Development of Software for Solving Systems of Nonlinear Equations

by

Karl Westerberg

EDRC 05-36-89
Carnegie Mellon University

DEVELOPMENT OF SOFTWARE FOR SOLVING SYSTEMS OF NONLINEAR EQUATIONS

Karl Westerberg

Engineering Design Research Center

Table of Contents

List of Figures

1. Introduction

This report discusses the development of a stand alone software package for solving large sparse systems of nonlinear equations. This work continues the devetopment effort toward a solving capability for the ASCEND mathematical modeling environment (Piela, 1989), and is based on a linear equation solver that we have previously developed (Westerberg, 1989). Readability, modularity, flexibility, and portability were the issues given top priority in the solver's design. Our chosen software/hardware platform was Pascal on an Apollo workstation. Although efficiency was also a concern, modularity took precedence; it was important for the software to easily incorporate multiple solving strategies.

This manuscript is divided into two distinct parts the first of which describes in general terms the development of the software described above. The second part located in appendix II is primarily intended for software developers and contains the Pascal include files for the software. Each include file contains a set of procedure declarations; and each declaration contains the parameters associated with the procedure, and a short description of what the procedure does.

2. Description of the software structure

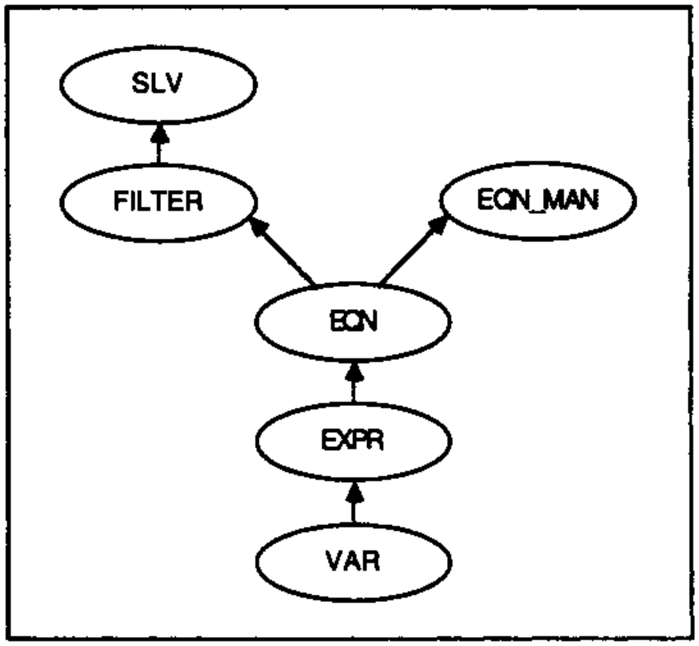

Figure 2-1: Module dependency tree

The module dependency tree is shown in figure 2-1. The modules which comprise the non-linear equation solver are as follows:

1. **var:** management of variables and variable tables.
2. **expr:** storage of general algebraic expressions.
3. **eqn:** management of equations and equation lists.
4. **eqnjnan:** higher level manipulation of equations, expressions and variables, e.g., evaluation of expressions and their derivatives.
5. **slv:** solution of a system of non-linear equations. The structure supports the general optimization problem, although the current implementation ignores the objective function and treats all constraints as equalities.
6. **filter:** contains some commonly used equation and variable filters.

The insert files can be found in the appendix II. The modules **llnsol, part, gauss,** and **mtx** are part of a linear equation solver which have been described in (Westerberg, 1989).

This solver differs from many others in that it incorporates an internal representation of an equation. Most other solvers require that the user write a routine which evaluates the equation (and possibly its derivatives), and then bind this routine with the solver which leaves the user with the problem of having to represent equations. By devising our own representation, we have relieved the user of the burden of writing complex code for performing operations on equations, such as evaluation of derivatives.

We have chosen to separate the representation and manipulation of equations into four modules: **var, expr, eqn,** and **eqn_man.** We felt that this separation constituted a set of abstractions which provided a high degree of flexibility in the design and re-use of this software. The **eqn** and **var** modules are responsible for managing collections of equations and variables, and were separated from the storage of expressions. The reason for making **eqnjnan** (equation manipulator) a separate module was that its functionality doesn't depend on the data representation.

We consider equations and expressions to be different. An equation is a statement which can be true or false; an expression represents a real number. An equation is composed of two expressions separated by a relational operator (such as * or <).

After some discussion we decided to export the structure of the variable record to the user. A complete data abstraction would have demanded that this record be completely hidden from the user and the fields be accessible only through procedure calls. However, we envisioned the user needing to add fields to the variable record. With an exported record definition this is straightforward; however, a hidden structure would require the user to write two additional procedures (one to fetch the value and another to set the value) in addition to adding the field to the record. Also, with a hidden structure, a reference to a variable would require a function call, which is considerably more expensive than dereferencing a pointer. The same arguments were applied to the equation record.

We anticipated that users might want to count the number, or obtain a list of variables, with a specified property (such as fixed). Rather than write a separate routine for each property, we allowed the user to pass in a function that takes a variable as an argument and returns true or false depending on whether the variable has the given property or not. We call such functions *filters,* and these filters can be written by the user. To assist in this task we wrote a **filter** module which contains some common filters (such as a filter accepting free/fixed variables). Again, the same arguments were applied to equations.

3. Implementation

There are three issues of implementation worthy of mention: storage of expressions, evaluation of expressions (and their derivatives), and the algorithm for solving systems of equations.

3.1. Evaluation

We evaluate expressions and all of the derivatives using a stack (Ponton, 1982). This involves scanning the expression in postfix for the tokens, (a token is either a constant, a variable reference, or an operator) and maintaining a function stack and possibly one or more derivative stacks. This method is numeric, and thus the derivatives need not be stored symbolically. However, the derivatives computed are exact. We extended the method to handle second derivatives, anticipating the need for them in the solution of optimization problems.

3.2. Storage

We had considered two different representations for equations: a threaded tree (Knuth, 1973) and an array (tokens stored in postfix order). The tree is a good for dealing with structural rearrangement, the array is better for evaluation. Our evaluation method required that the tokens be accessed in postfix order, and a tree representation (even threaded) would not provide them as quickly as an array representation.

Although the ability to perform structural rearrangement was important, most time would be spent in evaluation and so we adopted an array representation. However, in our representation each token includes pointers back to the root(s) of the subexpression(s). This allows us to do all operations associated with a tree representation efficiently and maintain an effective evaluation mechanism. We made these pointers relative, so that most of them would remain untouched, even if a large part of the expression was removed or replaced. Also, informal calculations suggested that the tree representation would require at least twice as much memory as the array representation (or more, since allocation of the tree would occur in small pieces, whereas the array would be allocated as one piece). Since we were designing a solver to handle large sparse systems, this difference could be on the order of a megabyte.

3.3. Solving

We implemented a variant of Newton's which searches along the Newton-gradient plane (Westerberg and Director, 1978). One can specify the maximum allowable step to be taken at a given iteration, and if the Newton step is too large, then a direction between the Newton and the gradient is efficiently selected so that the step is as close as desired to the maximum allowed. The solver adjusts the maximum step allowed after each iteration.

We found the method for adjusting the maximum step suggested in this algorithm to be unsuitable. Our experience was that whenever the solver rejected the Newton direction, the maximum step was reduced to virtually zero. However, the heuristic method for increasing the maximum step size was unable to increase it to reasonable values. In an attempt to correct this problem we replaced the complicated method with a simple one: we simply multiplied the maximum step by 1.5 if the previous step was accepted, and we divided it by 2 if the previous step was not accepted. In addition, after each successful step, we attempted to search in the Newton direction regardless of the value of the maximum step. This improved things somewhat, but with certain problems the maximum step got too small and the Newton predictions in that area were unacceptable. Furthermore, the step control was still based on arbitrary scaling factors such as 1.5 and 2 which have no theoretical basis.

We then investigated the Armijo line search (Armijo, 1966) as a method for step control. The Armijo line search begins by attempting a full Newton step. If the objective function does not decrease (by enough), the step is rejected and is reduced by a factor based on how much the objective function would have increased had the step been accepted. Otherwise, the step is accepted and the sequence starts over. We carried this idea over to the Newton-gradient method. We first tried a full Newton (by setting the maximum step to infinity). If the objective function reduced, the step was accepted and the maximum step was reset to infinity. Otherwise, the step was rejected and the maximum step was reduced using the

Armijo line search. We feel that this hybrid algorithm is better than either of the two algorithms it is based on.

In order to maintain numerical stability during computation, to provide a reasonable objective function, and to provide reasonable step length control, we needed to scale equations and variables. We scale variables using nominal values provided by the user (the nominal value is a field of the variable record), by multiplying each column in the jacobian by the nominal value of the corresponding variable. Having scaled the variables, the equations are scaled by normalizing them. The weights remain constant for a specified number of iterations and are then updated. We use the weighted sum of squares as our objective function. Step length control is based on the scaled step, so that variables with larger nominal values can take larger actual steps. It is important that the solver only use scaled values for making internal decisions, such as what step to try and whether to accept it. External decisions such as the convergence test should be decided independently of the weights. We therefore use the unweighted residuals to test convergence.

The solver offers the option of partitioning the problem into irreducible blocks and solving each irreducible block separately. This should increase the speed and reliability of solving problems which have non-trivial partitions. However, it is possible (although less likely than the reverse) that a problem cannot be solved when partitioned but can be solved as a single block. This could occur when the solution of all predecessors to a block and its initial values result in a numerically singular starting point for that block. The initial singularity could be avoided by solving the entire system. (Here is an example: try x=0 and cos(y)+x*sin(y)=0, starting at (x,y)=(1,0).) To avoid these pathological cases the user has the option of turning partitioning off.

4. Conclusions

We have developed a structured solver which should provide a good basis for further research. The solver has been tested on a large number of examples, including a set of standard problems given in (Ferraris and Tronconi, 1986). Its performance compares favorably with the results presented in that reference, and should improve as work progresses.

I. Solving example

In this section, we illustrate the use of the solving software on the following problem proposed by Powell (Powell, 1970).

```
10,000 vl v2        m i.o
exp(-vl) + exp(-v2)  * 1.0001
```

Starting point: vl = 0, v2 = 1

Firstly we create a variable table with two entries, and an equation list with two entries.

```
VAR
   vl,v2      : var$_t;
   var_table  : var$_table_t;
   el,e2      : eqn$__t;
   eqn__list  : eqn$_list_t;

var$__create__table (variable) ;
var$_create_var (var_table , 1 , {=>> vl) ;
var$_create_yar (variable , 2 , {=>} v2);
eqn$_create_list (eqn__list);
eqn$_create_eqn (eqn_list , 1 , {=>} el) ;
eqn$_create__eqn (eqn_list , 2 , {=>} «2);
```

The variable and equation entries are created with default values for most of their fields. To specify the structure of the problem completely, we need only modify the value field of both variables; and the two expression fields and the relation field of both equations. Variable values (initial) are specified as follows:

```
vl^.value := 0.0;
v2^.value := 1.0;
```

At present, specifying the structure of equations is somewhat tedious; however, an equation parser (to be written) could make this process much easier. The following illustrates how equations would be parsed.

```
VAR
   eqn_str : str$_atring_t;

eqn_str :« str$_form('10000 * «1 * #2 * 1.0') ;
parse(eqnjstr , el) ;
str$_destroy(eqn_str);
eqn_str :» str$_£orm('exp(-#l) + exp(-#2) = 1.0001');
parse(eqn_str , e2);
3tr$_jde3troy(eqn_str);
```

At present, the user can build equations in one of two ways: either hierarchically, or in postfix. We shall demonstrate both ways by constructing equation 1 in postfix, and equation 2 hierarchically.

Construction of equation 1:

```
VAR
   token  : expr$__token__t;
   tag    : expr$_tag_t;
   ok     : boolean;

{ Equation 1 in postfix: LHS: 10000 vl v2 v2 * *  RHS: 1.0 >
expr$_create_expr_injpostfix(tag);

token.op := expr$_const;
token.value := 10000;
expr$__append__token (tag, token);

token.op := expr$_var;
token.v := vl;
expr$__append_token(tag,token);

token.op := expr$__var;
token.v  := v2;
expr$__append_token(tag,token);
expr$_append__token(tag,token);

token.op := expr$_mul;
expr$_append_token(tag,token);
expr$_append_token(tag,token);

expr$__return__expr (tag, el^. LHS, ok);

expr$_create_exprMinjpostfix(tag);

token.op := expr$__const;
token.value := 1.0;
expr$__append_token(tag,token);

( If the above code is correct, then ok should certainly return true }
expr$_return__expr(tag,el^.RHS,ok);
```

Construction of equation 2:

```
VAR
   token        : expr$_tokenjt;
   exl,ex2,ex3  : expr$__t;       { Temporary expressions }

token.op := expr$_var;
token.v :« vl;
expr$_coznbine(token,exl);        { exl « vl }

token.op := expr$__neg;
expr$_combine(token,ex2,exl);   { ex2 = -vl, exl destroyed }

token.op := ejqpr$_exp;
expr$_combine(token,exl,ex2);   { exl = exp(-vl), ex2 destroyed }
```

```
    token.op :» expr$_var;
    token.v :» v2;
    expr$__conb±ne(token,ex2);       { ex2 » v2 }

    token.op :* expr$_neg;
    expr$__combine(token,ex3,ex2);   { ex3 • -v2, ex2 destroyed }

    token.op := expr$__exp;
    expr$__combine(token,ex2,ex3);   { ex2 • exp(-v2), ex3 destroyed }

    token.op := expr$_add;
    expr$_combine(token,e2^.LHS,exl,ex2);   { e2^.LHS * exp(-vl) + exp(-v2) }

    token.op := expr$__const;
    token.value := 1.0001;
    expr$_combine(token,e2^.RHS);   { e2^.RHS = 1.0001 }
```

As the above example suggests, coding the equations directly into a program may require a lot of statements. However, if the user is willing to read the equations from a file (or from the keyboard), a procedure can be written which requires that the user only enter relevant structural information. For example:

```
PROCEDURE input__expr ( OUT expr : expr$__t );
   VAR
      tag   : expr$_tag_t;
      token : expr$_token__t;
      vnum  : integer;
      ok    : boolean;

   BEGIN { input_expr }
      writeln('Enter in the expression in postfix.');
      writeln('Terminate using expr$_end.');

      WHILE true DO BEGIN
         write('Op -> ' ) ;
         readln(token.op);
         IF token.op » expr$_end THEN exit;
         IF token.op = expr$_const THEN BEGIN
            write('Constant ~> ');
            readln(token.value);
         END; { IF >
         IF token.op = expr$__var THEN BEGIN
            write('Variable number —> ' ) ;
            readln(vnum);
            token,v := var$_get_yar(var_table , vnum) ;
         END; { IF }
         expr$__append_token(tag,token) ;
      END; { WHILE }
      expr$_return__expr(tag,expr,ok) ;
   END;  { input_expr )
```

In order to solve the problem we must create a "solve system[1]* which references the variable table and equation list that we have created.

```
VAR
   system : slv$_system_t;

slv$_create(system);
slv$_set_eqn_list(system , eqn_list);
slv$_set_var_table(system , var_table);
```

The solve system is created with a set of default values for attributes, such as maximum number of iterations, and CPU time limit. If these values are acceptable, the problem is ready to be solved. If the default values are unacceptable, they can be easily changed. The following shows how to specify the CPU time limit.

```
VAR
   status : slv$_status_t;

{ Always get the status before modifying it, especially if you only want to
  modify some of the parameters }

slv$_get_status(system , status);
status.time_limit := 30.0;   { 30.0 CPU second time limit }
slv$_change_status(system , status);
```

The system is now ready to be solved. A call to the slv$_presolve procedure prepares the system of equations for solution, and a call to slv$_solve will attempt to solve the system.

```
slv$_presolve(system);
slv$_solve(system);
```

On completion of slv$_solve, the "solve status" should be checked to verify that the system of equations actually converged. This is done by making a call to slv$_change_status. Variable values at the solution can be obtained by examining v1\value and v2^.value.

II. Insert files

```
{ BEGIN var.lns.pas_____________________________________________________________>

{-------------------------------------------------------------------------------
| Prevent this file from being included more than once.
+------------------------------------------------------------------------------}
%IFDBF var$_already_inserted %THEN
   %BXIT           "
%BLSE
   %VAR  var$_already__inserted
%ENDIF

%INCLUDB  '/ascend/utilities/ins/str.ins.pas';
%INCLUDE '/ascend/utilities/ins/list.ins.pas' ;

{-------------------------------------------------------------------------------
| All variable numbers must be greater than zero.  It is recommended, but not
| required that all variable numbers also be no greater than mtx$_max__order_c.
| Higher level routines that work with a matrix will require variable numbers
| to be in this range.  VARIABLE NUMBERS SHOULD ONLY BE READ, NEVER CHANGED?
+------------------------------------------------------------------------------}

TYPE
{-------------------------------------------------------------------------------
| Warning:  If a field is added, then the assignment of its default value must
| also be added to internal routine var_create_var.  In addition, EVERY module
| above this one must be recompiled, even modules which don't use the new field.
+------------------------------------------------------------------------------}
   var$_spec_t     = RECORD
      number       : integer;         { Variable number (for reference in table) }
      vlabel       : str$_string_t;   { Variable label (NIL: no label) >
      value        : double;          { Current value of the variable >
      nominal      : double;          { Nominal value of the variable }
      lower_bound  : double;          { Lower bound of value }
      upper_bound  : double;          { Upper bound of value }
      fixed        : boolean;         { Fixed status >
      solved__for  : boolean;         ( Whether the variable has been solved for yet }
      incidence    : boolean;         { Whether the variable is incident }
      block        : integer;         { Which block the variable belongs to (solver) }
      destroyed    : boolean;         { Destroyed status }
   END;
   var$_t          = ^var$_spec_t;
   var$_table_t    = UNIV_PTR;
   var$_filter_t   = ^FUNCTION ( IN v : var$_t ) : boolean;
      { Filters need not be able to handle NILs >

%BJBCT;       {*****************************************************************}

PROCEDURE var$_create_table (    OUT   table : var$_table_t
                           );               EXTERN;
{........................................- ____________________________________
| Creates a variable table and returns a handle to it.  The table is initially
| ■■?ty.
|
| OUT
|    table   New variable table.
+------------------------------------------------------------------------------}


PROCEDURE var$ destroy_table ( IN OUT   table : var$_table_t
              ""             );               EXTERN;   "
{_______________________ - __________________________________.--------------------
| Destroys a variable table and all variables in it.
|
| IN
|    table   Variable table to be destroyed.
|
I OUT
|    table   MIL.
+-----------------------------------------------------------------------}


PROCEDURE var$_oreate_yar( IN       table   : var$_table_t
                         ; IN       number  : integer
                         ;    OUT   v       : var$_t
                         );                   EXTERN;
{-----------------------------------------------------------------------
| Creates a variable in the table and returns a pointer to it.  Default values
| of the fields are filled in.  If the number exists already, or is out of
| range, then the lowest available number is used instead.
|
| IN
|    table    The table in which the new variable is to be created,
|    number   Variable number to create.
|
| OUT
|    v        The new variable.
+-----------------------------------------------------------------------}


PROCEDURE var$_destroy_var< IN       table : var$_table t;
                            IN OUT   v     : var$_t    "
                          );                 EXTERN;
{-----------------------------------------------------------------------
| Destroys a variable and removes it from the table.
|
| IN
|    table   The table that the variable is in.
|    v       The variable to be destroyed.
|
| OUT
|    v       NIL.
+-----------------------------------------------------------------------}


FUNCTION var$_get_var ( IN       table  : var$_table_t
                      ; IN       number : integer   ""
                      )                 : var$_t
                      ;                   EXTERN;
{-----------------------------------------------------------------------
| Looks up the variable by number on the variable table and returns a pointer to
| it.  If the variable does not exist, then NIL is returned.
|
| IN
|    table    The variable table,
|    number   Variable number.
|
| RETURN      Pointer to the variable specified.
+-----------------------------------------------------------------------}


FUNCTION var$_exist( IN       table  : var$_table_t
                   ; IN       number : integer
                   )                 : boolean
                   ;                   EXTERN;
{-----------------------------------------------------------------------
| Returns true if the variable exists, false otherwise.
|
| IN
|    table    The variable table,
|    number   Variable number.
|
| RETURN      Whether or not there is a variable in the table with the given
|             number.
+-----------------------------------------------------------------------}
```

```
FUNCTION  var$_f  ind_label  (  IN  table  :  var$_table_t
            .""         ; IN      vlabel : atr$~string t
                        )                : var$~t     "
                        ;                 EXTERN;
{------------------------------------------------------------------------
| Searches the variable table for the given label and returns the first match.
| If there is no match, then NIL is returned.  A variable with a NIL label never
| results in a match.
|
| IN
|    table    The variable table.
|    vlabel   The desired variable label.
|
| RETURN      The variable with the given label.
|
| NOTES
|    All searches are case sensitive.  It is the responsibility of the user to
|    convert all labels to one case if a case insensitive search is desired.
+-----------------------------------------------------------------------}


PROCEDURE var$_number_of_yars ( IN       table         : var$_table_t
                              ;    OUT   number_yars   : integer
                              ;    OUT   lowest_varn   : integer
                              ;    OUT   highest_yarn  : integer
                              );                "       EXTERN;
{------------------------------------------------------------------------
| Returns the number of variables defined in the table as well as the lower and
| upper bound on the variable numbers.
|
| IN
|    table           Variable table.
|
| OUT
|    number__vars    The number of variables.
|    lowest_varn     Lowest defined variable number.
|    highest_varn    Highest defined variable number.
+-----------------------------------------------------------------------}


FUNCTION var$_count( IN       table  : var$_table_t
             ""     ; IN      filter : var$~filter_t
                    )                : integer
                    ;                 EXTERN;
{------------------------------------------------------------------------
| Returns the number of variables for which the filter returns true.
|
| IN
|    table    The variable table,
|    filter   The filter.
|
| RETURN      Number of variables for which the filter returns true.
+-----------------------------------------------------------------------}


FUNCTION var$_get vars( IN       table  : var$ table_t
                ""    ; IN      filter : var$~filter_t
                      )                : pl$ list t "
                      ;                 EXTlRN;  "
{------------------------------------------------------------------------
| Returns a list of variables for which the filter returns true.  The list will
| be in increasing order of variable number.
|
| IN
|    table    The variable table,
|    filter   The filter.
|
| RETURN      List of variables matching the filter.
+-----------------------------------------------------------------------}
```

```
%BJXCT;  {**********************************************************************}
```

```
{ BEGIN eqn.ins.pas -------------------------------------------------------}

{--------------------------------------------------------------------------
I Prevent this file from being included more than once.
+-------------------------------------------------------------------------}
%IFDEF eqn$_already_inserted %THEN
   %EXIT  ""
%BLSE
   %VAR eqn$_already_inserted
%BNDIF     ~

%INCLUDE '/ascend/utilities/ins/str.ins.pas';
%INCLUDB '/ascend/utilities/ins/list.ins.pas';
%INCLUDB '/ascend/solver/ins/var.ins.pas';
%INCLUDB '/ascend/solver/ins/expr.ins.pas';

{--------------------------------------------------------------------------
 | All equation number* oust be greater than zero.  It is recommended, but not
 | required that all equation numbers also be no greater than mtx$_max_order_c.
 | Higher level routines that work with a matrix will require equation numbers
 | to be in this range.  EQUATION NUMBERS SHOULD ONLY BB READ, NEVER CHANGED!
+-------------------------------------------------------------------------}

TYPE
   eqn$__relation__t • (eqn$_less , eqn$_equal , eqn$_greater);

{--------------------------------------------------------------------------
t Warning:  If a field is added, then the assignment of its default value must
 | also be added to internal routine eqn_create_eqn.  In addition, BVERY module
 | above this one must be recompiled, even modules which don't use the new field.
+-------------------------------------------------------------------------}
   eqn$_spec_t      = RECORD
      number        : integer;              { Equation number }
      elabel        : str$_string_t;        { Equation label (NIL: no label) >
      LHS,RHS       : expr$_t;              { LHS and RHS expressions }
      relation      : SBT OF eqn$_relation_t; { Required relation }
      residual      : double;               { LHS - RHS >
      satisfied     : boolean;              { Whether the relation is satisfied or not >
      included      : boolean;              { Whether equation is included }
      block         : integer;              { Which block the equation belongs (solver }
      destroyed     : boolean;              { Whether equation is destroyed }
   END;
   eqn$_t           « ^eqn$_spec_t;
   eqn$JList_t      • UNIV_PTR;
   eqn$_filter_t    * ^FUNCTION ( IN eqn : eqn$_t ) : boolean;
      { Filters need not be able to handle NILs }

%EJECT;       {*****************************************************************}

PROCEDURE eqn$_create_list (    OUT   list : eqn$_list_t
                           );               EXTERN;
{.............-.............-...................- ________________
| Creates an equation list and returns a handle to it.  The list is initially
| empty.
|
| OUT
|    list   New equation list.
+-------------------------------------------------------------------------}


PROCEDURE eqn$_destroy_list ( IN OUT   list : eqn$_list_t
                            );              EXTERN;
{--------------------------------------------------------------------------
| Destroys an equation list and all equations in it.  The expressions are all
| destroyed, but the variables within the expressions are not.
|
| IN
|    list   Equation list to be destroyed.
|
| OUT
|    li*t   NIL.
+------------------------------------------------------------------------------}


PROCEDURE eqn$_create_eqn ( IN       list    : eqnf_list_t
                          ; IN       number  : integer
                          ;    OUT   eqn     : eqn$_t
                          );                   EXTERN;
{-------------------------------------------------------------------------------
| Creates an equation in the equation list and returns a pointer to it.  Default
| values of the fields are filled in.  If the number exists already, or is out
| of range, then the lowest available number is used instead.
|
| IN
|    list     The equation list in which the new equation is to be created,
|    number   Equation number to create.
|
| OUT
|    eqn      The new equation.
+------------------------------------------------------------------------------}


PROCEDURE eqn$_destroy_eqn < IN       list  : eqn$_list_t
                           ; IN OUT   eqn   : eqn$_t
                           );               EXTERN;
{-------------------------------------------------------------------------------
| Destroys an equation and removes it from the list.  The expressions are
| destroyed, but not the variables within the expressions.
|
| IN
|    list   The equation list from which the equation is to be destroyed,
|    eqn    The equation to be destroyed.
|
I OUT
I    eqn    NIL.
+------------------------------------------------------------------------------}


FUNCTION eqn$_get_eqn (  IN       list    :  eqn$_list_t
                      ;  IN       number  :  integer
                      )                   :  eqn$_t
                      ;                      EXTERN;
{-------------------------------------------------------------------------------
| Looks up the equation by number on the equation list and returns a pointer to
| it.  If the equation does not exist, then NIL is returned.
|
| IN
|    list     The equation list,
|    number   Equation number.
|
| RETURN      Pointer to the equation specified.
+------------------------------------------------------------------------------}


FUNCTION eqn$_exist(  IN       list    :  eqn$_list_t
                   ;  IN       number  :  integer
                   )                   :  boolean
                   ;                      EXTERN;
{-------------------------------------------------------------------------------
| Returns true if the equation exists, false otherwise.
|
| IN
|    list     The equation list,
|    number   Variable number.
|
| RETURN      Whether or not there is an equation in the list with the given
|             number.
+------------------------------------------------------------------------------}
```

```
FUNCTION eqn$Jfind_label ( IN       list   : eqn$_list_t
                         ; IN       elabel :  str$~string_t
                         )                 : eqn$~t
                         ;                  EXTERN;
{-----------------------------------------------------------------------------
| Searches the equation list for the given label and returns the first match.
| If there is no match, then NIL is returned.  An equation with a NIL label
| never results in a match.
|
| IN
|    list     The equation list.
|    vlabel   The desired equation label.
|
| RETURN      The equation with the given label.
|
| NOTES
|    All searches are case sensitive.  It is the responsibility of the user to
|    convert all labels to one case if a case insensitive search is desired.
+----------------------------------------------------------------------------}


PROCEDURE eqn$_number__of_eqns ( IN       list          : eqn$_list_t
                               ;    OUT   number_eqns   : integer
                               ;    OUT   lowest_eqnn   : integer
                               ;    OUT   highest_eqnn  : integer
                               );               ~~      EXTERN;
{-----------------------------------------------------------------------------
| Returns the number of equations defined in the list as well as the lower and
| upper bound on the equation numbers.
|
| IN
|    list           Equation list.
|
| OUT
|    number_eqns    The number of equations.
|    lowest__eqnn   Lowest defined equation number.
|    highest_eqnn   Highest defined equation number.
+----------------------------------------------------------------------------}


FUNCTION eqn$_eount( IN       list   : eqn$_list_t
             ""    ; IN       filter : eqn$~filter_t
                   )                 : integer
                   ;                  EXTERN;
{-----------------------------------------------------------------------------
| Returns the number of equations for which the filter returns true.
|
| IN
|    list     The equation list,
|    filter   The filter.
|
| RETURN      Number of equations for which the filter returns true.
+----------------------------------------------------------------------------}


FUNCTION eqn$ get_eqns ( IN       list   : eqn$_list_t
             "         ; IN       filter : eqn$~filter_t
                       )                 : pl$_list t
                       ;                  EXTERN; ""
{-----------------------------------------------------------------------------
| Returns a list of equations for which the filter returns true.  The list will
| be in increasing order of equation number.
|
| IN
|    list     The equation list,
|    filter   The filter.
|
| RETURN      List of equations matching the filter.
+----------------------------------------------------------------------------}
```

```
%EJBCT;        {****************************************************************************}
```

```
{ BEGIN slv.ins.paa -----------------------------------------------------------}

{------------------------------------------------------------------------------
 | Prevent this file from being included more than once.
+...........................................................................----------}
%IFDRF slv$_already_inaerted %THEN
   %SXIT
%BLSB
   %VAR slv$_already_inserted
%BHDIF              ""

%INCLUDB '/aacend/utilities/ins/str.ins.pas' ;
%ZNCLUDB '/aacend/utilities/ins/list.ins.pas';
%INCLUDB '/ascend/solver/ins/var.ins.pas';
%INCLUDB '/ascend/solver/ins/expr.ins.pas';
%INCLUDE '/ascend/solver/ins/eqn.ins.pas';
%INCLUDB '/ascend/solver/ins/mtx.ins.pas';
%INCLUDB '/ascend/solver/ins/linsol.ins.pas';

{------------------------------------------------------------------------------
| There will be (in general) more than one solver.  Since all of them may be
| bound in at once (to give a choice of solvers at run-time), the routines must
| have different names for each solver, and consequently, different insert files
| are required.  The naming convention is slvn$_xxx, where n is the number of
| the solver.  The insert file is correspondingly slvn.ins.pas.  All insert
| files must have the same "structure".  In particular, when a modification is
| made, the contents of one insert file may be copied into another, with the
| only change being the substitution of "slvm$__" for "«lvn$_".
+-----------------------------------------------------------------------------}

TYPE
   slv$_convergence__t  = ( slv$_converged     , { System converged >
                            slv$_diverged      , { Solver gave up }
                            slv$__inconsistent , { Solver found definite inconsistency }
                            slv$_timeout       , { Iteration or time limit exceeded }
                            slv$_continue      );{ Solver still trying }

   slv$_technique_t    = ( slv$_newton           ,
                           slv$_newton_a;radient );

   slv$_status_t      «- RECORD
      ready_to_solve   :: boolean;  { System is ready to be solved }
      everything__ok   :: boolean;  { Everything is ok }
      over_defined     :: boolean;  { System is over-defined }
      under__defined   :: boolean;  { System is under-defined }
      struct_singular  :: boolean;  { System is structurally singular }
      convergence      :: slv$_conv●rg•nce_t;   { Convergence status }
      solver           :: integer;  { Number of the solver being used }
      technique        :: slv$_techniqae__t;     { Recommendation for next iteration }
      calc_ok          :: boolean;  { Jill numerical calculations ok }
      residual         :: double;   { Current total residual }
      iteration        :: integer32; { Current iteration count }
      cpu_elapsed      :: double;   { CPU time elapsed so far }
      block            :: RECORD
         num_of        :: integer;  { Number of blocks }
         current       :: integer;  { Current block number }
         size          :: integer;  { Size of current block }
         prev          :: integer;  { Total size of previous blocks >
         iteration     :: integer;  { Iteration count of current block >
      END;

      verbose          :: boolean;  { Verbose flag }
      partition        :: boolean;  { Whether or not to solve by blocks }
      time_limit       :: double;   { Time limit }
      iteration_limit  :: integer;  { Iteration limit >
      tolerance        :: RECORD    { Residual tolerance (per equation) }
         unsealed      :: double;
         scaled        :: double;
      END;                          { Both tolerances must be satisfied }
   END;

   slv$_system_t        - UNXVJPTR;  { System handle }

$EJECT; {**************************************************************************}

PROCEDURE slv$_create(    OUT   system : slv$_system_t
                     );                  EXTERN;
{--------------------------------------------------------------------------------
| Initializes the internal structure of a system and returns a handle to it.
| The system is initialized with no equation list, no variable table, and a zero
| objective function.
|
| OUT
|    system   Handle to a new system.
+-------------------------------------------------------------------------------}


PROCEDURE slv$_destroy ( IN OUT   system         : slv$_system_t
                       ; IN       destroy_others : boolean
                       );                          EXTERN;
{--------------------------------------------------------------------------------
| The internal structure of the system is destroyed.  If destroy_others is true,
| then the equation list and variable table are also destroyed, otherwise, they
| are both preserved.
|
| IN
|    system           System to be destroyed.
|    destroy__others   Whether or not to destroy equation list and variable table.
|
| OUT
|    system           NIL.
+-------------------------------------------------------------------------------}


PROCEDURE slv$_set_eqn_list ( IN       system   : slv$_system_t
                            ; IN      eqn_list : eqn$_list_t
                            );                  EXTERN;
{--------------------------------------------------------------------------------
| Establishes the equation list for the system.  This must be done before the
| system can be solved.
|
| IN
|    system     The system handle.
|    eqn_list   The equation list to be associated with the system.
+-------------------------------------------------------------------------------}


PROCEDURE slv$_set_var_table ( IN       system    : slv$_system_t
                             ; IN      var__table : var$_table_t
                             );                    EXTERN;
{--------------------------------------------------------------------------------
| Establishes the variable table for the system.  This must be done before the
| system can be solved.
|
| IN
|    system      The system handle.
|    var_table   The variable table to be associated with the system.
+-------------------------------------------------------------------------------}


PROCEDURE slv$_set_obj_function( IN       system : slv$_system_t
                               ; IN       obj    : expr$_t
                               );                  EXTERN;
{--------------------------------------------------------------------------------
| Establishes the objective function for the system.  This must be done for
| optimization problems.
|
| IN
```

```
I    system   The system handle.
|    obj      The objective function to be associated with the system.
•.................................-...................................>


PROCEDURE slv$_get_status ( IN       system : slv$_system_t
                          ;    OUT   status : slv$_status_t
                          );                  EXTERN;
{---------------------------------------------------------------------
| Returns the status associated with the system.
|
| IN
|    system   The system handle.
|
| OUT
|    status   The current status.
+--------------------------------------------------------------------}


PROCEDURE slv$_change_status ( IN       system : slv$ system t
                     "      ; IN       status : slv$~status~t
                            );                  EXTERN;
{---------------------------------------------------------------------
| Modifies the system status.  The only fields that can be modified are:
|
|    technique          Recommended technique.
|    partition          Whether or not to solve each block separately.
|    verbose            Verbose flag.
|    time_limit         CPU time limit.
|    iteration_limit    Iteration limit.
|    tolerance          Residual tolerance, per equation.
|
| All other fields of the status are ignored.
|
| IN
|    system   The system handle,
|    status   New status.
+--------------------------------------------------------------------}


PROCEDURE slv$_presolve( IN       system : slv$_system_t
                       );                  EXTERN;    ""
{______________________________-.....-..............................-....
| Prepares the system for solving.  This procedure must be called before the
| system is solved.  To insure proper behavior of the solver, the user cannot
| modify the equations or the variables in any way after this procedure is
| called, until the solver is done.
|
| IN
|    system   The system handle.
|
| The status can be obtained by slv$_get_status.  The status is affect as
| follows:
|
| OUT
|    everything_ok     True if everything is ok (i.e. the fields below are false).
|    over_defined      Whether or not the system is over-defined,
|    underjdefined     Whether or not the system is under-defined.
|    struct_singular   Whether or not the system is structurally singular.
|
| In addition, the fields ready_to__solve, convergence, technique, iteration, and
| cpu_elapsed are initialized for the first iteration.
+-------------------------------------------------------------------->


PROCEDURE slv$_copy_reorder ( IN       system : slv$_system_t
               ""           ; IN OUT   matrix : mtx$_matrix_t
                            );                  EXTERN;
{---------------------------------------------------------------------
```

```
| Copies the permutation used to reorder the system to the matrix.  All other
| aspects of the matrix are unaffected, except possibly for the order (see
| mtx$_copy_perm).  slv$_presolve must be executed first.
|
| IN
|    system   The system of equations which has been reordered,
|    matrix   The matrix to use.
|
| OUT
|    matrix   The re-permuted matrix.
+-----------------------------------------------------------------------------}


PROCEDURE slv$_get_jacobian_system( IN       system :  slv$_system_t
                                  ;    OUT   ms     :  linsol$_matrix_system_t
                                  );                  EXTERN;
{-----------------------------------------------------------------------------
| Returns the handle to the jacobian system.  It is recommended that at least
| one iteration is performed before getting this system.  If ms=NIL, then the
| matrix system is not available.
|
| IN
|    system   System of equations.
|
| OUT
|    ms       Handle to the jacobian system.
|
| NOTES
|    The user should not assume that the jacobian system has been solved.  If it
|    has and the user attempts to solve it, the linear equation solver will
|    return immediately.
+-----------------------------------------------------------------------------}


FUNCTION slv$_obj_function ( IN       system :  slv$_system_t
                           )                 :  double
                           ;                    EXTERN;
{-----------------------------------------------------------------------------
| Returns the value of the objective function.
|
| IN
|    system   System of equations.
|
| RETURN      Current value of the objective function.
+-----------------------------------------------------------------------------}


PROCEDURE slv$_iterate( IN       system :  slv$_system_t
                      );                   EXTERN;
{-----------------------------------------------------------------------------
| Performs one iteration.  slv$_presolve must be called before calling this
| routine.  If status.ready_to_solve is false, this procedure returns
| immediately.  This routine modifies the variable values in an attempt to
| satisfy all of the equations (or inequalities) in the equation list, and
| minimize the objective function.  In general, it must be called multiple
| times.
|
| IN
|    system   The system handle.
|
| The status can be obtained from slv$_get_status and modified using
| slv$_change_status.  It is used here as follows:
|
| IN
|    ready_to_solve    TRUE if the system is prepared to be solved,
|    technique         Technique to be used this iteration,
|    iteration         Number of previous iterations.
|    cpu_elapsed       CPU time elapsed in previous iterations,
|    verbose           Whether to display useful info on the screen.
|    time_limit        CPU tin* limit.
|    iteration_limit   Maximum number of iterations,
|    tolerance         Residual tolerance (per equation).
|
| OUT
|    ready_to_solve    TRUE if a next iteration is required.
|    everything_ok     TRUE if calc_ok, and ready_to_solve or converged.
|    convergence       Convergence status.
|    technique         Reoonendation for next iteration.
|    calc_ok           All numeric calculations ok.
|    residual          Norm of the residual vector.
|    iteration         Updated to reflect this iteration.
|    cpu__elapsed      Updated to reflect this iteration.
|
| In addition, the variable values and equation residuals are updated
| automatically.
|
| NOTES
|    Current implementation ignores the relations in the equations, treating
|    each equation as an equality condition.  Also, the objective function is
|    ignored.
+-----------------------------------------------------------------------------}


PROCEDURE slv$_solve( IN       system : slv$_system_t
                    );                 EXTERN;    ""
{------------------------------------------------------------------------------
| Repeatedly iterates the system until it is either solved, or the solver gave
| up (i.e. until status.ready_to__solve is false) .  See slv$ iterate for further
| details.                    ~  "                          ~
|
| IN
|    system   The system to solve.
+____________________________________________________________________________>
```

References

L Armijo. (1966). Minimization of Functions having Continuous Partial Derivatives. Pac/y/cJ.Maf/?., 76, 1-3.

Guido Buzzi Ferraris and Enrico Tronconi. (1986). BUNLSI - A Fortran program for solution of systems of nonlinear algebraic equations. *Computers and Chemical Engineering, 10(2),* 129-141.

Richard S. Varga and Michael A. Harrison (Eds.). Donald E. Knuth. (1973). *Fundamental Algorithms.* : Addison-Wesley.

Peter Piela. (1989). *An object-oriented computer environment for modeling and analysis.* Doctoral dissertation, Dept. of Chemical Engineering Carnegie Mellon University,

J.W. Ponton. (1982). The numerical evaluation of analytical derivatives. *Computers and Chemical Engineering,* 6(4), 331-333.

P. Rabinowitz (Ed.). M. J. D. Powell. (1970). *Numerical Methods for Nonlinear Algebraic Equations.* Gordon and Breach.

Karl Westerberg. (January 1989). *Development of software for solving systems of linear equations* (Tech. Rep.). Engineering Design Research Center, Carnegie Mellon University,

A.W. Westerberg and S.W. Director. (1978). A modified least squares algorithm for solving sparse n x n sets of nonlinear equations. *Computers and Chemical Engineering, 2,* 77-81.