Restructuring the BLAS Level 1 Routine for Computing the Modified Givens Transformation

Tim Hopkins (trh@ukc.ac.uk)
Computing Laboratory
University of Kent
Canterbury, CT2 7NF
Kent, UK.

November 19, 1997

Abstract

We look at how both logical restructuring and improvements available from successive versions of Fortran allow us to reduce the complexity (measured by a number of the commonly used software metrics) of the Level 1 BLAS code used to compute the modified Givens transformation. With these reductions in complexity we claim that we have improved both the maintainability and clarity of the code; in addition, we report a fix to a minor problem with the original code. The performance of two commercial Fortran restructuring tools is also reported.

1 Introduction

The Level 1 BLAS [LHKK79], originally published in Fortran 66 [ANS66], implemented a number of common vector operations and were designed to be used as building blocks for linear algebra software. Hopkins [Hop96] used knot counts [WHH79] and path counts [Nej88] to identify routines from the Level 1 BLAS which might benefit from code restructuring

Two sets of routines, `*NRM2`, used to compute the Euclidean norm of a vector and `*ROTMG`, for computing the modified Givens transformation, were identified as having extremely high metric values given their relatively low number of executable statements. The restructuring of the `*NRM2` routines, along with a dramatic decrease in the metric values, was reported by Hopkins [Hop96]; the `*ROTMG` routines are considered here.

Following a brief description of the software metrics used to compare versions of the `*ROTMG` routines, we present a flowgraph of the published code and look at how two Fortran code restructuring tools fared on this original source. We then compare the metric values obtained for the original and automatically restructured code with hand-coded Fortran 66 and Fortran 77 versions.

Section 5 looks at how the metric values may be reduced further by using Fortran 90 and we show how the use of some of the new facilities available in Fortran 90 may be used to improve these routines further.

Finally we look briefly at the testing of the new routine and report a fix to a minor problem in the original code.

2 Modified Givens Rotation Matrix

The input values to `*ROTMG`, d_1, d_2, x_1 and y_1, define a two-vector $[a_1, a_2]^T$ in the partitioned form as

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} d_1^{\frac{1}{2}} & 0 \\ 0 & d_2^{\frac{1}{2}} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

The routines then determine the *Modified Givens Rotation Matrix*, H, that transforms y_1 and thus a_2 to zero. Details of the computation may be found in Appendix A of [LHKK79].

The values of d_1 and d_2 are scaled to ensure that they are kept within the limits

$$\frac{1}{\gamma^2} \leq |d_i| \leq \gamma^2, \qquad i = 1, 2;$$

where the value of γ was originally chosen to be 4096 for portability reasons; see §5 for details of how this value may be computed using the new Fortran 90 environment enquiry functions. On output, the values of d_1, d_2 and x_1 are changed to represent the effect of the transformation while y_1, which would be zeroed by the transformation, is left unchanged.

In the case where the input vector is already in the correct form, i.e., $(c, 0)^T$, no scaling of the values of d_1 and d_2 takes place even if the input values are outside the limits given above.

Lawson and Hanson [LH74] detail the use of a negative value of d_2 to implement row removal in least squares procedures. The original code thus allows the value of d_2 to be negative as suggested by equation (27.48) on page 230 of [LH74].

3 Software Metrics

We use the following software metrics as indicators of how successful any restructuring we perform has been; a slightly more detailed description may be found in [Hop96].

1. Knot count [WHH79]: a knot is defined to occur in a segment of code whenever the paths associated with transfers of control intersect. The higher the number of knots in a piece of code the more difficult the code will be to read, understand and maintain. As an example, when coding in Fortran 66 the lack of a block IF construction meant that the equivalent code to implement a simple `IF-THEN-ELSE` construction required two `GOTO` statements and one knot.

2. Path count: this is based on the metric proposed by Nejmeh [Nej88] and provides a lower bound on the number of distinct paths through a section of code. This measure gives an estimate of the amount of effort required to thoroughly test the code. Nejmeh suggests a maximum value of 200 for any routine.

3. Cyclomatic Complexity [McC76]: this was one of the first software metrics to be proposed and is calculated as one more than the number of predicates in the code. It was originally proposed as a measure of testing effort although this has been questioned recently (see [She88] and [SI94] for details). This metric has been found to be largely unaffected by code restructuring and appears to be more successful as a measure of the underlying complexity of the algorithm. A routine with a high cyclomatic complexity value is thus generally considered to be in need of modularization. Myers [Mye77] suggests the use of a complexity interval whose lower bound is the cyclomatic complexity and whose upper bound is one more than the total number of conditions.

In addition to these three metrics we also consider the number of executable statements and the number of explicit `GOTO` statements in the routine.

All the software metric values stated in this paper were generated using QAFortran version 6.0 [Pro92].

4 Fortran 66 and Fortran 77

For each of the 46 routines listed on the BLAS reference card [Uni92], Table 1 shows the number of executable lines of code along with the values of the three metrics defined above. Although containing more executable statements than any of the other routines, the `*ROTMG` family of routines stands out as far as both knot and path counts are concerned. The high knot count of 104 in a routine containing just 131 executable statements suggests that the code is likely to be extremely difficult to understand and maintain. This fact is reinforced by Hanson and Krogh [HK87] where, in a paper detailing the translation of the Level BLAS into assembler, they state

> *Here, the subprograms [`SROTMG` and `DROTMG`] are provided in Fortran only, due to the complexity of their specification ...*

and by the control graph of the original code which is shown in Figure 1.

Routine	Exec Stat	Cyclomatic Interval	Knot Count	Path Count
`*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

Table 1: Metric Values for BLAS 1 Routines

The large number of possible paths through the routine, 196608, indicates that it will be difficult to be confident that the routine has been thoroughly tested. In addition the routine contains 34 explicit `GOTO` statements and 27 target labels.

Spag [Pol93], a software tool designed to improve the structure of Fortran 66 code by rearranging (and if necessary duplicating) statements and using Fortran 77 (or Fortran 90), produced some improvement in the metric values when applied to this original code. The knot count was reduced by more than a half and the path count was reduced by a factor of almost a hundred to 2304. Nag_struct [Num92], one of NAG's suite of Fortran 77 software tools, was unable to restructure the code due to multiple-entry loops being detected. However, it should be noted that, even with what appear to be big reductions in the metric values, the code produced by Spag is hardly any more comprehensible that the original.

Restructuring the code from scratch was far more successful. Even using Fortran 66 it was possible to reduce the knot count to 35 and the path count to 4096. This version used 22 explicit `GOTO` statements and contained 12 target labels. The cyclomatic complexity came down from 19 to 13 which is very unusual in any restructuring exercise; this would seem to imply that there were unnecessarily repeated tests taking place in the original code.

Both the commercial restructurers fared much better on this recoded Fortran 66 code, producing Fortran 77 versions with both knot and path counts reduced. The path count reported by QAFortran for the Spag restructuring is optimistically low. This is due to Spag restructuring a sequence of four `WHILE` statements (constructed with pairs of `IF` and `GOTO` statements) into a set of *nested* labelled `IF` statements with `GOTO`s. Since the path count metric used by QAFortran is unaffected by `GOTO` statements this has the effect of reducing the path count from 242 to 36.

The large knot counts associated with the Spag and Nag_struct versions are due mainly to long jumps out of nested block `IF`s. This may be avoided with Fortran 77 by more careful structuring and, although the path count is somewhat higher, the knot count, the number of

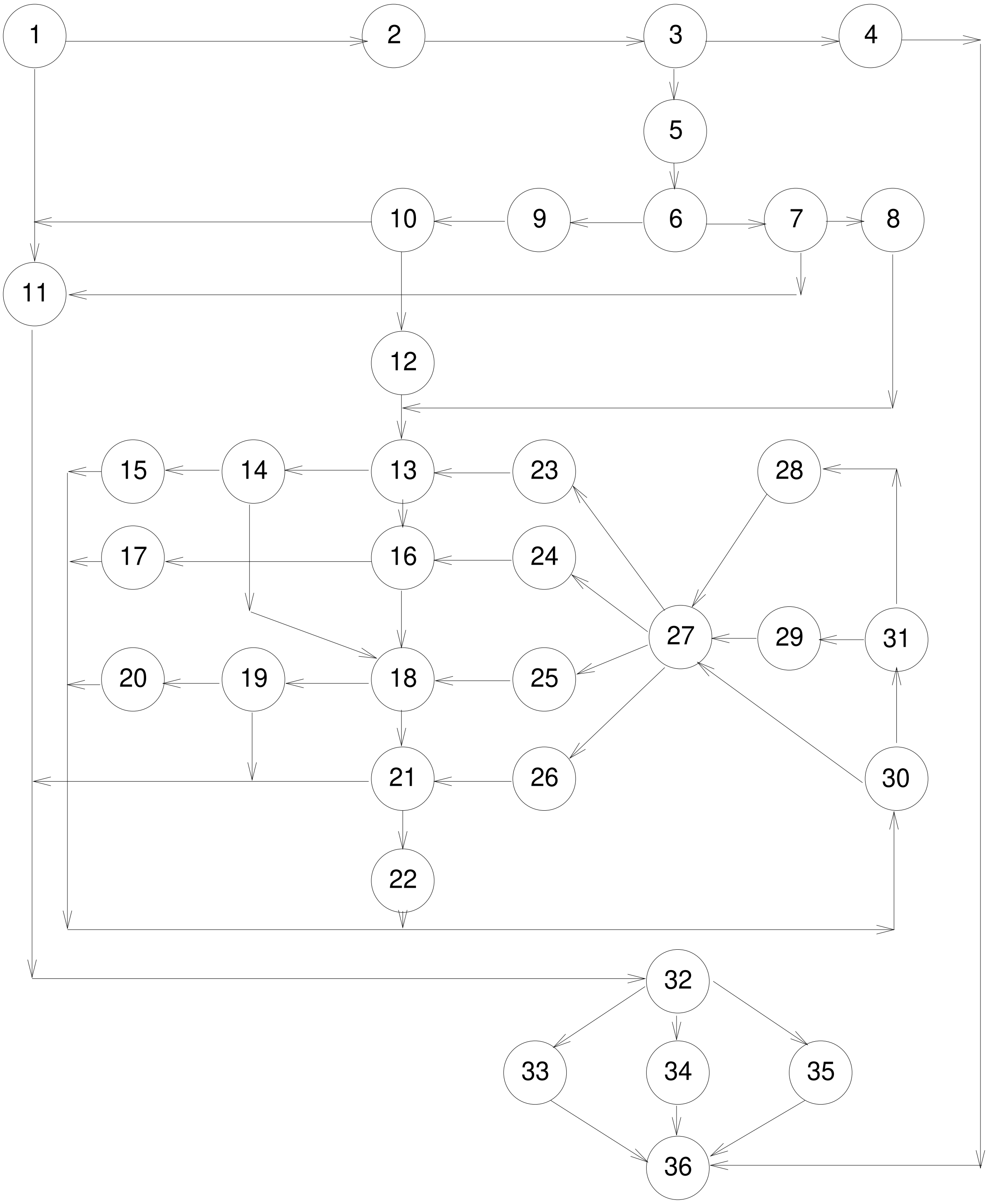

Figure 1: Flowgraph of originally published code

explicit GOTO statements and the number of labels are all reduced to four; all these are required for the construction of the four WHILE loops.

Figure 2 shows the flowgraph for the hand-coded Fortran 77 version of the routine and clearly shows the improvement in structure over the original.

5 Fortran90

Moving to Fortran 90 allowed us to replace the last four labels, knots and GOTO statements by four DO WHILE blocks. Another minor improvement to the code was the combination of a CASE statement and structure constructors to simplify the setting of the output matrix before exit. In addition the new TYPE construction provided us with a cleaner version of the *PARAM argument.

In the original Fortran 66 code this parameter is a real array of length five. The first element is used as a flag to indicate the type of 2×2 Givens Rotation Matrix that is being returned in the other four elements. The rotation matrix is stored by columns. The original possibilities were

	*PARAM				
	1	2	3	4	5
unit matrix	-2	1	0	0	1
rescaled	-1	h_{11}	h_{21}	h_{12}	h_{22}
A6	0	1	h_{21}	h_{12}	1
A7	1	h_{11}	1	-1	h_{22}

where A6 and A7 refer to the equations given in the Appendix to [LHKK79] and only the elements shown as h_{ij} are actually set by the routine. In the case of an error in the input data, the returned matrix is classified as rescaled and all elements are set to zero.

For the new Fortran 90 version of the code we defined the following type

```
TYPE:: SpGivensRotation
  INTEGER :: MatrixType
  REAL(sp) :: Rotation(2,2)
END TYPE SpGivensRotation
```

and the integer parameter values

```
INTEGER, PARAMETER :: clts=1, sltc=0, rescaled=-1, &
                      unit_matrix=-2, error=2
```

which are the only names used to set the MatrixType component of SpGivensRotation. A new value of MatrixType, error, was used to differentiate between a normally rescaled matrix and an error condition. We also set all four values of the rotation matrix whatever type of rotation matrix is generated. A similar definition is made for the double precision case.

Since Fortran 77 users have been provided with generic intrinsic functions, Fortran 90 allows such functionality in user defined routines. Thus another improvement we made was to produce a generic version of the routine, GROTMG. Basically this involves providing an interface to the two routines SROTMG and DROTMG with the system selecting the correct version based on the type of the actual arguments.

The Fortran 90 version was also altered to provide a single point of exit from the routine. This allowed the CASE statement to set all the possible settings of the SpGivensRotation variable. The extra cost here was an IF guard to the block of WHILE statements.

Finally, we use the newly introduced environment enquiry functions to set the value of γ and hence the values used to determine the range of values for which scaling will take place. The value used for γ^2 is

```
MIN(HUGE(0.0_wp), 1.0_wp/TINY(0.0_wp))*0.25
```

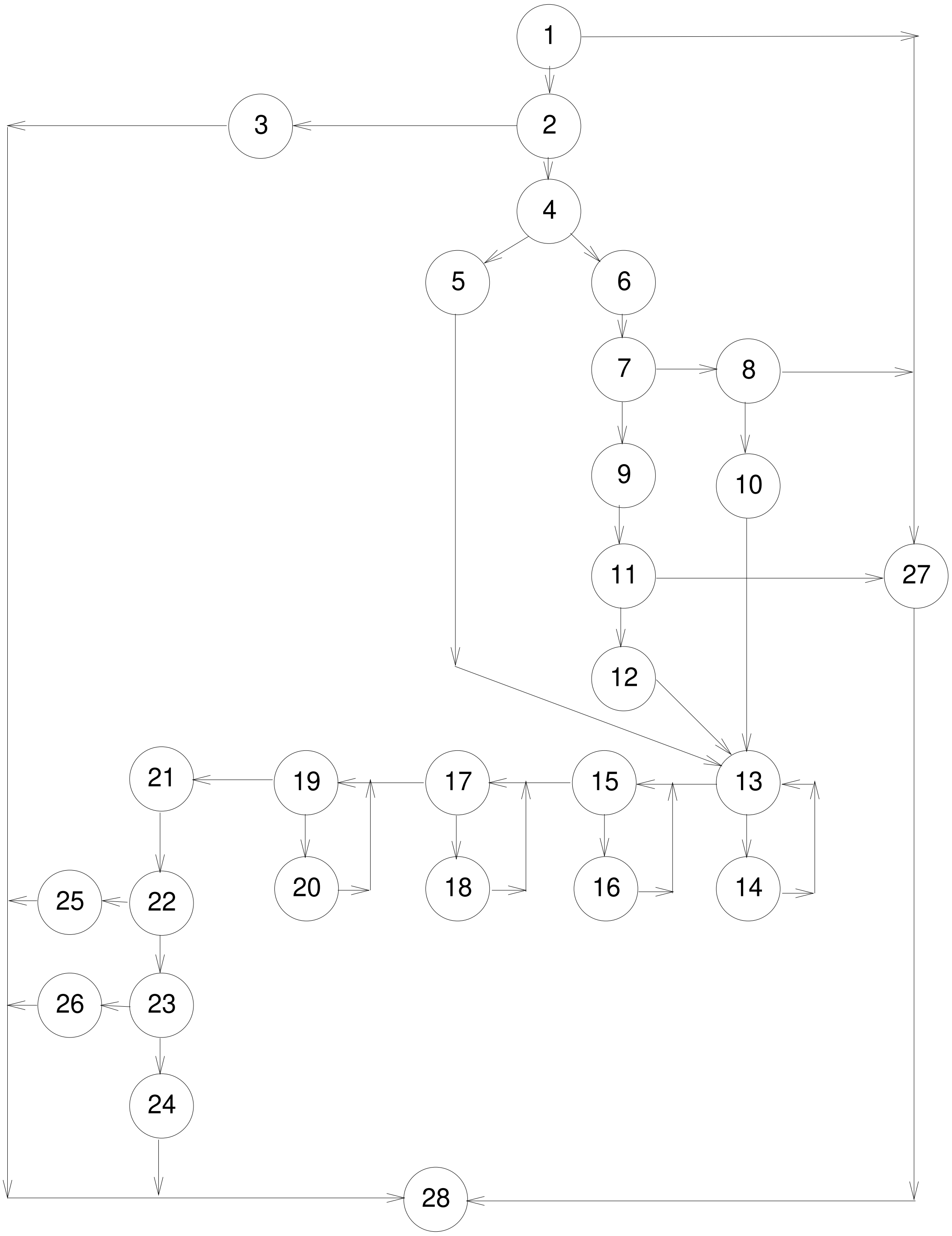

Figure 2: Flowgraph of recoded Fortran 66 code

n	Indy 4400 SC Nag f90 (2.2 260)			Sun Sparc LX Epc f90 (1.1.5.1)			Dec Alpha WS200 Digital f90 (2.0-1)		
	S	M	Mf90	S	M	Mf90	S	M	Mf90
50	0.1	0.1	0.1	3.0	2.7	2.0	0.1	0.2	0.2
100	0.7	0.8	0.9	19.1	17.2	11.5	0.3	0.3	0.4
200	10.0	9.9	10.7	157.4	130.8	75.9	5.8	5.9	6.0

Table 2: Comparison of standard Givens (S), Modified Givens (M) and the Fortran 90 generic version of the Modified Givens (Mf90) to triangularize a $2n \times n$ matrix using double precision. All times are in seconds.

where `wp` is the working precision of the floating point arithmetic. For IEEE standard floating-point arithmetic we obtain an exact representation for γ of 2^{62} (single precision) and 2^{510} (double precision). These values mean that scaling occurs far less frequently than with the original code whilst preserving numerical safety.

A listing of part of the final Fortran 90 implementation is given in the appendix.

6 Testing

When restructuring any code it is imperative that the new version produces the same results as the original, except, of course, where the original version was incorrect. We thus attempted to generate an exhaustive set of test data in order to be as confident as possible that all of the new versions we produced performed exactly as the original code. Note that, with the new settings for γ, the Fortran 90 version will generate results which differ from those produced by the original Fortran 66 code.

This exercise unearthed a minor error in the original code. For the input values x_1, $y_1 \neq 0$, $d_1 = 0$ and $d_2 > 0$ the original code returned the 'solution'

$$H = \begin{bmatrix} 0 & 1 \\ -1 & \frac{x_1}{y_1} \end{bmatrix}$$

This input data effectively generates an input vector of the form $\begin{bmatrix} 0 \\ r \end{bmatrix}$ whose correct transformation matrix is $H = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ and rescaling may take place dependent upon the size of d_2.

Using the profiling tool from the NAG suite of tools [Num92] on the rewritten Fortran 77 code we were able to check for statement coverage using our set of test data. It was found that all statements were executed at least once with the exception of the `GOTO 60` statement immediately before the statement labelled 30. In order to execute this statement the following two conditions need to hold simultaneously

$$|(d_1 \times x_1) \times x_1| > |(d_2 \times y_1) \times y_1| \tag{1}$$

and

$$1 - \left(\frac{(d_2 \times y_1)}{(d_1 \times x_1)}\right) \times \left(\frac{-y_1}{x_1}\right) \leq 0 \tag{2}$$

where the bracketing indicates the order in which the evaluations take place. It is obvious that condition (2) can be true only if $d_2 < 0$, additionally it would appear that we require some peculiar combination of rounding errors to allow both conditions to hold. Using IEEE arithmetic [IEE85] we have been unable to discover any set of input values which causes both conditions (1) and (2) to be true.

Finally, we repeated the timing experiment, performed in [LHKK79], to compare the efficiency of the modified plane rotation, both in its original and Fortran 90 forms, with the standard

Code Version	Language	Exec	Knots	Paths	Cyc. Int.	GOTO's	Labels
1. original	f66	131	104	196608	19:20	34	27
2. Spag on 1.	f77	120	48	2304	17:18	20	11
3. nag_struct on 1.	f77	Not restructured due to multiple-entry loop					
4. hand coded 1.	f66	103	35	4096	13:17	22	12
5. Spag on 4.	f77	105	30	36[1]	13:17	8	6
6. nag_struct on 4.	f77	114	22	241	13:17	8	6
7. hand coded 4.	f77	113	4	336	13:17	4	4
8. Fortran 90	f90	94	0	336	18:23	0	0

Table 3: Summary of code versions and associated metrics

Givens transformation. Both techniques were used to triangularize $2n \times n$ matrices $A = \{a_{ij}\}$ where $a_{ij} = (i+j-1)^{-1}$.

Table 2 gives a sample of the cpu times obtained for a number of compiler/platform combinations. Given the accuracy of the timing routines there is, for this particular problem, little or nothing to choose between the two methods for the majority of the compilers tested. This was especially the case when high optimization levels were selected. The Edinburgh Portable Compilers Fortran 90 compiler on the SUN Sparc LX did still show a gain from using the modified Givens method when full run time checking was switched on. The efficiency gains in this case are comparable to those reported in [LHKK79].

The effect of using the Fortran 90 generic version of the `ROTMG` routines was generally to increase the execution times very marginally.

7 Conclusion

We have shown how the combination of the knot and path count software metrics along with their number of executable statements in a subroutine allowed old Fortran code, that was difficult to understand and test comprehensively, to be identified. Table 3 provides a summary of the various versions of the routine generated along with the associated metric values.

The hand-coded Fortran 66 version (code 4 in Table 3) was better structured than the code produced by applying the Spag restructuring tool to the original code even though Spag's target language was Fortran 77. This is reflected by the lower knot count although it should be noted that the path count is actually larger for code 4.

Applying both restructurers to the hand-crafted version did produce a dramatic reduction in both the path count and the number of explicit `GOTO` statements used. The knot count remained high due mainly to a small number of long jumps out of deeply nested `IF` statements. This suggests that code 4 was a logically clearer implementation of the algorithm than the original code.

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.

The cyclomatic complexity interval values are interesting; it is very rare that this value is reduced by code restructuring. Indeed Shepperd & Ince [SI94] state that cyclomatic complexity is insensitive to the structure of the software. This implies that some of the tests in the original code are either repeated or unnecessary. The higher interval associated with the Fortran 90 code includes the extra test needed to set the value of γ and a small number of repeated tests (within the `CASE` statement) required to generate a consistent return strategy.

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 led to a significant reduction in the metric values although the resultant code was still as impenetrable.

[1] Optimistically low – see section 4 for details

An analysis of the knot and path counts for the 96 Level 2 and Level 3 BLAS ([DDHH88] and [DDDH90]) 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.

8 Acknowledgements

Thanks to Richard Hanson who kindly read a draft of this paper in super quick time and made some very useful comments. In particular, the use of the machine enquiry functions to set `GAMSQ` was his idea.

References

[ANS66] ANSI. *Programming Language Fortran X3.9-1966.* American National Standards Institute, New York, 1966.

[DDDH90] 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.

[DDHH88] 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.

[HK87] R. J. Hanson and F. T. Krogh. Translation of Algorithm 539: PC-BLAS basic linear algebra subprograms for Fortran usage with the INTEL8087 80287 numeric data processor. *ACM Transactions on Mathematical Software*, 13(3):311–317, September 1987.

[Hop96] T.R. Hopkins. Restructuring software: A case study. *Software–Practice and Experience*, 26(8):967–982, August 1996.

[IEE85] IEEE. *IEEE standard for binary floating-point arithmetic.* Institute of Electrical and Electronic Engineers, New York, ANSI/IEEE standard 754-1985 edition, 1985.

[LH74] C. L. Lawson and R. J. Hanson. *Solving least squares problems.* Series in automatic computation. Prentice-Hall, Englewood Cliffs, N.J., 1974.

[LHKK79] 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.

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[Mye77] G. J. Myers. An extension to the cyclomatic measure of program complexity. *Sigplan Notices*, 12(10):61–64, 1977.

[Nej88] B. A. Nejmeh. NPATH: A measure of execution path complexity and its applications. *Commun. ACM*, 31(2):188–200, 1988.

[Num92] Numerical Algorithms Group Ltd., Oxford, UK. *NAGWare f77 Tools*, second edition, September 1992.

[Pol93] Polyhedron Software, Oxford, UK. *plusFORT*, Revision B edition, 1993.

[Pro92] Programming Research Ltd, Hersham, Surrey. *QA Fortran 6.0*, 1992.

[She88] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3:30–36, March 1988.

[SI94] M. Shepperd and D. C. Ince. A critique of three metrics. *J. Systems Software*, 26:197–210, 1994.

[Uni92] University of Tennessee, Tennessee, US. *Basic Linear Algebra Subroutines: A Quick Reference Guide*, June 1992.

[WHH79] 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.

A Fortran 90 Version of the Restructured Code

```
    MODULE modified_givens_rotation
! .. Generic Interface Blocks ..
      INTERFACE grotmg
        MODULE PROCEDURE srotmg
        MODULE PROCEDURE drotmg
      END INTERFACE
! ..
! .. Intrinsic Functions ..
      INTRINSIC kind
! ..
! .. Parameters ..
      INTEGER, PARAMETER :: clts = 1
      INTEGER, PARAMETER :: dp = kind(1.0D0)
      INTEGER, PARAMETER :: error = 2, rescaled = -1, sltc = 0
      INTEGER, PARAMETER :: sp = kind(1.0E0)
      INTEGER, PARAMETER :: unit_matrix = -2
! ..
! .. Derived Type Declarations ..
      TYPE :: spgivensrotation
        INTEGER :: matrixtype
        REAL (sp) :: rotation(2,2)
      END TYPE spgivensrotation
      TYPE :: dpgivensrotation
        INTEGER :: matrixtype
        REAL (dp) :: rotation(2,2)
      END TYPE dpgivensrotation
! ..
    CONTAINS

      SUBROUTINE srotmg(sd1,sd2,sx1,sy1,sparam)
! .. Structure Arguments ..
        TYPE (spgivensrotation), INTENT (OUT) :: sparam
! ..
! .. Scalar Arguments ..
        REAL (sp), INTENT (INOUT) :: sd1, sd2, sx1
        REAL (sp), INTENT (IN) :: sy1
! ..
! .. Local Scalars ..
        REAL (sp), SAVE :: gamsq, rgamsq
        REAL (sp) :: sh11, sh12, sh21, sh22, sp1, sp2, sq1, sq2, stemp, su
        INTEGER :: sflag
! ..
```

```
! .. Intrinsic Functions ..
        INTRINSIC abs, huge, min, reshape, sqrt, tiny
! ..
! .. Parameters ..
        REAL (sp), PARAMETER :: one = 1.0_sp
        REAL (sp), PARAMETER :: quarter = 0.25_sp
        REAL (sp), PARAMETER :: zero = 0.0_sp
! ..
! .. Dependents ..
        REAL (sp), SAVE :: gam = zero
! ..
! Set the value of gam, gamsq, rgamsq on first call to the
! routine. These values are dependent on the underlying
! floating-point arithmetic and should only be computed
! once.
        IF (gam==zero) THEN
          gamsq = min(huge(one),one/tiny(one))*quarter
          gam = sqrt(gamsq)
          rgamsq = one/gamsq
        END IF
! NOTE: sd2 is allowed to be negative to allow for row removal
!       in least squares problems
! Test for illegal input sd1<0 -- return H as zero matrix with sflag=-1
! Set matrix to zero for error exit

        IF (sd1<zero) THEN
          sd1 = zero
          sd2 = zero
          sx1 = zero
          sflag = error

! Special cases
! Input vector is of the required form (c,0) where c can be zero
! Set H = I
        ELSE IF (sd2==zero .OR. sy1==zero) THEN
          sflag = unit_matrix

! Input vector is of the form (0,c) -- just need to reverse elements
! May need to scale d2 dependent values

        ELSE IF ((sd1==zero .OR. sx1==zero).AND. sd2>zero) THEN
          sflag = clts
          sh12 = one
          sh21 = -one
          sh11 = zero
          sh22 = zero
! set new x value to old y value
          sx1 = sy1
! swap d values
          su = sd1
          sd1 = sd2
          sd2 = su

! Compute required bits and pieces
        ELSE
          sp2 = sd2*sy1
          sp1 = sd1*sx1
          sq2 = sp2*sy1
```

```
          sq1 = sp1*sx1

! |c| > |s|; type zero matrix (diagonal elements one)
          IF (abs(sq1)>abs(sq2)) THEN
            sflag = sltc
            sh11 = one
            sh22 = one
            sh21 = -sy1/sx1
            sh12 = sp2/sp1
            su = one - sh12*sh21

! If su has underflowed -- sparam has already been set -- exit
            IF (su<=zero) THEN
              sd1 = zero
              sd2 = zero
              sx1 = zero
              sflag = error
            ELSE
              sd1 = sd1/su
              sd2 = sd2/su
              sx1 = sx1*su
            END IF
          ELSE

! |s| >= |c|; type 1 matrix (antidiagonal case)

            IF (sq2<zero) THEN
              sd1 = zero
              sd2 = zero
              sx1 = zero
              sflag = error
            ELSE
              sflag = clts
              sh21 = -one
              sh12 = one
              sh11 = sp1/sp2
              sh22 = sx1/sy1

! No possibility of underflow since sd2>0 if here
              su = one + sh11*sh22
              stemp = sd1/su
              sd1 = sd2/su
              sd2 = stemp
              sx1 = sy1*su
            END IF
          END IF
        END IF

! Scaling may be necessary -- matrices now become type -1
! Scale -- sd1

        IF (sflag/=error .AND. sflag/=unit_matrix) THEN
          DO WHILE (sd1<=rgamsq .AND. sd1/=zero)
            sflag = rescaled
            sd1 = (sd1*gam)*gam
            sx1 = sx1/gam
            sh11 = sh11/gam
            sh12 = sh12/gam
```

```
          END DO

          DO WHILE (sd1>gamsq)
            sflag = rescaled
            sd1 = (sd1/gam)/gam
            sx1 = sx1*gam
            sh11 = sh11*gam
            sh12 = sh12*gam
          END DO

! Scale -- sd2
          DO WHILE (abs(sd2)<=rgamsq .AND. sd2/=zero)
            sflag = rescaled
            sd2 = (sd2*gam)*gam
            sh21 = sh21/gam
            sh22 = sh22/gam
          END DO

          DO WHILE (abs(sd2)>gamsq)
            sflag = -one
            sd2 = (sd2/gam)/gam
            sh21 = sh21*gam
            sh22 = sh22*gam
          END DO
        END IF

! set sparam array and exit

        SELECT CASE (sflag)

        CASE (clts)
          sparam = spgivensrotation(clts,reshape((/sh11,-one,one,sh22/),(/2,2 &
            /)))
        CASE (sltc)
          sparam = spgivensrotation(sltc,reshape((/one,sh21,sh12,one/),(/2,2/) &
            ))
        CASE (rescaled)
          sparam = spgivensrotation(rescaled,reshape((/sh11,sh21,sh12,sh22/), &
            (/2,2/)))
        CASE (unit_matrix)
          sparam = spgivensrotation(unit_matrix,reshape((/one,zero,one,zero/), &
            (/2,2/)))
        CASE (error)
          sparam = spgivensrotation(error,reshape((/zero,zero,zero,zero/),(/2, &
            2/)))

        END SELECT
      END SUBROUTINE srotmg
!
! Double precision subroutine code omitted
!
    END MODULE modified_givens_rotation
```