Algorithm 830: Another Visit with Standard and Modified Givens Transformations and a Remark on Algorithm 539

RICHARD J. HANSON
Center for High Performance Software Research, Rice University
and
TIM HOPKINS
University of Kent, UK

First we report on a correction and improvement to the Level 1 Blas routine `srotmg` for computing the Modified Givens Transformation (MG). We then, in the light of the performance of the code on modern compiler/hardware combinations, reconsider the strategy of supplying separate routines to compute and apply the transformation. Finally, we show that the apparent savings in multiplies obtained by using MG rather than the Standard Givens Transformation (SG) do not always translate into reductions in execution time.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Numerical Algorithms*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra; G.4 [**Mathematical Software**]: —*Certification and testing; Efficiency; Algorithm design and analysis*

General Terms: Algorithms; Performance

Additional Key Words and Phrases: BLAS, Givens rotation, linear algebra

1. INTRODUCTION

The Standard Givens orthogonal transformation is a low-level operation of numerical linear algebra. This plane rotation transformation eliminates z_1 in the identity

$$GW = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} w_1 \dots w_n \\ z_1 \dots z_n \end{bmatrix}$$

Authors' addresses: R. J. Hanson, Center for High Performance Software Research, Rice University, 6100 Main Street, MS41, Houston, TX 77005-1892; email: koolhans@rice.edu; Tim Hopkins, Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK; email: t.r.hopkins@kent.ac.uk.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

Thus $c = w_1/r$, $s = z_1/r$, $r = \pm\left(w_1^2 + z_1^2\right)^{\frac{1}{2}}$ for $r \neq 0$. The orthogonality of G follows from $c^2 + s^2 = 1$. Applying the transformation to the columns requires $4(N-1)$ multiplies and $2(N-1)$ adds. Constructing the transformation requires at least four multiplies, one add, one divide, and one square-root operation. This is well-known material. Gentleman [Gentleman 1973] and Hammarling [Hammarling 1974], showed how the computation could be reorganized to reduce the operation count for applying the transformation to the remaining columns. The central idea is to regard the matrix in factored form

$$W = \begin{bmatrix} d_1^{\frac{1}{2}} & 0 \\ 0 & d_2^{\frac{1}{2}} \end{bmatrix} \begin{bmatrix} x_1 \ldots x_n \\ y_1 \ldots y_n \end{bmatrix} \equiv D^{\frac{1}{2}} X$$

The product is re-factored

$$GW = GD^{\frac{1}{2}}X = \tilde{D}^{\frac{1}{2}}HX \equiv \begin{bmatrix} \tilde{d}_1^{\frac{1}{2}} & 0 \\ 0 & \tilde{d}_2^{\frac{1}{2}} \end{bmatrix} HX$$

The matrix H is selected so that two of its four entries are units. This implies that $2(N-1)$ multiplies and $2(N-1)$ adds are needed to compute HX. This *Modified Givens* yields a saving of $2(N-1)$ multiplies compared with the *Standard Givens*. For MG, the additional cost of updating the diagonal terms and computing the non-unit entries of H was not regarded as significant in [Lawson et al. 1979b]. The decision to include the MG as a module was based on timings that were compelling. Our testing indicates that on modern computers, the *netlib* SG pair of routines (`_rotg` and `_rot`) are typically more efficient that the MG pair (`_rotmg` and `_rotm`). This seems to indicate that MG is relatively inefficient and, therefore, has little to recommend it. We believe that this is due to the coding itself and a critical mathematical detail. We implemented a Fortran 90 version of MG given in [Golub and Van Loan 1996] that is typically more efficient than our Fortran 90 version of SG. This is the result one expects since MG requires fewer operations per elimination step. We organized both routines so that a single subprogram call constructs and then applies the transformation. This saves the overhead of an additional call. The resulting codes are almost always more efficient that the corresponding *netlib* versions. Our version of SG includes an additional feature of a hyperbolic transformation, useful for dropping data, provided in both the old and new version of MG. Now SG and MG have the same functionality. Several ideas are presented here that enhance both methods. Some of these principles may be new, but that is hard to determine from the apocryphal literature. In the implementations of the most frequently taken paths through the sections of code that compute the basic plane rotation parameters MG trades five floating-point multiply/divide operations against two floating-point adds and a square root used by SG. Experiments on a number of currently available compiler/platform combinations failed to provide a definitive answer as to which of the two methods performs the better. Therefore, as part of the software accompanying this paper, a benchmark program has been provided to allow a software developer to make an informed decision on whether MG will provide better performance than the simpler SG based on both the run-time environment and typical problem size.

Recently, Anda and Park [Anda and Park 1994] point out the apparent reticence by software developers to use the modified plane rotation and they suggest a revised algorithm incorporating their idea of dynamic scaling to prevent problems with over- and underflows. Their algorithm for calculating the plane rotation requires an additional divide step compared to our implementation.

Their application step requires the same amount of work as our implementation but assumes that a form of "chaining" allows a completed sub-expression to be used again as a multiplier. Although this is a reasonable assumption for vector machines, for modern scalar machines it implies that an intermediate result must complete before the transformation is finished.

2. CODE UPDATE

The routines `srotmg` and `drotmg` given in [Lawson et al. 1979a] generate incorrect results when the input vector $\begin{bmatrix} w_1 \\ z_1 \end{bmatrix}$ is defined as $\begin{bmatrix} d_1^{\frac{1}{2}} & \\ & d_2^{\frac{1}{2}} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$ with $d_1 = 0$. This may be a contorted way of generating an input vector of the form $\begin{bmatrix} 0 \\ r \end{bmatrix}$ whose correct transformation matrix, H, is $\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$. The published code generates the result $H = \begin{bmatrix} 0 & 1 \\ -1 & \frac{x_1}{y_1} \end{bmatrix}$ by setting $c = 0$ and $s = 1$ in the equations (A7a) to (A7d) below. Unlike the case $x_1 = 0$ and $d_1 \neq 0$, the general formula cannot be used in this case since $\tilde{d}_1^{\frac{1}{2}}$ cannot be factored out of the equation (A5) in [Lawson et al. 1979b].

We have, thus, replaced the line

```
IF(.NOT. SD1 .LT. ZERO) GO TO 10
```

by

```
IF((SD1 .GT. ZERO) .AND. (SD2 .NE. ZERO)) GO TO 10
```

and introduced an error flag, `SFLAG = 2`, directly following the label 60. Either of the d_i, $i = 1, 2$ having value zero corresponds to an equality constraint in a least squares problem. A large weight on a row can be used to achieve an equality constraint. This corresponds to a small positive value of the reciprocal weight, for example, d_1. A value of zero for the reciprocal is treated as an error. A further discussion of this class of weighting methods may be found in Chapter 22 of [Lawson and Hanson 1995].

The value of `GAM`, which determines when scaling of the d_i values takes place, was chosen at a time when Fortran lacked any standard means of obtaining information about the underlying floating-point arithmetic. The value used in both the original single and double precision codes, 4096.0, was chosen deliberately small to ensure that the routine would operate correctly on all known systems. With access to new intrinsic inquiry functions in Fortran 90 it is now possible to set this value in a hardware dependent way

```
        GAMSQ = MIN(HUGE(ONE), ONE/TINY(ONE))*QUARTER
        GAM = SQRT(GAMSQ)
```

```
      RGAMSQ = ONE/GAMSQ
```

where `ONE` is set to either `1.0E0` or `1.0D0` and `QUARTER` to either `0.25E0` or `0.25D0` for single and double precision respectively.

For IEEE arithmetic this sets `GAM` to 4.62E+18 (single precision) and 3.35D+153 (double precision); thus reducing dramatically the number of times scaling may take place. In addition, for IEEE arithmetic, the values assigned to `GAM` and `RGAMSQ` are exact.

Since the assigned goto statement has been formally removed from the Fortran 95 language [ISO/IEC 1997], occurrences of the statement have been replaced by computed gotos in both the single and double precision routines. In addition, the use of the specific intrinsic function `DABS` has been replaced by the generic `ABS` in the double precision routine.

The original coding of these routines was extremely contorted and a more readable, corrected version, in Fortran 90, may be found in [Hopkins 1997].

3. DEVELOPMENT

3.1 Modified Givens

There is a saving in constructing MG by storing and updating the *reciprocal squares*, d_j^{-1}, $j = 1, 2$. In [Lawson et al. 1979b] the d_j are updated. The reformulation found in [Golub and Van Loan 1996] p. 221, eliminates two divides for each transformation.

Storage locations for these reciprocal squares are passed to the subprogram that constructs and applies the transformation. In this development we note that the mathematical quantities d_j^{-1}, $j = 1, 2$ each occupy a memory location. Thus we may associate a sign with these values. To designate that a datum is dropped, the sign of the location for d_1^{-1} is set negative. The sign of d_2^{-1} is always positive.

Note that the use of reciprocal squares is often what a user desires when calling MG routines to solve a linear least squares regression problem. If the quantity to be minimized is $\rho(x) = \sum_{i=1}^{m} |r_i/\sigma_i|^2$, $r_i = a_i^T x - b_i$, then the initialization for the reciprocal squares is $d_i^{-1} = \sigma_i^2$, $i = 1, \ldots, m$. Thus there is no need to directly apply scale factors σ_i^{-1} to the row vectors $\left[a_i^T : b_i\right]$. This scaling step occurs implicitly during the QR factorization leading to the weighted least squares estimate $x = \hat{x}$. If the scaling factors are the same for all the rows, or if the weighting is already applied to the row vectors, then initially set $d_i^{-1} = 1$, $i = 1, \ldots, m$.

Using the numbering given in [Lawson et al. 1979b], Equations A6a–A7g are recast in terms of the reciprocal squares.

The absolute value used below on the quantity d_1 defines the construction of the transformation for both cases, either when data is being added or dropped. We use the sub-expressions $p_1 = d_2^{-1} x_1$, $p_2 = d_1^{-1} y_1$.

```
If $p_1 x_1 \geq |p_2 y_2|$, then
    If $p_2 = 0$ then $H = I$, the identity matrix
    else
```

$$h_{11} = 1, \quad h_{21} = -y_1/x_1 \qquad \text{(A6a)}$$

$$h_{12} = -p_2/p_1, \quad h_{22} = 1 \qquad \text{(A6b)}$$

$$u = 1 - h_{21}h_{12} \tag{A6c}$$
$$d_1^{-1} \leftarrow d_1^{-1}u \tag{A6d}$$
$$d_2^{-1} \leftarrow d_2^{-1}u \tag{A6e}$$
$$x_1 \leftarrow x_1 u \tag{A6f}$$

If $p_1x_1 < |p_2y_2|$, then

$$h_{11} = p_1/p_2, \quad h_{21} = -1 \tag{A7a}$$
$$h_{12} = 1, \quad h_{22} = x_1/y_1 \tag{A7b}$$
$$u = 1 + h_{11}h_{22} \tag{A7c}$$
$$v = d_1^{-1}u \tag{A7d}$$
$$d_1^{-1} \leftarrow d_2^{-1}u \tag{A7e}$$
$$d_2^{-1} \leftarrow v \tag{A7f}$$
$$x_1 \leftarrow y_1 u \tag{A7g}$$

The relations A6d–A6e and A7d–A7f show that two of the floating point divides found in [Lawson et al. 1979b] can be replaced by two multiplies. this is important because divides are relatively slow on modern computers. The reciprocal squares increase by as much as a factor of two with each transformation. They can also decrease, when data is dropped, but this is unlikely to cause problems. A re-scaling of a row of the transformation may be required. Re-scaling of either row occurs if

$$\left|d_1^{-1}\right| + d_2^{-1} > \mathtt{GAM}$$

following the update steps A6d–A6e and A7d–A7f. This allows values of x_1 and y_1 to be as large as `SQRT(HUGE(ONE))` without overflow occurring.

Storing the MG transformations requires an integer flag, and two or four floating point values per elimination step. An integer flag and four floating point values are required only when re-scaling occurs, and this is now rare indeed, thanks to the large machine-dependent scale factor `GAMSQ`. The SG has an edge here, since the transformations can be saved in the same store as the input data, using the idea of Stewart documented in [Lawson et al. 1979b].

3.2 Standard Givens

The MG and SG are typically used in *row accumulation mode*. For some applications it is efficient to remove the row, once accumulated. This saves the expense of refactoring the least-squares matrix with the row removed. This is not a numerically stable process, so refactoring may be necessary under some circumstances. When using MG this mathematically requires $d_2 < 0$, which is flagged in our code by $d_1 < 0$.

For SG one can use *hyperbolic rotations* (see [Björck 1996], page 144)

$$H = \begin{bmatrix} ch & -sh \\ -sh & ch \end{bmatrix}, \qquad (ch)^2 - (sh)^2 = 1$$

The H matrix is constructed to zero the second component in a vector $(a, b)^T$,

$|a| > |b|$. This leads to

$$s = \frac{b}{a}, \qquad c = \sqrt{(1+s)(1-s)}, \qquad \sigma = ac$$

With $ch = c^{-1}$ and $sh = s^{-1}$, we see that

$$H \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \sigma \\ 0 \end{bmatrix},$$

and, incidentally, that $c^2 + s^2 = 1$. We may form the product

$$H \begin{bmatrix} u \\ v \end{bmatrix} \equiv \begin{bmatrix} x \\ y \end{bmatrix},$$

in a relationally stable way, by using the 2-step evaluation

$$\begin{aligned} x &= (ch)(u - sv) \\ y &= -sx + cv \end{aligned}$$

This is based on the intermediate Givens transformation

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} = \begin{bmatrix} u \\ y \end{bmatrix}$$

Either the error condition $|a| \le |b|$ or the computed value $(1+s)(1-s) \le 0$, is flagged in our code by returning $c = s = 0$.

The form of this row removal method goes back to Chambers [Chambers 1971] and was further refined by Stewart and Stewart [Stewart and Stewart 1998].

3.3 Applying the Rotations

Advocates of either MG or SG must observe that the primary efficiency issue is *access to the data* during the formation of products of the rotations. The fact that MG saves arithmetic may be important. This depends on the problem size and machine characteristics. But the choice is not as crucial as organizing the data access with a unit stride between matrix rows or columns wherein the rotation will be applied. A developer would choose MG or SG, depending on the problem size, and employ unit strides in the application phase. This will involve design of the matrix storage for some problems.

We have found that *loop unrolling* has little or no useful effect when high levels of optimization are used. Thus, as with the Level-1 BLAS routines `srotm` and `srot` no loop unrolling has been implemented.

Finally, we would mention that the MG code would not be at all suited to providing readable code in the case of a series of blocked rotations which are applied to a series of column vectors. The situation is improved by using SG by at least making the code easier to follow.

4. IMPLEMENTATION

We have implemented the new routines as a Fortran 90 module, *Givens_Rotations*. This provides two ways in which to call both the standard and modified Givens transformations; in each case the pair of Blas Level 1 routines are replaced by a single call that computes the rotation and applies it to a $2 \times N$ array of data.

First there are single and double precision versions (p`_rot` and p`_rotm` where p is either `s` or `d`) and, second, we have included a simple generic interface to both algorithms that allows the names `rot` and `rotm` to be used for either precision provided that the floating point parameters are all of the same type. The calling sequences

```
CALL rot(w1, z1, k, x, incx, y, incy, c, s)

CALL rotm(rd1, rd2, x1, x2, k, x, incx, y, incy, param)
```

are basically formed by combining the calling sequences of the original `rot/rotg` and `rotm/rotmg` pairs of routines. We note here that the common use, in Fortran 77, of an array element as an actual argument for a vector dummy argument is illegal when calling the generic interface.

We experimented with a pair of generic interfaces to these Fortran 77 compatible routines which required shorter calling sequences. By using assumed shape arrays it is possible to dispense with the `INCX` and `INCY` parameters since their effect may be obtained using array slices with a non-unit stride. Furthermore, the length of the data arrays may be ascertained from the array arguments using the `SIZE` intrinsic function. For the `rotm` routine we could also define a derived type to replace the `PARAM` array; this allows the use of an integer return code signifying the type of the transformation matrix rather than using the first element of the floating-point `PARAM` array. However, any gains obtained from having shorter, and cleaner, argument lists were far outweighed by increases in execution time. This was especially true when `INCX` and `INCY` did not have the value one. In this case we passed the actual arguments as array slices with a non-unit stride, and this, along with the additional routine call overhead, appeared to exact a high penalty with execution times up to five times slower than using the Fortran 77 compatible routines with non-unit increments. This additional overhead made these routines up to twenty times slower than the optimal.

On studying the assembler produced by the Sun f90 compiler for the `rotm` routines, we discovered that we could improve the overall execution speed by using the array elements of the `PARAM` array argument rather than locally defined simple variables in the transformation loops. The released software uses these array elements although this may not be optimal across all platform/compiler combinations.

There are a number of differences between the features available from the original BLAS routines and those presented here

(1) the standard transformation routines allow for row removal,

(2) the rotation is no longer saved in the standard transformation routines. Details of how this may be calculated can be found in [Lawson et al. 1979b].

(3) negative values for `INCX` and `INCY` are no longer catered for,

(4) the new routines require and return the reciprocal squares, d_i^{-1}, $i = 1, 2$,

(5) error returns from the standard transformation routines are signalled by setting $c = s = 0$,

(6) scaling only occurs in the new modified transformation routines when the reciprocal squares are extremely large,

Test routines, providing 100% statement coverage, were constructed for all the new user callable routines. These executed all but two of the basic blocks of code; these missed blocks test for an error condition that we have been unable to generate using IEEE arithmetic but may occur with other, less stringent, arithmetics.

5. BENCHMARKING AN IMPLEMENTATION

Single precision codes `s_rotm` and `s_rot` were written for the above MG algorithm and SG implementing hyperbolic transformations. Therefore either code can be used for adding and then dropping data from least squares problems. Note that the Level-1 BLAS codes `srotmg/srotm` allowed this add/drop step, but the pair `srotg/srot` can only *add* data. Including this extra capability levels the choice for a developer, who often wants to use Givens transformations for row operations on a dense matrix for a least squares problem. A natural choice is to store the matrix elements a_{ij} in Fortran assumed-size array locations `A(I,J)`. As our benchmarks show, this is *not* the optimal organization in terms of efficiency. The matrix elements are best organized in transposed form, with unit strides when applying the transformations. Other choices, including the choice of MG vs. SG, are secondary in importance. At first blush it appears obvious, therefore, to use MG instead of SG. The argument goes like this: *Eventually MG has fewer operations when applying the transformation. So when the problem is large enough, this choice will be more efficient that SG.*

Our benchmark program triangularizes $2n \times n$ random matrices using Givens transformations. However, extensive experimentation with this benchmark code failed to confirm our conjecture above regarding the superiority of the MG implementation. While different combinations of compiler options may greatly affect the run-time of the resultant executable code, it is not possible, due to the large number of options available, to conduct exhaustive trials. Combined with the advances in the provision of various levels of cache memory and the way compilers utilize such memory, it is impossible to provide a definitive answer as to which routine will be the most efficient for any particular hardware/compiler/compiler options combination without running a benchmarking program.

What our experiments did show was that, using unit strides, the original Blas Level 1 MG pair never produced the best performance. Each of the other routines, the proposed MG/SG and the original Blas Level 1 SG pair, performed best for at least one experiment. On large problems ($n > 100$) the benefit obtained from using the most efficient over the least efficient of the other three routines could be as high as 40%.

When it is critical to obtain maximum performance from an application which makes intensive use of Givens transformations, we thus propose that our benchmark program be run for typical size problems to determine the optimal routine for the hardware/compiler combination being used.

6. ACKNOWLEDGEMENTS

Richard Hanson was helped during the course of several discussions with Fred T. Krogh. We would also like to thank an anonymous referee for making us aware of Chambers' paper.

REFERENCES

Anda, A. A. and Park, H. 1994. Fast plane rotations with dynamic scaling. *SIAM J. Matrix Anal. Appl. 15,* 1 (Jan.), 162–174.

Björck, A. 1996. *Numerical Methods for Least-Squares Problems.* SIAM Publications, Philadelphia.

Chambers, J. M. 1971. Regression updating. *J. Amer. Statist. Assoc. 66*, 744–748.

Gentleman, W. M. 1973. Least squares computations by Givens transformations without square roots. *J. Inst. Maths Applics 12*, 329–336.

Golub, G. and Van Loan, C. 1996. *Matrix Computations*, 3rd ed. Johns Hopkins University Press, Baltimore.

Hammarling, S. 1974. A note on modifications to the Givens plane rotation. *J. Inst. Maths Applics 13*, 215–218.

Hopkins, T. 1997. Restructuring the BLAS Level-1 routine for computing the modified Givens transformation. *ACM SIGNUM 32,* 4 (Oct.), 2–14.

ISO/IEC. 1997. *Information Technology – Programming Languages – Fortran - Part 1: Base Language (ISO/IEC 1539-1:1997).* ISO/IEC Copyright Office, Geneva.

Lawson, C. and Hanson, R. 1995. *Solving Least Squares Problems*, Classics ed. SIAM, Philadelphia.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. 1979a. Algorithm 539: Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw. 5,* 3 (Sept.), 324–325.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. 1979b. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw. 5,* 3 (Sept.), 308–323.

Stewart, M. and Stewart, G. W. 1998. On hyperbolic triangularization: stability and pivoting. *SIAM J. Matrix Anal. Appl. 19,* 4, 847–860.

Received ??? ???; revised ??? ???; accepted October 2003