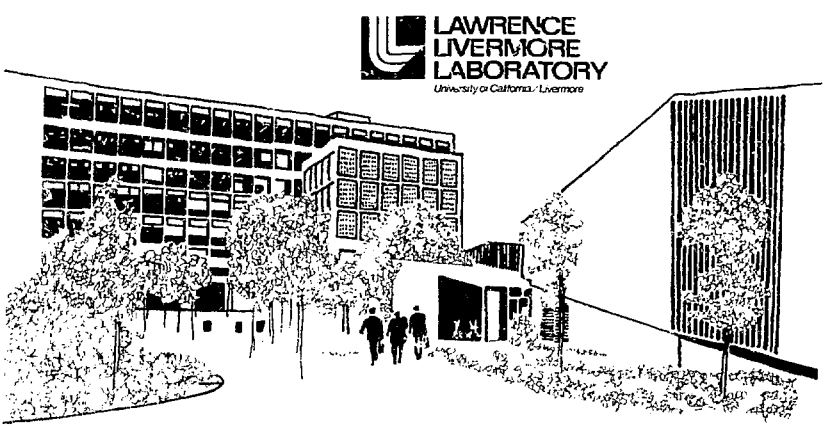
UCRL-51929

SOME OPTIMIZATIONS OF THE ANIMAL CODE

W. T. Fletcher

October 9, 1975

Prepared for U.S. Energy Research & Development Administration under contract No. W-7405-Eng-48



Printed in the United States of America
Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, Virginia 22151
Price: Printed Copy $___*; Microfiche $2.25

*Pages	NTIS Selling Price
1-50	$4.00
51-150	$5.45
151-325	$7.60
326-500	$10.60
501-1000	$13.60

Distribution Category
UC-32 21

LAWRENCE LIVERMORE LABORATORY
University of California/Livermore, California/94550

UCRL-51929

SOME OPTIMIZATIONS OF THE ANIMAL CODE

W. T. Fletcher

October 9, 1975

Some Optimizations of the Animal Code*

Abstract

Optimizing techniques were performed on a version of the ANIMAL code (MALAD1B) at the source-code (FORTRAN) level. Sample optimizing techniques and operations used in MALADOP -- the optimized version of the code -- are presented, along with a critique of some standard CDC 7600 optimizing techniques. The statistical analysis of total CPU time required for MALADOP and MALAD1B shows a run-time saving of 174 msec (almost 3%) in the code MALADOP during one time step. Optimizing strategies were used for these FORTRAN programming concepts:

- Simple DO-loops,
- Two- and three-dimensional arrays,
- Nested DO-loops in which an inner loop has a small limit,
- Computation of constants in a DO-loop, and
- Inefficient arithmetic operations (multiplications and divisions in FORTRAN expressions).

1. Introduction

The computer code ANIMAL - A New Implicit Magnetohydrodynamic Algorithm - is a descendant of the original code developed to study theta pinches and laser produced plasmas.[1] It has been used most recently to study toroidal theta pinches and sausage instabilities of straight z pinches.

This report presents the results of some optimizing techniques performed on a version of this code, called MALAD1B, at the source-code level. The optimized version of the code, called MALADOP, achieved substantial run-time savings on the CDC 7600. Sample optimizing techniques

*This work was done as part of the Summer Computer Institute, 1975, held at Lawrence Livermore Laboratory. The author is currently at Princeton University.

are presented. We also give a critique of some of the standard optimizing techniques described in Ref. 2, which we used to optimize the code. Certain cautions must be carefully observed when using some of these techniques; our observations should be beneficial to a new user at LLL performing optimizations for the first time.

2. Statistical Analysis of Run Time

An analysis of CPU time for MALAD1B and MALADOP is given below:

Code: MALADOP

No. of sample points taken in timing study: 1429

No. of time steps processed by MALADOP: 1

Total CPU: 5865 msec

Distribution of total CPU time, msec (main program and selected subroutines)

MAIN PROG	976.8
NXTZFLX	82.1
MAT2	1296.9
BOVAR	32.8
TRIANG	759.2
RFLX	274.9
ZFLX	57.5
TRANCO	86.2

Code: MALAD1B

No. of sample points taken in timing study: 1406

No. of time steps processed by MALAD1B: 1

Total CPU: 6039 msec

Distribution of total CPU time, msec (main program and selected subroutines)

MAIN PROG	979.3
NXTZFLX	77.3
MAT2	1318.6
BOVAR	60.1
TRIANG	807.5
RFLX	240.5
ZFLX	51.4
TRANCO	77.3

Total time saved in MALADOP during one time step: 174 msec

MAIN PROG, MAT2, and TRIANG account for more than one-half of the total CPU time used by the codes and most of the optimization was done here. Efficiency gain was greatest in the subroutine TRIANG, where the MALADOP version required 5% less time than the MALAD1B version. Forty percent of the total CPU savings achieved occurred in these three routines.

3. Sample Optimizing Techniques

This section gives examples of some optimizing techniques used in MALADOP to improve execution efficiency. Efforts to optimize centered on these FORTRAN concepts:

- Simple DO-loops,
- Two- and three-dimensional arrays,
- Nested DO-loops in which an inner loop has a small limit,
- Computation of constants in a DO-loop, and
- Inefficient arithmetic operations (multiplications and divisions) in FORTRAN expressions.

Example 1:

MALAD1B's Method

```
      DO
35    VAR (L+NDADT) = RL(L)
```

MALADOP's Method

```
      NRZ2 = (NRZ/2) * 2
      DO 35 L = 1, NRZ2, 2
      VAR(L+NDADT) = RZ(L)
35    VAR(L+1-NDADT) =
     1 RZ(L+1)
      IF(NRZ .GT. NRZ2)
     2 VAR(NDADT + NRZ)
     3 = RZ(NRZ)
```

In MALADOP's method, the frequency of execution of the DO-loop closing code of the FORTRAN compiler is halved.

Example 2:

MALAD1B's Method

```
      DIMENSION BC (10, 10, 2), B(10, 10)
      DO 650 LL = 1, NV
      DO 650 MM = 1,NV
650   B(LL,MM) = - BC(LL, MM, NNJ)
```

MALADOP's Method

```
      DIMENSION BC(10, 10, 2), B(10,10)
      NJM1 = 100 * (NNJ-1)
      DO 650 LL = 1, NV
      LLMM = LL
      LMNJ = LL + NJM1
      DO 650 MM = 1, NV
      B(LLMM) = - BC(LMNJ)
      LLMM = LLMM + 10
650   LMNJ = LMNJ + 10
```

This example illustrates the use of identities to transform multiple-dimensional arrays into one-dimensional arrays. MALADOP's method is twice as fast as the one used in MALAD1B, since expensive subscript computation has been simplified and memory fetches of each index LL, MM, NNJ within the loop are replaced by a single fetch of LLMM and LMNJ.

Example 3:

MALADIB's Method

```
      DO 41 J = 1, NDR
      DO 41 K = 1, NDZ
      DO 41 L = 1, NV
      JKL = J+NDADT+NDR*((K-1) +
     1      NDZ*(1-1))
      LJK = L+NTIME + NV*((J-1) +
     2      NDR*(K-1))
41    XLCM(LJK) = VAR(JKL)
```

MALADOP's Method

```
      NRNZ = NDR * NDZ
      NVDR = NV * NDR
      LJK1 = 0
      DO 41 J=1, NDR
      JKL2 = 0
      LJK2 = 0
      DO 42 K = 1, NDZ
      JKL = NDADT + J + JKL2
      LJK = NDADT + J + JKL2
      LJK = NTIME + LJK2 + LJK1 + 1
      DO 43 L + 1, NV
      XLCM(LJK) = VAR(JKL)
      LJK = LJK + 1
43    JKL = JKL + NRNZ
      LJK2 = LJK2 + NVDR
42    JKL2 = JKL2 + NDR
41    LJK1 = LJK1 + NV
```

This example illustrates the use of programming techniques to improve the efficiency of arithmetic operations appearing in a complicated FORTRAN expression within a DO-loop. The 2*NV*NDZ*NDR multiplications which occur in MALADIB's DO-loop have been replaced by two multiplications outside MALADOP's nest of loops and additions within the loops. Since integer addition requires only 12 7600 clock periods and integer multiplication requires 15 - 20 clock periods, the new method if is faster than the old method in MALADIB.

Example 4:

MALADIB's Method

```
      DIMENSION ABC (10, 10, 3)
      DO 760 L = 1, NV
      LL = J1VV (L)
      V(L) = V(LL)
      DO 760 M = 1, NV
      MM = J1VV (M)
      DO 760 N = 1, 3
760   ABC(L,M,N) = ABC (LL, MM, N)
```

MALADOP's Method

```
      DO 760 L = 1, NV
      LL = J1VV (L)
      V(L) = V (LL)
      LMN = L
      DO 760 M = 1, NV
      MM = J1VV (M)
      LLMN = LL + 10 * (MM - 1)
      ABC (LMN) = ABC (LLMN)
      ABC (LMN + 100) = ABC (LLMN + 100)
      ABC (LMN + 200) = ABC (LLMN + 200)
760   LMN = LMN + 10
```

Since even a simple DO-loop requires expensive stores and fetches, 'unwinding' DO-loops manually gains execution efficiency. The above example illustrates this technique.

Example 5:

MALADIB's Method

```
      DIMENSION B(10, 10), A(10, 10)
      DO 180 M = 1, NV
180   B(N,M) = B(N, M) / A(J,N)
```

MALADOP's Method

```
      NN10 = 11 * N - 10
      ANNS = 1./ A(NN10)
      NM2 = N
      DO 180 M = 1, NV
      B (NM2) = B(NM2) *ANNS
180   NM2 = NM2 + 10
```

Divides are quite slow (requiring 30 7600 clock periods) and when performed within a DO-loop, they can appreciably affect the time required to evaluate arithmetic expressions. This example illustrates a method which transforms floating-point divides into floating-point multiplications which require 15 CDC 7600 clock periods.

Example 6:

```
ROGDT = XDDT * RO * RTG/DT
V(6) + V(6) + EI * T21 + EI * ROGDT
```

MALADOP's Method

```
ROGDT = XDDT * RO * (RTG/DT)
V(6) = V(6) + EI * (T21 + ROGDT)
```

The expression ROGDT = XDDT * RO * (RTG/DT) is faster than ROGDT = XDDT * RO * RTG/DT, because in the former case, the divide is being done while XDDT and RO are fetched and multiplied. In the latter case, the divide is done last and the result is stored after a full 20-clock-cycle wait for the divide to complete.

4. Cautions

When a programmer is asked to optimize a code which he did not write originally and which he doesn't know much about, he must be extremely careful not to alter the logic of the original code. In this section, we present some troublesome statements where optimization of the MALADIB code produced logical errors. These examples serve as warnings to other programmers to avoid the author's errors and, equally important, serve as guides -- should an error occur -- to places to look for errors during the debugging state.

Example 7:

Old Method

```
      DO 208 L = 1, NBCH
208   K2BC (L) = 1
```

New Method

```
      NB2 = (NBCH/2) * 2
      DO 208 L = 1, NB2, 2
      K2BC (L) = 1
208   K2BC (L+1) = I
      IF (NBCH .GT. NB2) K2BC (NBCH) = 1
```

Remark:

When NBCH = 1, the new method is invalid, since it sets the value of the element K2BC(2) equal to 1. In the old method, this element does not receive the value of 1 in the DO-loop.

Example 8:

Old Method

```
      DIMENSION A(10,10), B(10,10),
     1    C(10,10)
      DO 160 LL = 1, NV
      B(N,LL) = B(N,LL) - A(NM1) *
     2    B(M,LL)
      IF (IF1 .NE. 2) GO TO 160
      DO 160 LL = 1, NV
      C(N,LL) = C(N,LL) - A(NM1) *
     3    C(M,LL)
160   A(NM1) = 0.0
```

New Method

```
      DIMENSION A(10,10), B(10,10),
     1    C(10,10
      NM = N
      MM = M
      LNM = N
      LMM = M
      DO 160 LL = 1, NV
      B(LNM) = B(LNM) - A(NM1) *
     2    B(LMM)
      IF (IF1 .NE. 2) GO TO 160
      DO 161 LL + 1, NV
      C(NM) = C(NM) - A(NM1) *
     3    C(MM)
      NM = NM + 10
161   MM + MM + 10
      A(NM1) = 0.0
      LMN = LMN + 10
160   LMM = LMM + 10
```

Remark:

The new method in the above example contains a logical error. When IF1 ≠ 2, control is transferred to statement number 160 containing the expression LMM = LMM + 10, but control should have been transferred to the statement containing the expression A(NM1) = 0.0. This error can be easily corrected.

General Remark:

References made to a multiple-dimension array using integer variable indices [e.g., A(I,J) or A(I,J,K)]

require considerable run-time computation. It will increase efficiency to transform these arrays to one-dimensional arrays, as illustrated in previous examples.

When the arrays are referenced with integer constant indices, however, no such run-time computation occurs (at least on LLL's PUTT compiler), so no advantage is gained by such a transformation.

(In this connection, it is interesting to note that LLL's PUTT compiler does not accept array indices of the form I * K. It does, however, accept indices of the form I + K.)

References

1. J. Lindemuth and J. Killeen, J. Comp. Phys. 13, 181 (1973).

2. F. McGirt, K. J. Melendez, and L. Rudsinski, "CDC 7600 FORTRAN Optimizing Techniques, Los Alamos Scientific Laboratory, Rept. LA-5219-MS, UC-32 (June, 1973).

CB/gw