ \Delta\nu \\ \Delta\bar{\lambda} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \Delta p \end{bmatrix}. \tag{11}$$

Since $\Delta w = \Delta p$, the step computed by this matrix (without the second row) is the same as the optimal step stated in (6).

1.2 Multiple Sequential Parameter Perturbations

In the derivations in the previous sections we considered changes to the parameter vector. However, in some cases we may be interested in making multiple parameter perturbations in a sequential manner. For example we may want to perturb the current solution $s(\mu; p_0)$ using the parameter vectors $p_1, \ldots, p_{n_{\text{pert}}}$. This amounts to solving system (6) with different right hand sides $N_p(s(\mu; p_0))$ (Eq. (5)). Note that, because we already have (4) factorized at the solution, it is very cheap to obtain the n_{pert} sensitivities. With them and using Equation (8) we can determine the approximated solutions $s(\mu; p_1)$, …, $s\left(\mu; p_{n_{\text{pert}}}\right)$.

2 Usage

In the following sections we describe how the *sIPOPT* library can be used through the AMPL interface. However, we also provide examples for the C++ interface in the examples folder of the distribution. To help illustrate the use of *sIPOPT* the following NLP, taken from [6], will be used:

$$
\begin{aligned}
\min \quad & x_1^2 + x_2^2 + x_3^2 \\
\text{s.t.} \quad & 6x_1 + 3x_2 + 2x_3 - p_1 = 0 \\
& p_2 x_1 + x_2 - x_3 - 1 = 0 \\
& x_1, x_2, x_3 \geq 0,
\end{aligned} \tag{12}
$$

with variables x_1, x_2, and x_3 and parameters p_1, and p_2. Since IPOPT does not distinguish variables from parameters, we reformulate the NLP as (9) by introducing equations that fix the parameters p_1 and p_2 to their nominal values $p_{1,a}$ and $p_{2,a}$.

$$
\begin{aligned}
\min \quad & x_1^2 + x_2^2 + x_3^2 && \text{(13a)} \\
\text{s.t.} \quad & 6x_1 + 3x_2 + 2x_3 - p_1 = 0 && \text{(13b)} \\
& p_2 x_1 + x_2 - x_3 - 1 = 0 && \text{(13c)} \\
& p_1 = p_{1,a} && \text{(13d)} \\
& p_2 = p_{2,a} && \text{(13e)} \\
& x_1, x_2, x_3 \geq 0. && \text{(13f)}
\end{aligned}
$$

For (13), the KKT conditions are:

$$
\begin{aligned}
2x_1 + 6\lambda_1 + p_2\lambda_2 - \nu_1 &= 0 && \text{(14)} \\
2x_2 + 3\lambda_1 + \lambda_2 - \nu_2 &= 0 && \text{(15)} \\
2x_3 + 2\lambda_1 - \lambda_2 - \nu_3 &= 0 && \text{(16)} \\
-\lambda_1 + \lambda_3 &= 0 && \text{(17)} \\
\lambda_2 x_1 + \lambda_4 &= 0 && \text{(18)} \\
6x_1 + 3x_2 + 2x_3 - p_1 &= 0 && \text{(19)} \\
p_2 x_1 + x_2 - x_3 - 1 &= 0 && \text{(20)} \\
p_1 - p_{1,a} &= 0 && \text{(21)} \\
p_2 - p_{2,a} &= 0 && \text{(22)} \\
\nu_1 x_1 - \mu &= 0 && \text{(23)} \\
\nu_2 x_2 - \mu &= 0 && \text{(24)} \\
\nu_3 x_3 - \mu &= 0 && \text{(25)} \\
x_1, x_2, x_3, \nu_1, \nu_2, \nu_3 &\geq 0, && \text{(26)}
\end{aligned}
$$

and the corresponding Newton step is

$$\begin{bmatrix}
2 & & & & \lambda_2 & 6 & p_2 & & -1 & & & \\
 & 2 & & & & 3 & 1 & & & -1 & & \\
 & & 2 & & & 2 & -1 & & & & -1 & \\
 & & & & & -1 & & 1 & & & & \\
\lambda_2 & & & & & & x_1 & & 1 & & & \\
6 & 3 & 2 & -1 & & & & & & & & \\
p_2 & 1 & -1 & & x_1 & & & & & & & \\
 & & & 1 & & & & & & & & \\
 & & & & 1 & & & & & & & \\
\nu_1 & & & & & & & & x_1 & & & \\
 & \nu_2 & & & & & & & & x_2 & & \\
 & & \nu_3 & & & & & & & & x_3 &
\end{bmatrix}
\begin{bmatrix}
\Delta x_1 \\ \Delta x_2 \\ \Delta x_3 \\ \Delta p_1 \\ \Delta p_2 \\ \Delta \lambda_1 \\ \Delta \lambda_2 \\ \Delta \lambda_3 \\ \Delta \lambda_4 \\ \Delta \nu_1 \\ \Delta \nu_2 \\ \Delta \nu_3
\end{bmatrix}
= -
\begin{bmatrix}
2x_1^* + 6\lambda_1^* + p_2\lambda_2^* - \nu_1^* \\
2x_2^* + 3\lambda_1^* + \lambda_2^* - \nu_2^* \\
2x_3^* + 2\lambda_1^* - \lambda_2^* - \nu_3^* \\
-\lambda_1^* + \lambda_3^* \\
\lambda_2^* x_1^* + \lambda_4^* \\
6x_1^* + 3x_2^* + 2x_3^* - p_1^* \\
p_2^* x_1^* + x_2^* - x_3^* - 1 \\
p_1^* - p_{1,a} \\
p_2^* - p_{2,a} \\
\nu_1^* x_1^* - \mu \\
\nu_2^* x_2^* - \mu \\
\nu_3^* x_3^* - \mu
\end{bmatrix} \tag{27}$$

where the right hand side is zero at the solution.

2.1 AMPL Interface

In this section we will show how to use *sIPOPT* through the AMPL interface [7]. This is the preferred method for using IPOPT, because this allows us to take advantage of the exact first and second order derivatives provided by the modeling language. The first thing to do is to write the problem in the AMPL language as shown in code listing 1.

```
reset ;

# Define parameters
param et1p ;
param et2p ;

# Original parameter values
let et1p := 5 ;
let et2p := 1 ;

# Define variables, with bounds and initial guess
var x1 >= 0, := 0.15 ;
var x2 >= 0, := 0.15 ;
var x3 >= 0, := 0.00 ;

# objective function
```

```
minimize objf: x1^2 + x2^2 + x3^2 ;

# constraints
subject to

r1:     6*x1 + 3*x2 + 2*x3 - et1p = 0 ;
r2: et2p*x1 +   x2 -   x3 - 1     = 0 ;

# Define solver and Ampl options in this case we don't want Ampl's
# presolve to accidentally remove artificial variables.
options solver ipopt_sens ;
option presolve 0 ;

# Solve problem
solve ;
```

Code Listing 1: AMPL code for Problem 13.

We can now proceed to modify the above code to add the information needed to use *sIPOPT*. For this we need to create the following suffixes. These will be used to communicate the nominal and perturbed values of the parameters, and also some will serve as flags to indicate to IPOPT which are the artificial constraints that were added.

sens_state_0 This is used to enumerate the parameters that will be perturbed. It takes values from 1 to length(p), and the values may not be repeated. Note that the order of the values is crucial.

sens_state_1 This is similar to **sens_state_0**, but it now indicates the order for the parameters at the perturbed value. This suffix should have the same values as **sens_state_0**. It takes values from 1 to length(p), and the values may no be repeated.

sens_state_value_1 This is used to communicate the values of the perturbed parameters. It has to be set for the same variables as **sens_state_1**.

sens_init_constr This is a flag that indicates the constraint is artificial, e.g., $w - p = 0$ in Problem (10). If the constraint is artificial, set this suffix to 1 (no indexing is necessary).

Once these suffixes have been set, we must enable *sIPOPT* by setting the *run_sens* to '*yes*'. Note that this option can alternatively be set in the ipopt.opt file. In addition, to ensure that AMPL's presolve feature does not eliminate the initial value constraints, we disable it. Thus, the modified code is

```
reset ;
```

```
# Suffixes for sensitivity update
suffix sens_state_0, IN;
suffix sens_state_1, IN;
suffix sens_state_value_1, IN;
suffix sens_sol_state_1, OUT;
suffix sens_init_constr, IN;

# Original value of parameters
param et1p ;
param et2p ;

# Original parameter values
let et1p := 5 ;
let et2p := 1 ;

# Define variables, with bounds and initial guess
var x1 >= 0, := 0.15 ;
var x2 >= 0, := 0.15 ;
var x3 >= 0, := 0.00 ;

# Artificial variables so IPOPT sees the parameters
var et1 ;
var et2 ;

# objective function
minimize objf: x1^2 + x2^2 + x3^2 ;

# constraints
subject to

r1: 6*x1 + 3*x2 + 2*x3 - et1 = 0 ;
r2: et2*x1 + x2 - x3 - 1 = 0 ;

# Artificial constraints to pass parameters to IPOPT
r3: et1 = et1p ;
r4: et2 = et2p ;

# Define solver and Ampl options in this case we don't want Ampl's
```

```
# presolve to accidentally remove artificial variables.
options solver ipopt_sens ;
option presolve 0;

# define an order to the parameters that will change.
# In step 0, only et1 changes, and has position 1
let et1.sens_state_0 := 1 ;

# in the first step/change et1 has position 1
let et1.sens_state_1 := 1 ;

# Perturbed value of parameter et1 (in step 1)
let et1.sens_state_value_1 := 4.5 ;

# In step 0, et2 has position 1
let et2.sens_state_0 := 2 ;

# in the second step/change et1 has position 1
let et2.sens_state_1 := 2 ;

# Perturbed value of parameter et2 (in step 1)
let et2.sens_state_value_1 := 1 ;

# Artificial constraints
let r3.sens_init_constr := 1 ;
let r4.sens_init_constr := 1 ;

# solve problem
solve ;
```

Code Listing 2: AMPL code for sensitivity update of Problem 13.

After the algorithm has completed successfully, the perturbed solution is stored in the following AMPL suffixes:

sens_sol_state_1 This holds the updated variables, as well as, the updated constraint multiplier values computed in the sensitivity update.

sens_sol_state_1_z_L This suffix holds updated lower bound multipliers.

sens_sol_state_1_z_U This suffix holds updated upper bound multipliers.

For example we could append the following code to Listing 2 in order to print both the nominal solution, as well as the updated values.

```
#*********************************************
# Print nominal solution and bound multipliers
#*********************************************
display x1, x2, x3, et1, et2 ;
display x1.ipopt_zU_out, x2.ipopt_zU_out, x3.ipopt_zU_out,
        et1.ipopt_zU_out, et2.ipopt_zU_out ;

display x1.ipopt_zL_out, x2.ipopt_zL_out, x3.ipopt_zL_out,
        et1.ipopt_zL_out, et2.ipopt_zL_out ;

# Constraint multipliers
display r1, r2, r3, r4 ;

#************************
# Print updated solution
#************************
display x1.sens_sol_state_1, x2.sens_sol_state_1,
        x3.sens_sol_state_1, et1.sens_sol_state_1,
        et2.sens_sol_state_1 ;

display x1.sens_sol_state_1_z_U, x2.sens_sol_state_1_z_U,
        x3.sens_sol_state_1_z_U,
        et1.sens_sol_state_1_z_U, et2.sens_sol_state_1_z_U ;

display x1.sens_sol_state_1_z_L, x2.sens_sol_state_1_z_L,
        x3.sens_sol_state_1_z_L,
        et1.sens_sol_state_1_z_L, et2.sens_sol_state_1_z_L ;

# and updated constraint multipliers
display r1.sens_sol_state_1, r2.sens_sol_state_1,
        r3.sens_sol_state_1, r4.sens_sol_state_1 ;
```

Code Listing 3: AMPL code to print updated solution.

An example implementation of the above is provided in the directory:

`$IPOPT/Ipopt/contrib/sIPOPT/examples/parametric_ampl`.

3 Reduced Hessian

An important byproduct of the sensitivity calculation is information related to the Hessian of the Lagrange function pertinent to the second order conditions. At the solution of (1) we again consider the sensitivity system, $MS = N_{rh}$, and partition the variables into free and bounded variables, i.e., $x^* = [x_f^T \; x_b^T]$ where $x_f^* > 0, x_b^* = 0$. Assuming strict complementarity (SC), the IFT sensitivity system using (4) can be partitioned with:

$$M = \begin{bmatrix} W_{ff}(x^*, \lambda^*) & W_{fb}(x^*, \lambda^*) & A_f(x^*) & -I_f & 0 \\ W_{bf}(x^*, \lambda^*) & W_{bb}(x^*, \lambda^*) & A_b(x^*) & 0 & -I_b \\ A_f(x^*)^T & A_b(x^*))^T & 0 & 0 & 0 \\ 0 & 0 & 0 & X_f^* & 0 \\ 0 & V_b^* & 0 & 0 & 0 \end{bmatrix}, S = \begin{bmatrix} S_{x_f} \\ S_{x_b} \\ S_\lambda \\ S_{\nu_f} \\ S_{\nu_b} \end{bmatrix}, \text{ and } N_{rh} = \begin{bmatrix} E \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{28}$$

From (28) it is easy to see that $S_{x_b} = 0, S_{\nu_f} = 0$. These variables and the last two rows can therefore be removed, leading to:

$$\begin{bmatrix} W_{ff}(x^*, \lambda^*) & A_f(x^*) & 0 \\ A_f(x^*))^T & 0 & 0 \\ W_{bf}(x^*, \lambda^*) & A_b(x^*) & -I_b \end{bmatrix} \begin{bmatrix} S_{x_f} \\ S_\lambda \\ S_{\nu_b} \end{bmatrix} = \begin{bmatrix} E \\ 0 \\ 0 \end{bmatrix}$$

For a chosen set of $n_I \leq n_x - m$ independent variables with elements reordered at the end of the x vector, A_D nonsingular, $E^T = [0 \mid I_{n_I}]$ and the matrices defined in (28), the reduced Hessian can be found directly by solving $MS = N_{rh}$. As described in [2], the reduced Hessian can be extracted easily from the rows of S. Thus taking advantage of the implementation described in Section 1.1 for sensitivity based updates, we can obtain an approximation of the reduced Hessian via back solves involving the factorized KKT matrix.

4 Usage

In the following sections we describe the usage of the reduced Hessian calculator using the AMPL. We also provide examples of the C++ interface in the examples folder.

4.1 AMPL Interface

The usage of the reduced Hessian calculation is similar to the sensitivity updates described above. The critical step here is deciding which variables will be independent variables at the optimal solution. Theses independent variables are then identified with the suffix **red_hessian**.

This suffix provides an enumeration of the independent variables, thus it needs to take ordered values from $1..n_I$, where n_I is the number of independent variables. The columns of the inverse reduced Hessian will be printed to the screen, and their order is determined by the ordering of these indices.

To enable reduced Hessian calculations we need to set the option The algorithm is enabled by setting the solver option *compute_red_hessian* to '*yes*'. Using Example 1 defined by Problem (12), we illustrate the use of the reduced Hessian calculator. The code for this is shown in Listing 4.

```
reset ;

# Define reduced Hessian suffixes
suffix red_hessian, IN ;

# Define parameters
param et1 ;
param et2 ;

# Parameter values
let et1p := 5 ;
let et2p := 1 ;

# Define variables, with bounds and initial guess
var x1 >= 0, := 0.15 ;
var x2 >= 0, := 0.15 ;
var x3 >= 0, := 0.00 ;

# objective function
minimize objf: x1^2 + x2^2 + x3^2 ;

# constraints
subject to

r1:    6*x1 + 3*x2 + 2*x3 - et1p = 0 ;
r2: et2p*x1 +   x2 -   x3 - 1    = 0 ;

# Define solver and Ampl options in this case we don't want Ampl's
# presolve to accidentally remove artificial variables.
options solver ipopt_sens ;
option presolve 0 ;

# Define free variables
let x3.red_hess := 1 ;
```

```
# Solve problem
solve ;
```

Code Listing 4: AMPL code for Problem 13.

5 C++ Interface

The C++ interface is very simple to apply to an existing `Ipopt::TNLP` implementation. The member function `TNLP::::get_var_con_metadata` in Ipopt provides a feature very similar to that of AMPL suffixes.

The steps taken to make a TNLP class ready for using the *sIPOPT* code are similar to those used in AMPL. First, the parameter values are defined with artificial variables and constraints. Note that because of this the Jacobian and Hessian computations have to be adjusted accordingly. Finally, the suffixes need to be set the same way they would in AMPL as described above. This is done using member function `TNLP::::get_var_con_metadata`. This is illustrated in examples `examples/redhess_cpp` and `examples/parametric_cpp`.

6 Installation

The first step to install the software is to install IPOPT, once this is done installing *sIPOPT* is very simple. IPOPT's installation instructions can be found in `http://www.coin-or.org/Ipopt/documentation/`. Also note that in the following we refer to `$IPOPT` as the main folder, where the Ipopt, ThirdParty, BuildTools, ..., folders are located. If you wish to use the AMPL interface, make sure that your IPOPT installation also includes it. To do this you need to download the ASL library, with the `get.ASL` script located in `$IPOPT/ThirdParty/ASL`. Finally, we assume that you created a build folder to install IPOPT in `$IPOPT/build/`.

Once IPOPT has been compiled and installed, we can proceed to build *sIPOPT*. To do this go to the `$IPOPT/build/Ipopt/contrib/sIPOPT/` folder, and type `make` there.

```
$ cd $IPOPT/build/Ipopt/contrib/sIPOPT

$ make
```

If no errors are shown after compilation you can proceed to install the libraries and to generate the AMPL executable. To do this type

```
$ make install
```

This should copy the generated libraries (`libsipopt.*`) to `$IPOPT/build/lib`, and the AMPL executable (`ipopt_sens`) to `$IPOPT/build/bin/`.

7 Options

There are several new options that can be set in the `ipopt.opt` file, that determine the behavior of the *sIPOPT* code. The more important options are the ones enable the execution of the post-optimal *sIPOPT*code. These are

```
run_sens yes
```

to enable sensitivity computations, and

```
compute_red_hessian yes
```

to enable the computation of the reduced Hessian.

Other options are:

`n_sens_steps` In general, the update can be done sequentially for any number of parameters. The valid range for this integer option is $1 \leq$ `n_sens_steps` $\leq \infty$, and the default value is 1. Please see Section 1.2 for more details on this.

`sens_boundcheck` If set to `yes`, this option turns on the bound correction algorithm (see Section 2.4 in the implementation paper). The default value of this string option is `no`.

`sens_bound_eps` This option makes sure that only variables that violate the bound by more than `sens_bound_eps` are considered as real violations. Otherwise, the bound checking might continue until the full active set has been covered. This is only used if the `sens_boundcheck` is set to `yes`. The valid range of this real valued option is: $0 \leq$ `sens_bound_eps` $\leq \infty$, and the default is 10^{-3}.

`sens_max_pdpert` For certain problems, IPOPT uses inertia correction of the primal dual matrix to achieve better convergence properties. This inertia correction changes the matrix and renders it useless for the use with *sIPOPT*. This option sets an upper bound, which the inertia correction may have. If any of the inertia correction values is above this bound, the *sIPOPT* algorithm is aborted. The valid range of this real valued option is: $0 \leq$ `sens_max_pdpert` $\leq \infty$, and the default is 10^{-3}. Please see Section 2.2 of the IPOPT implementation paper [8], and Section **??** of the *sIPOPT* implementation paper [2] for more details.

`rh_eigendecomp` If this option is set to `yes`, the reduced Hessian code will compute the eigenvalue decomposition of the reduced Hessian matrix. The default value of this string option is `no`.

`sens_allow_inexact_backsolve` This option is used to enable or disable IPOPT's Iterative Refinement. See Section 3.10 of the IPOPT implementation paper [8]. By default this string option is set to `yes` (do not do iterative refinement), and it can take values of `yes` or `no`.

References

[1] Fiacco, A.V., *Introduction to Sensitivity and Stability Analysis in Nonlinear Programming*, volume 165 of *Mathematics in Science and Engineering*. Academic Press, **1983**.

[2] Pirnay, H.; López-Negrete, R.; and Biegler, L.T., Optimal Sensitivity Based on IPOPT, **2011**.

[3] Forsgren, A.; Gill, P.E.; and Wright, M.H., Interior Point Methods for Nonlinear Optimization. *SIAM Review* 44(4), **2002**, pp. 525–597.

[4] Biegler, L.T., *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*. SIAM, **2010**.

[5] Nocedal, J. and Wright, S., *Numerical Optimization*. Operations Research and Financial Engineering, 2nd edition, Springer, New York, **2006**.

[6] Ganesh, N. and Biegler, L.T., A reduced hessian strategy for sensitivity analysis of optimal flowsheets. *AIChE* 33, **1987**, pp. 282–296.

[7] Fourer, R.; Gay, D.M.; and Kernighan, B.W., *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, Pacific Grove, **2002**.

[8] Wächter, A. and Biegler, L.T., On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming. *Mathematical Programming* 106(1), **2006**, pp. 25–57.

A Summary of Suffixes

In this section we summarize the suffixes that need to be set for sensitivity updates, or reduced Hessian calculations.

Sensitivity Calculations: Set the option `run_sens` to `yes`.

Some suffixes will need to be defined by the user, while others are automatically generated by *sIPOPT*. Moreover, some of the suffixes need to be indexed by $\{i : 1 \leq i \leq \texttt{n\_sens\_steps}\}$. Also note that the direction column below is used to indicate to AMPL if the suffix will be sent to the solver, or passed by the solver to AMPL. More information on this can be found in [7].

Defined by User		
Suffix	Direction	Description
sens_state_0	IN	This is used to enumerate the parameters that will be perturbed. It takes values from 1 to length(p), and the values may not be repeated. Note that the order of the values is crucial.
sens_state_*i*	IN	This is similar to **sens_state_0**, but it now indicates the order for the parameters at the perturbed value. You must define one for each $\{i : 1 \leq i \leq$ `n_sens_steps`$\}$. This suffix should have the same values as **sens_state_0**. It takes values from 1 to length(p), and the values may no be repeated.
sens_state_value_*i*	IN	This is used to communicate the values of the perturbed parameters. You must define one for each $\{i : 1 \leq i \leq$ `n_sens_steps`$\}$. It has to be set for the same variables as **sens_state_1**.
sens_init_constr	IN	This is a flag that indicates the constraint is artificial, e.g., $w - p = 0$ in Problem (10). If the constraint is artificial, set this suffix to 1 (no indexing is necessary).
Defined by *sIPOPT*		
sens_sol_state_*i*	OUT	This holds the updated variables, as well as, the updated constraint multiplier values computed in the sensitivity update. One for each $\{i : 1 \leq i \leq$ `n_sens_steps`$\}$ will be defined.
sens_sol_state_*i*_z_L	OUT	This suffix holds updated lower bound multipliers. One for each $\{i : 1 \leq i \leq$ `n_sens_steps`$\}$ will be defined.
sens_sol_state_*i*_z_U	OUT	This suffix holds updated upper bound multipliers. One for each $\{i : 1 \leq i \leq$ `n_sens_steps`$\}$ will be defined.

Reduced Hessian Calculations: Set the option `compute_red_hessian` to `yes`.

Suffix	Direction	Description
red_hessian	IN	This is used to enumerate the independent variables, thus it needs to take ordered values from $1..n_I$, and n_I is the number of independent variables.