ADVANCES IN COMPUTER METHODS FOR PARTIAL DIFFERENTIAL EQUATIONS • III
R. VICHNEVETSKY and R.S. STEPLEMAN (Editors)
Publ. I M A C S - 1979

On the Constructive Approach to Programming : The Case of "Partial Choleski Factorisation" (A Tool for Static Condensation in Structural Analysis)

Alain BOSSAVIT
Bertrand MEYER
Electricité de France, Direction des Etudes et Recherches, Service IMA
1 avenue du Général de Gaulle 92141 CLAMART FRANCE

I - INTRODUCTION

From our experience in solving PDE's for engineering problems, we feel confident that the following statements will generally be agreed upon :

- There is a noticeable gap between *algorithms* presented in textbooks, and their implementation as *programs*.

- Too much unproductive work is done by structural engineers, physicists, numerical analysts, etc., to code again and again close variants of standard algorithms.

The two points are closely related, we think. The fact that, in spite of the existence and availability of popular finite-element codes, in spite of the broad marketing of well documented subroutine libraries, so many people continue to code their own versions of standard methods, is not to be blamed on human weaknesses only. It is also due to a tendency to reorganize old methods into new combinations in order to answer new questions. Thus, slight modifications of existing implementations are always required. But the general obscurity and poor readability of programs make these adaptations harder than they should be, and would be, if the gap we mentioned could be bridged.

Our contention is that this gap can be reduced, thanks to the advances in Computer Science in the recent years. We shall concentrate here on algorithms in linear algebra, relying on results obtained in the domain of proof-oriented program construction techniques, after the work of Hoare (2) and Dijkstra (1). In this approach, methods which were originally used for proving properties of existing programs are applied instead to the top-down design of new ones ; programs and their proofs will be developed in parallel.

The example we discuss is related to structural analysis by the finite-element method. It is often necessary to perform a "static condensation", which is nothing else than elimination of selected variables in a linear system. Though not new (3), the idea is currently in the development stage as far as software is concerned (4), and our implementation is (we hope) novel.

To avoid mixing all difficulties, we shall first state the basic concepts on a toy example (the square root), then proceed with the formal specification of the static condensation problem, the design of the program, and a brief account of the FORTRAN implementation.

II - PROGRAMS AS PREDICATE TRANSFORMERS - BASIC CONCEPTS

Underlined words below denote the few concepts we need. Consider the following text :

```
program square root (input a : REAL ; output x : REAL)
    | {P : a > 0}
    |
    | find x
    |
    | {Q : x^2 = a and x > 0}
```

This is the text of a tautologically correct "program". The notation should be self-explanatory : sentences surrounded by braces { and } are treated as comments ; most often, they will be predicates, i.e. properties which must be verified by the variables of the program (x and a in our example). Texts such as *find x* are statements ; successive statements will be separated by semicolons. Proper indentation will help display the logical structure. A construct of the form $\{P\}\ A\ \{Q\}$ is used to express that Q is true after execution of A if P was true before.

The approach known as top-down design calls now for refinements of the statement *find x*, which must eventually be expressed in terms of elementary actions. A seemingly very productive heuristics toward that end is uncoupling (or embedding) : let us introduce another variable y, also of type *REAL*, and a new predicate I which is a weaker form of the intended conclusion Q

$$(I) \qquad xy = a \ \underline{and}\ x > 0 \ \underline{and}\ y > 0$$

so that $Q = I \ \underline{and}\ C$, where C is

$$(C) \qquad x = y \text{ (within a prescribed margin of accuracy).}$$

A possible way to proceed is to start with such a state of the variables where I is true, and, while keeping it true, to try to reach the "goal" C. One natural way to do this is to look for a program of the form

```
{P}
establish I ;
while not C do
    | bring x and y closer to each other ;
    | restore I
{C and I}
```

which we can assert is correct (i.e. will bring the variables from a state where the initial predicate is verified to a state where the final predicate is verified) provided the loop terminates properly.

This is shown by noting that I is a loop invariant, i.e.

(2) $\{I \text{ and not } C\}\ A\ \{I\}$

where A stands for the two statements of the loop body. Hoare's axioms for the *while* loop imply that, if the loop terminates, we can deduce from (2) that

(3) $\{I\}$ *while not* C *do* A $\{I \text{ and } C\}$

which ensures the validity of program (1).

A third step in the top-down design will be to express more precisely how I is to be "established" (for instance by the two assignments $x \leftarrow a\ ;\ y \leftarrow 1$), how x and y are to be "brought closer to each other" (one possible way is the assignment $x \leftarrow (x+y)/2$), and how I will then be "restored" ($y \leftarrow a/x$).

The program now obtained (a version of Newton's algorithm) is in a form very close to any current programming language, and we may stop here. But the process could be pursued further down if this proved necessary. For instance, if ordinary floating-point division were not available, we might have to refine the statement $y \leftarrow a/x$ into a loop so designed as to achieve the final predicate $xy = a$.

III - STATIC CONDENSATION

We consider now the $n \times n$ rigidity matrix of a structure, in block form

$$(4) \qquad \begin{vmatrix} A & B \\ B^t & C \end{vmatrix}$$

where the last blocks correspond to a subset $E \subset [1,n]$ of the variables, containing only those which may interact with other parts of a higher-level structure, and are called "internal" for that reason. It is often useful to factorize such a matrix in the following way :

$$(5) \qquad \begin{vmatrix} A & B \\ B^t & C \end{vmatrix} = \begin{vmatrix} S & 0 \\ T & H \end{vmatrix} \begin{vmatrix} S^t & T^t \\ 0 & 1 \end{vmatrix}$$

The matrix $H (H = C - TT^t)$, which links displacements (or other generalized variables) of external type and related forces may be used in a higher-level assembly process and T and S allow one to go back to the elementary displacements once external variables are known. It seems interesting to allow "external variables" to include load parameters, Lagrange multipliers, right-hand members in general. Thus, a software tool capable of solving (5), which may be called a "condenser" (it is an extension of the classical "solver"), appears as an essential piece of any finite-element code where sub-structuring is to be implemented.

It is not practical, however, to number the variables in such a way that internal variables will always appear first as in (5) : identical sub-structures may have different sets of external variables (fig. 1), so that a different set of pointers to the same file must be used for each condensation. If interval variables are treated first (for example through a Cholesky routine), T being computed next, and then H, a lot of page faults will occur if the file is not entirely in core.

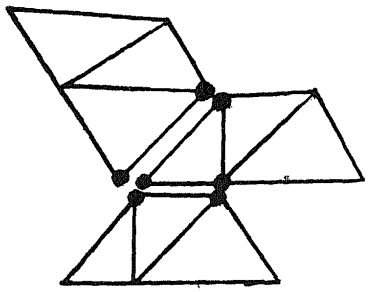

Figure 1 : Assembly of three identical substructures, with different sets of external variables

The problem to be solved may thus be sstated as follows : given a $n \times n$ symmetric matrix a, and a subset E of $[1,n]$, find a symmetric $n \times n$ matrix s such that

$$I(n) \quad \left| \begin{array}{l} \textit{if } 1 \leq j \leq i \leq n \textit{ then} \\ \quad \textit{if } j \notin E \textit{ then} \\ \quad\quad \sum'_{k \leq j} s_{ik}\, s_{jk} = a_{ij}\ , \\ \quad \textit{if } j \in E \textit{ and } i \notin E \textit{ then} \\ \quad\quad \sum'_{k \leq i} s_{ik}\, s_{jk} = a_{ij}\ , \\ \quad \textit{if } j \in E \textit{ and } i \notin E \textit{ then} \\ \quad\quad s_{ij} = a_{ij} - \sum'_{k \leq n} s_{ik}\, s_{jk} \end{array} \right.$$

where $\sum'$ stands for a summation where external indices are omitted. Entries s_{ij} correspond to S in (5) when i and j are both internal, to T when one of them is external, to H when both are external.

The above predicate, which we call $I(n)$ for reasons which will be made clear below, is the final predicate of the program that is to be built. Obviously, it requires positive definitions of the internal-internal part of a, which will be assumed as initial predicate, although checked in practice by the program itself.

We shall add the following requirement : assuming that a and s are stored in "symmetric mode", i.e. line after line, in one-dimensional arrays, the processing should be as "sequential" as possible. This will guide the search for a suitable loop-invariant.

Only Cholesky factorization is considered below ; using the LDL^t variant would not be more difficult.

IV - TOP-DOWN DESIGN OF THE SELECTIVE FACTORIZATION

We embed the final predicate $I(n)$ in a family of predicates $I(l)$, obtained by substitution of l for n in $I(n)$; $I(0)$ is trivially true.

Choosing the "uncoupling"

(6) $I(n) = I(l)$ *and* $(l=n)$

then leads to the following program :

```
program selective_Cholesky (input a : SYMMETRIC_MATRIX,
                                E : subset of
                                        INTEGER;
                            output s :
                                        SYMMETRIC_MATRIX)
    variable l : INTEGER ;
    l ← 0 ;      {I(0)}
    while  l < n  do
        l ← l+1 ;
        restore I(l)
    {I(n)}
```

For the second refinement step, we look for a sequence A of actions such that

(7) $\{I(l-1)\}\ \ A\ \ \{I(l)\}$

It is sufficient to find A such that :

(8)
$\{1 \leq j \leq i \leq l-1$ *and* $i \in E$ *and* $j \in E \Rightarrow s_{ij} = \sum' s_{ik}\, s_{jk}\}$
A
$\{J(l)\}$

where $J(l)$ is defined as

$J(l)$:
if $1 \leq j \leq l$ *then*
  if $j \notin E$ *then* $\sum'_{k \leq j} s_{lk}\, s_{jk} = a_{lj}$,
  if $j \in E$ *and* $l \notin E$ *then* $\sum'_{k \leq l} s_{lk}\, s_{jk} = a_{lj}$
  if $j \in E$ *and* $l \in E$ *then*
    for all $i \geq j$ *and* $i \in E$
      $s_{ij} = a_{ij} - \sum_{k \leq l} s_{ik}\, s_{jk}$

In order to solve problem (8), we again use the "uncoupling" strategy. Let l, an *INTEGER*, and y, a *REAL*, be new variables. We can express $J(l)$ as :

(9) $J(l) = J(l,r,y)$ *and* $r=l$ *and* $y=s_{ll}$ *and* $K(l)$

where $J(l,r,y)$ is :

$J(l,r,y)$:
if $i \leq j \leq r$ *and* $r \leq l$ *then*
  if $j \notin E$ *then* $\sum'_{k<l} s_{lk}\, s_{jk} = a_{lj}$,
  if $j \in E$ *and* $l \notin E$ *then* $\sum'_{k<l} s_{lk}\, s_{jk} + y\, s_{jl} = a_{lj}$

and $K(l)$ as :

$K(l)$:
if $l \notin E$ *and* $1 \leq j \leq i \leq l$ *and* $i \notin E$ *and* $j \notin E$ *then*
  $s_{ij} = a_{ij} - \sum'_{k \leq l} s_{ik}\, s_{jk}$

Examination of (9) suggests that action A should be constructed as the composition of a *while* loop admitting $J(l,r,y)$ as an invariant, and suitable actions which will ensure $y = s_{ll}$ *and* $K(l)$. In other words, we shall look for an A of the following form :

```
A
    r ← 0 ; y ← 1 ;
    while  r < l  do
        r ← r+1 ;
        restore J(l,r,y) ;
    ensure  y = s_ll  while maintaining  J(l,l,y) ;
    ensure  K(l)
```

We could pursue the process further, and show how the last sub-actions of A can be systematically developed ; the approach of proof-directed top-down program design should be clear by now, however, and we shall give the final product without further justification. The program for selective Choleski factorization, expressed in our notation, appears on the next page.

```
program selective_choleski
        (input  a : SYMMETRIC_MATRIX,
                e : FUNCTION (INTEGER → LOGICAL) ;
         output s : SYMMETRIC_MATRIX)

{input assertion :
      the internal-internal part of a is
                              positive-definite}

variables
  n, l, r, i, j : INTEGER,
  y             : REAL ;

n ← order (a) ; l ← 0 ;

{I(l)}

while  l < n  do     {loop invariant : I(l)}
  l ← l+1 ;
  {restore I(l) : }
    r ← 0 ; y ← 1   {J(l,r,y) is satisfied}
    while  r < l  do   {loop invariant :
                                  J(l,r,y)}
      r ← r+1 ;
      {restore J(l,r,y) :}
        if  not  e(v)  then
            if  r < l  then
                s_lr ← (a_lr - Σ'_{k<r} s_lk s_rk)/s_rr
            else
                s_ll ← √(a_ll - Σ'_{k<l} s_lk²)
                {positive definiteness
                 guarantees that the square
                 root is defined}
        elsif  not  e(l)  then
            s_lr ← (a_lr - Σ'_{k<l} a_lk a_rk) / y  ;
        if  not  e(l)  then
            y ← s_ll ;
            for  0 < r < l  while e(r)  do
                s_lr ← s_lr / y ;
            for  1 ≤ j ≤ i ≤ l
            while  e(i)  and  e(j)
            do
                s_ij ← s_ij - s_li s_lj
        {J (l,r,y) }
    {I(l) }
```

V - A FORTRAN IMPLEMENTATION OF THE SELECTIVE FACTORIZATION.

Included below is a FORTRAN program which is little more than a litteral translation of the above routine. It embodies a few implementation decisions ; in particular :

- s and a have been stored as one-dimensional arrays, where the relative position of an element (i, j) is computed via a function *ADRESS (I, J)*. In the implementation below, it is assumed that s and a are full and fit in core. If skyline is considered, and/or if secondary storage is required, only *ADRESS* has to be modified.

- the $\sum'$ operator is effected through a call to the *SCLPRD* subroutine, which must be consistent with *ADRESS* (pursuing the approach further, we could have defined these functions as belonging to the same "virtual machine", or "abstract data type").

VI - CONCLUSION

We hope to have shown that program design, all the way from the search for an algorithm to the writing and documenting of the final code, can proceed in a disciplined way not unlike the classical mathematical discourse.

We certainly do not mean to imply, however, that this is a smooth and easy process ; indeed, there are choices involved at each step, and sometimes there is no better way to resolve them than trial and error. Uncoupling (9), for instance, requires a good deal of insight, and we had to try a few blind alleys before we found it. Top-down proof-directed program design is certainly not a magical recipe for solving numerical problems ; what the method does bring, however, is a better control of the whole programming process, a better relationship between mathematical methods and their implementations, and, of course, better programs, easier to code, debug, document and modify.

REFERENCES

(1) Dijkstra, E. W. : *A Discipline of Programming* ; Prentice-Hall, 1976.

(2) Hoare, C. A. R. : *An Axiomatic Basis for Computer Programming* ; CACM, 12, 10, 1969, pp. 576-583.

(3) Kron, G. : *Diakoptics, Piecewise Solution of Large Systems* ; London, McDonald, 1963.

(4) Wilson, E. L., and Dovey, H. H. : *Solution or Reduction of Equilibrium Equations for Large Complex Structural Systems* ; Advances in Engineering Software, 1, 1, 1978, pp. 19-26.

APPENDIX

A FORTRAN Program for the Selective Factorization

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      SUBROUTINE
     .                             S E L C H O
     .
     .                       (N,A,E,S,POSDEF,RANK)
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C                                FUNCTION
C
C        SELECTIVE CHOLESKI FACTORIZATION (I.E. DEALING WITH "INTERNAL"
C     VARIABLES ONLY, THOSE FOR WHICH E(I) = .FALSE.), WITH SIMULTANEOUS
C     DETERMINATION OF INTERNAL-EXTERNAL RELATIONS AND OF THE "CONDENSED
C     MATRIX' WHERE ONLY EXTERNAL VARIABLES ARE CONSIDERED.
C        S AND A MAY BE THE SAME REAL ARGUMENT IN THE CALLING PROGRAM.
C        "SEQUENTIAL" VARIANT OF THE ALGORITHM, SUITABLE FOR AN OUT-OF-
C     CORE IMPLEMENTATION. THIS PROGRAM IS NOT WRITTEN FOR ACTUAL USE,
C     BUT COULD SERVE AS A TEMPLATE FOR AN OPERATIONAL VERSION.
C
C                            FORMAL ARGUMENTS
C
C ************************************************************************
C        TYPE         NAME        ARRAY ?                MEANING
C ****************|*********|************|*******************************
C  INPUT
C
      REAL            A          ( 1 )
C                                           | SYMMETRIC POSITIVE-DEFINITE
C                                           | MATRIX. 1-D ARRAY STORAGE :
C                                           | A(I,J) IS IN A(ADRESS(I,J))
      INTEGER         N
C                                           | ORDER OF A.
      LOGICAL         E          ( 1 )
C                                           | E(I) <==> I IS "EXTERNAL".
C
C  OUTPUT
C
      REAL            S          ( 1 )
C                                           | OUTPUT OF THE SELECTIVE FACTO
C                                           | RIZATION. STORED LIKE A.
      LOGICAL         POSDEF
C                                           | TRUE IF A IS INDEED POSITIVE
C                                           | DEFINITE.
      INTEGER         RANK
C                                           | N IF POSDEF ; ELSE, RANK OF
C                                           | THE LINE WHERE A NEGATIVE
C                                           | DIAGONAL TERM APPEARED.
C ************************************************************************
C
C                         THIS UNIT MAKES CALLS TO :
C                        REAL FUNCTION SCLPRD(S,E,I,J)
C
C////////////////////////////////////////////////////////////////////////
C     THE FOLLOWING IS TRUE AFTER EXECUTION OF SELCHO :
C
C    IF POSDEF, THEN
C     |    IF 1 <= J <= I <= N, THEN
C     |    |    IF NON-E(J), THEN
C     |    |     |  SIGMA( SIK*SJK | K <= J, NON-E(K)) = AIJ
C     |    |    IF NON-E(I) & E(J), THEN
C     |    |     |  SIGMA( SIK*SJK | K <= I, NON-E(K)) = AIJ
C     |    |    IF E(I) & E(J), THEN
C     |    |     |  SIJ = AIJ - SIGMA( SIK*SJK | K <= N, NON-E(K))
C
C       WITH SYMMETRY : S(I,J) = S(J,I) AND SAME FOR A .
C
C////////////////////////////////////////////////////////////////////////
C     LOCAL VARIABLES :
          INTEGER AD, I, J, L, R
          REAL RADIC, Y
C     ARITHMETICALLY DEFINED FUNCTION :
          INTEGER ADRESS
          ADRESS(I,J) = (I*(I-1))/2 + J
C
C
      POSDEF = .TRUE.
      L = 0
C     /WHILE L < N (I.E. FOR EACH LINE OF A), REPEAT/
1     IF(L.GE.N) GOTO 11
          L = L + 1
C         /RESTORE THE VALIDITY OF THE LOOP-INVARIANT I(L)/
              R = 0
C             /WHILE R < L (I.E. FOR ALL LINES UP TO L),REPEAT/
2             IF(R.GE.L) GOTO 6
                  R = R + 1
C                 /COMPUTE SLR IN THE INTERNAL-INTERNAL CASE, AND
C                 SLL*SLR IN THE INTERNAL-EXTERNAL CASE/
                      AD = ADRESS(L,R)
C                     /IF R INTERNAL, THEN/
                      IF(E(R)) GOTO 4
C                         /IF R < L, COMPUTE SLR/
                          IF(R.EQ.L) GOTO 3
                              S(AD) = (A(AD)-SCLPRD(S,E,L,R))/
                                   S(ADRESS(R,R))
3                         CONTINUE
C                         /IF R = L, COMPUTE THE PIVOT/
                              RANK = L
                              RADIC = A(AD) - SCLPRD(S,E,L,R)
                              POSDEF = RADIC.GT.0.
                              IF(.NOT.POSDEF) GOTO 13
                              S(AD) = SQRT(RADIC)
                          GOTO 5
C                     /ELSE, IF L INTERNAL, THEN/
4                         IF(.NOT.E(L)) S(AD)=A(AD)-SCLPRD(S,E,L,R)
5                 GOTO 2
6             CONTINUE
              IF(E(L)) GOTO 11
C             /IF L INTERNAL/
C                 /ADJUST THE INTERNAL-EXTERNAL TERMS IN LINE L/
C                 /(THE PIVOT IS NOW KNOWN)/
                      Y = S(AD)
                      R = 0
C                     /FOR EACH  EXTERNAL R < L, REPEAT/
7                     IF(R.GE.L) GOTO 9
                          R = R + 1
                          IF(.NOT.E(R)) GOTO 8
                              AD = ADRESS(L,R)
                              S(AD) = S(AD) / Y
8                         GOTO 7
9                     CONTINUE
C                 /INCREMENT THE EXTERNAL-EXTERNAL ENTRIES/
                      DO 10 I = 1,L
                      DO 10 J = 1,I
C                     /FOR EACH EXTERNAL-EXTERNAL (I,J), REPEAT/
                          IF(E(I).AND.E(J)) S(ADRESS(I,J))=S(ADRESS
                          (I,J))-S(ADRESS(L,I))*S(ADRESS(L,J))
10                        CONTINUE
11            CONTINUE
              GOTO 1
12        CONTINUE
C
13    RETURN
      END
```

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      REAL FUNCTION
     .                             S C L P R D
     .                         ( S ,  E ,  I ,  J )
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      REAL S(1)
      LOGICAL E(1)
      INTEGER I, J
C SCALAR PRODUCT OF TWO LINES OF S, SKIPPING EXTERNAL INDICES. SEE
C COMMENTS IN "SELCHO".
      INTEGER IAD,JAD
      DOUBLE PRECISION PS
C
      PS = 0.
      IAD = (I*(I-1))/2 + 1
      JAD = (J*(J-1))/2 + 1
      K = 1
C
C     /WHILE K < J, REPEAT/
1     IF(K.GE.J) GOTO 2
          IF(.NOT.E(K)) SCLPRD = SCLPRD + S(IAD + K) * S(JAD + K)
          K = K + 1
          GOTO 1
2     RETURN
      END
```