SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 0(0), 0 (XXX 1900)

Restructuring Software: A Case Study

TIM HOPKINS
Computing Laboratory, University of Kent, Canterbury, CT2 7NF, Kent, UK
trh@ukc.ac.uk

SUMMARY

We use knot count and path count metrics to identify which routines in the Level 1 BLAS **might benefit from code restructuring. We then consider how both logical restructuring and the improvements in the facilities available from successive versions of Fortran have allowed us to improve both the complexity of the code as measured by knot count, path count and cyclomatic complexity, and the user interface of one of the identified routines which compute the Euclidean norm of a vector. With these reductions in complexity we hope that we have contributed to improvements in the maintainability and clarity of the code. Software complexity metrics and the control graph are used to quantify and provide a visual guide to the quality of the software, and the performance of a Fortran code restructuring tool is reported. Finally we give some indication of the cost of the extra numerical robustness offered by the** BLAS **routine over the use of new Fortran 90 intrinsic functions.**

KEY WORDS Software complexity metrics Fortran Euclidean norm Software tools

INTRODUCTION

Three of the most important qualities of software, as far as the end user is concerned, are robustness, reliability and efficiency. Those interested in fixing bugs and upgrading software are also keen that the code be easy to understand, test, and maintain. Details of these, and other important software factors may be found in Fenton[1] and Watts[2].

The ability to produce clear, readable code depends both on the implementation language and the programmer. The programming language needs to have the control and data structures available to allow the coder to generate a clean and simple, yet efficient, implementation of the algorithm. In general, the fewer facilities a language provides the more difficult it is for a programmer to generate such code.

Fortran has evolved from the very simple 'high level' language of the 1950's, through Fortran 66, the first programming language to be standardised[3], and Fortran 77[4], a relatively minor modernisation, to the much larger and more complex Fortran 90[5].

The Level 1 Basic Linear Algebra Subroutines (BLAS)[6], originally published in Fortran 66, implemented a number of common vector operations and were designed to be used as building blocks for linear algebra software. Routines were provided to deal with the different numerical data types available in Standard Fortran 66 (default precision for real and complex and double precision real). The routines were designed to be both numerically robust and efficient (many used loop unrolling[7] to obtain better performance), although for maximal efficiency it was expected that specially tuned versions would be made available for particular hardware platforms.

0038–0644/00/000000–01$5.50

Received 0 XXX 0000
Revised 0 XXX 0000

First we describe briefly three software metrics and look at how they were used to help identify which of the Level 1 BLAS were likely to be difficult to understand, test and maintain. We then consider how restructuring the code affected the metrics to ascertain whether the chosen measures offer any help in quantifying the improvements made to the code. We also report on how well two Fortran code restructurers fared on the original and rewritten Fortran 66 routines. A Fortran 77 version of one of the routines is presented. We then consider how the new facilities offered by Fortran 90 may be used to improve both the code and the user interface. Finally we look at the impact which the universal use of IEEE floating-point arithmetic may have on the underlying method and the cost of numerical robustness.

SOFTWARE COMPLEXITY MEASURES

In attempting to assist in the production of high quality software, various methods have been proposed to limit the complexity of individual code modules and many methods of measuring code complexity have been advocated (see Zuse[8] for an extensive bibliography). In order to help in quantifying the effects of any changes we make to the software, we will consider the use of three of these metrics

1. knot count[9] which provides some indication of coding clarity,
2. path count (a variant on the metric proposed by Nejmeh [10]) which gives an estimate of the effort required for testing,
3. cyclomatic complexity[11] which also attempts to estimate the testing effort required.

In addition we will look at a pictorial representation of the code in the form of a control graph. The program control graph is a directed graph consisting of basic blocks of code (nodes) connected by directed arcs (edges) corresponding to the flow of control between these nodes. A basic block of code is a sequence of executable statements containing no decisions or transfers of control. Cyclomatic complexity is related to the control graph and is defined to be the cyclomatic number

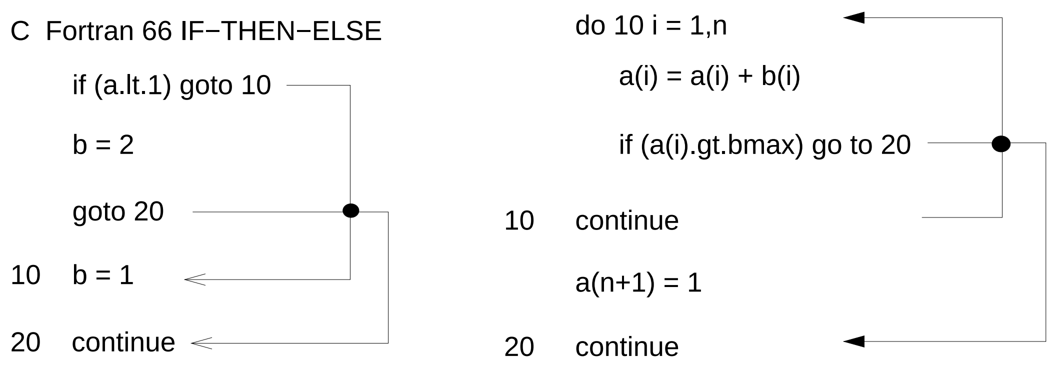
$$V(G) = \text{edges } - \text{nodes } + 2$$

This may be shown to be equivalent to the number of predicates used in the code plus one. It is generally accepted that compound predicates contribute more to program complexity than simple predicates and Myers [12] suggests the use of a complexity interval whose lower bound is $V(G)$ and whose upper bound is one more than the total number of conditions.

Work by Shepperd[13] and Shepperd & Ince[14] has highlighted a number of theoretical problems with the use of cyclomatic complexity as a software metric; we consider these further in the Conclusion.

The knot count of Woodward *et al*[9] gives a good indication of program clarity. It is dependent both on the implementation language and the order of statements in the code. A knot is said to occur in a piece of code whenever the paths associated with two transfers of control intersect (see Figure 1 for two examples). The greater the number of knots in a program unit the less intelligible the code is likely to be.

The static path counts given below are similar to Nejmeh's NPATH statistic[10] and provide an upper bound for the number of possible static paths through the code. Note that some paths so defined may not be executable. The path count complexity of a function is defined as the product of the path complexities of the individual constructions. Thus, for example, the path complexity of a simple if-then-else statement is 2 while three consecutive if-then-else statements would have an associated value of $2^3 = 8$. Three nested if statements would have



```
C  Fortran 66 IF−THEN−ELSE
       if (a.lt.1) goto 10
       b = 2
       goto 20
10     b = 1
20     continue
```

```
       do 10 i = 1,n
          a(i) = a(i) + b(i)
          if (a(i).gt.bmax) go to 20
10     continue
       a(n+1) = 1
20     continue
```

Figure 1. Examples of Knots in Fortran Code

a path count of 4. This metric provides a useful measure of the effort required to test the code stringently as well as giving an indication of code maintainability. A reduction in the path count caused by restructuring code would imply the elimination of paths through the code which were originally either impossible to execute or irrelevant to the computation. This would be likely to reduce the time spent in the testing phase of the software development.

All the values of the software metrics stated in this paper have been generated using QAFortran[15].

THE LEVEL 1 BLAS

The original BLAS publication [6] did not define double precision complex routines. In addition a number of subprograms, for example, multiple precision versions and several mixed precision routines, no longer appear on the BLAS reference card [16] which is now widely considered to be the official definition of the BLAS.

Table I gives the number of lines of executable code in each of the 46 routines listed on the card along with the values of the three metrics defined above. Where routines exist for more than one data type the values given are for the subprogram generating the largest values.

The `*NRM2` and `*ROTMG` families of routines stand out as having very high knot and path counts. Given the low numbers of executable statements, these values indicate that the routines are likely to be difficult to understand, due to the high knot count, and difficult to test thoroughly, due to the large number of possible paths through the code.

To study the effects of code restructuring on these metrics we have chosen a routine from the shorter of the two families, `SNRM2`. In fact the metric values for `SNRM2` are somewhat less than the values given in Table I for the routine which acts on complex input data. The restructuring detailed below is equally applicable to the complex data routine but is easier to follow, as is the description of the underlying algorithm, when applied to the real data case.

Table I. Metric Values for BLAS 1 Routines

Routine	LOC	McCabe	Knots	Paths
*ROTG	22	5:6	2	16
*ROTMG	121	18:18	92	98304
*ROT	22	7:8	1	8
*ROTM	84	13:15	17	144
*SWAP	37	10:11	2	16
*SCAL	22	8:9	2	8
*COPY	31	10:11	2	16
*AXPY	29	11:12	2	16
*DOT	29	10:11	4	32
*DOTU	22	7:8	1	8
*DOTC	22	7:8	1	8
*xDOT	23	7:8	3	16
*NRM2	48	18:19	64	10240
*ASUM	22	8:9	4	8
I*AMAX	22	8:9	3	8

THE EUCLIDEAN NORM ROUTINES

Given an n-vector, $\{x_i\}_{i=1}^n$, the Euclidean norm is defined as

$$||x||_2 = \left(\sum_{i=1}^{n} x_1^2 \right)^{\frac{1}{2}} \tag{1}$$

There are four BLAS routines available for this calculation; one for each of the numerical data types mentioned above. The underlying algorithm is relatively simple and may be considered as a compromise between the efficiency of a naive implementation of (1) as a simple square-and-add loop, and the provable numerical robustness of Blue's algorithm[17]. The BLAS routines form partial sums of squares as

$$\frac{1}{x_{\text{max}}^2} \sum_{i=1}^{r} x_i^2$$

where

$$x_{\text{max}} = \begin{cases} 1 \text{ if } cutlo < z < cuthi \\ z \text{ if } z > cuthi \text{ or } z < cutlo \end{cases}$$

and

$$z = \text{max}_{i=1..r} |x_i|$$

Thus as each element is added into the partial sum there may be, at worst, a change of scale. The values of $cutlo$ and $cuthi$ are dependent on the machine arithmetic and are set in the published code to values which are hopefully applicable to all machines in order to assist with portability (see also below in the section on THE FUTURE OF THE *NRM2 ROUTINES). The original, published code for the single precision version, `snrm2`, with the introductory comments stripped out and with some minor corrections, is presented in Figure 2. (This is the code currently available from *netlib*[18].)

```
      REAL FUNCTION SNRM2 ( N, SX, INCX)
      INTEGER I, INCX, IX, J, N, NEXT
      REAL   SX(1),  CUTLO, CUTHI, HITEST, SUM, XMAX, ZERO, ONE
      DATA   ZERO, ONE /0.0E0, 1.0E0/
      DATA CUTLO, CUTHI / 4.441E-16,  1.304E19 /
      IF(N .GT. 0 .AND. INCX.GT.0) GO TO 10
         SNRM2  = ZERO
         GO TO 300
   10 ASSIGN 30 TO NEXT
      SUM = ZERO
      I = 1
      IX = 1
C                                                 BEGIN MAIN LOOP
   20    GO TO NEXT,(30, 50, 70, 110)
   30 IF( ABS(SX(I)) .GT. CUTLO) GO TO 85
      ASSIGN 50 TO NEXT
      XMAX = ZERO
C                        PHASE 1.  SUM IS ZERO
   50 IF( SX(I) .EQ. ZERO) GO TO 200
      IF( ABS(SX(I)) .GT. CUTLO) GO TO 85
C                                PREPARE FOR PHASE 2.
      ASSIGN 70 TO NEXT
      GO TO 105
C                                PREPARE FOR PHASE 4.
  100 CONTINUE
      IX = J
      ASSIGN 110 TO NEXT
      SUM = (SUM / SX(I)) / SX(I)
  105 XMAX = ABS(SX(I))
      GO TO 115
C                   PHASE 2.  SUM IS SMALL.
C                             SCALE TO AVOID DESTRUCTIVE UNDERFLOW.
   70 IF( ABS(SX(I)) .GT. CUTLO ) GO TO 75
C                     COMMON CODE FOR PHASES 2 AND 4.
C                     IN PHASE 4 SUM IS LARGE.  SCALE TO AVOID OVERFLOW.
  110 IF( ABS(SX(I)) .LE. XMAX ) GO TO 115
         SUM = ONE + SUM * (XMAX / SX(I))**2
         XMAX = ABS(SX(I))
         GO TO 200
  115 SUM = SUM + (SX(I)/XMAX)**2
      GO TO 200
C                  PREPARE FOR PHASE 3.
   75 SUM = (SUM * XMAX) * XMAX
C     FOR REAL OR D.P. SET HITEST = CUTHI
C     FOR COMPLEX      SET HITEST = CUTHI
   85 HITEST = CUTHI/FLOAT( N )
C                   PHASE 3.  SUM IS MID-RANGE.  NO SCALING.
      DO 95 J = IX, N
         IF(ABS(SX(I)) .GE. HITEST) GO TO 100
         SUM = SUM + SX(I)**2
         I = I + INCX
   95 CONTINUE
      SNRM2 = SQRT( SUM )
      GO TO 300
  200 CONTINUE
      IX = IX + 1
      I = I + INCX
      IF( IX .LE. N ) GO TO 20
C              END OF MAIN LOOP.
C              COMPUTE SQUARE ROOT AND ADJUST FOR SCALING.
      SNRM2 = XMAX * SQRT(SUM)
  300 CONTINUE
      RETURN
      END
```

Figure 2. Original Fortran 66 Code for `snrm2`

FORTRAN 66 TO FORTRAN 77

However harmful it may be considered [19], the use of explicit GOTO statements and labels was unavoidable in Fortran 66 due to the extremely limited control structures that were provided. For example, there was no IF-ELSE IF-ELSE block available; thus even a simple IF-THEN-ELSE construction required two labels and two GOTO statements, and generated a knot. In restructuring the code we looked primarily at reducing the knot count; we also report the number of explicit GOTO statements and target labels (i.e., those labels that appear explicitly in a GOTO statement) in the code as these contribute both to the knot count and to the ease of understanding of the code. It was hoped that the path count and cyclomatic complexity values would provide an indication of the effect of any restructuring had on the test effort required. Any reduction in these metrics could be viewed as a reduction in the work required to ensure a comprehensive path coverage was achieved.

Figure 3 shows the control graph of the source given in Figure 2 and reflects the complexity of the code. Each circle in the control graph represents a basic block. The numbers in parentheses denote the lines of code, given in the equivalent source code listing, which make up each basic block; the other number is the block number and is included for ease of reference. The most telling of the metric values is the knot count which, at 41, is extremely high for a routine containing just 44 executable statements. The cyclomatic complexity interval is (13:14) and the path count is 1024, again very high for such a short routine and far in excess of the 200 maximum for a routine quoted in Nejmeh[10] for a similar metric. In addition the code contains 15 explicit gotos and 13 target labels.

Spag[20], a software tool designed to improve the structure of Fortran 66 by rearranging (and if necessary duplicating) statements and using Fortran 77 constructions, produced some improvements. The knot count was almost halved to 23, the cyclomatic complexity interval was reduced to (10:11) and the path count was reduced to 256. Nag_struct, one of NAG's suite of Fortran 77 software tools[21], only managed a reduction of one in the knot count although the path count was reduced by a factor of almost three.

Restructuring the original code by hand was more successful. This new Fortran 66 version had a knot count of just 17 and a path count of 256. The cyclomatic complexity interval was (11:16), showing that, although the number of predicates was reduced, the logical complexity of some of them had increased. It is interesting to note here that spag, even using Fortran 77 as its target language, did not manage to reduce the number of knots to this level.

The hand coded Fortran 77 version given in Figure 4 has a knot count of 2 whilst the path count has been reduced to 65. The cyclomatic complexity interval remains the same as for the input Fortran 66 code. There is now a single explicit GOTO and just three labels, two of which are used as do-loop terminators. The other label, both knots and the explicit GOTO are required to test for a zero input vector and the possible return of a zero value.

Figure 5 shows the control graph for the hand coded Fortran 66 version of the routine and clearly shows the improvement in structure over the original. The implicit use of IF-THEN-ELSE constructs is now obvious and it is the translation of these into explicit Fortran 77 constructs which accounts for a large proportion of the reduction in knots. Both commercial restructurers fared much better on this code, generating versions with almost as few knots (4 in each case) and paths (spag, 96 and nag_struct, 65) as the hand crafted Fortran 77.

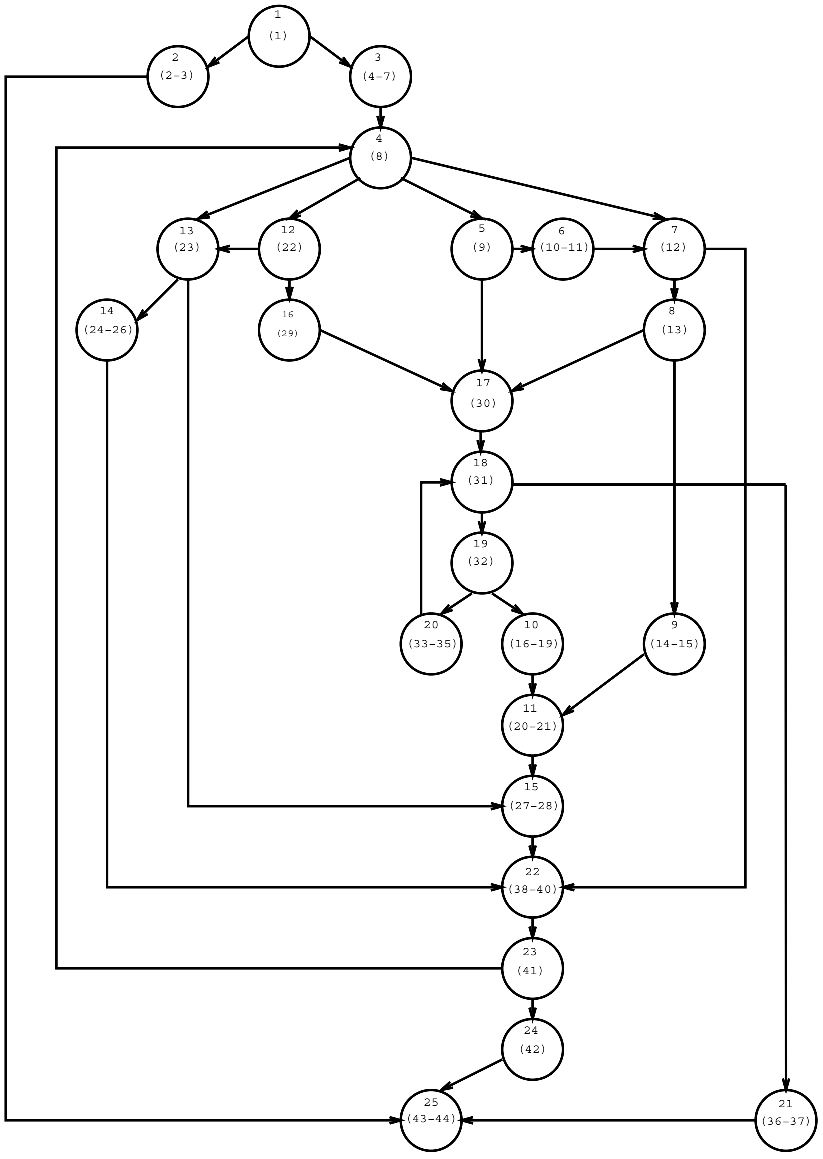


Figure 3. Control graph for the Original Fortran code

FORTRAN 90

Although the new features of Fortran 90 do not provide more than a cosmetic improvement to the final Fortran 77 code (for example, use of the new relational operators) the ENDDO and EXIT statements do allow the removal of the remaining three labels albeit with a little contortion in the case of the break out from the loop testing for the zero vector.

However the new language does offer scope for improving the user interface. Most users

```
      REAL FUNCTION SNRM2(N, SX, INCX)
      REAL CUTLO, CUTHI, ZERO, ONE, HITEST, SUM, XMAX, SX(*)
      PARAMETER (CUTLO=8.232E-11, CUTHI=1.304E19, ZERO=0.0E0,
     *               ONE=1.0E0)
      INTEGER INCX, N, I, NN
      LOGICAL TINY, HUGE
*
      IF (N.LE.0 .OR. INCX.LT.1)THEN
        SNRM2=ZERO
      ELSE
        NN = N*INCX
        HITEST = CUTHI/FLOAT(N)
        DO 10 I=1,NN,INCX
          IF (SX(I) .NE. ZERO) GO TO 20
10      CONTINUE
* ZERO VECTOR
        SNRM2 = ZERO
        RETURN
20      TINY = ABS(SX(I)) .LE. CUTLO
        HUGE = ABS(SX(I)) .GE. HITEST
        XMAX = ABS(SX(I))
        SUM = ZERO
        IF (.NOT.TINY .AND. (.NOT.HUGE)) THEN
* MIDRANGE ... NO SCALING
          SUM = SUM + SX(I)*SX(I)
        ELSE
* NEED TO SCALE
          SUM =ONE
        ENDIF
        DO 30 I=I+INCX,NN,INCX
* TRANSITION FROM TINY (SCALED) TO MIDRANGE (UNSCALED)
          IF (TINY .AND. ABS(SX(I)).GT.CUTLO) THEN
            TINY = .FALSE.
            SUM = (SUM*XMAX)*XMAX
          ENDIF
          IF(.NOT.TINY .AND. (.NOT.HUGE)) THEN
* TRANSITION FROM MID-RANGE TO HUGE
            IF(ABS(SX(I)).GE.HITEST)THEN
              HUGE = .TRUE.
              XMAX = ABS(SX(I))
              SUM=ONE + (SUM/XMAX)/XMAX
            ELSE
* NO TRANSITION (I.E. MIDRANGE)
              SUM=SUM+SX(I)*SX(I)
            ENDIF
          ELSE
            IF(ABS(SX(I)) .LE. XMAX) THEN
* NO NEED TO CHANGE SCALE
              SUM = SUM +(SX(I)/XMAX)**2
            ELSE
* NEED TO SCALE UPWARDS
              SUM = ONE + SUM*(XMAX/SX(I))**2
              XMAX = ABS(SX(I))
            ENDIF
          ENDIF
30      CONTINUE
        IF(TINY .OR. HUGE) THEN
* SCALE RESULT
          SNRM2 = XMAX*SQRT(SUM)
        ELSE
          SNRM2 = SQRT(SUM)
        ENDIF
      ENDIF
      END
```

Figure 4. Restructured Fortran 77 Code for `snrm2`

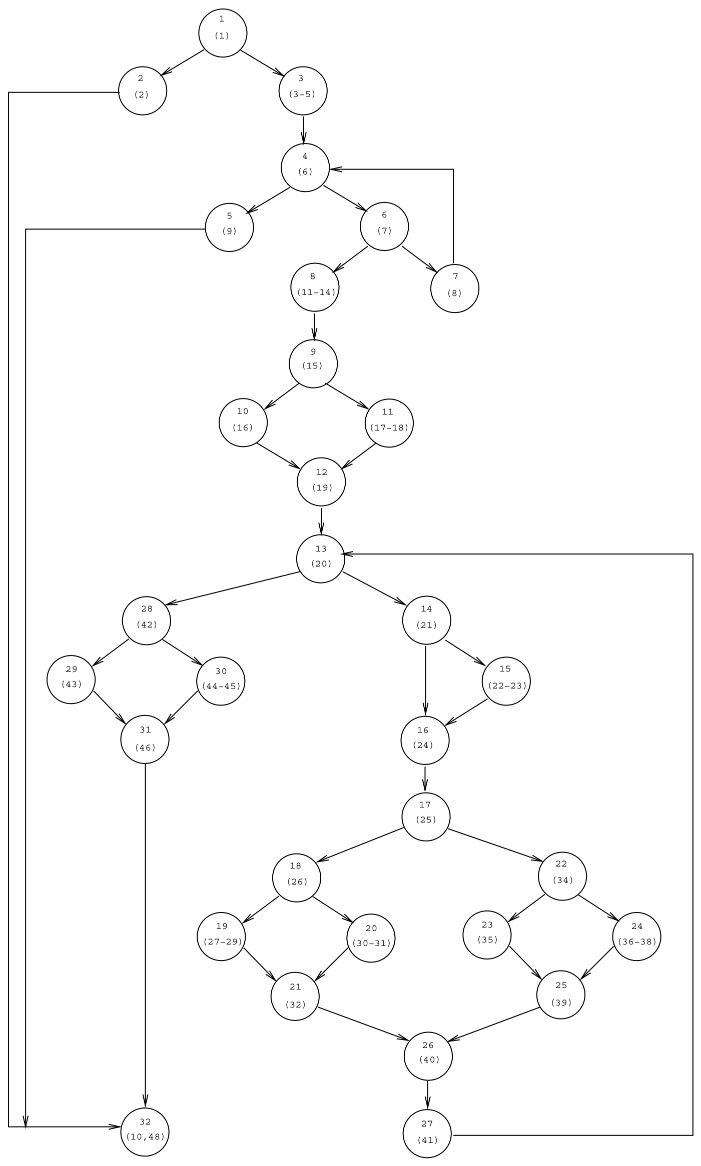


Figure 5. Control graph for the Restructured Fortran 66 code

of the `*nrm2` routines have no need of the `incx` parameter, the most common requirement being to compute the norm of an entire vector. Certainly users would be happier with a call to a single routine rather than having to call different routines for different types of vector; after all Fortran 77 compilers allow this for intrinsic functions like `sin` and `cos`. Being able to write

```
x_norm = nrm2(x,n)
```

where the value returned is given by (1) no matter what the type of `x` would be a step in the right direction. In fact Fortran 90 provides the ability to use

```
x_norm = nrm2(x)
```

where, provided the vector `x` is defined as an assumed-shape array, the routine can obtain the length of the vector, `n`, internally. Additionally, the other two arguments may still be provided if required and the original Fortran 77 routines may be used inside a wrapper routine.

```
function snrm2_opt(sx, n, incx) result (sp_norm)
  integer, optional, intent(in) :: n, incx
  real, intent(in) :: sx(:)
  real :: sp_norm

  integer :: local_n, local_incx
  real :: snrm2

  if (present(n)) then
    localn = n
  else
    local_n = size(sx)
  endif

  if (present(incx)) then
    local_incx = incx
  else
    local_incx = 1
  endif

  sp_norm = snrm2(local_n, sx, local_incx)
end function snrm2_opt
```

Figure 6. Fortran 90 Wrapper Routine for `snrm2`

The three new facilities provided by Fortran 90 which are helpful here are assumed-shape (allocatable) arrays, optional arguments, and user defined generic functions. The maximum size of an array no longer needs to be determined at compile time; dynamic allocation allows the correct amount of storage to be reserved for the particular problem being solved and the size of such arrays may be ascertained when they are passed as parameters to subprograms. Thus

```
real, dimension(:), allocatable :: vector
read *, size_of_problem
allocate (vector(size_of_problem))
```

defines a real array of length `size_of_problem`. A wrapper routine, with `n` and `incx` as optional arguments can be used to generate default values when these parameters are not explicitly provided by the user. The version of a wrapper routine implementing this for a real vector, `snrm2_opt`, is given in Figure 6.

The change in the order of the parameters in `snrm2_opt` is important as the non-optional argument (the input vector) must come first. Similar wrappers may be constructed for the other vector types. It should be noted that explicit use of the double precision type statement has been superseded in Fortran 90 by the use of a real declaration with an appropriate kind selector.

Finally, a generic function, `nrm2`, may be constructed as shown in Figure 7. We note that within `nrm2` the `incx` parameter only needs to be retained for backwards compatibility, since Fortran 90 allows a stride value as part of the triplet used to define an array section. We may thus replace, for example,

```
        x_norm = snrm2(n,x,incx)
```

with

```
        nn = 1 + (n-1)*incx
        x_norm = nrm2(x(1:nn:incx))
```

The interface module needs to be USE'd in all program units and modules wishing to refer to `nrm2`. The improvements in the interface can be obtained without translating the underlying `*nrm2` functions into Fortran 90. Thus Fortran 90 may be viewed as providing added value to existing code.

```
module generic_nrm2
interface nrm2
  function snrm2_opt(sx, n, incx) result (sp_norm)
    integer, optional, intent(in) :: n, incx
    real, intent(in) :: sx(:)
    real :: sp_norm
  end function snrm2_opt

  function dnrm2_opt(dx, n, incx) result (dp_norm)
    integer, optional, intent(in) :: n, incx
    double precision, intent(in) :: dx(:)
    double precision :: dp_norm
  end function dnrm2_opt

  function cnrm2_opt(cx, n, incx) result (c_norm)
    integer, optional, intent(in) :: n, incx
    complex, intent(in) :: cx(:)
    real :: c_norm
  end function cnrm2_opt

  function znrm2_opt(zx, n, incx) result (z_norm)
    integer, optional, intent(in) :: n, incx
    complex*16, intent(in) :: zx(:)
    double precision :: z_norm
  end function znrm2_opt

end interface

end module generic_nrm2
```

Figure 7. Fortran 90 Generic Function for Computing the Euclidean Norm of a Vector

THE FUTURE OF THE `*NRM2` ROUTINES

The four BLAS routines, `*nrm2`, scale partial sums to avoid unnecessary overflows, underflows and possible loss of accuracy. Thus, for example, the 3-vector whose elements are all

$\sqrt{maxreal/2}$ would cause an overflow if a simple square-and-add loop were used. This is unreasonable since the final result, $\sqrt{3}\sqrt{maxreal/2}$, is representable by the floating-point system.

There are two developments since the original routine was published which are worthy of mention. First the IEEE floating-point standard[22] and second the vector and matrix operations available in Fortran 90.

With the increased adoption of the IEEE floating-point standard it may be assumed that the parameters `cutlo` and `cuthi` may soon be replaced by their IEEE values rather than using the given universal values. These two parameters are defined as

$$\begin{aligned} cutlo &= sqrt(u/eps) \\ cuthi &= sqrt(v) \end{aligned}$$

where u and v are the smallest and largest positive numbers representable and eps is the smallest number such that $1 + eps > 1$. (It should be noted that these values may be obtained via the new Fortran 90 numerical inquiry functions `tiny`, `huge`, and `epsilon`.) Table II gives the values as published, which are claimed to be applicable over all machines, along with values specifically associated with the IEEE single and double precision definitions. The probability of scaling taking place in a double precision IEEE routine would appear to be extremely small for most practical applications.

Table II. `cutlo` and `cuthi` Values

	Published[6]		IEEE Standard	
	Single	Double	Single	Double
cutlo	$4.4e-16$	$8.2e-11$	$3.1e-16$	$1.0e-146$
cuthi	$1.3e+19$	$1.3e+19$	$1.8e+19$	$1.3e+154$

Second, Fortran 90 provides two new intrinsic functions, `dot_product` and `sum` which could be used to generate the required norm. These are both generic in the same sense as the `nrm2` function defined above. Thus

$$\begin{aligned} \mathtt{nrm2(x)} &\equiv \mathtt{sqrt(dot\_product(x,x))} \\ &\equiv \mathtt{sqrt(sum(x*x))} \end{aligned}$$

It should be noted that neither the `dot_product` nor the `sum` function offers the numerical robustness of the `nrm2` routines in providing accurate results for a very wide range of numerical inputs.

The overheads involved in using `snrm2`, rather than a simple square-and-add loop in the case of data requiring no scaling, mainly consist of the additional tests at lines 21, 25 and 26 in Figure 4. To gauge the extra cost involved a comparison was made of the execution times of the four methods

1. `snrm2`,
2. a simple square-and-add code,
3. the Fortran 90 intrinsic function, `dot_product`,
4. the Fortran 90 intrinsic function, `sum`.

In each case the vector of length 1000000 was computed 50 times. Three compilers were used, SUN f77 (version 1.4), NAG f90 (version 2.0a(264)) and EPC f90 (version 1.0.1) all running on a SUN 4/670MP under SunOS 4.3.

Table III shows the timings, in seconds, obtained using the Unix time command. Several runs of each program were made and the times averaged. The optimised timings are for the highest level of optimisation available and the non-optimised timings are for debug mode using the `-g` flag. (Note that all three compilers offer more stringent checking in the form of array bound, and in some cases, unassigned variable checks.)

Overall the cost of `snrm2` compared to the square-and-add code is a factor of around two to three. The effectiveness of the `dot_product` intrinsic function was not consistent, being much faster than square-and-add using the EPC compiler and about the same using the NAG compiler. The `sum` routine had little to offer. The `snrm2` code appears to be slightly more susceptible to optimisation than either the square-and-add, or `dot_product` and `sum` codes.

In order to ascertain quantitatively where the extra time was being spent in `snrm2`, rtp[23], a real time profiler, was used to obtain timing information at a statement level. This showed that the execution of the statement

```
sum = sum + sx(i)*sx(i)
```

accounted for approximately 15% of the total execution time of the `snrm2` code and 43% with the simple square-and-add routine. For `snrm2` approximately 30% of the time was used in performing the tests at lines 25 and 26 in Figure 4. It should be noted that rtp uses code sampling to obtain its timings; care was thus taken to ensure that the results obtained were as independent as possible of the sampling frequency.

Table III. Execution times in seconds for vector norm computations

Compiler	mode	`snrm2`	square-and-add	`dot_product`	`sum`
Sun f77	non-opt	114.4	64.0	–	–
Sun f77	opt	62.7	22.4	–	–
Nag f90	non-opt	151.1	63.3	75.0	130.2
Nag f90	opt	69.2	32.9	30.2	49.1
Epc f90	non-opt	264.3	128.4	29.9	67.3
Epc f90	opt	80.2	44.1	13.9	42.5

CONCLUSION

We have looked at how the use of a combination of software metrics (knot and path counts) along with the number of executable lines of code allowed the identification of old Fortran code that was difficult both to understand and to test comprehensively. This would imply that the code would also be hard to maintain. TableIV provides a summary of the various versions of the routine generated along with the associated metric values.

The hand restructured Fortran 66 version, code 4. in Table IV, appeared to be more structured than the codes produced from the original SNRM2 using the code restructuring tools whose target language is Fortran 77. This was highlighted by the fact that both the knot and path counts were reduced to almost ‘optimal’ values when the same tools were applied to the hand-crafted code. This suggests that code 4. was a logically clearer implementation of the algorithm than the original code. The use of Fortran 90 allowed a significant improvement in the user interface.

Table IV. Summary of code versions and associated metrics

Code Version	Language	LOC	Knots	Paths	Cyclomatic	GOTO's	Labels
1. original	f66	44	41	1024	13:14	15	13
2. spag on 1.	f77	47	23	256	10:11	12	10
3. nag_struct on 1.	f77	60	40	385	13:14	12	11
4. hand coded 1.	f66	43	17	256	11:16	15	10
5. spag on 4.	f77	48	4	96	11:16	2	2
6. nag_struct on 4.	f77	48	4	65	11:16	2	2
7. hand coded 4.	f77	48	2	65	11:16	1	3
8. Fortran 90	f90	53	0	65	11:16	0	0

In addition we would assert that the reduction in the path count can be translated into a significant saving in the effort required to produce adequate test data for the code.

It is interesting to note that the cyclomatic complexity interval of the original unstructured code is contained within the interval for the best reported Fortran 77 version. Given that the metric was being applied to Fortran code, for which it was originally intended, and that the path count has been reduced by a factor of almost 16, the values obtained for the cyclomatic complexity clearly do not reflect the reduction in testing effort obtained by restructuring. This also reinforces the point made by Shepperd & Ince[14] that cyclomatic complexity is insensitive to the structure of the software.

In the case of 'dusty deck' Fortran 66 code, automatic restructurers may be able to reduce both the knot and path counts although the extent to which they are successful is very dependent on the way in which the original code was structured. It is worth noting here that the metrics do not always, in themselves, completely reflect improvements; applying spag to the original code lead to a significant reduction in the metric values although the resultant code was still difficult to understand.

Similar reductions in both knot and path counts have also been obtained for the routine SROTMG (see Hopkins[24] for details).

An analysis of the knot and path counts for the 96 Level 2 and Level 3 BLAS[25][26] both developed in Fortran 77, reveals no knots and a maximum path count of 6912 for a 140 line routine. These routines generally contain more executable statements than the Level 1 routines. However the path and knot counts indicate that they are likely to be easier to understand and test than several of the shorter BLAS Level 1 routines. This would suggest that using a combination of number of executable statements with path and knot counts may be helpful in identifying code that is likely to be difficult to understand and maintain.

The overheads of numerical robustness of code were considered and quantified for several commercial compilers; certainly savings would be made in safe cases where vector norm calculations form a substantial part of the computational effort. The final generic function illustrates how end-users may benefit from the backwards compatibility and the new facilities offered by Fortran 90 which allow important core routines, written in Fortran 66 and 77, to be successfully repackaged.

ACKNOWLEDGEMENTS

Thanks to Michel Olagnon from IFREMER whose comments on an earlier draft of this paper were most useful and to the two anonymous referees whose comments improved the paper

considerably.

REFERENCES

1. N. E. Fenton, *Software Metrics: A Rigorous Approach*, Chapman & Hall, London, 1991.
2. R. Watts, *Measuring Software Quality*, NCC Publications, Manchester, UK, 1987.
3. ANSI, *Programming Language Fortran X3.9-1966*, American National Standards Institute, New York, 1966.
4. ANSI, *Programming Language Fortran X3.9-1978*, American National Standards Institute, New York, 1979.
5. ISO/IEC, *Information Technology – Programming Languages – Fortran (ISO/IEC 1539:1991(E))*, ISO/IEC Copyright Office, Geneva, 1991.
6. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, 'Basic linear algebra subprograms for Fortran usage', *ACM Trans. Math. Softw.*, **5**, 308–323, (1979).
7. J. J. Dongarra and A. R. Hinds, 'Unrolling loops in FORTRAN', *Software–Practice and Experience*, **9**, 219–226, (1979).
8. H. Zuse, *Software complexity: measures and methods*, W. de Gruyter, Berlin, 1991.
9. M. R. Woodward, M. A. Hennell, and D. Hedley, 'A measure to control flow complexity in program text', *IEEE Transactions on Software Engineering*, **SE-5**, (1), 45–50, (1979).
10. B. A. Nejmeh, 'NPATH: A measure of execution path complexity and its applications', *Commun. ACM*, **31**, (2), 188–200, (1988).
11. T. J. McCabe, 'A complexity measure', *IEEE Transactions on Software Engineering*, **SE-2**, (4), 308–320, (1976).
12. G. J. Myers, 'An extension to the cyclomatic measure of program complexity', *Sigplan Notices*, **12**, (10), 61–64, (1977).
13. M. Shepperd, 'A critique of cyclomatic complexity as a software metric', *Software Engineering Journal*, 30–36, (March 1988).
14. M. Shepperd and D. C. Ince, 'A critique of three metrics', *J. Systems Software*, 197–210, (1994).
15. Programming Research Ltd, Hersham, Surrey, *QA Fortran 6.0*, 1992.
16. University of Tennessee, Tennessee, US, *Basic Linear Algebra Subroutines: A Quick Reference Guide*, June 1992.
17. J. L. Blue, 'A portable Fortran program to find the Euclidean norm of a vector', *ACM Transactions on Mathematical Software*, **4**, (1), 15–23, (1978).
18. J. J. Dongarra and E. Grosse, 'Distribution of mathematical software via electronic mail', *Commun. ACM*, **30**, (5), 403–407, (1987).
19. E. W. Dijkstra, 'Goto statement considered harmful', *Communications ACM*, **11**, (3), 147–149, (March 1968).
20. Polyhedron Software, Oxford, UK, *plusFORT*, Revision B edition, 1993.
21. Numerical Algorithms Group Ltd., Oxford, UK, *NAGWare f77 Tools*, second edition, September 1992.
22. IEEE, *IEEE standard for binary floating-point arithmetic*, Institute of Electrical and Electronic Engineers, New York, ANSI/IEEE standard 754-1985 edition, 1985.
23. D. J. Barnes, M. T. Russell, and M. C. Wheadon, 'Developing and adapting UNIX tools for workstations', *EUUG Conference Proceedings*, 1988, pp. 321–333.
24. T. R. Hopkins, 'Restructuring the BLAS level 1 fast Givens routines', Technical report, Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK, (In preparation).
25. J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, 'Algorithm 656: An extended set of basic linear algebra subprograms: Model implementation and test programs', *ACM Trans. Math. Softw.*, **14**, (1), 18–32, (March 1988).
26. J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, 'Algorithm 679: A set of level 3 basic linear algebra subprograms', *ACM Trans. Math. Softw.*, **16**, (1), 18–28, (March 1990).