Modellierung eines Beispiels in IPOPT

Franziska Krüger

19. März 2010

Inhaltsverzeichnis

1 Das Beispiel

Wir betrachten ein einfaches Beispiel einer nichtlinearen Optimierungsaufgabe, um die verschiedenen Möglichkeiten der Problemmodellierung in IPOPT darzustellen.
Betrachte also

$$\begin{cases} \min\limits_{x\in\mathbb{R}^4} & x^{(1)}x^{(4)}(x^{(1)}+x^{(2)}+x^{(3)})+x^{(3)} \\ \text{mit} & x^{(1)}x^{(2)}x^{(3)}x^{(4)} \geq 25 \\ & (x^{(1)})^2+(x^{(2)})^2+(x^{(3)})^2+(x^{(4)})^2 = 40 \\ & 1 \leq x \leq 5 \end{cases}$$

1.1 Modellierung in AMPL

1.1.1 hs71.mod

```
# Copyright (C) 2009, International Business Machines
#
# This file is part of the Ipopt open source package, published under
# the Common Public License
#
# Author:  Andreas Waechter         IBM       2009-04-03

# This is a model of Example 71 from
#
# Hock, W, and Schittkowski, K,
# Test Examples for Nonlinear Programming Codes,
# Lecture Notes in Economics and Mathematical Systems.
# Springer Verlag, 1981.

##############################################################################

# Definition of the variables with bounds
var x {i in 1..4}, >= 1, <= 5;

# objective function
minimize obj: x[1]*x[4]*(x[1] + x[2] + x[3]) + x[3];

# and the constraints
```

```
subject to c1: x[1]*x[2]*x[3]*x[4] >= 25;
subject to c2: x[1]^2+x[2]^2+x[3]^2+x[4]^2 = 40;

# Now we set the starting point:

let x[1] := 1;
let x[2] := 5;
let x[3] := 5;
let x[4] := 1;
```

1.2 Modellierung in C

1.2.1 hs071_c.c

```
/* Copyright (C) 2005, 2006 International Business Machines and others.
 * All Rights Reserved.
 * This code is published under the Common Public License.
 *
 * $Id: hs071_c.c 1324 2008-09-16 14:19:26Z andreasw $
 *
 * Authors:  Carl Laird, Andreas Waechter     IBM    2005-08-17
 */

#include "IpStdCInterface.h"
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>

/* Function Declarations */
Bool eval_f(Index n, Number* x, Bool new_x,
            Number* obj_value, UserDataPtr user_data);

Bool eval_grad_f(Index n, Number* x, Bool new_x,
                 Number* grad_f, UserDataPtr user_data);

Bool eval_g(Index n, Number* x, Bool new_x,
            Index m, Number* g, UserDataPtr user_data);

Bool eval_jac_g(Index n, Number *x, Bool new_x,
                Index m, Index nele_jac,
                Index *iRow, Index *jCol, Number *values,
                UserDataPtr user_data);

Bool eval_h(Index n, Number *x, Bool new_x, Number obj_factor,
            Index m, Number *lambda, Bool new_lambda,
            Index nele_hess, Index *iRow, Index *jCol,
            Number *values, UserDataPtr user_data);

/* Main Program */
int main()
{
  Index n=-1;                          /* number of variables */
```

```
Index m=-1;                          /* number of constraints */
Number* x_L = NULL;                  /* lower bounds on x */
Number* x_U = NULL;                  /* upper bounds on x */
Number* g_L = NULL;                  /* lower bounds on g */
Number* g_U = NULL;                  /* upper bounds on g */
IpoptProblem nlp = NULL;             /* IpoptProblem */
enum ApplicationReturnStatus status; /* Solve return code */
Number* x = NULL;                    /* starting point and solution vector */
Number* mult_x_L = NULL;             /* lower bound multipliers
   at the solution */
Number* mult_x_U = NULL;             /* upper bound multipliers
   at the solution */
Number obj;                          /* objective value */
Index i;                             /* generic counter */

/* Number of nonzeros in the Jacobian of the constraints */
Index nele_jac = 8;
/* Number of nonzeros in the Hessian of the Lagrangian (lower or
   upper triangual part only) */
Index nele_hess = 10;
/* indexing style for matrices */
Index index_style = 0; /* C-style; start counting of rows and column
     indices at 0 */

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

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

/* create the IpoptProblem */
nlp = CreateIpoptProblem(n, x_L, x_U, m, g_L, g_U, 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", 1e-7);
  AddIpoptStrOption(nlp, "mu_strategy", "adaptive");
  AddIpoptStrOption(nlp, "output_file", "ipopt.out");

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

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

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

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

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

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

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

  return 0;
}
```

```
/* Function Implementations */
Bool eval_f(Index n, Number* x, Bool new_x,
            Number* obj_value, UserDataPtr user_data)
{
  assert(n == 4);

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

  return TRUE;
}

Bool eval_grad_f(Index n, Number* x, Bool new_x,
                 Number* grad_f, UserDataPtr user_data)
{
  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;
}

Bool eval_g(Index n, Number* x, Bool new_x,
            Index m, Number* g, UserDataPtr user_data)
{
  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;
}

Bool eval_jac_g(Index n, Number *x, Bool new_x,
                Index m, Index nele_jac,
                Index *iRow, Index *jCol, Number *values,
                UserDataPtr user_data)
{
  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;
}

Bool eval_h(Index n, Number *x, Bool new_x, Number obj_factor,
            Index m, Number *lambda, Bool new_lambda,
            Index nele_hess, Index *iRow, Index *jCol,
            Number *values, UserDataPtr user_data)
{
  Index idx = 0; /* nonzero element counter */
  Index row = 0; /* row counter for loop */
  Index 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 < 4; row++) {
      for (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;
}
```

1.3 Modellierung in C++

1.3.1 hs071_nlp.hpp

```
// Copyright (C) 2005, 2007 International Business Machines and others.
// All Rights Reserved.
// This code is published under the Common Public License.
//
// $Id: hs071_nlp.hpp 1324 2008-09-16 14:19:26Z andreasw $
//
```

```
// Authors:  Carl Laird, Andreas Waechter     IBM    2005-08-09

#ifndef __HS071_NLP_HPP__
#define __HS071_NLP_HPP__

#include "IpTNLP.hpp"

using namespace Ipopt;

/** C++ Example NLP for interfacing a problem with IPOPT.
 *  HS071_NLP implements a C++ example of problem 71 of the
 *  Hock-Schittkowski test suite. This example is designed to go
 *  along with the tutorial document and show how to interface
 *  with IPOPT through the TNLP interface.
 *
 * Problem hs071 looks like this
 *
 *     min   x1*x4*(x1 + x2 + x3)  +  x3
 *     s.t.  x1*x2*x3*x4                   >=  25
 *           x1**2 + x2**2 + x3**2 + x4**2  =  40
 *           1 <=  x1,x2,x3,x4  <= 5
 *
 *     Starting point:
 *        x = (1, 5, 5, 1)
 *
 *     Optimal solution:
 *        x = (1.00000000, 4.74299963, 3.82114998, 1.37940829)
 *
 *
 */
class HS071_NLP : public TNLP
{
public:
  /** default constructor */
  HS071_NLP();

  /** default destructor */
  virtual ~HS071_NLP();

  /**@name Overloaded from TNLP */
  //@{
  /** Method to return some info about the nlp */
  virtual bool get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                            Index& nnz_h_lag, IndexStyleEnum& index_style);

  /** Method to return the bounds for my problem */
  virtual bool get_bounds_info(Index n, Number* x_l, Number* x_u,
                               Index m, Number* g_l, Number* g_u);

  /** Method to return the starting point for the algorithm */
  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);

  /** Method to return the objective value */
  virtual bool eval_f(Index n, const Number* x, bool new_x, Number& obj_value);

  /** Method to return the gradient of the objective */
  virtual bool eval_grad_f(Index n, const Number* x, bool new_x, Number* grad_f);

  /** Method to return the constraint residuals */
  virtual bool eval_g(Index n, const Number* x, bool new_x, Index m, Number* g);

  /** Method to return:
   *   1) The structure of the jacobian (if "values" is NULL)
   *   2) The values of the jacobian (if "values" is not NULL)
   */
  virtual bool eval_jac_g(Index n, const Number* x, bool new_x,
                          Index m, Index nele_jac, Index* iRow, Index *jCol,
                          Number* values);

  /** Method to return:
   *   1) The structure of the hessian of the lagrangian (if "values" is NULL)
   *   2) The values of the hessian of the lagrangian (if "values" is not NULL)
   */
  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);

  //@}

  /** @name Solution Methods */
  //@{
  /** This method is called when the algorithm is complete so the TNLP can store/write the solution
  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);
  //@}

private:
  /**@name Methods to block default compiler methods.
   * The compiler automatically generates the following three methods.
   *  Since the default compiler implementation is generally not what
   *  you want (for all but the most simple classes), we usually
   *  put the declarations of these methods in the private section
   *  and never implement them. This prevents the compiler from
   *  implementing an incorrect "default" behavior without us
   *  knowing. (See Scott Meyers book, "Effective C++")
   *
```

```
   */
  //@{
  //  HS071_NLP();
  HS071_NLP(const HS071_NLP&);
  HS071_NLP& operator=(const HS071_NLP&);
  //@}
};


#endif
```

1.3.2 hs071_main.cpp

```
// Copyright (C) 2005, 2009 International Business Machines and others.
// All Rights Reserved.
// This code is published under the Common Public License.
//
// $Id: hs071_main.cpp 1597 2009-10-29 15:23:18Z andreasw $
//
// Authors:  Carl Laird, Andreas Waechter     IBM    2005-08-10

#include "IpIpoptApplication.hpp"
#include "hs071_nlp.hpp"

// for printf
#ifdef HAVE_CSTDIO
# include <cstdio>
#else
# ifdef HAVE_STDIO_H
#  include <stdio.h>
# else
#  error "don't have header file for stdio"
# endif
#endif

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", 1e-7);
  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");

  // Intialize the IpoptApplication and process the options
  ApplicationReturnStatus status;
  status = app->Initialize();
  if (status != Solve_Succeeded) {
    printf("\n\n*** Error during initialization!\n");
    return (int) status;
  }

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

  if (status == Solve_Succeeded) {
    printf("\n\n*** The problem solved!\n");
  }
  else {
    printf("\n\n*** The problem FAILED!\n");
  }

  // As the SmartPtrs go out of scope, the reference count
  // will be decremented and the objects will automatically
  // be deleted.

  return (int) status;
}
```

1.3.3 hs071_nlp.cpp

```
// Copyright (C) 2005, 2006 International Business Machines and others.
// All Rights Reserved.
// This code is published under the Common Public License.
//
// $Id: hs071_nlp.cpp 1324 2008-09-16 14:19:26Z andreasw $
//
// Authors:  Carl Laird, Andreas Waechter     IBM    2005-08-16

#include "hs071_nlp.hpp"

// for printf
#ifdef HAVE_CSTDIO
# include <cstdio>
#else
# ifdef HAVE_STDIO_H
#  include <stdio.h>
# else
#  error "don't have header file for stdio"
```

```
# endif
#endif

using namespace Ipopt;

// constructor
HS071_NLP::HS071_NLP()
{}

//destructor
HS071_NLP::~HS071_NLP()
{}

// 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;
}

// 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;
}

// 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;
}

// 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;
}

// 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;
}

// 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;
}

// 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)
{
  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;
}

//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)
{
  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;
}

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

  printf("\n\nSolution of the bound multipliers, z_L and z_U\n");
  for (Index i=0; i<n; i++) {
    printf("z_L[%d] = %e\n", i, z_L[i]);
  }
  for (Index i=0; i<n; i++) {
    printf("z_U[%d] = %e\n", i, z_U[i]);
  }

  printf("\n\nObjective value\n");
```

```
  printf("f(x*) = %e\n", obj_value);

  printf("\nFinal value of the constraints:\n");
  for (Index i=0; i<m ;i++) {
    printf("g(%d) = %e\n", i, g[i]);
  }
}
```

1.4 Modellierung in Fortran

1.4.1 hs071_f.f

```
C Copyright (C) 2002, 2007 Carnegie Mellon University and others.
C All Rights Reserved.
C This code is published under the Common Public License.
C
C    $Id: hs071_f.f.in 991 2007-06-09 07:20:55Z andreasw $
C
C =============================================================================
C
C     This is an example for the usage of IPOPT.
C     It implements problem 71 from the Hock-Schittkowski test suite:
C
C     min   x1*x4*(x1 + x2 + x3)  +  x3
C     s.t.  x1*x2*x3*x4                   >=  25
C           x1**2 + x2**2 + x3**2 + x4**2  =  40
C           1 <=  x1,x2,x3,x4  <= 5
C
C     Starting point:
C        x = (1, 5, 5, 1)
C
C     Optimal solution:
C        x = (1.00000000, 4.74299963, 3.82114998, 1.37940829)
C
C =============================================================================
C
C
C =============================================================================
C
C                            Main driver program
C
C =============================================================================
C
      program example
C
      implicit none
C
C     include the Ipopt return codes
C
      include 'IpReturnCodes.inc'
C
C     Size of the problem (number of variables and equality constraints)
```

```
C
      integer     N,     M,     NELE_JAC,     NELE_HESS,      IDX_STY
      parameter  (N = 4, M = 2, NELE_JAC = 8, NELE_HESS = 10)
      parameter  (IDX_STY = 1 )
C
C     Space for multipliers and constraints
C
      double precision LAM(M)
      double precision G(M)
C
C     Vector of variables
C
      double precision X(N)
C
C     Vector of lower and upper bounds
C
      double precision X_L(N), X_U(N), Z_L(N), Z_U(N)
      double precision G_L(M), G_U(M)
C
C     Private data for evaluation routines
C     This could be used to pass double precision and integer arrays untouched
C     to the evaluation subroutines EVAL_*
C
      double precision DAT(2)
      integer IDAT(1)
C
C     Place for storing the Ipopt Problem Handle
C
CC     for 32 bit platforms
C      integer IPROBLEM
C      integer IPCREATE
C     for 64 bit platforms:
      integer*8 IPROBLEM
      integer*8 IPCREATE
C
      integer IERR
      integer IPSOLVE, IPADDSTROPTION
      integer IPADDNUMOPTION, IPADDINTOPTION
      integer IPOPENOUTPUTFILE
C
      double precision F
      integer i
C
C     The following are the Fortran routines for computing the model
C     functions and their derivatives - their code can be found furhter
C     down in this file.
C
      external EV_F, EV_G, EV_GRAD_F, EV_JAC_G, EV_HESS
C
C     Set initial point and bounds:
C
      data X   / 1d0, 5d0, 5d0, 1d0/
```

```
      data X_L / 1d0, 1d0, 1d0, 1d0 /
      data X_U / 5d0, 5d0, 5d0, 5d0 /
C
C     Set bounds for the constraints
C
      data G_L / 25d0, 40d0 /
      data G_U / 1d40, 40d0 /
C
C     First create a handle for the Ipopt problem (and read the options
C     file)
C
      IPROBLEM = IPCREATE(N, X_L, X_U, M, G_L, G_U, NELE_JAC, NELE_HESS,
     1     IDX_STY, EV_F, EV_G, EV_GRAD_F, EV_JAC_G, EV_HESS)
      if (IPROBLEM.eq.0) then
         write(*,*) 'Error creating an Ipopt Problem handle.'
         stop
      endif
C
C     Open an output file
C
      IERR = IPOPENOUTPUTFILE(IPROBLEM, 'IPOPT.OUT', 5)
      if (IERR.ne.0 ) then
         write(*,*) 'Error opening the Ipopt output file.'
         goto 9000
      endif
C
C     Note: The following options are only examples, they might not be
C           suitable for your optimization problem.
C
C     Set a string option
C
      IERR = IPADDSTROPTION(IPROBLEM, 'mu_strategy', 'adaptive')
      if (IERR.ne.0 ) goto 9990
C
C     Set an integer option
C
      IERR = IPADDINTOPTION(IPROBLEM, 'max_iter', 3000)
      if (IERR.ne.0 ) goto 9990
C
C     Set a double precision option
C
      IERR = IPADDNUMOPTION(IPROBLEM, 'tol', 1.d-7)
      if (IERR.ne.0 ) goto 9990
C
C     As a simple example, we pass the constants in the constraints to
C     the EVAL_C routine via the "private" DAT array.
C
      DAT(1) = 0.d0
      DAT(2) = 0.d0
C
C     Call optimization routine
C
```

```
      IERR = IPSOLVE(IPROBLEM, X, G, F, LAM, Z_L, Z_U, IDAT, DAT)
C
C     Output:
C
      if( IERR.eq.IP_SOLVE_SUCCEEDED ) then
         write(*,*)
         write(*,*) 'The solution was found.'
         write(*,*)
         write(*,*) 'The final value of the objective function is ',F
         write(*,*)
         write(*,*) 'The optimal values of X are:'
         write(*,*)
         do i = 1, N
            write(*,*) 'X  (',i,') = ',X(i)
         enddo
         write(*,*)
         write(*,*) 'The multipliers for the lower bounds are:'
         write(*,*)
         do i = 1, N
            write(*,*) 'Z_L(',i,') = ',Z_L(i)
         enddo
         write(*,*)
         write(*,*) 'The multipliers for the upper bounds are:'
         write(*,*)
         do i = 1, N
            write(*,*) 'Z_U(',i,') = ',Z_U(i)
         enddo
         write(*,*)
         write(*,*) 'The multipliers for the equality constraints are:'
         write(*,*)
         do i = 1, M
            write(*,*) 'LAM(',i,') = ',LAM(i)
         enddo
         write(*,*)
      else
         write(*,*)
         write(*,*) 'An error occoured.'
         write(*,*) 'The error code is ',IERR
         write(*,*)
      endif
C
 9000 continue
C
C     Clean up
C
      call IPFREE(IPROBLEM)
      stop
C
 9990 continue
      write(*,*) 'Error setting an option'
      goto 9000
      end
```

```
C
C =============================================================================
C
C                    Computation of objective function
C
C =============================================================================
C
      subroutine EV_F(N, X, NEW_X, F, IDAT, DAT, IERR)
      implicit none
      integer N, NEW_X
      double precision F, X(N)
      double precision DAT(*)
      integer IDAT(*)
      integer IERR
      F = X(1)*X(4)*(X(1)+X(2)+X(3)) + X(3)
      IERR = 0
      return
      end
C
C =============================================================================
C
C                Computation of gradient of objective function
C
C =============================================================================
C
      subroutine EV_GRAD_F(N, X, NEW_X, GRAD, IDAT, DAT, IERR)
      implicit none
      integer N, NEW_X
      double precision GRAD(N), X(N)
      double precision DAT(*)
      integer IDAT(*)
      integer IERR
      GRAD(1) = X(4)*(2d0*X(1)+X(2)+X(3))
      GRAD(2) = X(1)*X(4)
      GRAD(3) = X(1)*X(4) + 1d0
      GRAD(4) = X(1)*(X(1)+X(2)+X(3))
      IERR = 0
      return
      end
C
C =============================================================================
C
C                     Computation of equality constraints
C
C =============================================================================
C
      subroutine EV_G(N, X, NEW_X, M, G, IDAT, DAT, IERR)
      implicit none
      integer N, NEW_X, M
      double precision G(M), X(N)
      double precision DAT(*)
      integer IDAT(*)
```

```
      integer IERR
      G(1) = X(1)*X(2)*X(3)*X(4) - DAT(1)
      G(2) = X(1)**2 + X(2)**2 + X(3)**2 + X(4)**2 - DAT(2)
      IERR = 0
      return
      end
C
C =============================================================================
C
C                Computation of Jacobian of equality constraints
C
C =============================================================================
C
      subroutine EV_JAC_G(TASK, N, X, NEW_X, M, NZ, ACON, AVAR, A,
     1     IDAT, DAT, IERR)
      integer TASK, N, NEW_X, M, NZ
      double precision X(N), A(NZ)
      integer ACON(NZ), AVAR(NZ), I
      double precision DAT(*)
      integer IDAT(*)
      integer IERR
C
C     structure of Jacobian:
C
      integer AVAR1(8), ACON1(8)
      data  AVAR1 /1, 2, 3, 4, 1, 2, 3, 4/
      data  ACON1 /1, 1, 1, 1, 2, 2, 2, 2/
      save  AVAR1, ACON1
C
      if( TASK.eq.0 ) then
        do I = 1, 8
          AVAR(I) = AVAR1(I)
          ACON(I) = ACON1(I)
        enddo
      else
        A(1) = X(2)*X(3)*X(4)
        A(2) = X(1)*X(3)*X(4)
        A(3) = X(1)*X(2)*X(4)
        A(4) = X(1)*X(2)*X(3)
        A(5) = 2d0*X(1)
        A(6) = 2d0*X(2)
        A(7) = 2d0*X(3)
        A(8) = 2d0*X(4)
      endif
      IERR = 0
      return
      end
C
C =============================================================================
C
C                Computation of Hessian of Lagrangian
C
```

```
C =============================================================================
C
      subroutine EV_HESS(TASK, N, X, NEW_X, OBJFACT, M, LAM, NEW_LAM,
     1     NNZH, IRNH, ICNH, HESS, IDAT, DAT, IERR)
      implicit none
      integer TASK, N, NEW_X, M, NEW_LAM, NNZH, i
      double precision X(N), OBJFACT, LAM(M), HESS(NNZH)
      integer IRNH(NNZH), ICNH(NNZH)
      double precision DAT(*)
      integer IDAT(*)
      integer IERR
C
C     structure of Hessian:
C
      integer IRNH1(10), ICNH1(10)
      data  IRNH1 /1, 2, 2, 3, 3, 3, 4, 4, 4, 4/
      data  ICNH1 /1, 1, 2, 1, 2, 3, 1, 2, 3, 4/
      save  IRNH1, ICNH1

      if( TASK.eq.0 ) then
         do i = 1, 10
            IRNH(i) = IRNH1(i)
            ICNH(i) = ICNH1(i)
         enddo
      else
         do i = 1, 10
            HESS(i) = 0d0
         enddo
C
C     objective function
C
         HESS(1) = OBJFACT * 2d0*X(4)
         HESS(2) = OBJFACT * X(4)
         HESS(4) = OBJFACT * X(4)
         HESS(7) = OBJFACT * (2d0*X(1) + X(2) + X(3))
         HESS(8) = OBJFACT * X(1)
         HESS(9) = OBJFACT * X(1)
C
C     first constraint
C
         HESS(2) = HESS(2) + LAM(1) * X(3)*X(4)
         HESS(4) = HESS(4) + LAM(1) * X(2)*X(4)
         HESS(5) = HESS(5) + LAM(1) * X(1)*X(4)
         HESS(7) = HESS(7) + LAM(1) * X(2)*X(3)
         HESS(8) = HESS(8) + LAM(1) * X(1)*X(3)
         HESS(9) = HESS(9) + LAM(1) * X(1)*X(2)
C
C     second constraint
C
         HESS(1) = HESS(1) + LAM(2) * 2d0
         HESS(3) = HESS(3) + LAM(2) * 2d0
         HESS(6) = HESS(6) + LAM(2) * 2d0
```

```
   HESS(10)= HESS(10)+ LAM(2) * 2d0
endif
IERR = 0
return
end
```