

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 design the user to build up a problem with a sequence of subroutine cde 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.

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


```
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.

(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)**2A1(e, xc)1(ti 2A1(x2-1(ti 02)1()1*)12/*)19+()"1(31)1(5e))1(1*)
```

`r(m)11by`


```
int v[3];  
double (*F)(int, double*, void*);  
double (*dF)(int, int, double*, void*);  
double (*ddF)(int, int, int, double*, void*);  
void *data;
```

```
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...Vecton(...);
```

```
void *data;  
void (*freedata)(void*);  
  
nv=3; v[0]=3; v[1]=10; v[2]=9;  
l=1.; u=10.;  
rc=NLPAddEqualityConstraint(P, nam2, nv, v, F, dF, ddF,  
                             data, freedata);
```

The data

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 –

$$f(\mathbf{x}) - s$$


```
NLPSetObjectiveGroupFunction(g, gf);  
NLPSetEqualityConstraintGroup(n);  
NLPSetObjectiveGroup(n);
```

different GroupFunctions), and freedata is a routine that is 1, toled when the GroupFunction is freed.

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

```
ef=NLCreatElementFunctionByString(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 evaluate F and its derivatives (ddF may be NULL). If ddF is NULL, ddF0 gives an initial guess at the Hessian

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

3.1.8 Matrices

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

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

2 Example

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

minimize
(x


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

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);

```



```
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.);
```

object, then set the initial guess and ask for the minimization to be performed.

