NLPAPI: An API to Nonlinear Programming Problems. **User's Guide**

Michael E. Henderson
IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598
`mhender@watson.ibm.com`

June 30, 2003

1 Introduction

This API provides a way to create and access Nonlinear Programming Problems of the form

minimize $O(\mathbf{x})$	Objective Function
subject to :	
$l_i \leq x_i \leq u_i$	Simple Bounds
$c_i(\mathbf{x}) = 0$	Equality Constraints
$L_i \leq c_i(\mathbf{x}) \leq U_i$	Inequality Constraints

The API began as an interface to the LANCELOT optimization code. LANCELOT is a Fortran program that solves nonlinear optimization problems using a trust region method. It has its own input description language called SIF, which is run through a program called the "SIF Decoder", which produces a set of Fortran subroutines and data files. These are compiled and linked against a supplied main program, which reads the data files and calls the subroutines in order to solve the problem, then writes the solution to a file. Parameters controlling the execution are read from another file.

We wanted to use LANCELOT on a problem whose constraints involved functions that were computed by an external code (a circuit simulator), and so could not be expressed in the SIF language. There is a hook to allow "external" functions in SIF, and this worked, but was awkward. I designed this API to replace the SIF decoder. It allows the user to build up a problem with a sequence of subroutine calls, to which the user may pass pointers to those external routines. The user may also set LANCELOT's parameters, and then invoke LANCELOT to solve the problem.

LANCELOT is documented in the book "LANCELOT: a Fortran Package for Large-Scale Nonlinear Optimization (Release A)", by A. R. Conn, N. I. M. Gould and Ph. L. Toint, *Springer Series in Computational Mathematics, Volume 17, Springer Verlag (Heidelberg, New York), ISBN 3-540-55470-X, 1992.* This document only covers the API.

The project has grown somewhat, into an API which can be used to define a nonlinear program that might be presented to any solver. In fact, we have so far only used LANCELOT and IPOPT, but a CUTE interface is in the works, and that will allow a number of solvers to be used.

The CUTE environment is documented in the paper "CUTE: Constrained and Unconstrained Testing Environment", by I. Bongartz, A. R. Conn, Nick Gould, and Ph. L. Toint, ACM Transactions on Mathematical Software, Vol. 21, No. 1, March 1995, Pages 123–160. A web page on a follow-on called "CUTEr" is available at `http://hsl.rl.ac.uk/cuter-www/`.

2 Overview

The API deals with minimization (or maximization) problems of the form

$$\begin{array}{c}\text{minimize } O(\mathbf{x})\text{subject to :}\\ l_i \leq x_i \leq u_i \qquad c_i(\mathbf{x}) = 0 \qquad L_i \leq c_i(\mathbf{x}) \leq U_i\end{array}$$

To define the problem the user creates an `NLProblem` that holds all of the information defining the problem. The problem is initially empty, and the user provides a function for the objective, and adds equality and inequality constraints. Each of these, the objective and the constraints involve scalar valued functions of the problem variables $\mathbf{x}$.

2.1 Functions

The NLPAPI provides three methods of defining these functions. They may be defined by an expression in a string (easy for small problems, but slow and cumbersome for large problems), by subroutines which evaluate the function (good for medium sized problems), and finally in LANCELOT's group partially separable form (designed for handling large problems, but needlessly complicated for small problems).

2.2 Defining functions by means of an expression in a string

The user can define a function by means of a string containing an expression. For example

```
"(x1-x2)**2+(x1+x2-10)**2/9+(x3-5)**2"
```

Since unknowns (in this case `x1`, `x2` and `x3`) may appear in any order, a second string is used which declares what the argument of the function is. That is,

```
F("[x1,x2,x3]") = "(x1-x2)**2+(x1+x2-10)**2/9+(x3-5)**2"
```

This list of variable names is a comma separated list of identifiers, delimited by square brackets. Finally, since typically not all problem variables will appear in the expression, there must be a mapping from the problem variables to the identifiers. This is done with an array of integers, `v` which lists the `nv` variables whose values are substituted for the variables `"[x1,x2,x3]"`. For example

```
nv=3;v[0]=0;v[1]=10;v[2]=9;
```

would indicate that to evaluate this function the idenitifier `x1` is assigned to the value of the first problem variable (note that we start counting at zero), `x2` the value of the x_{10}, and `x2` to x_9. The problem variables may appear in any order, and may even appear more than once.

To set the objective to this function would require code looking somthing like:

```
int v[3];
```

```
P=NLCreateProblem(...);

v[0]=0;v[1]=10;v[2]=9;
NLPSetObjectiveByString(P,"Obj",3,v,
    "[x1,x2,x3]","(x1-x2)**2+(x1+x2-10)**2/9+(x3-5)**2");
```

(The `NLPSetObjectiveByString` is described more completely below.)

2.3 Defining functions by means of subroutines

Functions may also be defined by providing subroutines which evaluate the functions and optionaly it's fist and second derivatives. (Some solvers provide differencing or updates for estimating derivatives.) The subroutines (e.g. `F`, `dF` and `ddF`) are called back when the objective is evaluated. The arguments the the subroutines are:

```
double F(int nv, double *x, void *data)
   int nv     The number of entries in x, as provided by
                     the user in the SetObjective call.
   double *x  An array with the values of the coordinates
                     of x.
   void *data The data pointer provided by the user in the
                     SetObjective call.

double dF(int i, int nv, double *x, void *data);

double ddF(int i, int j, int nv, double *x, void *data);
```

and all return a scalar result. `dF` evaluates the partial derivative $\partial F/\partial x_i$, and `ddF` evaluates the second partial derivatives $\partial^2 F/\partial x_i \partial x_j$.

The `data` variable allows the user to associate a data block with the objective, that is passed to the functions when they are evaluated. The `freedata` routine is called when the problem is free'd, so that the data block can be released. The array `v` lists the `nv` problem variables on which the objective (or constraint, since the same form is used for those) depends.This allows a simple form of sparsity).

The correspondance between the arguments of F and the problem variable is handled in the same way as for the expression. So to set the objective this way would require code:

```
v[0]=0;v[1]=10;v[2]=9;
rc=NLPSetObjective(P,name,3,v,F,dF,ddF,data,freedata);
```

2.4 Defining functions in group partially separable form

And the last option for defining a function is in group partially separable form. This represents the function as the sum of a number of *groups* –

$$f(\mathbf{x}) = \sum_{i=1}^{ng} \frac{1}{s_i} g_i(\sum_{j=1}^{ne_i} w_{ij} f_{ij}(R\mathbf{e}_{ij}) + < a_i, \mathbf{x} > - b_i)$$

The $g_i : \mathbb{R} \rightarrow \mathbb{R}$ are called group functions, s_i the group scale, and the argument of the group function is called the *group element*. The group element consists of three parts. The $f_{ij} : \mathbb{R}^{e_{ij}} \rightarrow \mathbb{R}$ are called nonlinear elements, $< a_i, \mathbf{x} >$ is the linear element, and b_i is called the constant element. Note that the constant element appears with a minus sign.

The objective function is created when the problem is created. Constraint functions are created by routines like NLPAddNonlinearEqualityConstraint. When the function is first created it consists of a single *trivial* group (i.e. the group function is the identity, the group scale is 1, there are no nonlinear elements, and the linear and constant elements are zero).

The group functions g_i and element functions f_{ij} may be defined by string or subroutine in the same way as an entire function. (The secret is that those routines define a trivial group with a single nonlinear element!) This is done with the `NLPCreateGroupFunction`, `NLPCreateGroupFunctionByString`, and `NLPCreateElementFunction`, and `NLPCreateElementFunctionByString` routines. A nonlinear element associates a *range transformation R* and selection of element variables e from the problem variables with an element function.

The basic procedure for defining a function this way is add the required number of groups, then to set the group scales and group functions, the linear and constant elements, and finally to add a set of nonloinear elements (though the order isn't important). The groups are referred to by the order in which they were added to the function.

The code for the example is a little more complicated using this method

```
P=NLCreateProblem(...);
g=NLCreateGroupFunction(P,"L2",gSq,dgSq,ddgSq,NULL,NULL);
f=NLCreateElementFunction(P,"fSq",1,NULL,fSq,dfSq,ddfSq,NULL,NULL);
```

The objective function $(x_1 - x_2)^2 + (x_1 + x_2 - 10)^2 + (x_3 - 5)^2$, consists of three groups, all with the same group function, but different linear parts.
$(x_1 - x_2)^2$:

```
group=NLPAddGroupToObjective(P,"OBJ1","L2");
rc=NLPSetObjectiveGroupFunction(P,group,g);
a=NLCreateVector(3);
rc=NLVSetC(a,0,1.);
rc=NLVSetC(a,10,-1.);
rc=NLVSetC(a,9,0.);
rc=NLPSetObjectiveGroupA(P,group,a);
NLFreeVector(a);
```

$(x_1 + x_2 - 10)^2$:

```
group=NLPAddGroupToObjective(P,"OBJ2","L2");
rc=NLPSetObjectiveGroupFunction(P,group,g);
a=NLCreateVector(3);
rc=NLVSetC(a,0,1.);
rc=NLVSetC(a,10,1.);
rc=NLVSetC(a,9,0.);
rc=NLPSetObjectiveGroupScale(P,group,9.);
rc=NLPSetObjectiveGroupA(P,group,a);
rc=NLPSetObjectiveGroupB(P,group,10.);
NLFreeVector(a);
```

$(x_3 - 5)^2$:

```
group=NLPAddGroupToObjective(P,"OBJ3","L2");
rc=NLPSetObjectiveGroupFunction(P,group,g);
a=NLCreateVector(3);
rc=NLVSetC(a,0,0.);
rc=NLVSetC(a,10,0.);
```

```
rc=NLVSetC(a,9,1.);
rc=NLPSetObjectiveGroupA(P,group,a);
rc=NLPSetObjectiveGroupB(P,group,5.);
NLFreeVector(a);
```

2.5 Solving a Problem

To invoke a solver the user creates a solver, e.g. an `NLLancelot` data structure. This has a set of parameters and a routines for invoking the solver. Default values are assigned to the parameters controlling the solver, and the user can set parameters as needed.

3 The API

The API consists of two main pieces: routines for defining the problem, and routines for setting up a solver and solving a problem.

3.1 Defining The Nonlinear Optimization Problem

To begin the user creates a problem –

```
#include <NLPAPI.h>

NLProblem P;
P=NLCreateProblem("MyProblem",1004);
```

The first argument is a name that is assigned to the problem. The second is the number of variables in the problem.

Each variable has a name, and may have simple bounds. The default for variable names is `"X%d"`. So in the example, the default is `"X1"`, ... `"X1004"`. The names can be changed, and simple bounds imposed with the routines:

```
int i;
double u,l;

rc=NLPSetVariableName(P,550,"V");
rc=NLPSetSimpleBounds(P,i,l,u);
```

```
rc=NLPSetLowerSimpleBound(P,i,l);
rc=NLPSetUpperSimpleBound(P,i,u);
```

3.1.1 The Objective

Each problem has an objective function. This can be set with the `NLPSet-Objective` or `NLPSetObjectiveByString` routine, or can be built as the sum of a number of groups.

```
NLProblem P;
char name[]="Obj";
int v[];
double (*F)(int,double*,void*);
double (*dF)(int,int,double*,void*);
double (*ddF)(int,int,int,double*,void*);
void *data;
void (*freedata)(void*);

v[0]=0;v[1]=45;v[2]=9;
rc=NLPSetObjective(P,name,3,v,F,dF,ddF,data,freedata);
```

The subroutines `F`, `dF` and `ddF` are called back when the objective is evaluated. The arguments to `F` are:

```
double F(int nv, double *x, void *data)
   int nv     The number of entries in x, as provided by
                     the user in the SetObjective call.
   double *x  An array with the values of the coordinates
                     of x.
   void *data The data pointer provided by the user in the
                     SetObjective call.
```

The `data` variable allows the user to associate a data block with the objective, that is passed to the functions when they are evaluated. The `freedata` routine is called when the problem is free'd, so that the data block can be released. The array `v` lists the `nv` problem variables on which the objective (or constraint, since the same form is used for those) depends.This allows a simple form of sparsity).

`dF` evaluates the partial derivatives of `f`, and has an additional integer argument (the first argument), which indicates which partial derivative to return. `ddF` evaluates the second partial derivatives, and has two additional integer arguments (the first two).

Alternatively, the user can define the objective by means of a string containing an expression:

```
NLPSetObjectiveByString(P,name,nv,v,
    "[x1,x2,x3]",
    "(x1-x2)**2+(x1+x2-10)**2/9+(x3-5)**2");
```

In this case the array `v` lists the `nv` variables whose values are substituted for the variables `"[x1,x2,x3]"`. So in the last string `x1` is the value of the first problem variable. `x1` is the value of the 46th problem variable, and so on.

Finally, the objective may be defined via LANCELOT's Group Partial Separable form. When the problem is first created the objective has a single group with trivial group function, no nonlinearelements, and linear and constant elements both zero. Additional groups ban be added using the `NLPAddGroupToObjective` routine. Each group has a group function g_i, a group scale s_i, a linear element, $\mathbf{a}_i.\mathbf{x}$, a constant element b_i, and a nonlinear element $N_i(\mathbf{x})$. (See the sections below on "Groups").

$$O(\mathbf{x}) = \sum \frac{1}{s_i} g_i(N_i(\mathbf{x}) + \mathbf{a}_i.\mathbf{x} - b_i)$$

The various pieces are added or set with the routines

```
NLPSetObjectiveGroupFunction
NLPSetObjectiveGroupScale
NLPSetObjectiveGroupA
NLPSetObjectiveGroupB
NLPAddNonlinearElementToObjectiveGroup
```

The nonlinear element is itself composed of a sum of element functions:

$$N(\mathbf{x}) = \sum w_i f_i(\sum R_{ij} e_j)$$

for a more complete description see the section below on nonlinear elements.

The main advantage of this form is that the derivatives of the objective (and other functions) can be expressed in terms of the derivatives of simpler functions. This is a convenience, but if the derivatives are approximated by differencing it can also substantially reduce the number of operations needed to approximate the derivatives of the objective.

The objective can be evaluated using the routines:

```
double o;
NLVector v,g;
NLMatrix H;

o=NLPEvaluateObjective(P,v);

g=NLCreate...Vector(...);
NLPEvaluateGradientOfObjective(P,v,g);

H=NLCreate...Matrix(...);
NLPEvaluateHessianOfObjective(P,v,H);
```

The idea here is that the user creates either a sparse or dense vector, and passes it to the routine which computes the gradient, which fills in the appropriate values, or a matrix in one of several formats, and passes it to the routine which evaluates the Hessian. (See the sections below on vectors and matrices.)

3.1.2 Inequality Constraints

A problem may have a number of inequality constraints. They are handled almost exactly as the objective was handled, but have in addition upper and lower bounds. Inequality constraints are added with the `NLPAddInequality-Constraint` or `NLPAddInequalityConstraintByString` routine, or can be built as the sum of a number of groups.

```
NLProblem P;
char name[]="InEq0";
double l,u;
int nv;
int v[3];
double (*F)(int,double*,void*);
double (*dF)(int,int,double*,void*);
double (*ddF)(int,int,int,double*,void*);
void *data;
void (*freedata)(void*);

nv=3;v[0]=3;v[1]=10;v[2]=9;
l=1.;u=10.;
rc=NLPAddInequalityConstraint(P,name,l,u,nv,v,F,dF,ddF,
                                        data,freedata);
```

The `data` variable allows the user to associate a data block with the constraint, that is passed to the functions when they are evaluated. The `freedata` routine is called when the problem is free'd, so that the data block can be released. The array `v` lists the `nv` variables on which the function depends.

Alternatively, the user can define the constraint by means of a string containing an expression:

```
rc=NLPAddInequalityConstraintByString(P,name,l,u,nv,v,
    "[x1,x2,x3]","48-x1**2-x2**2-x3**2");
```

The array `v` lists the `nv` variables whose values are substituted for the variables `"[x1,x2,x3]"`.

When the constraint is first created it has a single, empty group. Additional groups can be added using the `NLPAddGroupToInequalityConstraint` routine. Each group has a group function, a scale, a linear element, a constant element, and a set of nonlinear elements. (See the section below on "Groups").

```
NLPAddNonlinearInequalityConstraint(P,name);

NLPAddLinearInequalityConstraint(P,name,a,b);

NLPSetInequalityConstraintBounds
NLPSetInequalityConstraintUpperBound
NLPSetInequalityConstraintLowerBound
```

```
 NLPSetInequalityConstraintGroupFunction
 NLPSetInequalityConstraintGroupScale
 NLPSetInequalityConstraintGroupA
 NLPSetInequalityConstraintGroupB
 NLPAddNonlinearElementToInequalityConstraintGroup
```

Inequality constraints can be evaluated using the routines:

```
int c;
double o;
NLVector v,g;
NLMatrix H;

o=NLPEvaluateInequalityConstraint(P,c,v);

g=NLCreate...Vector(...);
NLPEvaluateGradientOfInequalityConstraint(P,c,v,g);

H=NLCreate...Matrix(...);
NLPEvaluateHessianOfInequalityConstraint(P,c,v,H);
```

3.1.3 Equality Constraints

Equality constraints are handled exactly as the inequality constraints are handled, but without the upper and lower bounds. Equality constraints are added with the `NLPAddEqualityConstraint` or `NLPAddEqualityConstraint-ByString` routine, or can be built as the sum of a number of groups.

```
NLProblem P;
char name[]="Eq0";
double l,u;
int nv;
int *v;
double (*F)(int,double*,void*);
double (*dF)(int,int,double*,void*);
double (*ddF)(int,int,int,double*,void*);
```

```
void *data;
void (*freedata)(void*);

nv=3;v[0]=3;v[1]=10;v[2]=9;
l=1.;u=10.;
rc=NLPAddEqualityConstraint(P,name,nv,v,F,dF,ddF,
                                        data,freedata);
```

The `data` variable allows the user to associate a data block with the constraint, that is passed to the functions when they are evaluated. The `freedata` routine is called when the problem is free'd, so that the data block can be released. The array `v` lists the `nv` variables on which the function depends.

Alternatively, the user can define the constraint by means of a string containing an expression:

```
rc=NLPAddEqualityConstraintByString(P,name,nv,v,
    "[x1,x2,x3]",
    "48-x1**2-x2**2-x3**2");
```

Again, the array `v` lists the `nv` variables whose values are substituted for the variables `"[x1,x2,x3]"`.

When the constraint is first created it has a single, empty group. Additional groups are added using the `NLPAddGroupToEqualityConstraint` routine. Each group has a group function, a scale, a linear element, a constant element, and a set of nonlinear elements. (See the section below on "Groups").

```
NLPAddNonlinearEqualityConstraint(P,name);

NLPAddLinearEqualityConstraint(P,name,a,b);

NLPSetEqualityConstraintGroupFunction
NLPSetEqualityConstraintGroupScale
NLPSetEqualityConstraintGroupA
NLPSetEqualityConstraintGroupB
NLPAddNonlinearElementToEqualityConstraintGroup
```

Equality constraints can be evaluated using the routines:

```
int c;
double o;
NLVector v,g;
NLMatrix H;

o=NLPEvaluateEqualityConstraint(P,c,v);

g=NLCreate...Vector(...);
NLPEvaluateGradientOfEqualityConstraint(P,c,v,g);

H=NLCreate...Matrix(...);
NLPEvaluateHessianOfEqualityConstraint(P,c,v,H);
```

3.1.4 Transformations of the Problem

Several common operations on problems are provided. They are not necessary, but may be of help. A sophisticated user may of course write their own.

The first operation simply creates a copy of the problem:

```
NLProblem Q;

Q=NLCopyProblem(P);
```

The other transformations below change the problem, so a copy can be useful to compare results.

Next there is a transformation which looks for simple bounds where the upper and lower bounds are identical, and adds a linear equality constraint which requires that the variable take that value, and removes the simple bounds.

```
NLEliminateFixedVariables(P);
```

This is useful for interior point techniques, which replace simple bounds by log barriers, and have problems dealing with identical bounds.

Inequalities are sometimes dealt with by introducing extra variables called slacks. That is,

$$l \leq f(\mathbf{x}) \leq u$$

is replaced by an equality constraint and simple bounds on the slack –

$$\begin{array}{l} f(\mathbf{x}) - s = 0 \\ 0 \leq s \leq u - l \end{array}$$

These operations take a problem with inequality and equality constraints and convert it to a problem with only equality constraints –

```
    NLPConvertToEqualityAndBoundsOnly(P);
```

Finally, equalities are sometimes eliminated by introducing a quadratic penalty and Lagrange multipliers. That is,

$$\begin{array}{lr} \text{minimize} & O(\mathbf{x}) \\ \text{subjectto} & f(\mathbf{x}) = 0 \end{array}$$

becomes

$$\text{minimize} \quad O(\mathbf{x}) + \frac{1}{2\mu} f^2(\mathbf{x}) - \lambda f(\mathbf{x})$$

In the limit of small penalty parameter μ, and minimizing w.r.t. both $\mathbf{x}$ and λ, the solution of this problem is the same as the solution of the original problem.

This is a little complicated, because of the group structure. In fact, the terms added to the objective are

$$\text{minimize} \quad O(\mathbf{x}) + \frac{1}{2\mu} \sum_i \left(f_i(\mathbf{x}) - \mu\lambda_i\right)^2$$

Notice that this perserves the group structure (see the floowing sections) *if* the constraint has only one group and the trivial group function. This is required to use this transformation. If this is true the groups added to the objective get a group function $g(x) = x^2$, inherit the nonlinear and linear elements of the constraint. The constant element of the group is increased by $\mu\lambda_i$, and the group scale is squared and then multiplied by 2μ. This means that when the penalty parameter μ or the Lagrange multipliers (λ_i's) are changed we must know which groups in the objective are penalties, and or each, what the original constant element and group scale were. Therefore the routine which creates the terms in the objective requires arrays that it can store this information in –

```
int    g[nc];  /* the ids of the added groups */
double mu;
double l[nc];  /* Lagrange multipliers */
double b[nc];  /* Constant elements */
double s[nc];  /* Group scales */

nc=NLPGetNumberOfEqualityConstraints(P);

NLCreateAugmentedLagrangian(P, mu,l, g,b,s);

NLSetLambaAndMuInAugmentedLagrangian(P, nc, mu,l, g,b,s);
```

3.1.5 Groups

In LANCELOT, functions are made of a sum of *groups*. Each group is a scaled scalar function of a nonlinear function of the problem variables. A group is of the form

$$\sum \frac{1}{s_i} g_i(N_i(\mathbf{x}) + \mathbf{a}_i.\mathbf{x} - b_i)$$

When a group is created the group function g_i is the identity, and the group scale is 1. In addition, there are no nonlinear elements (the $N_i(\mathbf{x})$), and the linear element $\mathbf{a}_i.\mathbf{x}$ and the constant element b_i are zero.

The user creates a group when he adds a constraint, or when he adds a group to the objective or constraint. The groups are associated with a constraint or the objective, so the user sets, e.g. group 3 in the objective, or group 2 in equality constraint 10, and so on. The group scale, linear and constant elements are set with routines

```
int g;
int c;
NLGroupFunction gf;
double s;
NLVector a;
double b;
```

```
NLPSetObjectiveGroupFunction(g,gf);
NLPSetEqualityConstraintGroupFunction(c,g,gf);
NLPSetInequalityConstraintGroupFunction(c,g,gf);

NLPSetObjectiveGroupScale(g,s);
NLPSetEqualityConstraintGroupScale(c,g,s);
NLPSetInequalityConstraintGroupScale(c,g,s);

NLPSetObjectiveGroupA(g,a);
NLPSetEqualityConstraintGroupA(c,g,a);
NLPSetInequalityConstraintGroupA(c,g,a);

NLPSetObjectiveGroupB(g,b);
NLPSetEqualityConstraintGroupB(c,g,b);
NLPSetInequalityConstraintGroupB(c,g,b);
```

The NLVector data structure is described below (dense or spares vectors). The NLGroupFunction data structure represents a scalar function (with first and second derivatives). It can be created by passing routines, or by way of a string.

```
NLGroupFunction g;
NLProblem P;
double (*G)(double,void*);
double (*dG)(double,void*);
double (*ddG)(double,void*);
void *data;
void (*freedata)(void*);

gf=NLCreateGroupFunction(P,"type",G,dG,ddG,
                                  data,freedata);
```

The type is a string associated with the group. G,dG, and ddG are functions which evaluate the group function, and its first and second derivatives. If dG and/or ddG is NULL centered differencing is used. The data is a block of memory passed to the functions (so that the same G etc. can be used in different GroupFunctions), and freedata is a routine that is called when the GroupFunction is freed.

A second method of creating a group function is "ByString". For example:

```
g=NLCreateGroupFunctionByString(P,"type",
                                          "s","sin(s)*cos(2*s)");
```

Each "CreateGroupFunction" should be matched with a "NLFreeGroupFunction" later on in the users code. Reference counting is used, so a group function that is passed to Set...GroupFunction can be safely "Free'd" immediately afterward.

3.1.6 Nonlinear Elements

Nonlinear elements are scalar valued functions of a subset of the problem variables. A group has a list of nonlinear elements whose values are summed, then added to the value of the linear and constant elements to give the argument to the group function. Each nonlinear element is of the form:

$$N(\mathbf{x}) = \sum_i w_i f_i(\sum_j R_{ij} e_j)$$

The element weight w_i is a scalar (default is $w_i = 1$). The element function f_i is a scalar valued function of a set of *internal variables*, which are a linear combination of the element variables e_j (a subset of the problem variables). The range transformation R_{ij} relates element variables to internal variables.

An element function is created with one of the routines:

```
NLElementFunction ef;
NLMatrix R;
double (*F)(int,double*,void*);
double (*dF)(int,int,double*,void*);
double (*ddF)(int,int,int,double*,void*);
void *data;
void (*freedata)(void*);
NLMatrix ddF0;

ef=NLCreateElementFunction(P,"etype",n,R,F,dF,ddF,
                                       data,freedata);
```

```
ef=NLCreateElementFunctionWithInitialHessian(P,"etype",
                                       n,R,F,dF,ddF,
                                       data,freedata,
                                       ddF0);

ef=NLCreateElementFunctionByString(P,"etype",n,R,
                                       "[x,y,z,w]",
                                       "x**2+y**2-z*w");
```

Here, `n` is the number of element variables, `R` the range transformation (or NULL), F, dF, and ddF are routines which evalute F and its derivatives (ddF may be NULL). If ddF is NULL, ddF0 gives an initial guess at the Hessian which is then updated using rank one updates. Note that the updates are done on the derivatives w.r.t. the internal variables, and that the derivatives are derivatives w.r.t the internal variables.

NLMatrices, which are used to represent the range transformation and the initial Hessian are described below.

A nonlinear element is created with the routine:

```
NLNonlinearElement N;
NLElementFunction ef;
int *vars;

N=NLCreateNonlinearElement(P,"type",ef,vars);
```

the `vars` array gives a list of the problem variables (by number, starting with 0!) which become the element variables.

A nonlinear element can be added to a group using the appropriate routine:

```
int c;
int g;
double w;
NLNonlinearElement N;

NLPAddNonlinearElementToObjectiveGroup(P,g,w,N):
```

```
NLPAddNonlinearElementToEqualityConstraintGroup(P,c,g,w,N):
NLPAddNonlinearElementToInequalityConstraintGroup(P,c,g,w,N):
```

where of course, w is the element weight.

Each "NLCreateElementFunction..." and "NLCreateNonlinearElement..." should be paired with a "NLFreeElementFunction" and "NLFreeNonlinearElement" (see Memory Management below).

3.1.7 Vectors

The NLVector is a data structure for ... vectors! There are two "types" of vector currently supported: sparse and dense. Sparse vectors are stored as a list of non-zero coordinates, dense vectors as a contiguous array of coordinates. They are created using the routines:

```
NLVector NLCreateVector(int n);
NLVector NLCreateVectorWithSparseData(int n,int nz,
                                      int *el,double *vl);

NLVector NLCreateDenseVector(int n);
NLVector NLCreateVectorWithFullData(int n,double *vl);
NLVector NLCreateDenseWrappedVector(int n,double *data);
```

The first two create sparse vectors (n is the dimension of the vector, nz the number of nonzeroes, and the coordinate el[i] is given by vl[i]). The third routine creates a vector whose coordinates are all zero. The fourth routine creates a dense vector, and the coordinates in vl are copied into a new array. The dense wrapped vector stores a pointer to the data array. This allows the user to change the vector by changing the data array.

Access to the vector is provided through routines like:

```
c=NLVGetC(v,i);
NLVSetC(v,i,c);
```

these have high overhead, so I'd recommend instead that you use a wrapped dense vector, or do your manipulations before creating the vector. Internal routines access the data directly when they can, so avoid the overhead.

3.1.8 Matrices

NLMatrices are similar to the NLVectors. There are dense matrices, and two kinds of sparse matrices currently supported,

```
int n,m;
double *data;

NLMatrix NLCreateMatrix(n,m);
NLMatrix NLCreateMatrixWithData(n,m,data);
NLMatrix NLCreateDenseWrappedMatrix(n,m,data);

NLMatrix NLCreateSparseMatrix(n,m);
NLMatrix NLCreateWSMPSparseMatrix(n);
```

The first three constructors are for dense matrices (i.e. range transformations). The first creates and $n \times m$ matrix with zero elements. The send copies the array data into the matrix, and the third uses a pointer to the data array (so that changing data changes the elements of the array). The dense matrices are stored by column, à la FORTRAN, so that element i, j is located in entry $data[i + n * j]$.

The first sparse format (`NLCreateSparseMatrix`) stores a list of elements, together with the associated row and column. The second sparse format stores the matrix as sparse rows. Access to the vector is provided through routines like:

```
NLMatrix A;
Aij=NLMGetElement(A,i,j);
NLMSetElement(A,i,j,Aij);
```

for sparse formats setting an element creates a nonzero element (if Aij is nonzero).

3.1.9 Memory Management

When the problem is no longer needed

```
NLFreeProblem(P);
```

releases the storage. It calls `NLFree..` for all of the groups, element functions, and so on which are stored in the problem. When the user creates one of these data structures a ”reference count” associated with it is set to ”1”. When the problem stores a pointer to the data structure the reference count is increased by one. The ”NLFree...” routines decreases the reference count by one and if the count is zero, releases the memory used by the data structure. For example:

```
g=NLCreateGroupFunction(...);      ref count = 1
NLPSetObjectiveGroupFunction(...); ref count = 2
NLFreeGroupFunction(...);          ref count = 1 not yet

NLFreeProblem(...);                ref count = 0 DELETE g!
```

This ensures that memory is released, but not until everyone who is using it is done with it. It relies on an “honor code”. If you decided to free the group twice in the code segment above you could get some nice side effects.

3.1.10 Error Handling

Most routines return a return code that indicates whether the operation was successful. If the routine creates or returns a data structure an invalid value is returned if the routine is not successful. In addition a simple error handling is also provided.

```
int NLGetNErrors();
```

returns the total number of errors that have occured, and

```
void NLClearErrors();
```

resets the count. Individual errors can be examined with the routines

```
int NLGetErrorSev(int n);
char *NLGetErrorRoutine(int n);
char *NLGetErrorMsg(int n);
int NLGetErrorLine(int n);
char *NLGetErrorFile(int n);
```

The severity is 4, 8 or 12, the Routine is the routine which issued the error, and the line and file give the line of source code where it was issued. The message usually gives information about what caused the error (usually an invalid argument to the routine).

4 Solving a Nonlinear Optimization Problem Solver with LANCELOT

The LANCELOT Nonlinear Optimization Problem Solver consists of a parameter list. The problem solver is created by a subroutine call, and has a set of default parameters, which can be modified. The user invokes LANCELOT by passing a problem (see above) and a starting guess. The same problem solver may be used on different problems, and several problem solver may be created.

```
NLLancelot L=NLCreateLancelot();
```

Various parameters, like how much to print to the screen, are given defaults that can be queried or set via additional subroutine calls. For example

```
void LNSetPrintLevel(Lancelot,int);
```

gives the value for the `PRINT-LEVEL` line in the `SPEC.SPC` file.

When the "LNMinimize" or "LNMaximize" routine is called, a `SPEC.SPC` file containing the current parameters is dumped, as well as an `OUTSDIF.d` file containing information about the problem. A global variable is set to point to the problem being solved, and the Lancelot main program is invoked. This calls back to `ELFUNS`, `GROUPS`, etc., which refer to the problem referenced by the global pointer to provide information to Lancelot. When Lancelot terminates the `SOLUTION.d` file is read to get the solution, which is sent back to the user.

```
LNMinimize(L,P,v0,v);
```

5 Example

We will develop the code for creating and solving HS65. HS65 is the problem:

$$\begin{array}{l} \text{minimize} \\ \quad (x_1 - x_2)^2 + (x_1 + x_2 - 10)^2/9 + (x_3 - 5)^2 \\ \text{subject to} \\ \quad -4.5 \leq x_1 \leq 4.5 \\ \quad -4.5 \leq x_2 \leq 4.5 \\ \quad -5. \leq x_3 \leq 5. \\ \quad 48 - x_1^2 - x_2^2 - x_3^2 \geq 0 \end{array}$$

First we do this using the SetObjective/AddConstraint routines, then in group partially separable form.

5.1 Using the SetObjective/AddConstraint routines

This should be fairly clear. First we include the NLPAPI header file and declare some variables –

```
#include <NLPAPI.h>

int main(int argc, char *argv[])
 {
  NLProblem P;
  int v[3];
```

Then we create the problem, giving it the name "HS65" –

```
  P=NLCreateProblem("HS65",3);
```

and change the names of the varables (although these are the default names anyway) and set the bounds on the variables.

```
  NLPSetVariableName(P,0,"X1");
  NLPSetSimpleBounds(P,0,-4.5,4.5);

  NLPSetVariableName(P,1,"X2");
```

```
    NLPSetSimpleBounds(P,1,-4.5,4.5);

    NLPSetVariableName(P,2,"X3");
    NLPSetSimpleBounds(P,2,-5.,5.);
```

Next we specify the objective function (which in this case depends on all three problem variables)

```
    v[0]=0;v[1]=1;v[2]=2;
    NLPSetObjectiveByString(P,"Obj",3,v,
        "[x1,x2,x3]","(x1-x2)**2+(x1+x2-10)**2/9+(x3-5)**2");
```

and add an inequality constraint

```
    v[0]=0;v[1]=1;v[2]=2;
    NLPAddInequalityConstraintByString(P,"I1",0.,1.e40,3,v,
        "[x1,x2,x3]","48-x1**2-x2**2-x3**2");
```

And that' all there is to it.

If instead we had subroutines to evaluate the objective (`o` and `do` and `ddo` to evaluate the first and second derivatives) and constraint (`c`, `dc` and `ddc`) the code would change slightly:

```
    v[0]=0;v[1]=1;v[2]=2;
    NLPSetObjectiveByString(P,"Obj",3,v,o,do,ddo,NULL,NULL):
    NLPAddInequalityConstraintByString(P,"I1",0.,1.e40,3,v,
        c,dc,ddc,NULL,NULL);
```

5.2 Using the AddGroup routines

This approach uses the same definition as the SIF file for HS65 would. The objective consists of three groups with the same group function, none has any nonlinear elements, and the first has no constant part to the linear element.

So we will define one `LNGroupFunction`, passing it routines which square the argument and evaluate the derivatives. We need one `LNElementFunction`, for the single non-linear element in the constraint. This evaluates the same function as the group function, but element functions, unlike groups, which take a scalar argument, take a vector as argument.

First we include the API prototypes:

```
#include <NLPAPI.h>
```

Then define two sets of three functions, which will be used for the group and element functions and their derivatives.

```
double gSq(double x){return(x*x);}
double dgSq(double x){return(2*x);}
double ddgSq(double x){return(2);}

double fSq(int n,double *x){return(x[0]*x[0]);}
double dfSq(int i,int n,double *x){return(2*x[0]);}
double ddfSq(int i,int j,int n,double *x){return(2);}
```

The main program and declarations –

```
int main(int argc,char *argv[])
 {
  NLProblem P;
  NLGroupFunction g;
  NLElementFunction f;
  NLNonlinearElement ne;
  int group;
  NLVector a;
  double x0[3];
  NLLancelot Lan;
  double x[3];
  int constraint;
  int element;
  int v[1];
  int i;
  int rc;
```

We are now ready to create the problem and a group and element function

```
  P=NLCreateProblem("HS65",3);
  g=NLCreateGroupFunction(P,"L2",gSq,dgSq,ddgSq,NULL,NULL);
  f=NLCreateElementFunction(P,"fSq",1,NULL,fSq,dfSq,ddfSq,NULL,NULL);
```

Note that for `NLCreateElementFunction` the number of internal variables is passed (i.e. the actual number of unknowns used by the function), as well as a "range transformation", which by default is the identity. The last two arguments allow data to be passed to the element function. The second argument is an "element function type", and should be unique to the element function.

The objective function $(x_1 - x_2)^2 + (x_1 + x_2 - 10)^2 + (x_3 - 5)^2$, consists of three groups, all with the same group function, but different linear parts. $(x_1 - x_2)^2$:

```
  group=NLPAddGroupToObjective(P,"OBJ1","L2");
  rc=NLPSetObjectiveGroupFunction(P,group,g);
  a=NLCreateVector(3);
  rc=NLVSetC(a,0,1.);
  rc=NLVSetC(a,1,-1.);
  rc=NLVSetC(a,2,0.);
  rc=NLPSetObjectiveGroupA(P,group,a);
  NLFreeVector(a);
```

$(x_1 + x_2 - 10)^2$:

```
  group=NLPAddGroupToObjective(P,"OBJ2","L2");
  rc=NLPSetObjectiveGroupFunction(P,group,g);
  a=NLCreateVector(3);
  rc=NLVSetC(a,0,1.);
  rc=NLVSetC(a,1,1.);
  rc=NLVSetC(a,2,0.);
  rc=NLPSetObjectiveGroupScale(P,group,9.);
  rc=NLPSetObjectiveGroupA(P,group,a);
  rc=NLPSetObjectiveGroupB(P,group,10.);
  NLFreeVector(a);
```

$(x_3 - 5)^2$:

```
  group=NLPAddGroupToObjective(P,"OBJ3","L2");
  rc=NLPSetObjectiveGroupFunction(P,group,g);
  a=NLCreateVector(3);
  rc=NLVSetC(a,0,0.);
  rc=NLVSetC(a,1,0.);
  rc=NLVSetC(a,2,1.);
```

```
rc=NLPSetObjectiveGroupA(P,group,a);
rc=NLPSetObjectiveGroupB(P,group,5.);
NLFreeVector(a);
```

Next come bounds on the variables:

```
rc=NLPSetSimpleBounds(P,0,-4.5,4.5);
rc=NLPSetSimpleBounds(P,1,-4.5,4.5);
rc=NLPSetSimpleBounds(P,2,-5.,5.);
```

And finally the single nonlinear inequality constraint, which is the trivial group with three nonlinear elements. The default bounds on the constraint are 0 on the left, and ∞ on the right, so we need not change the bounds.

```
constraint=NLPAddNonlinearInequalityConstraint(P,"C1");
rc=NLPSetInequalityConstraintB(P,constraint,-48.);
v[0]=0;
ne=NLCreateNonlinearElement(P,"Sq1",f,v);
element=NLPAddNonlinearElementToInequalityConstraint
                                (P,constraint,-1.,ne);
NLFreeNonlinearElement(P,ne);

v[0]=1;
ne=NLCreateNonlinearElement(P,"Sq2",f,v);
element=NLPAddNonlinearElementToInequalityConstraint
                                (P,constraint,-1.,ne);
NLFreeNonlinearElement(P,ne);

v[0]=2;
ne=NLCreateNonlinearElement(P,"Sq3",f,v);
element=NLPAddNonlinearElementToInequalityConstraint
                                (P,constraint,-1.,ne);
NLFreeNonlinearElement(P,ne);
```

5.3 Invoking LANCELOT to solve the problem

No matter which method was used to define the problem, the invocation of LANCELOT (Oh great and powerful LANCELOT, we pray that you find a solution ...) is the same. First we call the constructor for the NLLancelot object, then set the initial guess and ask for the minimization to be performed.

```
  Lan=NLCreateLancelot();

  x0[0]=-5.;
  x0[1]=5.;
  x0[2]=0.;
  rc=LNMinimize(Lan,P,x0,(double*)NULL,x);
```

The rest of the example simply prints the solution

```
  printf("Solution is (");
  for(i=0;i<3;i++)
   {
    if(i>0)printf(",");
    printf("%lf",x[i]);
   }
  printf(")\n");
```

and any errors that may have occured. I've embedded a couple, just to be tricky. We can test for an error with the return codes, or with the `LNGetError` function.

```
  printf("There were %d errors\n",NLGetNErrors());
  if(NLError())
   {
    for(i=0;i<NLGetNErrors();i++)
     {
      printf(" %d line %d, file %s, Sev: %d\n",i,
           NLGetErrorLine(i),NLGetErrorFile(i),NLGetErrorSev(i));
      printf("    Routine: \"%s\"\n",NLGetErrorRoutine(i));
      printf("    Msg: \"%s\"\n",NLGetErrorMsg(i));
     }
   }
```

And the final step, which really should be done, is to return all the memory used to the system.

```
  NLClearErrors();
```

```
 NLFreeGroupFunction(g);
 NLFreeElementFunction(f);
 NLFreeLancelot(Lan);
 NLFreeProblem(P);
 return(0);
}
```

The program produced the following output –

```
 objective function value =   9.53528856489003D-01

            X0000001          3.65046233137863D+00
            X0000002          3.65046219722889D+00
            X0000003          4.62041670283010D+00
            C1                0.00000000000000D+00
Solution is (3.650460,3.650460,4.620420)
There were 0 errors
```