

Appendix B: Code of Port II

B.1 CapeTNLP.hpp

```
// CapeTNLP.hpp --- Definition of class CCapeTNLP
//
// Copyright (C) 2005, International Business Machines and others.
// All Rights Reserved.
// This code is published under the Common Public License.
//
// Authors: Yidong Lang CMU, Carl Laird CMU, Andreas Waechter IBM
// Date: 04-20-2005

#ifndef __CAPETNLPNET_HPP__
#define __CAPETNLPNET_HPP__

#include "GCOIpoptWrapper_NET.h"

#include "IpTNLP.hpp"
#include "IpUtils.hpp"
#include "IpJournalist.hpp"
#include "IpException.hpp"
#include "IpSmartPtr.hpp"

using namespace Ipopt;

class CCapeTNLP: public TNLP
{
// YDL Define data type for CAPE-OPEN
private:
    CapeLong nv,niv,nlv,nliv,nc,nlc,nlz,nnz,nlzof,nnzof;//ICapeMINLPSize
    CapeArrayLong cids;//GetMINLPConstraintBounds
    CapeArrayLong vids;//GetMINLPVariableBounds
    CapeArrayDouble LB,UB;//GetMINLPConstraintBounds

    CapeArrayLong otype;// GetMINLPObjectiveFunctionType
    CapeString structuretype;//GetMINLPStructure
    CapeArrayLong rowindex,columnindex,objindex;
    CapeArrayDouble v;//GetMINLPObjectiveFunctionDerivativesValues
    CapeArrayDouble vals;//GetMINLPConstraintDerivativeValues
    CapeArrayDouble values;//SetMINLPVariableValues
    CapeArrayBoolean isinteger;//GetMINLPVariableTypes

// YDL
    CapeLong nhess ; // GetMINLPHessianStructure
    CapeArrayLong hRowId, hColId ;
    CapeArrayDouble hvalues ;
    CapeDouble objval_nI;

//--

    ICapeMINLP *m_pMinIp;

    bool optim;
    bool boundsError;
```

```

private:
    //@ Size of the model
    int m_objType;
    int m_n, m_m, m_nnz_jac_g, m_nnz_h_lag;
    int m_nlz_jac;

    double *m_lcoefObj;
    int *m_jColObj;

    int *m_iRow, *m_jCol;
    double *m_Jac_lin;

    int *m_iRow_h, *m_jCol_h;

//-----
public:
    /** default constructor */
    CCapeTNLP(ICapeMINLP *pMinlp);

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

    /** @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);

    /** 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);

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 */
virtual void finalize_solution(ApplicationReturnStatus 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);

private:
/** Solution Vectors */
    Number* x_sol_;
    Number* z_L_sol_;
    Number* z_U_sol_;
    Number* g_sol_;
    Number* lambda_sol_;
    Number obj_sol_;
    int mem_sol_;

// CCapeNLP();
    CCapeTNLP(const CCapeTNLP&);
    CCapeTNLP& operator=(const CCapeTNLP&);
    int SendVariablesOut(int type, double *x)
};

#endif

```

B.2 CapeTNLP_net.cpp

```

// CapeTNLP_net.cpp --- Implementation of class CCapeTNLP
// Copyright (C) 2004, International Business Machines and others.
// All Rights Reserved.
// This code is published under the Common Public License.
//
// Authors: Yi-dong Lang CMU, Carl Laird CMU, Andreas Waechter IBM
// Date: 04-20-2005

#include "stdafx.h"

#include <string.h>
#include <iostream>
#include <comdef.h>
#include "csafe.h"

#include "CapeTNLP_net.hpp"
#include "IpBlas.hpp"

int ret;

VARIANT auxVar,auxVarLB,auxVarUB,auxRow,auxCol,auxObj,auxValues,
        auxVarType;

```

```

const int dimensionC=1;
long lIndexC[dimensionC];

const int dimensionV=1;
long lIndexV[dimensionV];

const int dimensionH=1;
long lIndexH[dimensionH];

CSafeArray csaCids;
CSafeArray csaVids;
CSafeArray csaValues;

CapeLong otype;

using namespace lpopt;

/* Constructor. */
CCapeTNLP::CCapeTNLP(ICapeMINLP *pMinlp)
    :TNLP(),
      m_lcoefObj(NULL),
      m_iRow(NULL),
      m_jCol(NULL),
      m_Jac_lin(NULL)
{
    VariantInit(&auxVar);
    VariantInit(&auxVarLB);
    VariantInit(&auxVarUB);
    VariantInit(&auxRow);
    VariantInit(&auxCol);
    VariantInit(&auxObj);
    VariantInit(&auxValues);
    VariantInit(&auxVarType);

    // to be used by VB
    VariantInit(&vids);
    VariantInit(&cids);
    VariantInit(&LB);
    VariantInit(&UB);
    VariantInit(&rowindex);
    VariantInit(&columnindex);
    VariantInit(&objindex);
    VariantInit(&v);
    VariantInit(&vals);
    VariantInit(&values);
    VariantInit(&isinteger);

    VariantInit(&hvalues);

    m_pMinlp = pMinlp;

    //////////////////////////////////////
    //          Get size of the model
    //////////////////////////////////////

```

```

m_pMinlp->GetMINLPSize(&nv,&niv,&nlv,&nliv,&nc,&nlc,&nlz,&nnz,
&nlfzof,&nnzof);

m_n = nv;
m_m = nc;
m_nnz_jac_g = nlz + nnz;

VariantInit(&hRowId);
VariantInit(&hColId);

m_pMinlp->GetMINLPHessianStructure(&nhess,&hRowId,&hColId);

m_nnz_h_lag = nhess;
m_iRow_h = new int[nhess];
m_jCol_h = new int[nhess];

for (int i=0; i<nhess;i++)
{
    lIndexH[0] = i;
    SafeArrayGetElement(hRowId.parray , lIndexH, &auxRow);
    SafeArrayGetElement(hColId.parray , lIndexH, &auxCol);
    m_iRow_h[i] = auxRow.lVal;
    m_jCol_h[i] = auxCol.lVal;
}
////////////////////////////////////
// Initialize variables for safearray
////////////////////////////////////
auxVar.vt=VT_I4;
auxVarLB.vt=VT_R8;
auxVarUB.vt=VT_R8;
auxRow.vt=VT_I4,
auxCol.vt=VT_I4,
auxObj.vt=VT_I4;
auxValues.vt=VT_R8;
auxVarType.vt=VT_BOOL;

long safeBoundsC[2*dimensionC] = {0,nc};
csaCids.Init(1,VT_VARIANT,safeBoundsC);

long safeBoundsV[2*dimensionV]={0,nv};
csaVids.Init(1,VT_VARIANT,safeBoundsV);
csaValues.Init(1,VT_VARIANT,safeBoundsV);

for ( i=0; i<nv; i++)
{//ask for all the variables... nv
    lIndexV[0] = i;
    auxVar.lVal= i+1;//CapeOpen starts at one.
    csaVids.SetElement(lIndexV,auxVar);
}

VariantInit(&vids);
vids.vt=VT_ARRAY|VT_VARIANT;
csaVids.GetSafeArray(& (vids.parray));

```

```

    for (i=0; i<nc; i++)
    {
        //ask for all the constraints... nc
        lIndexC[0] = i;
        auxVar.IVal= i+1;//CapeOpen starts at one.
        csaCids.SetElement(lIndexC,auxVar);
    }

VariantInit(&cids);
cids.vt=VT_ARRAY | VT_VARIANT;
csaCids.GetSafeArray(&(cids.parray));

////////////////////////////////////
//////////////////////////////////// Structure of the NLP Problem //////////////////////////////////////
////////////////////////////////////

    _bstr_t structuretype= TEXT("LINEAR");
    m_pMinlp->GetMINLPStructure(structuretype,&rowindex,
        &columnindex,&objindex);

//
//-----Linear part
    m_iRow = new int [m_nnz_jac_g];
    m_jCol = new int [m_nnz_jac_g];
    m_jColObj = new int[nlzof+nnzof];

    for ( i=0; i<nlz;i++)
    {
        lIndexV[0] = i;
        SafeArrayGetElement(rowindex.parray , lIndexV, &auxRow);
        SafeArrayGetElement(columnindex.parray , lIndexV, &auxCol);
        m_iRow[i] = auxRow.IVal;
        m_jCol[i] = auxCol.IVal;
    }
    for (i=0; i<nlzof;i++)
    {
        lIndexV[0] = i;
        SafeArrayGetElement(objindex.parray , lIndexV, &auxObj);
        m_jColObj[i] = auxObj.IVal;
    }
//----- Nonlinear part
    structuretype= TEXT("NONLINEAR");
    m_pMinlp->GetMINLPStructure(structuretype,&rowindex,
        &columnindex,&objindex);

    for (i=0; i<nnz;i++)
    {
        lIndexV[0] = i;
        SafeArrayGetElement(rowindex.parray , lIndexV, &auxRow);
        SafeArrayGetElement(columnindex.parray , lIndexV, &auxCol);
        m_iRow[nlz+i] = auxRow.IVal;
        m_jCol[nlz+i] = auxCol.IVal;
    }
    for (i=0; i<nnzof;i++)
    {
        lIndexV[0] = i;
        SafeArrayGetElement(objindex.parray , lIndexV, &auxObj);
        m_jColObj[nlzof+i] = auxObj.IVal;
    }
}

```

```

////////////////////////////////////
//          Get Objective function type
////////////////////////////////////
    CapeLong otype;
    m_pMinlp->GetMINLPObjectiveFunctionType(&otype);

    m_objType = otype ;//(Max = -1; Min = 1)

////////////////////////////////////
//          Get coefficients of linear terms in objective function
////////////////////////////////////
    _bstr_t stype= TEXT("LINEAR");
    m_pMinlp->GetMINLPObjectiveFunctionDerivativeValues(stype,&v);

    m_lcoefObj = new double [nlzof];

    for (i=0; i<nlzof;i++)
    {
        lIndexV[0] = i;
        SafeArrayGetElement(v.parray , lIndexV, &auxValues);
        m_lcoefObj[i] = auxValues.dblVal;
    }
////////////////////////////////////
//          Coefficients for the Constraints
////////////////////////////////////

    _bstr_t structype= TEXT("LINEAR");

    m_pMinlp->GetMINLPConstraintDerivativeValues(structype,
                                                cids,&vals);

    m_Jac_lin = new double [nlz];

    for (i=0; i<nlz;i++){
        lIndexV[0] = i;
        SafeArrayGetElement(vals.parray , lIndexV, &auxValues);
        m_Jac_lin[i] = auxValues.dblVal;
    }
}

CCapeTNLP::~~CCapeTNLP()
{
    // delete pointers for private vectors
    delete [] m_lcoefObj ;
    m_lcoefObj = NULL;

    delete [] m_jColObj;
    m_jColObj = NULL;

    delete [] m_iRow, m_jCol;
    m_iRow = NULL;
    m_jCol = NULL;

    delete [] m_Jac_lin;
    m_Jac_lin = NULL;
}

```

```

delete [] m_Jac_lin;
m_Jac_lin = NULL;

delete [] m_iRow_h, m_jCol_h;
m_iRow_h = NULL;
m_jCol_h = NULL;

}
#####
//

bool CCapeTNLP::get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                             Index& nnz_h_lag)
{
    n=m_n;
    m=m_m;
    nnz_h_lag = m_nnz_h_lag;
    nnz_jac_g = m_nnz_jac_g;

    return true;
}
#####

bool CCapeTNLP::get_bounds_info(Index n, Number* x_l, Number* x_u,
                                 Index m, Number* g_l, Number* g_u)
{
    //////////////////////////////////////
    //////////////////////////////////////Upper and lower bounds for Variables //////////////////////////////////////
    //////////////////////////////////////

    boundsError=false;
    m_pMinlp->GetMINLPVariableBounds(vids,&LB,&UB);

    for (int i=0; i<n;i++)
    {
        lIndexV[0] = i;
        SafeArrayGetElement(UB.parray , lIndexV, &auxVarUB);
        SafeArrayGetElement(LB.parray , lIndexV, &auxVarLB);
        SafeArrayGetElement(isinteger.parray,lIndexV,&auxVarType);

        if (auxVarLB.dblVal<=auxVarUB.dblVal)
        {
            x_l[i] = auxVarLB.dblVal ;
            x_u[i] = auxVarUB.dblVal ;
        }
    }

    //////////////////////////////////////
    //////////////////////////////////////Upper and lower bounds for Constraints////////////////////////////////////
    //////////////////////////////////////

    boundsError=false;

    m_pMinlp->GetMINLPConstraintBounds(cids,&LB,&UB);

    for ( i=0; i<m;i++)
    {
        lIndexC[0] = i;

```

```

        SafeArrayGetElement(UB.parray , lIndexC, &auxVarUB);
        SafeArrayGetElement(LB.parray , lIndexC, &auxVarLB);
        g_l[i] = auxVarLB.dblVal ;
        g_u[i] = auxVarUB.dblVal ;
    }

    return true;
}

#####

bool CCapeTNLP::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 others if
    // you wish.

    assert(init_x == true);
    assert(init_z == false);
    assert(init_lambda == false);

    m_pMinlp->GetMINLPVariableValues(vids,&values);

    for (int i=0; i<n;i++)
    {
        lIndexV[0] = i;
        SafeArrayGetElement(values.parray , lIndexV, &auxVar);
        x[i] = auxVar.dblVal ;
    }

    return true;
}

#####

bool CCapeTNLP::eval_f(Index n, const Number* x, bool new_x,
                       Number& obj_value)
{
    double *x_double;
    x_double = (double*) x;
    double obj_nl = 0;

    // Calculate linear part of objective function

    double obj_lin = 0;

    for(int i=0; i<nIzof; i++)
    {
        obj_lin = obj_lin + x_double[m_jColObj[i]-1]*m_lcoefObj[i];
        //Index in MINLP starting from 1
    }

    // Send x into MINLP

```

```

        CCapeTNLP::SendVariablesOut(0, x_double);

// Then have MINLP calculate the objective function

        m_pMinlp->GetMINLPNonlinearObjectiveFunctionValue(&objval_nl);
        obj_nl = objval_nl;

// Assemble objective with linear and nonlinear parts

        obj_value = m_objType*(obj_nl + obj_lin);

    return true;
}

#####

bool CCapeTNLP::eval_grad_f(Index n, const Number* x, bool new_x,
                            Number* grad_f)
{
    double    *x_double;
    x_double = (double *) x;
    assert( x_double );

// reset grad_f
    for (int i=0; i<n; i++)
    {
        grad_f[i] = 0;
    }

// Linear part
    for(i=0; i<nlzof; i++)
    {
        grad_f[m_jColObj[i]-1] = m_lcoefObj[i] ;
        //Index in MINLP starting from 1
    }

// Send x into MINLP by CCapeTNLP::
    SendVariablesOut(0, x_double);

// NonLinear Part
    _bstr_t stype= TEXT("NONLINEAR");
    m_pMinlp->GetMINLPObjectiveFunctionDerivativeValues(stype,&v);

    for ( i=0; i<nnzof;i++)
    {
        lIndexV[0] = i;
        SafeArrayGetElement(v.parray , lIndexV, &auxValues);
        grad_f[m_jColObj[nlzof+i]-1] = auxValues.dblVal;
        //Index in MINLP starting from 1
    }
    return true;
}

#####

```

```

bool CCapeTNLP::eval_g(Index n, const Number* x, bool new_x, Index m,
                        Number* g)
{
    double *x_double;
    x_double = (double *) x;

    assert( x_double );
    double *g_lin = NULL;
    double *g_nl = NULL;

    g_lin = new double[m];
    g_nl = new double[m];
    for(int l=0;l<m;l++)
    {
        g_lin[l] = 0;
        g_nl[l] = 0;
    }
    // Send x into MINLP CCapeTNLP::
    SendVariablesOut(0, x_double);

    // Linear part
    for (int i=0;i<m;i++)
    {
        for (int j=0; j<nLz; j++)
        {
            if (m_iRow[j] == i+1) //Index in MINLP starting from 1
            {
                for(int k=0;k<n;k++)
                {
                    if (m_jCol[j] == k+1)
                    {
                        g_lin[i] = g_lin[i] + m_Jac_lin[j]*x_double[k];
                    }
                }
            }
        }
    }

    // NonLinear Part

    m_pMinlp->GetMINLPNonlinearConstraintValues(cids,&v);

    for ( i=0; i<m;i++)
    {
        lIndexC[0] = i;
        SafeArrayGetElement(v.parray , lIndexC, &auxValues);
        g_nl[i] = auxValues.dblVal;
        g[i] = g_nl[i] + g_lin[i];
    }

    delete [] g_lin, g_nl;
    g_lin = NULL;
    g_nl = NULL;

    return true;
}

```

```
#####
```

```
bool CCapeTNLP::eval_jac_g(Index n, const Number* x, bool new_x,  
                           Index m, Index nele_jac, Index* iRow, Index *jCol,  
                           Number* values)
```

```
{  
    double *x_double = NULL;  
    double *jacValues = NULL;  
  
    assert(nele_jac == m_nnz_jac_g);  
    if (values == NULL)  
    {  
        for(int i=0; i<nele_jac; i++)  
        {  
            iRow[i] = m_iRow[i];  
            jCol[i] = m_jCol[i];  
        }  
    }
```

```
// in IPOPT the indices start from one which is the same as in Cape-
```

```
// Open
```

```
}  
else
```

```
{  
    x_double = (double *) x;  
    assert( x_double );
```

```
// Linear
```

```
for(int i=0; i<nlnz; i++)  
{  
    values[i] = m_Jac_lin[i];  
}
```

```
// Send x into MINLP
```

```
CCapeTNLP::SendVariablesOut(0, x_double);
```

```
// NonLinear
```

```
_bstr_t structtype= TEXT("NONLINEAR");  
m_pMinlp->GetMINLPConstraintDerivativeValues(structtype,  
                                              cids,&vals);
```

```
for (i=0; i<nnz;i++)  
{  
    lIndexV[0] = i;  
    SafeArrayGetElement(vals.parray , lIndexV, &auxValues);  
    values[nlz+i] = auxValues.dblVal;  
}
```

```
return true;
```

```
}
```

```
#####
```

```
bool CCapeTNLP::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)
    {
        for (int i=0; i<nele_hess; i++)
            { // Both IPOPT and MINLP starting from 1
                iRow[i] = m_iRow_h[i];
                jCol[i] = m_jCol_h[i];
            }
    }
    else
    {
        double *x_double, *x_lambda;
        x_double = (double *) x;
        x_lambda = (double *) lambda;
        assert( x_double );
        assert(x_lambda );
        // Send x and lambda into MINLP
        CCapeTNLP::SendVariablesOut(0, x_double);
        CCapeTNLP::SendVariablesOut(1, x_lambda);
        // Send obj_factor
        VARIANT auxVarVal;
        VariantInit(&auxVarVal);
        auxVarVal.vt=VT_R8;
        auxVarVal.dblVal = obj_factor;
        m_pMinlp->SetMINLPHessianValues(auxVarVal);
        //used for set obj_factor to MINLP

        // Get Hessian from MINLP
        m_pMinlp->GetMINLPHessianValues(&hvalues);
        for (int i=0; i<nhess;i++)
        {
            lIndexH[0] = i;
            SafeArrayGetElement(hvalues.parray , lIndexH, &auxValues);
            values[i] = auxValues.dblVal;
        }
    }
    return true;
}
#####

void CCapeTNLP::finalize_solution(ApplicationReturnStatus 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)
{
    lpBlasDcopy(n, x, 1, x_sol_, 1);

    SendVariablesOut(0, x_sol_);
}

#####

int CCapeTNLP::SendVariablesOut(int type, double* x)
{
    const int dimensionV=1;
    long lIndexV[dimensionV];

```

```

int ns;

if (type == 0) // Send Variables
{
    ns = nv;
}
if (type == 1) // Send Lambda
{
    ns = nc ;
}

long safeBoundsV[2*dimensionV]={0,ns};

CSafeArray csaVids;
CSafeArray csaValues;
csaVids.Init(1,VT_VARIANT,safeBoundsV);
csaValues.Init(1,VT_VARIANT,safeBoundsV);

VARIANT auxVar,auxVarVal;
VariantInit(&auxVar);VariantInit(&auxVarVal);
auxVar.vt=VT_I4;
auxVarVal.vt=VT_R8;

for (int i=0; i<ns; i++){//ask for all the variables... nv
    lIndexV[0] = i;
    auxVar.lVal= i+1;//CapeOpen starts at one.
    csaVids.SetElement(lIndexV,auxVar);
    auxVarVal.dblVal = x[i] ;
    csaValues.SetElement(lIndexV,auxVarVal);
}

VariantInit(&vids);
vids.vt=VT_ARRAY | VT_VARIANT;
csaVids.GetSafeArray(& (vids.parray));
VariantInit(&values);
values.vt=VT_ARRAY | VT_VARIANT;
csaValues.GetSafeArray(&(values.parray));
if (type == 0) m_pMinlp->SetMINLPVariableValues(vids,values);
if (type == 1) {
    _bstr_t lmttype = "CONSTRAINT";
    m_pMinlp->SetMINLPLagrangeMultipliers(lmttype,vids, values);
}
return 0;
}

```