Interfacing your NLP to Ipopt

Method to request the initial information about the problem.

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 six interfaces to Ipopt, namely the AMPL modeling language [5] interface, and the C++, C, Fortran, Java, and R code interfaces. AMPL is a modeling language tool that allows users to write their optimization problem in a syntax that resembles the way the problem would be written mathematically. Once the problem has been formulated in AMPL, the problem can be easily solved using the (already compiled) Ipopt AMPL solver executable, ipopt. Interfacing your problem by directly linking code requires more effort to write, but can be far more efficient for large problems.

We will illustrate how to use each of the four interfaces using an example problem, number 71 from the Hock-Schittkowsky test suite [6],

$$\begin{aligned} \min_{x\in\Re^4} &\quad x_1 x_4 (x_1 + x_2 + x_3) + x_3 \\ \text{s.t.} &\quad x_1 x_2 x_3 x_4 \geq 25 \\ &\quad x_1^2 + x_2^2 + x_3^2 + x_4^2 = 40 \\ &\quad 1 \leq x_1, x_2, x_3, x_4 \leq 5, \end{aligned} \tag{HS071}$$

with the starting point

$$x_0 = (1, 5, 5, 1)$$

and the optimal solution

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

You can find further, less documented examples for using Ipopt from your own source code in the Ipopt/examples subdirectory.

Using Ipopt through AMPL

Using the AMPL solver executable is by far the easiest way to solve a problem with Ipopt. The user must simply formulate the problem in AMPL syntax, and solve the problem through the AMPL environment. There are drawbacks, however. AMPL is a 3rd party package and, as such, must be appropriately licensed (a free student version for limited problem size is available from the AMPL website. 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 [5].

The problem presented in (HS071) can be solved with Ipopt with the following AMPL model.

```
# 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 **Compiling and Installing Ipopt**) 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 **Ipopt Options** and **Ipopt Output**.

Using Ipopt from the command line

It is possible to solve AMPL problems with Ipopt directly from the command line. However, this requires a file in format `.nl` produced by ampl. If you have a model and data loaded in Ampl, you can create the corresponding `.nl` file with name, say, `myprob.nl` by using the Ampl command:

```
write gmyprob
```

There is a small `.nl` file available in the Ipopt distribution. It is located at `Ipopt/test/mytoy.nl`. We use this file in the remainder of this section. We assume that the file `mytoy.nl` is in the current directory and that the command `ipopt` is a shortcut for running the Ipopt binary available in the `bin` directory of the installation of Ipopt.

We list below commands to perform basic tasks from the Linux prompt.

- To solve mytoy.nl from the Linux prompt, use:
  ```
  ipopt mytoy
  ```
- To see all command line options for Ipopt, use:
  ```
  ipopt -=
  ```
- To see more detailed information on all options for Ipopt:
  ```
  ipopt mytoy 'print_options_documentation yes'
  ```
- To run ipopt, setting the maximum number of iterations to 2 and print level to 4:
  ```
  ipopt mytoy 'max_iter 2 print_level 4'
  ```

If many options are to be set, they can be collected in a file `ipopt.opt` that is automatically read by Ipopt, if present, in the current directory, see also **Ipopt Options**.

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 following information is 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 function
    - sparsity structure of the Jacobian of the constraints
    - sparsity structure of the Hessian of the Lagrangian function
5. Evaluation of Problem Functions

   Information evaluated using a given point (x, λ, σ_f coming from Ipopt)
    - Objective function, $f(x)$
    - Gradient of the objective, $\nabla f(x)$
    - Constraint function values, $g(x)$
    - Jacobian of the constraints, $\nabla g(x)^T$
    - Hessian of the Lagrangian function, $\sigma_f \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x)$
      (this is not required if a quasi-Newton options is chosen to approximate the second derivatives)

The problem dimensions and bounds are straightforward and come solely from the problem definition. The initial starting point is used by the algorithm when it begins iterating to solve the problem. If Ipopt has difficulty converging, or if it converges to a locally infeasible point, adjusting the starting point may help. Depending on the starting point, Ipopt may also converge to different local solutions.

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

As Ipopt iterates, it will need the values for problem functions 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 (HS071).

The gradient of the objective $f(x)$ is given by

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

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

$$\begin{bmatrix} x_2 x_3 x_4 & x_1 x_3 x_4 & x_1 x_2 x_4 & x_1 x_2 x_3 \\ 2x_1 & 2x_2 & 2x_3 & 2x_4 \end{bmatrix}.$$

We also need to determine the Hessian of the Lagrangian. (If a quasi-Newton option is chosen to approximate the second derivatives, this is not required. However, if second derivatives can be computed, it is often worthwhile to let

Ipopt use them, since the algorithm is then usually more robust and converges faster. More on the quasi-Newton approximation in **Quasi-Newton Approximation of Second Derivatives**.) The Lagrangian function for the NLP (HS071) is defined as $f(x) + g(x)^T \lambda$ and the Hessian of the Lagrangian function is, technically, $\nabla^2 f(x_k) + \sum_{i=1}^m \lambda_i \nabla^2 g_i(x_k)$. However, we introduce a factor (σ_f) in front of the objective term so that Ipopt can ask for the Hessian of the objective or the constraints independently, if required. Thus, for Ipopt the symbolic form of the Hessian of the Lagrangian is

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

and for the example problem this becomes

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

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

The remaining sections of the tutorial will lead you through the coding required to solve example problem (HS071) using, first C++, then C, and finally Fortran. Completed versions of these examples can be found in `$IPOPTDIR/Ipopt/examples` under `hs071_cpp`, `hs071_c`, and `hs071_f`.

As a user, you are responsible for coding two sections of the program that solves a problem using Ipopt: the main executable (e.g., main) and the problem representation. Typically, you will write an executable that prepares the problem, and then passes control over to Ipopt through an Optimize or Solve call. In this call, you will give Ipopt everything that it requires to call back to your code whenever it needs functions evaluated (like the objective function, the Jacobian of the constraints, etc.). In each of the three sections that follow (C++, C, and Fortran), we will first discuss how to code the problem representation, and then how to code the executable.

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, **Ipopt::TNLP**. For the executable (the main function) we will make the call to Ipopt through the **Ipopt::IpoptApplication** class. In addition, we will also be using the **Ipopt::SmartPtr** class which implements a reference counting pointer that takes care of memory management (object deletion) for you (for details, see **The Smart Pointer Implementation: SmartPtr<T>**).

After `make install` (see **Compiling and Installing Ipopt**), the header files that define these classes are installed in `$PREFIX/include/coin-or`.

Coding the Problem Representation

We provide the required information 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 **Ipopt::TNLP** interface. If you have any difficulty as the implementation proceeds, have a look at the completed example in the `Ipopt/examples/hs071_cpp` directory.

Start by creating a new directory `MyExample` under examples 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 namespace **Ipopt**, and create the declaration of the HS071_NLP class, inheriting off of TNLP. Have a look at the **Ipopt::TNLP** class; 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 namespace **Ipopt**. A full version of these files can be found in the `Ipopt/examples/hs071_cpp` directory.

It is very easy to make mistakes in the implementation of the function evaluation methods, in particular regarding the derivatives. Ipopt has a feature that can help you to debug the derivative code, using finite differences, see **Derivative Checker**.

Note that the return value of any bool-valued function should be true, unless an error occurred, for example, because the value of a problem function could not be evaluated at the required point.

Ipopt::TNLP::get_nlp_info

```
   virtual bool get_nlp_info(
      Index&          n,
      Index&          m,
      Index&          nnz_jac_g,
      Index&          nnz_h_lag,
      IndexStyleEnum& index_style
   ) = 0;
```

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.

Parameters

n	(out) Storage for the number of variables x
m	(out) Storage for the number of constraints $g(x)$
nnz_jac_g	(out) Storage for the number of nonzero entries in the Jacobian
nnz_h_lag	(out) Storage for the number of nonzero entries in the Hessian
index_style	(out) Storage for the index style, the numbering style used for row/col entries in the sparse matrix format (TNLP::C_STYLE: 0-based, TNLP::FORTRAN_STYLE: 1-based; see also **Triplet Format for Sparse Matrices**)

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

```
// returns the size of the 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;
}
```

Ipopt::TNLP::get_bounds_info

```
   virtual bool get_bounds_info(
      Index   n,
      Number* x_l,
      Number* x_u,
      Index   m,
      Number* g_l,
      Number* g_u
   ) = 0;
```

Method to request bounds on the variables and constraints.

Parameters

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

Returns

true if success, false otherwise.

The values of n and m that were specified in TNLP::get_nlp_info are passed here for debug checking. Setting a lower bound to a value less than or equal to the value of the option **nlp_lower_bound_inf** will cause Ipopt to assume no lower bound. Likewise, specifying the upper bound above or equal to the value of the option **nlp_upper_bound_inf** will cause Ipopt to assume no upper bound. These options are set to -10^{19} and 10^{19}, respectively, by default, but may be modified by changing these options.

In our example, the first constraint has a lower bound of 25 and no upper bound, so we set the lower bound of constraint 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.

```
// returns the variable bounds
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;
}
```

Ipopt::TNLP::get_starting_point

```
   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
   ) = 0;
```

Method to request the starting point before iterating.

Parameters

n	(in) the number of variables x in the problem; it will have the same value that was specified in TNLP::get_nlp_info
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 $g(x)$ in the problem; it will have the same value that was specified in TNLP::get_nlp_info
init_lambda	(in) if true, this method must provide an initial value for the constraint multipliers λ

lambda (out) the initial values for the constraint multipliers, λ

Returns

true if success, false otherwise.

The boolean variables indicate whether the algorithm requires to have x, z_L/z_u, and lambda initialized, respectively. If, for some reason, the algorithm requires initializations that cannot be provided, false should be returned and Ipopt will stop. The default options only require initial values for the primal variables x.

Note, that the initial values for bound multiplier components for absent bounds ($x^L_i = -\infty$ or $x^U_i = \infty$) are ignored.

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

```
// returns the initial point for the problem
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
   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;
}
```

Ipopt::TNLP::eval_f

```
virtual bool eval_f(
   Index         n,
   const Number* x,
   bool          new_x,
   Number&       obj_value
) = 0;
```

Method to request the value of the objective function.

Parameters

n (in) the number of variables x in the problem; it will have the same value that was specified in TNLP::get_nlp_info

x (in) the values for the primal variables x at which the objective function $f(x)$ is to be evaluated

new_x (in) false if any evaluation method (`eval_*`) was previously called with the same values in x, true otherwise. 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.

obj_value (out) storage for the value of the objective function $f(x)$

Returns

true if success, false otherwise.

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

```
// returns the value of the objective function
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;
}
```

Ipopt::TNLP::eval_grad_f

```
virtual bool eval_grad_f(
   Index         n,
   const Number* x,
   bool          new_x,
   Number*       grad_f
) = 0;
```

Method to request the gradient of the objective function.

Parameters

n (in) the number of variables x in the problem; it will have the same value that was specified in TNLP::get_nlp_info

x (in) the values for the primal variables x at which the gradient $\nabla f(x)$ is to be evaluated

new_x (in) false if any evaluation method (`eval_*`) was previously called with the same values in x, true otherwise; see also TNLP::eval_f

grad_f (out) array to store values of 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]`).

Returns

true if success, false otherwise.

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

```
// return the gradient of the objective function grad_{x} f(x)
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;
}
```

Ipopt::TNLP::eval_g

```
   virtual bool eval_g(
      Index         n,
      const Number* x,
      bool          new_x,
      Index         m,
      Number*       g
   ) = 0;
```

Method to request the constraint values.

Parameters

n	(in) the number of variables x in the problem; it will have the same value that was specified in TNLP::get_nlp_info
x	(in) the values for the primal variables x at which the constraint functions $g(x)$ are to be evaluated
new_x	(in) false if any evaluation method (`eval_*`) was previously called with the same values in x, true otherwise; see also TNLP::eval_f
m	(in) the number of constraints $g(x)$ in the problem; it will have the same value that was specified in TNLP::get_nlp_info
g	(out) array to store constraint function values $g(x)$, do not add or subtract the bound values g^L or g^U.

Returns

true if success, false otherwise.

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

```
// return the value of the constraints: g(x)
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;
}
```

Ipopt::TNLP::eval_jac_g

```
   virtual bool eval_jac_g(
      Index         n,
      const Number* x,
      bool          new_x,
      Index         m,
      Index         nele_jac,
      Index*        iRow,
      Index*        jCol,
      Number*       values
   ) = 0;
```

Method to request either the sparsity structure or the values of the Jacobian of the constraints. The Jacobian is the matrix of derivatives where the derivative of constraint function g_i with respect to variable x_j is placed in row i and column j. See **Triplet Format for Sparse Matrices** for a discussion of the sparse matrix format used in this method.

Parameters

n	(in) the number of variables x in the problem; it will have the same value that was specified in TNLP::get_nlp_info
x	(in) first call: NULL; later calls: the values for the primal variables x at which the constraint Jacobian $\nabla g(x)^T$ is to be evaluated
new_x	(in) false if any evaluation method (`eval_*`) was previously called with the same values in x, true otherwise; see also TNLP::eval_f
m	(in) the number of constraints $g(x)$ in the problem; it will have the same value that was specified in TNLP::get_nlp_info
nele_jac	(in) the number of nonzero elements in the Jacobian; it will have the same value that was specified in TNLP::get_nlp_info
iRow	(out) first call: array of length nele_jac to store the row indices of entries in the Jacobian of the constraints; later calls: NULL
jCol	(out) first call: array of length nele_jac to store the column indices of entries in the Jacobian of the constraints; later calls: NULL
values	(out) first call: NULL; later calls: array of length nele_jac to store the values of the entries in the Jacobian of the constraints

Returns

true if success, false otherwise.

Note

The arrays iRow and jCol only need to be filled once. If the iRow and jCol arguments are not NULL (first call to this function), then Ipopt expects that the sparsity structure of the Jacobian (the row and column indices only) are written into iRow and jCol. At this call, the arguments `x` and `values` will be NULL. If the arguments `x` and `values` are not NULL, then Ipopt expects that the value of the Jacobian as calculated from array `x` is stored in array `values` (using the same order as used when specifying the sparsity structure). At this call, the arguments `iRow` and `jCol` will be NULL.

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

```
// return the structure or values of the Jacobian
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
)
{
   assert(n == 4);
   assert(m == 2);

   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;
}
```

Ipopt::TNLP::eval_h

```
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
)
```

Method to request either the sparsity structure or the values of the Hessian of the Lagrangian. The Hessian matrix that Ipopt uses is

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

for the given values for x, σ_f, and λ. See **Triplet Format for Sparse Matrices** for a discussion of the sparse matrix format used in this method.

Parameters

n	(in) the number of variables x in the problem; it will have the same value that was specified in TNLP::get_nlp_info
x	(in) first call: NULL; later calls: the values for the primal variables x at which the Hessian is to be evaluated
new_x	(in) false if any evaluation method (`eval_*`) was previously called with the same values in x, true otherwise; see also TNLP::eval_f
obj_factor	(in) factor σ_f in front of the objective term in the Hessian
m	(in) the number of constraints $g(x)$ in the problem; it will have the same value that was specified in TNLP::get_nlp_info
lambda	(in) the values for the constraint multipliers λ at which the Hessian is to be evaluated
new_lambda	(in) false if any evaluation method was previously called with the same values in lambda, true otherwise
nele_hess	(in) the number of nonzero elements in the Hessian; it will have the same value that was specified in TNLP::get_nlp_info
iRow	(out) first call: array of length nele_hess to store the row indices of entries in the Hessian; later calls: NULL
jCol	(out) first call: array of length nele_hess to store the column indices of entries in the Hessian; later calls: NULL
values	(out) first call: NULL; later calls: array of length nele_hess to store the values of the entries in the Hessian

Returns

true if success, false otherwise.

Note

The arrays iRow and jCol only need to be filled once. If the iRow and jCol arguments are not NULL (first call to this function), then Ipopt expects that the sparsity structure of the Hessian (the row and column indices only) are written into iRow and jCol. At this call, the arguments `x`, `lambda`, and `values` will be NULL. If the arguments `x`, `lambda`, and `values` are not NULL, then Ipopt expects that the value of the Hessian as calculated from arrays `x` and `lambda` are stored in array `values` (using the same order as used when specifying the sparsity structure). At this call, the arguments `iRow` and `jCol` will be NULL.

Attention

As this matrix is symmetric, Ipopt expects that only the lower diagonal entries are specified.

A default implementation is provided, in case the user wants to set quasi-Newton approximations to estimate the second derivatives and doesn't not need to implement this method.

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.

```
//return the structure or values of the Hessian
```

```
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
)
{
   assert(n == 4);
   assert(m == 2);

   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;
}
```

Ipopt::TNLP::finalize_solution

This is the only method that is not mentioned in the beginning of this section.

```
   virtual void finalize_solution(
      SolverReturn               status,
      Index                      n,
      const Number*              x,
      const Number*              z_L,
      const Number*              z_U,
      Index                      m,
      const Number*              g,
      const Number*              lambda,
      Number                     obj_value,
      const IpoptData*           ip_data,
      IpoptCalculatedQuantities* ip_cq
   ) = 0;
```

This method is called when the algorithm has finished (successfully or not) so the TNLP can digest the outcome, e.g., store/write the solution, if any.

Parameters

status	(in) gives the status of the algorithm • SUCCESS: Algorithm terminated successfully at a locally optimal point, satisfying the convergence tolerances (can be specified by options). • MAXITER_EXCEEDED: Maximum number of iterations exceeded (can be specified by an option). • CPUTIME_EXCEEDED: Maximum number of CPU seconds exceeded (can be specified by an option). • STOP_AT_TINY_STEP: Algorithm proceeds with very little progress. • STOP_AT_ACCEPTABLE_POINT: Algorithm stopped at a point that was converged, not to "desired" tolerances, but to "acceptable" tolerances (see the acceptable-... options). • LOCAL_INFEASIBILITY: Algorithm converged to a point of local infeasibility. Problem may be infeasible. • USER_REQUESTED_STOP: The user call-back function TNLP::intermediate_callback returned false, i.e., the user code requested a premature termination of the optimization. • DIVERGING_ITERATES: It seems that the iterates diverge. • RESTORATION_FAILURE: Restoration phase failed, algorithm doesn't know how to proceed. • ERROR_IN_STEP_COMPUTATION: An unrecoverable error occurred while Ipopt tried to compute the search direction. • INVALID_NUMBER_DETECTED: Algorithm received an invalid number (such as NaN or Inf) from the NLP; see also option check_derivatives_for_nan_inf). • INTERNAL_ERROR: An unknown internal error occurred.
n	(in) the number of variables x in the problem; it will have the same value that was specified in TNLP::get_nlp_info
x	(in) the final values for the primal variables
z_L	(in) the final values for the lower bound multipliers

z_U	(in) the final values for the upper bound multipliers
m	(in) the number of constraints $g(x)$ in the problem; it will have the same value that was specified in TNLP::get_nlp_info
g	(in) the final values of the constraint functions
lambda	(in) the final values of the constraint multipliers
obj_value	(in) the final value of the objective function
ip_data	(in) provided for expert users
ip_cq	(in) provided for expert users

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,
   const IpoptData*           ip_data,
   IpoptCalculatedQuantities* ip_cq
)
{
   // 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
   std::cout << std::endl << std::endl << "Solution of the primal variables, x" << std::endl;
   for( Index i = 0; i < n; i++ )
   {
      std::cout << "x[" << i << "] = " << x[i] << std::endl;
   }

   std::cout << std::endl << std::endl << "Solution of the bound multipliers, z_L and z_U" <<
        std::endl;
   for( Index i = 0; i < n; i++ )
   {
      std::cout << "z_L[" << i << "] = " << z_L[i] << std::endl;
   }
   for( Index i = 0; i < n; i++ )
   {
      std::cout << "z_U[" << i << "] = " << z_U[i] << std::endl;
   }

   std::cout << std::endl << std::endl << "Objective value" << std::endl;
   std::cout << "f(x*) = " << obj_value << std::endl;

   std::cout << std::endl << "Final value of the constraints:" << std::endl;
   for( Index i = 0; i < m; i++ )
   {
      std::cout << "g(" << i << ") = " << g[i] << std::endl;
   }
}
```

This is all that is required for our HS071_NLP class and the coding of the problem representation.

Coding the Executable

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

(**Ipopt::IpoptApplication**), initialize it, and ask the solver to find a solution. We always use the **Ipopt::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 **The Smart Pointer Implementation: SmartPtr<T>**.

Create the file `MyExample.cpp` in the `MyExample` directory. Include the header files `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"

#include <iostream>

using namespace Ipopt;

int main(
   int    /*argv*/,
   char** /*argc*/
)
{
   // Create a new instance of your nlp
   //  (use a SmartPtr, not raw)
   SmartPtr<TNLP> mynlp = new HS071_NLP();

   // Create a new instance of IpoptApplication
   //  (use a SmartPtr, not raw)
   // We are using the factory, since this allows us to compile this
   // example with an Ipopt Windows DLL
   SmartPtr<IpoptApplication> app = IpoptApplicationFactory();

   // Change some options
   // Note: The following choices are only examples, they might not be
   //       suitable for your optimization problem.
   app->Options()->SetNumericValue("tol", 3.82e-6);
   app->Options()->SetStringValue("mu_strategy", "adaptive");
   app->Options()->SetStringValue("output_file", "ipopt.out");
   // The following overwrites the default name (ipopt.opt) of the options file
   // app->Options()->SetStringValue("option_file_name", "hs071.opt");

   // Initialize the IpoptApplication and process the options
   ApplicationReturnStatus status;
   status = app->Initialize();
   if( status != Solve_Succeeded )
   {
      std::cout << std::endl << std::endl << "*** Error during initialization!" << std::endl;
      return (int) status;
   }

   // Ask Ipopt to solve the problem
   status = app->OptimizeTNLP(mynlp);

   if( status == Solve_Succeeded )
   {
      std::cout << std::endl << std::endl << "*** The problem solved!" << std::endl;
   }
   else
   {
      std::cout << std::endl << std::endl << "*** The problem FAILED!" << std::endl;
   }

   // 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, **Ipopt::IpoptApplication**. You could use new to create a new application object, but if you want to make sure that your code would also work with a Windows DLL, you need to use the factory, as done in the example above. The call to `app->Initialize(...)` will initialize that object and process the options (particularly the output related options). The call to `app->OptimizeTNLP(...)` will run Ipopt and try to solve the problem. By default, Ipopt will write its progress to the console, and return the SolverReturn status.

Compiling and Testing the Example

Our next task is to compile and test the code. If you are familiar with the compiler and linker used on your system, you can build the code, telling the linker about the necessary libraries, as obtained via `pkg-config --lflags ipopt`. The build system already created a sample makefile. Copy `Ipopt/examples/hs071_cpp/Makefile` into your `MyExample` directory. This makefile was created for the `hs071_cpp` code, but it can be easily modified for your example problem. Edit the file, making the following changes:

- change the EXE variable
  ```
  EXE = my_example
  ```
- change the OBJS variable
  ```
  OBJS = HS071_NLP.o MyExample.o
  ```

The code should compile easily with

```
$ make
```

Now run the executable with

```
$ ./my_example
```

and you should see output resembling the following:

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit https://github.com/coin-or/Ipopt
******************************************************************************

Number of nonzeros in equality constraint Jacobian...:        4
Number of nonzeros in inequality constraint Jacobian.:        4
Number of nonzeros in Lagrangian Hessian.............:       10

Total number of variables............................:        4
                     variables with only lower bounds:        0
                variables with lower and upper bounds:        4
                     variables with only upper bounds:        0
Total number of equality constraints.................:        1
Total number of inequality constraints...............:        1
        inequality constraints with only lower bounds:        1
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  1.6109693e+01 1.12e+01 5.28e-01   0.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  1.7410406e+01 8.38e-01 2.25e+01  -0.3 7.97e-01    -  3.19e-01 1.00e+00f  1
```

```
   2  1.8001613e+01 1.06e-02 4.96e+00  -0.3 5.60e-02   2.0 9.97e-01 1.00e+00h  1
   3  1.7199482e+01 9.04e-02 4.24e-01  -1.0 9.91e-01    -  9.98e-01 1.00e+00f  1
   4  1.6940955e+01 2.09e-01 4.58e-02  -1.4 2.88e-01    -  9.66e-01 1.00e+00h  1
   5  1.7003411e+01 2.29e-02 8.42e-03  -2.9 7.03e-02    -  9.68e-01 1.00e+00h  1
   6  1.7013974e+01 2.59e-04 8.65e-05  -4.5 6.22e-03    -  1.00e+00 1.00e+00h  1
   7  1.7014017e+01 2.26e-07 5.71e-08  -8.0 1.43e-04    -  1.00e-00 1.00e+00h  1
   8  1.7014017e+01 4.62e-14 9.09e-14  -8.0 6.95e-08    -  1.00e+00 1.00e+00h  1

Number of Iterations....: 8

Number of objective function evaluations             = 9
Number of objective gradient evaluations             = 9
Number of equality constraint evaluations            = 9
Number of inequality constraint evaluations          = 9
Number of equality constraint Jacobian evaluations   = 9
Number of inequality constraint Jacobian evaluations = 9
Number of Lagrangian Hessian evaluations             = 8
Total CPU secs in IPOPT (w/o function evaluations)   =      0.220
Total CPU secs in NLP function evaluations           =      0.000

EXIT: Optimal Solution Found.


Solution of the primal variables, x
x[0] = 1.000000e+00
x[1] = 4.743000e+00
x[2] = 3.821150e+00
x[3] = 1.379408e+00


Solution of the bound multipliers, z_L and z_U
z_L[0] = 1.087871e+00
z_L[1] = 2.428776e-09
z_L[2] = 3.222413e-09
z_L[3] = 2.396076e-08
z_U[0] = 2.272727e-09
z_U[1] = 3.537314e-08
z_U[2] = 7.711676e-09
z_U[3] = 2.510890e-09


Objective value
f(x*) = 1.701402e+01


*** The problem solved!
```

This completes the basic C++ tutorial, but see **Ipopt Output** for explanation on the standard console output of Ipopt and **Ipopt Options** for information about the use of options to customize the behavior of Ipopt.

The `Ipopt/examples/ScalableProblems` directory contains other NLP problems coded in C++.

Additional methods in TNLP

The following methods are available for additional features that are not explained in the example. Default implementations for those methods are provided, so that a user can safely ignore them, unless she wants to make use of those features. From these features, only the intermediate callback is already available in the C and Fortran interfaces.

Ipopt::TNLP::intermediate_callback

```
virtual bool intermediate_callback(
   AlgorithmMode              mode,
   Index                      iter,
   Number                     obj_value,
   Number                     inf_pr,
   Number                     inf_du,
   Number                     mu,
   Number                     d_norm,
   Number                     regularization_size,
   Number                     alpha_du,
   Number                     alpha_pr,
   Index                      ls_trials,
   const IpoptData*           ip_data,
   IpoptCalculatedQuantities* ip_cq
)
```

Intermediate Callback method for the user. This method is called once per iteration (during the convergence check), and can be used to obtain information about the optimization status while Ipopt solves the problem, and also to request a premature termination.

The information provided by the entities in the argument list correspond to what Ipopt prints in the iteration summary (see also **Ipopt Output**), except for inf_pr, which by default corresponds to the original problem in the log but to the scaled internal problem in this callback. Further information can be obtained from the ip_data and ip_cq objects. The current iterate and violations of feasibility and optimality can be accessed via the methods **Ipopt::TNLP::get_curr_iterate()** and **Ipopt::TNLP::get_curr_violations()**. These methods translate values for the *internal representation* of the problem from `ip_data` and `ip_cq` objects into the TNLP representation.

Returns

If this method returns false, Ipopt will terminate with the User_Requested_Stop status.

It is not required to implement (overload) this method. The default implementation always returns true.

In our example, we optionally print the current iterate and its violation of optimality conditions to the screen.

```
bool HS071_NLP::intermediate_callback(
   AlgorithmMode              mode,
   Index                      iter,
   Number                     obj_value,
   Number                     inf_pr,
   Number                     inf_du,
   Number                     mu,
   Number                     d_norm,
   Number                     regularization_size,
   Number                     alpha_du,
   Number                     alpha_pr,
   Index                      ls_trials,
   const IpoptData*           ip_data,
   IpoptCalculatedQuantities* ip_cq
)
{
   if( !printiterate_ )
   {
      return true;
   }
   Number x[4];
   Number x_L_viol[4];
   Number x_U_viol[4];
   Number z_L[4];
   Number z_U[4];
   Number compl_x_L[4];
   Number compl_x_U[4];
   Number grad_lag_x[4];

   Number g[2];
   Number lambda[2];
   Number constraint_violation[2];
   Number compl_g[2];

   bool have_iter = get_curr_iterate(ip_data, ip_cq, false, 4, x, z_L, z_U, 2, g, lambda);
   bool have_viol = get_curr_violations(ip_data, ip_cq, false, 4, x_L_viol, x_U_viol, compl_x_L,
        compl_x_U, grad_lag_x, 2, constraint_violation, compl_g);

   printf("Current iterate:\n");
   printf("  %-12s %-12s %-12s %-12s %-12s %-12s %-12s\n", "x", "z_L", "z_U", "bound_viol",
        "compl_x_L", "compl_x_U", "grad_lag_x");
   for( int i = 0; i < 4; ++i )
   {
      if( have_iter )
      {
         printf("  %-12g %-12g %-12g", x[i], z_L[i], z_U[i]);
      }
      else
      {
         printf("  %-12s %-12s %-12s", "n/a", "n/a", "n/a");
      }
      if( have_viol )
      {
         printf(" %-12g %-12g %-12g %-12g\n", x_L_viol[i] > x_U_viol[i] ? x_L_viol[i] : x_U_viol[i],
        compl_x_L[i], compl_x_U[i], grad_lag_x[i]);
      }
      else
      {
         printf(" %-12s %-12s %-12s %-12s\n", "n/a", "n/a", "n/a", "n/a");
      }
   }

   printf("  %-12s %-12s %-12s %-12s\n", "g(x)", "lambda", "constr_viol", "compl_g");
   for( int i = 0; i < 2; ++i )
   {
      if( have_iter )
      {
         printf("  %-12g %-12g", g[i], lambda[i]);
      }
      else
      {
         printf("  %-12s %-12s", "n/a", "n/a");
      }
      if( have_viol )
      {
         printf(" %-12g %-12g\n", constraint_violation[i], compl_g[i]);
      }
      else
      {
         printf(" %-12s %-12s\n", "n/a", "n/a");
      }
   }

   return true;
}
```

Ipopt::TNLP::get_scaling_parameters

```
   virtual bool get_scaling_parameters(
      Number& obj_scaling,
      bool&   use_x_scaling,
      Index   n,
      Number* x_scaling,
      bool&   use_g_scaling,
      Index   m,
      Number* g_scaling
   )
```

Method to request scaling parameters. This is only called if the options are set to retrieve user scaling, that is, if nlp_scaling_method is chosen as "user-scaling". The method should provide scaling factors for the objective function as well as for the optimization variables and/or constraints. The return value should be true, unless an error occurred, and the program is to be aborted.

The value returned in obj_scaling determines, how Ipopt should internally scale the objective function. For example, if this number is chosen to be 10, then Ipopt solves internally an optimization problem that has 10 times the value of the original objective function provided by the TNLP. In particular, if this value is negative, then Ipopt will maximize the objective function instead of minimizing it.

The scaling factors for the variables can be returned in x_scaling, which has the same length as x in the other TNLP methods, and the factors are ordered like x. use_x_scaling needs to be set to true, if Ipopt should scale the variables. If it is false, no internal scaling of the variables is done. Similarly, the scaling factors for the constraints can be returned in g_scaling, and this scaling is activated by setting use_g_scaling to true.

As a guideline, we suggest to scale the optimization problem (either directly in the original formulation, or after using scaling factors) so that all sensitivities, i.e., all non-zero first partial derivatives, are typically of the order 0.1-10.

Ipopt::TNLP::get_number_of_nonlinear_variables

This method is called only if the **quasi-Newton approximation** is selected.

```
   virtual Index get_number_of_nonlinear_variables()
```

Return the number of variables that appear nonlinearly in the objective function or in at least one constraint function. If -1 is returned as number of nonlinear variables, Ipopt assumes that all variables are nonlinear. Otherwise, it calls get_list_of_nonlinear_variables with an array into which the indices of the nonlinear variables should be written - the array has the length num_nonlin_vars, which is identical with the return value of get_number_of_nonlinear_variables(). It is assumed that the indices are counted starting with 1 in the FORTRAN_STYLE, and 0 for the C_STYLE.

The default implementation returns -1, i.e., all variables are assumed to be nonlinear.

Ipopt::TNLP::get_list_of_nonlinear_variables

This method is called only if the **quasi-Newton approximation** is selected.

```
   virtual bool get_list_of_nonlinear_variables(
      Index  num_nonlin_vars,
      Index* pos_nonlin_vars
   )
```

Return the indices of all nonlinear variables. This method is called only if limited-memory quasi-Newton option is used and get_number_of_nonlinear_variables() returned a positive number. This number is provided in parameter num_nonlin_var.

The method must store the indices of all nonlinear variables in pos_nonlin_vars, where the numbering starts with 0 order 1, depending on the numbering style determined in get_nlp_info.

Ipopt::TNLP::get_variables_linearity

```
virtual bool get_variables_linearity(
   Index          n,
   LinearityType* var_types
)
```

Method to request the variables linearity. This method is never called by Ipopt, but is used by Bonmin to get information about which variables occur only in linear terms. Ipopt passes the array var_types of length at least n, which should be filled with the appropriate linearity type of the variables (TNLP::LINEAR or TNLP::NON_LINEAR).

The default implementation just returns false and does not fill the array.

Ipopt::TNLP::get_constraints_linearity

```
virtual bool get_constraints_linearity(
   Index          m,
   LinearityType* const_types
)
```

Method to request the constraints linearity. This method is never called by Ipopt, but is used by Bonmin to get information about which constraints are linear. Ipopt passes the array const_types of size m, which should be filled with the appropriate linearity type of the constraints (TNLP::LINEAR or TNLP::NON_LINEAR).

The default implementation just returns false and does not fill the array.

Ipopt::TNLP::get_var_con_metadata

```
virtual bool get_var_con_metadata(
   Index                   n,
   StringMetaDataMapType&  var_string_md,
   IntegerMetaDataMapType& var_integer_md,
   NumericMetaDataMapType& var_numeric_md,
   Index                   m,
   StringMetaDataMapType&  con_string_md,
   IntegerMetaDataMapType& con_integer_md,
   NumericMetaDataMapType& con_numeric_md
)
```

Method to request meta data for the variables and the constraints. This method is used to pass meta data about variables or constraints to Ipopt. The data can be either of integer, numeric, or string type. Ipopt passes this data on to its internal problem representation. The meta data type is a std::map with std::string as key type and a std::vector as value type. So far, Ipopt itself makes only use of string meta data under the key idx_names. With this key, variable and constraint names can be passed to Ipopt, which are shown when printing internal vector or matrix data structures if Ipopt is run with a high value for the option. This allows a user to identify the original variables and constraints corresponding to Ipopt's internal problem representation.

If this method is not overloaded, the default implementation does not set any meta data and returns false.

Ipopt::TNLP::finalize_metadata

```
virtual void finalize_metadata(
   Index                         n,
   const StringMetaDataMapType&  var_string_md,
   const IntegerMetaDataMapType& var_integer_md,
   const NumericMetaDataMapType& var_numeric_md,
   Index                         m,
   const StringMetaDataMapType&  con_string_md,
   const IntegerMetaDataMapType& con_integer_md,
   const NumericMetaDataMapType& con_numeric_md
)
```

> This method returns any metadata collected during the run of the algorithm. This method is called just before finalize_solution is called. The returned data includes the metadata provided by TNLP::get_var_con_metadata. Each metadata can be of type string, integer, or numeric. It can be associated to either the variables or the constraints. The metadata that was associated with the primal variable vector is stored in `var_..._md`. The metadata associated with the constraint multipliers is stored in `con_..._md`. The metadata associated with the bound multipliers is stored in `var_..._md`, with the suffixes "_z_L", and "_z_U", denoting lower and upper bounds.
>
> If the user doesn't overload this method in her implementation of the class derived from TNLP, the default implementation does nothing.

Ipopt::TNLP::get_warm_start_iterate

```
virtual bool get_warm_start_iterate(
   IteratesVector& warm_start_iterate
)
{
   (void) warm_start_iterate;
   return false;
}
```

> Method to provide an Ipopt warm start iterate which is already in the form Ipopt requires it internally for warm starts. This method is only for expert users. The default implementation does not provide a warm start iterate and returns false.

The C Interface

The C interface for Ipopt is declared in the header file **IpStdCInterface.h**, which is found in `$PREFIX/include/coin-or`; while reading this section, it will be helpful to have a look at this file.

In order to solve an optimization problem with the C interface, one has to create an **IpoptProblem** with the function **CreateIpoptProblem**, which later has to be passed to the **IpoptSolve** function. **IpoptProblem** is a pointer to a C structure; you should not access this structure directly, but only through the functions provided in the C interface.

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

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

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

After an **IpoptProblem** has been created, you can set algorithmic options for Ipopt (see **Ipopt Options**) using the functions **AddIpoptStrOption**, **AddIpoptNumOption**, and **AddIpoptIntOption**. Finally, the Ipopt algorithm is called with **IpoptSolve**, giving Ipopt the **IpoptProblem**, the starting point, and arrays to store the solution values (primal and dual variables), if desired. Finally, after everything is done, **FreeIpoptProblem** should be called to release internal memory that is still allocated inside Ipopt.

In the remainder of this section we discuss how the example problem (HS071) can be solved using the C interface. A completed version of this example can be found in `Ipopt/examples/hs071_c`.

In order to implement the example problem on your own, create a new directory `MyCExample` and create a new file, `hs071_c.c`. Here, include the interface header file **IpStdCInterface.h**, along with other necessary header files, such as `stdlib.h` and `assert.h`. Add the prototypes and implementations for the five callback functions. Have a look at the C++ implementation for `eval_f`, `eval_g`, `eval_grad_f`, `eval_jac_g`, and `eval_h` in **Coding the Problem Representation**. The C implementations have somewhat different prototypes, but are implemented almost identically to the C++ code. See the completed example in `Ipopt/examples/hs071_c/hs071_c.c` if you are not sure how to do this.

We now need to implement the main function, create the **IpoptProblem**, set options, and call **IpoptSolve**. The **CreateIpoptProblem** function requires the problem dimensions, the variable and constraint bounds, and the function pointers to the callback routines. The **IpoptSolve** function requires the **IpoptProblem**, the starting point, and allocated arrays for the solution. The main function from the example is shown next and discussed below.

```
int main()
{
   ipindex n = -1;                          /* number of variables */
   ipindex m = -1;                          /* number of constraints */
   ipindex nele_jac;                        /* number of nonzeros in the Jacobian of the constraints */
   ipindex nele_hess;                       /* number of nonzeros in the Hessian of the Lagrangian (lower
        or upper triangular part only) */
   ipindex index_style;                     /* indexing style for matrices */
   ipnumber* x_L = NULL;                    /* lower bounds on x */
   ipnumber* x_U = NULL;                    /* upper bounds on x */
   ipnumber* g_L = NULL;                    /* lower bounds on g */
   ipnumber* g_U = NULL;                    /* upper bounds on g */
   IpoptProblem nlp = NULL;                 /* IpoptProblem */
   enum ApplicationReturnStatus status; /* Solve return code */
   ipnumber* x = NULL;                      /* starting point and solution vector */
   ipnumber* mult_g = NULL;                 /* constraint multipliers at the solution */
   ipnumber* mult_x_L = NULL;               /* lower bound multipliers at the solution */
   ipnumber* mult_x_U = NULL;               /* upper bound multipliers at the solution */
   ipnumber obj;                            /* objective value */
   struct MyUserData user_data;             /* our user data for the function evaluations */
   ipindex i;                               /* generic counter */

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

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

   /* set the number of nonzeros in the Jacobian and Hessian */
```

```
nele_jac = 8;
nele_hess = 10;

/* set the indexing style to C-style (start counting of rows and column indices at 0) */
index_style = 0;

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

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

/* Set some options.  Note the following ones are only examples,
 * they might not be suitable for your problem.
 */
AddIpoptNumOption(nlp, "tol", 3.82e-6);
AddIpoptStrOption(nlp, "mu_strategy", "adaptive");
AddIpoptStrOption(nlp, "output_file", "ipopt.out");

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

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

/* Initialize the user data */
user_data.g_offset[0] = 0.;
user_data.g_offset[1] = 0.;
user_data.nlp = nlp;

/* Set the callback method for intermediate user-control.
 * This is not required, just gives you some intermediate control in
 * case you need it.
 */
/* SetIntermediateCallback(nlp, intermediate_cb); */

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

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

   printf("\n\nSolution of the constraint multipliers, lambda\n");
   for( i = 0; i < m; i++ )
   {
      printf("lambda[%d] = %e\n", (int)i, mult_g[i]);
   }
   printf("\n\nSolution of the bound multipliers, z_L and z_U\n");
   for( i = 0; i < n; i++ )
   {
      printf("z_L[%d] = %e\n", (int)i, mult_x_L[i]);
   }
   for( i = 0; i < n; i++ )
   {
      printf("z_U[%d] = %e\n", (int)i, mult_x_U[i]);
   }

   printf("\n\nObjective value\nf(x*) = %e\n", obj);
}
```

```
   else
   {
      printf("\n\nERROR OCCURRED DURING IPOPT OPTIMIZATION.\n");
   }

   /* Now we are going to solve this problem again, but with slightly
    * modified constraints.  We change the constraint offset of the
    * first constraint a bit, and resolve the problem using the warm
    * start option.
    */
   user_data.g_offset[0] = 0.2;

   if( status == Solve_Succeeded )
   {
      /* Now resolve with a warmstart. */
      AddIpoptStrOption(nlp, "warm_start_init_point", "yes");
      /* The following option reduce the automatic modification of the
       * starting point done my Ipopt.
       */
      AddIpoptNumOption(nlp, "bound_push", 1e-5);
      AddIpoptNumOption(nlp, "bound_frac", 1e-5);
      status = IpoptSolve(nlp, x, NULL, &obj, mult_g, mult_x_L, mult_x_U, &user_data);

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

         printf("\n\nSolution of the constraint multipliers, lambda\n");
         for( i = 0; i < m; i++ )
         {
            printf("lambda[%d] = %e\n", (int)i, mult_g[i]);
         }
         printf("\n\nSolution of the bound multipliers, z_L and z_U\n");
         for( i = 0; i < n; i++ )
         {
            printf("z_L[%d] = %e\n", (int)i, mult_x_L[i]);
         }
         for( i = 0; i < n; i++ )
         {
            printf("z_U[%d] = %e\n", (int)i, mult_x_U[i]);
         }

         printf("\n\nObjective value\nf(x*) = %e\n", obj);
      }
      else
      {
         printf("\n\nERROR OCCURRED DURING IPOPT OPTIMIZATION WITH WARM START.\n");
      }
   }

   /* free allocated memory */
   FreeIpoptProblem(nlp);
   free(x);
   free(mult_g);
   free(mult_x_L);
   free(mult_x_U);

   return (status == Solve_Succeeded) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Here, we declare all the necessary variables and set the dimensions of the problem. The problem has 4 variables, so we set n and allocate space for the variable bounds (don't forget to call free for each of your malloc calls before the end of the program). We then set the values for the variable bounds.

The problem has 2 constraints, so we set m and allocate space for the constraint bounds. The first constraint has a lower bound of 25 and no upper bound. Here we set the upper bound to `2e19`. Ipopt interprets any number greater than or equal to the value of option **nlp_upper_bound_inf** as infinity. The default value of **nlp_lower_bound_inf** and **nlp_upper_bound_inf** is `-1e19` and `1e19`, respectively, and can be changed through Ipopt options. The second constraint is an equality with right hand side 40, so we set both the upper and the lower bound to 40.

Next, we set the number of elements in the Jacobian and Hessian. The Jacobian has 8 nonzero entries. For the Hessian we only specify the number of nonzeros in the lower-triangular part, which is 10. See **Triplet Format for Sparse Matrices** for a description of the sparse matrix format. Finally, we set the style for the row and column indices of the matrices to 0 to indicate C-Style, that is, start indexing at 0.

We next create an instance of the **IpoptProblem** by calling **CreateIpoptProblem**, giving it the problem dimensions and the variable and constraint bounds. We also include the references to each of our callback functions. Ipopt uses these function pointers to ask for evaluation of the NLP when required.

After freeing the bound arrays that are no longer required, the next three lines illustrate how to change the value of options through the interface. Ipopt options can also be changed by creating an `ipopt.opt` file (see **Ipopt Options**). We next allocate space for the initial point and set the values as given in the problem definition.

The call to **IpoptSolve** can provide information about the solution, but most of this is optional. Here, we want values for the constraint and variable bound multipliers at the solution and thus allocate space for these.

Further, we initialize the `user_data` that will be accessed in the function evaluation callbacks.

We can now make the call to **IpoptSolve** and find the solution of the problem. We pass in the **IpoptProblem**, the starting point `x` (Ipopt will use this array to return the solution or final point as well). The next 5 arguments are pointers so Ipopt can fill in values at the solution. If these pointers are set to `NULL`, Ipopt will ignore that entry. For example, here we do not want the constraint function values at the solution, so we set this entry to `NULL`. We do want the value of the objective, and the multipliers for the constraints and variable bounds. The last argument is a pointer to our example-specific "user data". Any pointer that is passed in here will be passed on to the callback functions.

The return code is an **ApplicationReturnStatus** enumeration, see the header file ReturnCodes_inc.h which is installed along **IpStdCInterface.h** in the Ipopt include directory. After the optimizer terminates, we check the status and print the solution if successful.

If the solve succeeded, we resolve a slightly modified version of the problem. We use both the primal solution point and the dual multipliers to warm-start Ipopt for this second solve. Additionally, to improve the warmstart, we reduce the amount by which Ipopt pushes the starting point into the interior of the variable bounds by setting the options `bound_push` and `bound_frac`. If the solve was successful, we print the solution again.

Finally, we free the **IpoptProblem** and the remaining memory and return from `main`.

The Fortran Interface

The Fortran interface is essentially a wrapper of the **C Interface**. The way to hook up Ipopt in a Fortran program is very similar to how it is done for the C interface, and the functions of the Fortran interface correspond one-to-one to the those of the C and C++ interface, including their arguments. You can find an implementation of the example problem (HS071) in `$IPOPTDIR/Ipopt/examples/hs071_f`.

The only special things to consider are:

- The return value of the function `IPCREATE` is of type `INTEGER` that must be large enough to capture a pointer on the particular machine. This means, that you have to declare the "handle" for the `IpoptProblem` as `INTEGER*8` if your program is compiled in 64-bit mode. All other `INTEGER`-type variables must be of the regular type.

- For the call of `IPSOLVE` (which is the function that is to be called to run Ipopt), all arrays, including those for the dual variables, must be given (in contrast to the C interface). The return value `IERR` of this function indicates the outcome of the optimization (see the include file `IpReturnCodes.inc` in the Ipopt include directory).
- The return value `IERR` of the remaining functions has to be set to zero, unless there was a problem during execution of the function call.
- The callback functions (`EV_*` in the example) include the arguments `IDAT` and `DAT`, which are `INTEGER` and `DOUBLE PRECISION` arrays that are passed unmodified between the main program calling `IPSOLVE` and the evaluation subroutines `EV_*` (similarly to `UserDataPtr` arguments in the C interface). These arrays can be used to pass "private" data between the main program and the user-provided Fortran subroutines.

  The last argument of the `EV_*` subroutines, `IERR`, is to be set to 0 by the user on return, unless there was a problem during the evaluation of the optimization problem function/derivative for the given point `X` (then it should return a non-zero value).

The Java Interface JIpopt

This section is based on documentation by Rafael de Pelegrini Soares (VRTech Industrial Technologies).

The Java Interface was written and contributed by Rafael de Pelegrini Soares and later updated by Tong Kewei (Beihang University). It offers an abstract base class **Ipopt** with basic methods to specify an NLP, set a number of Ipopt options, to request Ipopt to solve the NLP, and to retrieve a found solution, if any. A HTML documentation of all available interface methods of the **Ipopt** class can also be generated via javadoc by executing `make javadoc` in the Ipopt build directory (`$IPOPTDIR`).

If the Ipopt build includes the Java interface, then a JAR file `org.coinor.ipopt.jar` containing class **org.coinor.Ipopt** will have been installed in `$PREFIX/share/java`. To use the Java interface, make sure that this jar file is part of your Java classpath. A sample makefile that compiles and runs example HS071 can be found in `$IPOPTDIR/examples/hs071_java`.

In the following, we discuss necessary steps to implement example (HS071) with `JIpopt`. We create a new Java source file HS071.java and define a class `HS071` that extends the class **Ipopt**. In the class constructor, we call the **org.coinor.Ipopt.create()** method of `JIpopt`, which works analogously to **Ipopt::TNLP::get_nlp_info()** of the C++ interface. It initializes an **Ipopt::IpoptApplication** object and informs `JIpopt` about the problem size (number of variables, constraints, nonzeros in Jacobian and Hessian).

```
   public HS071()
   {
      /* Number of nonzeros in the Jacobian of the constraints */
      nele_jac = 8;

      /* Number of nonzeros in the Hessian of the Lagrangian (lower or
       * upper triangual part only)
       */
      nele_hess = 10;

      /* Number of variables */
      n = 4;

      /* Number of constraints */
      m = 2;

      /* Index style for the irow/jcol elements */
      int index_style = Ipopt.C_STYLE;
      /* Whether to print iterate in intermediate_callback */
      printiterate = false;

      /* create the IpoptProblem */
      create(n, m, nele_jac, nele_hess, index_style);
   }
```

Next, we add callback functions that are called by `JIpopt` to obtain variable bounds, constraint sides, and a starting point:

```
   protected boolean get_bounds_info(
      int      n,
      double[] x_L,
      double[] x_U,
      int      m,
      double[] g_L,
      double[] g_U)
   {
      assert n == this.n;
      assert m == this.m;

      /* set the values of the variable bounds */
      for( int i = 0; i < n; ++i )
      {
         x_L[i] = 1.0;
         x_U[i] = 5.0;
      }

      /* set the values of the constraint bounds */
      g_L[0] = 25.0;
      g_U[0] = 2e19;
      g_L[1] = 40.0;
      g_U[1] = 40.0;

      return true;
   }
```

```
   protected boolean get_starting_point(
      int      n,
      boolean  init_x,
      double[] x,
      boolean  init_z,
      double[] z_L,
      double[] z_U,
      int      m,
      boolean  init_lambda,
      double[] lambda)
   {
      assert init_z == false;
      assert init_lambda = false;

      if( init_x )
      {
         x[0] = 1.0;
         x[1] = 5.0;
         x[2] = 5.0;
         x[3] = 1.0;
      }

      return true;
   }
```

In the following, we implement the evaluation methods in a way that is very similar to the C++ interface:

```
   protected boolean eval_f(
      int      n,
      double[] x,
      boolean  new_x,
      double[] obj_value)
   {
      assert n == this.n;

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

```java
      return true;
   }

   protected boolean eval_grad_f(
      int      n,
      double[] x,
      boolean  new_x,
      double[] grad_f)
   {
      assert n == this.n;

      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;
   }

   protected boolean eval_g(
      int      n,
      double[] x,
      boolean  new_x,
      int      m,
      double[] g)
   {
      assert n == this.n;
      assert m == this.m;

      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;
   }

   protected boolean eval_jac_g(
      int      n,
      double[] x,
      boolean  new_x,
      int      m,
      int      nele_jac,
      int[]    iRow,
      int[]    jCol,
      double[] values)
   {
      assert n == this.n;
      assert m == this.m;

      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] = 2.0 * x[0];           /* 1,0 */
         values[5] = 2.0 * x[1];           /* 1,1 */
         values[6] = 2.0 * x[2];           /* 1,2 */
         values[7] = 2.0 * x[3];           /* 1,3 */
      }

      return true;
   }

   protected boolean eval_h(
      int      n,
      double[] x,
      boolean  new_x,
      double   obj_factor,
      int      m,
      double[] lambda,
      boolean  new_lambda,
      int      nele_hess,
      int[]    iRow,
      int[]    jCol,
      double[] values)
   {
      assert n == this.n;
      assert m == this.m;

      int idx = 0; /* nonzero element counter */
      int row = 0; /* row counter for loop */
      int col = 0; /* col counter for loop */

      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 */
         idx = 0;
         for( row = 0; row < n; ++row )
            for (col = 0; col <= row; ++col)
            {
               iRow[idx] = row;
               jCol[idx] = col;
               ++idx;
            }

         assert idx == nele_hess;
         assert nele_hess == this.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 * (2.0 * x[3]);               /* 0,0 */
         values[1] = obj_factor * (x[3]);                     /* 1,0 */
         values[2] = 0.0;                                     /* 1,1 */
         values[3] = obj_factor * (x[3]);                     /* 2,0 */
         values[4] = 0.0;                                     /* 2,1 */
         values[5] = 0.0;                                     /* 2,2 */
         values[6] = obj_factor * (2.0 * 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] = 0.0;                                     /* 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;                    /* 0,0 */
        values[2] += lambda[1] * 2.0;                    /* 1,1 */
        values[5] += lambda[1] * 2.0;                    /* 2,2 */
        values[9] += lambda[1] * 2.0;                    /* 3,3 */
     }

     return true;
  }
```

For this example, we override the callback that is called by Ipopt in every iteration and can be used to indicate to Ipopt whether to stop prematurely. We use this callback to print the current iterate and its violations of primal and dual feasibility (obtained via **get_curr_iterate()** and **get_curr_violations()**, resp.):

```
  public boolean intermediate_callback(
     int      algorithmmode,
     int      iter,
     double   obj_value,
     double   inf_pr,
     double   inf_du,
     double   mu,
     double   d_norm,
     double   regularization_size,
     double   alpha_du,
     double   alpha_pr,
     int      ls_trials,
     long     ip_data,
     long     ip_cq)
  {
     if( !printiterate )
     {
        return true;
     }

     double x[] = new double[n];
     double z_L[] = new double[n];
     double z_U[] = new double[n];
     double x_L_viol[] = new double[n];
     double x_U_viol[] = new double[n];
     double compl_x_L[] = new double[n];
     double compl_x_U[] = new double[n];
     double grad_lag_x[] = new double[n];
     double g[] = new double[m];
     double lambda[] = new double[m];
     double constr_viol[] = new double[m];
     double compl_g[] = new double[m];

     boolean have_iter = get_curr_iterate(ip_data, ip_cq, false, n, x, z_L, z_U, m, g, lambda);
     boolean have_viol = get_curr_violations(ip_data, ip_cq, false, n, x_L_viol, x_U_viol,
       compl_x_L, compl_x_U, grad_lag_x, m, constr_viol, compl_g);

     System.out.println("Current iterate at iteration " + iter + ":");
     System.out.println("  x z_L z_U bound_viol compl_x_L compl_x_U grad_lag_x");
     for( int i = 0; i < n; ++i )
     {
        if( have_iter )
        {
           System.out.print("  " + x[i] + " " + z_L[i] + " " + z_U[i]);
        }
        else
        {
           System.out.print("  n/a n/a n/a");
        }
        if( have_viol )
        {
           System.out.println(" " + Math.max(x_L_viol[i], x_U_viol[i]) + " " + compl_x_L[i] + " " +
       compl_x_U[i] + " " + grad_lag_x[i]);
        }
        else
        {
           System.out.println("  n/a/ n/a n/a n/a");
        }
     }
     System.out.println("  g(x) lambda constr_viol compl_g");
     for( int i = 0; i < m; ++i )
```

```
      {
         if( have_iter )
         {
            System.out.print("  " + g[i] + " " + lambda[i]);
         }
         else
         {
            System.out.print("  n/a n/a");
         }
         if( have_viol )
         {
            System.out.println(" " + constr_viol[i] + " " + compl_g[i]);
         }
         else
         {
            System.out.println(" n/a + n/a");
         }
      }

      return true;
   }
```

Finally, we add a main routine to run this example. The main routines creates an instance of our class, calls the solve method **OptimizeNLP**, and prints the solution.

```
   public static void main(String[] args)
   {
      // Create the problem
      HS071 hs071 = new HS071();

      // Set some options
      // hs071.setNumericOption("tol",1E-7);
      // hs071.setStringOption("nlp_scaling_method","user-scaling");
      // hs071.setStringOption("print_options_documentation","yes");
      // hs071.setStringOption("warm_start_init_point","yes");
      // hs071.setNumericOption("warm_start_bound_push",1e-9);
      // hs071.setNumericOption("warm_start_bound_frac",1e-9);
      // hs071.setNumericOption("warm_start_slack_bound_frac",1e-9);
      // hs071.setNumericOption("warm_start_slack_bound_push",1e-9);
      // hs071.setNumericOption("warm_start_mult_bound_push",1e-9);

      // enable printing of current iterate in intermediate_callback
      // hs071.printiterate = true;

      // Solve the problem
      int status = hs071.OptimizeNLP();

      // Print the solution
      if( status == SOLVE_SUCCEEDED )
      {
         System.out.println("\n\n*** The problem solved!");
      }
      else
      {
         System.out.println("\n\n*** The problem was not solved successfully!");
      }

      double obj = hs071.getObjectiveValue();
      System.out.println("\nObjective Value = " + obj + "\n");

      double x[] = hs071.getVariableValues();
      hs071.print(x, "Primal Variable Values:");

      double constraints[] = hs071.getConstraintValues();
      hs071.print(constraints, "Constraint Values:");

      double MLB[] = hs071.getLowerBoundMultipliers();
      hs071.print(MLB, "Dual Multipliers for Variable Lower Bounds:");

      double MUB[] = hs071.getUpperBoundMultipliers();
      hs071.print(MUB, "Dual Multipliers for Variable Upper Bounds:");

      double lam[] = hs071.getConstraintMultipliers();
      hs071.print(lam, "Dual Multipliers for Constraints:");
   }
```

The **OptimizeNLP** method returns the Ipopt solve status as integer, which indicates whether the problem was solved successfully. Further, the methods **getObjectiveValue()**, **getVariableValues()**, **getConstraintMultipliers()**, **getLowerBoundMultipliers()**, and **getUpperBoundMultipliers()** can be used to obtain the objective value, the primal solution value of the variables, and dual solution values.

The R Interface ipoptr

This section is based on documentation by Jelmer Ypma (University College London).

The `ipoptr` package (see also **Compiling and Installing the R Interface ipoptr**) offers a R function `ipoptr` which takes an NLP specification, a starting point, and Ipopt options as input and returns information about an Ipopt run (status, message, ...) and a solution point.

In the following, we discuss necessary steps to implement example (HS071) with `ipoptr`. A more detailed documentation of `ipoptr` is available in contrib/RInterface/inst/doc/ipoptr.pdf.

First, we define the objective function and its gradient

```
> eval_f <- function( x ) {
    return( x[1]*x[4]*(x[1] + x[2] + x[3]) + x[3] )
  }
> eval_grad_f <- function( x ) {
    return( c( x[1] * x[4] + x[4] * (x[1] + x[2] + x[3]),
               x[1] * x[4],
               x[1] * x[4] + 1.0,
               x[1] * (x[1] + x[2] + x[3]) ) )
  }
```

Then we define a function that returns the value of the two constraints. We define the bounds of the constraints (in this case the `g_L` and `g_U` are 25 and 40) later.

```
> # constraint functions
> eval_g <- function( x ) {
     return( c( x[1] * x[2] * x[3] * x[4],
                x[1]^2 + x[2]^2 + x[3]^2 + x[4]^2 ) )
  }
```

Then we define the structure of the Jacobian, which is a dense matrix in this case, and function to evaluate it

```
> eval_jac_g_structure <- list( c(1,2,3,4), c(1,2,3,4) )
> eval_jac_g <- function( x ) {
     return( c ( x[2]*x[3]*x[4],
                 x[1]*x[3]*x[4],
                 x[1]*x[2]*x[4],
                 x[1]*x[2]*x[3],
                 2.0*x[1],
                 2.0*x[2],
                 2.0*x[3],
                 2.0*x[4] ) )
  }
```

The Hessian is also dense, but it looks slightly more complicated because we have to take into account the Hessian of the objective function and of the constraints at the same time, although you could write a function to calculate them both separately and then return the combined result in `eval_h`.

```
> # The Hessian for this problem is actually dense,
> # This is a symmetric matrix, fill the lower left triangle only.
> eval_h_structure <- list( c(1), c(1,2), c(1,2,3), c(1,2,3,4) )
> eval_h <- function( x, obj_factor, hessian_lambda ) {
     values <- numeric(10)
     values[1] = obj_factor * (2*x[4]) # 1,1
```

```
    values[2] = obj_factor * (x[4])   # 2,1
    values[3] = 0                     # 2,2

    values[4] = obj_factor * (x[4])   # 3,1
    values[5] = 0                     # 4,2
    values[6] = 0                     # 3,3

    values[7] = obj_factor * (2*x[1] + x[2] + x[3]) # 4,1
    values[8] = obj_factor * (x[1])                 # 4,2
    values[9] = obj_factor * (x[1])                 # 4,3
    values[10] = 0                                  # 4,4

    # add the portion for the first constraint
    values[2] = values[2] + hessian_lambda[1] * (x[3] * x[4]) # 2,1

    values[4] = values[4] + hessian_lambda[1] * (x[2] * x[4]) # 3,1
    values[5] = values[5] + hessian_lambda[1] * (x[1] * x[4]) # 3,2

    values[7] = values[7] + hessian_lambda[1] * (x[2] * x[3]) # 4,1
    values[8] = values[8] + hessian_lambda[1] * (x[1] * x[3]) # 4,2
    values[9] = values[9] + hessian_lambda[1] * (x[1] * x[2]) # 4,3

    # add the portion for the second constraint
    values[1] = values[1] + hessian_lambda[2] * 2 # 1,1
    values[3] = values[3] + hessian_lambda[2] * 2 # 2,2
    values[6] = values[6] + hessian_lambda[2] * 2 # 3,3
    values[10] = values[10] + hessian_lambda[2] * 2 # 4,4

    return ( values )
  }
```

After the hard part is done, we only have to define the initial values, the lower and upper bounds of the control variables, and the lower and upper bounds of the constraints. If a variable or a constraint does not have lower or upper bounds, the values `-Inf` or `Inf` can be used. If the upper and lower bounds of a constraint are equal, Ipopt recognizes this as an equality constraint and acts accordingly.

```
> # initial values
> x0 <- c( 1, 5, 5, 1 )
> # lower and upper bounds of control
> lb <- c( 1, 1, 1, 1 )
> ub <- c( 5, 5, 5, 5 )
> # lower and upper bounds of constraints
> constraint_lb <- c(  25, 40 )
> constraint_ub <- c( Inf, 40 )
```

Finally, we can call Ipopt with the `ipoptr` function. In order to redirect the Ipopt output into a file, we use Ipopt's and options.

```
> opts <- list("print_level" = 0,
               "file_print_level" = 12,
               "output_file" = "hs071_nlp.out")
> print( ipoptr( x0 = x0,
                 eval_f = eval_f,
                 eval_grad_f = eval_grad_f,
                 lb = lb,
                 ub = ub,
                 eval_g = eval_g,
                 eval_jac_g = eval_jac_g,
                 constraint_lb = constraint_lb,
                 constraint_ub = constraint_ub,
                 eval_jac_g_structure = eval_jac_g_structure,
                 eval_h = eval_h,
                 eval_h_structure = eval_h_structure,
                 opts = opts) )

Call:

ipoptr(x0 = x0, eval_f = eval_f, eval_grad_f = eval_grad_f, lb = lb,
  ub = ub, eval_g = eval_g, eval_jac_g = eval_jac_g,
  eval_jac_g_structure = eval_jac_g_structure, constraint_lb = constraint_lb,
  constraint_ub = constraint_ub, eval_h = eval_h, eval_h_structure = eval_h_structure,
  opts = opts)
```

```
Ipopt solver status: 0 ( SUCCESS: Algorithm terminated
successfully at a locally optimal point, satisfying the
convergence tolerances (can be specified by options). )

Number of Iterations....: 8
Optimal value of objective function:  17.0140171451792
Optimal value of controls: 1 4.743 3.82115 1.379408
```

To pass additional data to the evaluation routines, one can either supply additional arguments to the user defined functions and `ipoptr` or define an environment that holds the data and pass this environment to `ipoptr`. Both methods are shown in the file `tests/parameters.R` that comes with `ipoptr`.

As a very simple example, suppose we want to find the minimum of $f(x) = a_1 x^2 + a_2 x + a_3$ for different values of the parameters a_1, a_2 and a_3.

First, we define the objective function and its gradient using, assuming that there is some variable `params` that contains the values of the parameters.

```
> eval_f_ex1 <- function(x, params) {
      return( params[1]*x^2 + params[2]*x + params[3] )
  }
> eval_grad_f_ex1 <- function(x, params) {
      return( 2*params[1]*x + params[2] )
  }
```

Note, that the first parameter should always be the control variable. All of the user-defined functions should contain the same set of additional parameters. You have to supply them as input argument to all functions, even if you're not using them in some of the functions.

Then we can solve the problem for a specific set of parameters, in this case $a_1 = 1$, $a_2 = 2$, and $a_3 = 3$, from initial value $x_0 = 0$, with the following command

```
> # solve using ipoptr with additional parameters
> ipoptr(x0          = 0,
         eval_f      = eval_f_ex1,
         eval_grad_f = eval_grad_f_ex1,
         opts        = list("print_level"=0),
         params      = c(1,2,3) )

Call:

ipoptr(x0 = 0, eval_f = eval_f_ex1, eval_grad_f = eval_grad_f_ex1,
    opts = list(print_level = 0), params = c(1, 2, 3))


Ipopt solver status: 0 ( SUCCESS: Algorithm terminated
successfully at a locally optimal point, satisfying the
convergence tolerances (can be specified by options). )

Number of Iterations....: 1
Optimal value of objective function:  2
Optimal value of controls: -1
```

For the second method, we don't have to supply the parameters as additional arguments to the function.

```
> eval_f_ex2 <- function(x) {
      return( params[1]*x^2 + params[2]*x + params[3] )
  }
> eval_grad_f_ex2 <- function(x) {
      return( 2*params[1]*x + params[2] )
  }
```

Instead, we define an environment that contains specific values of `params`:

```
> # define a new environment that contains params
> auxdata        <- new.env()
> auxdata$params <- c(1,2,3)
```

To solve this we supply `auxdata` as an argument to `ipoptr`, which will take care of evaluating the functions in the correct environment, so that the auxiliary data is available.

```
> # pass the environment that should be used to evaluate functions to ipoptr
> ipoptr(x0                 = 0,
         eval_f             = eval_f_ex2,
         eval_grad_f        = eval_grad_f_ex2,
         ipoptr_environment = auxdata,
         opts               = list("print_level"=0) )

Call:

ipoptr(x0 = 0, eval_f = eval_f_ex2, eval_grad_f = eval_grad_f_ex2,
    opts = list(print_level = 0), ipoptr_environment = auxdata)


Ipopt solver status: 0 ( SUCCESS: Algorithm terminated
successfully at a locally optimal point, satisfying the
convergence tolerances (can be specified by options). )

Number of Iterations....: 1
Optimal value of objective function:  2
Optimal value of controls: -1
```