Appendix 2

Appendix 1

9 References

1. J. J. Dongarra, F. G. Gustavson, and A. Karp (January 1984), "Implementing Linear Algebra Algorithms for Dense Matrices on a Pipeline Machine," Siam Review, Vol 26, No. 1, pp.91-113.

2. A. A. Dubrulle, H. G. Kolsky, and R. G. Scarborough (1985), "How to Write Good Vectorizable FORTRAN," Technical Report G320-3396, IBM Scientific Center, Palo Alto, CA.

3. A. A. Dubrulle (June 1988), "A Version of Eispack for the IBM 3090VF," Technical Report G320-3510, IBM Scientific Center, Palo Alto, CA.

4. A. Padegs, B. B. Moore, R. M. Smith and W. Buchholz (1988), "The IBM System/370 Vector Architecture: Design Considerations," IEEE Trans. Comp., Vol. 37, No. 5.

5. S. G. Tucker (1986), "The IBM 3090 System: An Overview," IBM Systems Journal, Vol. 25, No. 1,pp. 4-19.

6. H. H. Wang (March 1986), "Introduction to Vectorizing Techniques on the IBM 3090 Vector Facility," Technical Report G320-3489, IBM Scientific Center, Palo Alto, CA.

7. J. H. Wilkinson and C. Reinsch (1971), "Handbook for Automatic Computation, Vol. II, Linear Algebra," Springer-Verlag, New York.

8 Conclusions

Two new versions of the EISPACK subroutine package have been tailored for the 3090-VF. One was produced by Augustin Dubrulle of IBM Palo Alto Scientific Center in double precision and intended to be most efficient for extremely large problems. The other was produced by Pleasant Valley Software in single precision with the purpose of attaining efficiency for smaller problems. The encoding of the PVS version has revealed several important transformations that can be important in making similar programs more efficient for the 3090-VF.

The best examples of the latter technique are `elmbak` and `combak`. With only very minor modifications and the application of a few vector directives, the code ran approximately 6 and 7 (respectively) times faster for $N = 200$. Notice that the important subroutines `hqr` and `hqr2` performed 70% and 90%, respectively, better at dimension 100 with the PVS versions than the with NATS. Alternatively, PVS `orthes` performed only 20% better at dimension 100 and `ortran` performed no better than its NATS version. Since improvements for small dimension matrices were realized with PASC double precision versions of `eltran`, `htribk`, and `ortran` but not with the PVS versions, we are led to believe that the significant stride-one reorientation that was employed for the PASC codes are the correct approach to employ even for single precision.

7 Correctness Testing

Both PASC and PVS codes were subjected to tests to ensure their correctness. These tests were created by modifying the thirteen NATS-supplied drivers. These programs originally read a large set of test matrices, called a sequence of EISPACK modules to produce eigenvalues and/or eigenvectors, compared results across alternative paths, and computed residuals. Each test produces a figure of merit that essentially is the residual norm scaled by the the product of ten, the machine precision, the matrix dimension, and the matrix norm. These tests have been employed since 1972 to determine the correctness of versions of EISPACK on various machines.

The modifications employed allowed for a set of random perturbations to be applied to each of the input matrices. For certain situations (e. g. symmetric matrices, tridiagonal matrices) the perturbations had to reflect the character of the original matrix. A large and varied set of tests could be performed with this procedure.

Although the codes were targeted for the 3090-VF they were also run (employing simulation) on Sun 3 and Sun 4 workstations and on a VAX 11/780. Testing on the alternative equipment with different floating point precisions and ranges as well as on the 3090-VF was intended to reveal subtle errors. In fact, they did since several of the original PASC codes (since recoded) showed difficulties with overflows that had not previously been uncovered.

The final versions of all PASC and PVS routines perform within the acceptable tolerances on all tests.

than a day was spent per routine in the recoding of the NATS original and in testing. These routines were timed against the PASC routines that had been carefully crafted for the 3090-VF from the original Algol source. The corresponding PASC codes were not examined until after the tests. The seven routines were selected simply to represent the spectrum of EISPACK capabilities.

The testing involved matrices of dimensions 50, 60, 70, 80, 100, 150, 200, and 300. What was seen was that 26 of the PASC routines outperformed the original NATS by at least 10% at dimension 300 (although because of the large standard deviation associated with the timings for `qzhes`, that routine should possibly be excluded). The PASC versions of the important procedures `hqr` and `hqr2` achieved about 130% and 90% speedup over the NATS codes, respectively. However, the timings of the similarly important `ortran` showed no significant differences. Such was also the case for `orthes` where PASC was compared to the recoded NATS. In general, the PASC code was about 40% faster than recoded NATS `qzit` and 20% faster than recoded NATS `tql1`. Ten of the other tests produced timings within 10% of each other at dimension 300 and one, the original NATS `qzval`, actually ran about 10% faster than the PASC code.

6.2 PVS vs. NATS (single precision)

There were two separate runs, one for PVS, one for NATS. The same drivers and timing routines were used for each run. The test matrices had dimensions 10 through 200 in steps of 10, and their elements were uniformly distributed over the interval $[-1 \ldots 1]$. Both versions were compiled by VS FORTRAN Version 2, Release 3. All code was compiled with flags to enable the production of vector code at the highest level of optimization. Each routine was run several times for each value of N to obtain more reliable results. Although the CPU times were obtained during off-peak hours, some routines (e.g. `figi1, figi2, and qzval`) required time on the order of the clock granularity. As a result, the distributions of the times for such routines were characterized by unusually high standard deviations.

A table entry with a "z" appended to the name of a subroutine (e.g. `qzitz`) documents the performance of that routine when it was called with a flag indicating that transformations should be accumulated. The corresponding entry without the trailing "z" in the name contains times for the calls when no accumulation was performed.

PVS routines that performed outstandingly well did so because the PVS routine had been converted from the NATS stride-N code to all stride-one code, either through conventional means (recoding) or through judicious application of compiler directives.

- **qzhes** - Two elementary row transformations were vectorized.
- **qzit** - As in `qzhes`, two elementary row transformations were vectorized. A "prefer vector" directive was applied to a row summation, vectorizing it as well.
- **qzvec** - A "prefer vector" directive was applied to a single row oriented inner product.
- **reduc, reduc2** - A single row inner product was vectorized.
- **trbak1** - A "prefer vector" directive and an "ignore recrdeps" directive were applied to a loop at nesting level 2. The innermost loops are mixed row and column operations.
- **trbak3,tred2** - The same problem experienced in the development of `trbak1` was noted for each of these routines. The same solution was applied in both cases. In the NATS `trbak3`, all operations were column oriented, so no compiler directives were needed. For `tred2`, transformation #6 was also applied.
- **tred1, tred3** - In each case, transformation #6 was applied. In `tred1` a section of code resembling a 3-way row exchange was vectorized.

6 Timings

Comparison testing was done on PASC code versus NATS (i. e. the double precision versions) and PVS code versus NATS (i. e. the single precision versions). The tables are found in Appendices 1 and 2. Times are recorded in seconds, and for each table, the number of samples per value of N is in parentheses following the name of the subroutine.

6.1 PASC vs. NATS (double precision)

The first four routines listed in the table (`balanc`, `balbak`, `cbak2`, and `cbal`) were not part of PASC: the timings are actually NATS vs. NATS and are included only for purposes of calibration. It should also be noted that seven of the routines to be compared to the PASC routines were actually NATS routines that had been modified to run more efficiently on the 3090-VF. They were `comhes`, `elmhes`, `orthes`, `qzhes`, `qzit`, `tql1`, and `tred1`. These seven were involved in an experiment to assess how productive a moderate amount of recoding effort would be. In the experiment no more

nested (level 3) loops show an increase in performance of from 10 to 15% over the NATS `htribk`. It should be noted that the loops at nesting level 3 performed *mixed* row and column complex inner products and elementary transformations.

- **figi, figi2** - see discussion in section 4
- **hqr** - "prefer vector" directives were used to vectorize a diagonal shift and an initialization of a diagonal to zero. The use of "prefer vector" directives and the renaming of temporary variables permitted vectorization of many additional row and column operations.
- **hqr2** - The transformations that were applied to `hqr` applied equally well here. Some transformations specific to `hqr2` involved additional renamings and the application of a "prefer vector" directive to a matrix multiply.
- **htrib3** - The sole transformation consisted of changing the signs of scale factors in a complex row/column saxpy.
- **htridi, htrid3** - In `htridi`, "prefer vector" directives were used on the following complex, row-oriented operations: sum, saxpy, sdot, scale. To achieve the same results in `htrid3`, "ignore recrdeps" directives had to be used in each case as well.
- **imtql1, imtql2, imtqlv, tql1, tql2** - The same transformation was performed for each of these routines. Invocations of the function `pythag` were replaced by equivalent in-line code. The major benefit in making such a replacement is that it permitted the vectorization of a loop at nesting level 2.
- **minfit, svd** - In two places in each routine, an "ignore recrdeps" directive was given to vectorize a doubly nested loop which performed at each iteration a column inner product and a column saxpy. A "prefer vector" directive was needed to vectorize a row scale, a row sum, and, with the help of an "ignore recrdeps" directive, a combined column and row initialization. The only transformation not performed in both cases affected `minfit`: a matrix initialization loop was modified so that the array was traversed by column rather than by row.
- **orthes** - As mentioned in section 4, the transformations in `orthes` involved negating scale factors in elementary transformations and modifying inner products to use positive strides. Here, both loops at nesting level 2 were vectorized, in spite of the fact that in one all operations were column oriented. (This, the authors believe, was a mistake and probably led to a degradation in performance.)

was given. `Comhes`, the complex analog of `elmhes`, was modified in exactly the same manner with one exception: an auxiliary routine, `cdiv`, was coded in-line to obtain the same degree of vectorization.

- **comlr, comlr2** - The transformations applied to `comlr` and `comlr2` were very similar. Both used "prefer vector" directives to vectorize row exchanges, elementary row transformations, and a diagonal shift. Numerous applications of the "rename overused temporaries" rule permitted vectorization of several row and column exchanges. Two additional transformations affected `comlr2` alone. A matrix norm was reformulated to permit its vectorization, and a "prefer vector" directive was applied to a mixed row/column inner product.

- **comqr, comqr2** - Much like `comlr` and `comlr2`, these two routines are similar enough in functionality that most transformations successfully applied to one are applied to the other as well. They shared a directive to encourage vectorization of a diagonal shift and the effective equivalent of an "ignore recrdeps" directive to permit vector operation in a pair of complex row saxpys. In `comqr2` alone, a matrix norm was reformulated to permit its vectorization, and a "prefer vector" directive was applied to a mixed row/column inner product.

- **corth** - Two transformations were applied to `corth`: a complex inner product was modified to use a positive stride, a pair of complex elementary transformations (one row-, the other column-oriented) were converted according to transformation #6.

- **eltran, htribk** - Here are two cases in which initial, injudicious application of the "prefer vector" directive resulted in noticeably anomalous performance. In `eltran's` case, the compiler was instructed to generate vector instructions for the innermost loop, which performs a row copy (not an interchange). The timing data (the PVS version executed from 10% to 20% more slowly than the NATS code) made it quite clear that, in this case, the scalar code is much more efficient. Removal of the offending directive yielded code effectively identical to NATS `eltran`. The moral here is, of course, that "copy" loops (in contrast to those performing vector-exchanges) generally execute more efficiently in scalar mode.

  In the case of `htribk`, the "prefer vector" directive was first applied to a loop that the compiler would have vectorized in any case. That loop was at nesting level 2. The timings of the code with the directives applied to the more deeply

- When a temporary variable (often with a name like `ip1` or `mm1`) was defined only to be used a single time (e.g. the initial or terminal index of a `do`-loop), it was replaced by its definition. Although efficiency was not appreciably affected by such changes, the resultant code was more readable.

- Many spurious ANSI 66 style zero-pass branches were removed. The removal of such branches is not mentioned below unless it was critical to loop rearrangement and/or vectorization, since only in that case is performance significantly affected.

- Nearly every reference to the function `pythag` was mechanically replaced by an in-line equivalent. For the few cases in which the replacement code was in a critical path, further refinements were made to decrease the number of floating operations in an inner loop. Those cases are noted below.

Of the 58 remaining routines, several were not substantially modified. Most unmodified routines were not timed, and none are discussed below. That is, if none of the transformations that were applied to a routine affected its vectorization, it was neither timed nor is it discussed below. Neither of the first two transformations affect vectorization, and in only a few cases was the removal of a branch required to achieve vectorization.

- **invit** - One "prefer vector" directive was used on a row summation in the computation of the infinity norm. A second was used to initialize a row to all zeros. The renaming of a temporary variable involved in a column exchange permitted the vectorization of that operation.

- **cinvit** - Renaming a temporary variable removed a compiler-induced dependence on a row exchange. Then, "prefer vector" directives forced the vectorization of the associated loop and one performing a complex row saxpy. Finally, transformation #6 was applied to the saxpy.

- **tinvit** - The single significant transformation applied to `tinvit` was to issue a "prefer vector" directive for the sole eligible loop.

- **combak, elmbak** - The application of an "ignore recrdeps" directive and the removal of a branch around an inner loop permitted a loop interchange and subsequent vectorization.

- **comhes, elmhes** - `Elmhes` was rewritten to perform all column operations and to make only one pass over the input matrices. One "ignore recrdeps" directive

cannot possibly be a dependence on the scale factor, `a(j,m-1)`, so the problem must lie with the term `a(i,j)`. Studying the inner loop in isolation we see that depending on the relationship between the indices `m` and `j`, there may indeed be a recursive dependency.[3] Now, stepping back to consider both loops, it is easy to determine that `j` will always be at least one greater than `m`, so actually there is no dependence.

```
         do 140 j = m+1, igh
            do 139 i = 1, igh
  139          a(i,m) = a(i,m) + a(j,m-1)*a(i,j)
  140    continue
```

The vector directive to ignore recursive dependencies in the code below serves to inform the compiler that `a(i,m)` does *not* depend recursively upon `a(i,j)`. Thus, the inner loop is vectorized. Caution must be used when specifying 'ignore' directives, since if improperly applied, they (unlike 'prefer' directives) may profoundly change the semantics of affected subroutines.

```
         do 140 j = m+1, igh
c"    ( ignore recrdeps
            do 139 i = 1, igh
  139          a(i,m) = a(i,m) + a(j,m-1)*a(i,j)
  140    continue
```

5 Timings of individual PVS routines

In this section, we shall discuss the individual PVS routines, relating which transformations (from section 4) were applied and the effects they had on vectorization. There are 75 separate subroutines and functions in the NATS package. Of those, 13 are simply driver subroutines and 4 are the utility subprograms `cdiv`, `csroot`, `epslon`, and `pythag`. No drivers or utilities were modified. In nearly every other routine, some of the following general transformations were used to convert NATS code into PVS code.

- ANSI 66 style loops with negative stride were converted to FORTRAN 77 `do`-loops with negative increments.

[3] If a loop on i computes a_i as a function of a_{i-1}, then a_i is said to be recursively dependent on a_{i-1}

```
      do 130 j = m, n
         f = 0.0d0
         do 110 ii = m, igh
            i = mp - ii
            f = f + ort(i) * a(i,j)
110      continue
         f = f / h
         do 120 i = m, igh
120      a(i,j) = a(i,j) - f * ort(i)
130   continue
```

was transformed into the PVS code:

```
      do 130 j = m, n
         f = 0.0d0
         do 110 i = m, igh
            f = f + ort(i) * a(i,j)
110      continue
         f = -f * h
         do 120 i = m, igh
120         a(i,j) = a(i,j) + f * ort(i)
130   continue
```

Note that in the NATS version, the inner product is computed with a negative stride, and in the elementary transformation the product `f * ort(i)` is subtracted from `a(i,j)`. Making the two small changes to obtain the PVS code yielded more than a 10% performance improvement for N as small as 100.[2]

7. **Use judiciously the "ignore recrdeps" compiler directive.** Let us examine a section of code taken in part from the PVS routine, `elmhes`. At first glance, the code would seem to be well-suited to vectorization, since the inner loop performs elementary transformations on columns. Unfortunately, the compiler reports that there *may* be a recursive dependency in the innermost loop, and does not generate vector instructions for it. We examine the innermost loop and note that there

[2]Note that there was a third change: the division by `h` in NATS was replaced in PVS by a multiplication by its inverse. Although multiplies are cheaper than divides on the 3090-vf, such a change does not yield a discernible performance improvement in this case.

```
 60    if (t(i,1).eq.0.0e0 .and. t(i-1,3).eq.0.0e0) go to 80
       ierr = -(3 * n + i)
 80    e(i) = sqrt(e2(i))
 90    d(i) = t(i,2)
100 continue
```

It is possible, in this case, to perform computation and testing in separate loops without duplicating any computation. Thus, at least the computation loop may be vectorized. The only expense is that when bad input is detected, more computation will have been performed before returning, since the computation loop must precede the loop for input validation. The net performance gain for this transformation is an average of about 50% for N in the range 50 to 150.

```
      d(1) = t(1,2)
      do 100 i = 2, n
         e2(i) = t(i,1) * t(i-1,3)
         e(i) = sqrt(abs(e2(i)))
         d(i) = t(i,2)
  100 continue
      do 200 i = 2, n
         if (e2(i) .gt. 0.e0) go to 200
            if (e2(i) .lt. 0.e0) go to 1000
               if (t(i,1).eq.0.e0.and.t(i-1,3).eq.0.e0) go to 200
               ierr = -(3 * n + i)
  200 continue
```

6. **When performing elementary vector operations, use the form $Y = Y + sX$ rather than $Y = Y - sX$.** If necessary, the sign of the scale factor s is changed before entering the loop. A related transformation is to convert inner products with negative stride to (mathematically) equivalent inner products with positive stride.[1] In general, negative stride loops should be avoided, since they are less efficiently vectorized. Both of these transformations were central to producing the PVS routine, `orthes`. For example, the NATS code `orthes` :

[1] However, using a positive stride when forming inner products of vectors from a graded matrix may result in an unacceptable loss of accuracy. See [Wilkinson, pp.339-358] for an in-depth discussion on this topic.

The following code is very similar to a part of the NATS routine, `elmhes`. The test and jump are used mainly to avoid the execution of the inner loop when it would have no net effect. Unfortunately, such "optimizations" impair the compiler's ability to recognize opportunities for vectorization. As in the previous example, removing the `if` statement permits vectorization of the outer loop without changing the semantics of the code.

```
      do 160 i = 1, n
         y = a(i,mm1)
         if (y .eq. 0.0e0) go to 160
         y = y / x
         a(i,mm1) = y
         do 140 j = m, n
  140       a(i,j) = a(i,j) - y * a(m,j)
  160 continue
```

Here, there is a lesson more generally applicable than "eliminate explicit branches to enable vectorization." Unless the above code is targeted for use in an application involving fairly sparse matrices, the scale factor, `y`, will seldom be 0. Clearly, executing the inner loop with `y` = 0 is a waste of time, but if `y` is rarely 0, the degradation (admittedly slight) of average performance is not worth the occasional "best case" improvements. Another argument in favor of removing the `go to` is that for most modern compilers, structured code is more easily and efficiently optimized.

A final example demonstrates dramatically how the improper use of explicit branches can inhibit vectorization. The EISPACK subroutine, `figi`, does not perform many operations (only $O(n)$). It is unusual in that it performs nearly as many operations to test the integrity of input data as it performs operations on those data. The NATS version of `figi` interleaves the error testing and computation. Since one of the error branches (to the `1000` label) exits the loop, the entire loop must be executed in scalar mode. Here is the body of the NATS code.

```
      do 100 i = 1, n
         if (i .eq. 1) go to 90
         e2(i) = t(i,1) * t(i-1,3)
         if (e2(i)) 1000, 60, 80
```

```
      else if (Z .ne. 0.0e0) then
         X = abs(Z)*sqrt((Y/Z)**2 + 1.0e0)
      else
         X = abs(Y)
      endif
```

If, for example, it is known that `Z` is non-zero or if either of `Y` or `Z` is a constant, the above section of code may be appropriately simplified. It is important to make in-line any small, auxiliary routines that appear in critical inner loops, since their presence will prohibit the vectorization of that or any containing loop. In the case of `pythag`, nearly all occurrences were replaced because the code using the hardware `sqrt` was much more efficient.

5. **Convert ANSI 66 FORTRAN-style, zero-pass loops to FORTRAN 77.** ANSI 66 FORTRAN did not specify whether the body of a `do`-loop should be executed when the initial index exceeded the final index. This forced the writers of portable code to insert just before any `do`-loop with such initial and final indices, a logical `if` that would cause a jump around the loop. In FORTRAN 77, it is guaranteed that the bodies of such loops will not be executed, so removing the associated logical `if` statements will not change the semantics of a FORTRAN 77 program. The removal of the offending `if` statement may permit a vector compiler to rearrange and vectorize loops that would otherwise have been executed in scalar mode.

   In the example below, the test and branch before the `do 140` loop prohibit the "loop interchange" that would have permitted vectorization on the outer loop. By default, the inner loop is not vectorized because it performs a row operation, and scalar code would execute more quickly in general. Simply removing the `if` statement permits vectorization of the outer loop, effectively yielding the more efficient *column* vector operations.

```
      do 160 i = 1, n
         if (k.gt.m) go to 160
         do 140 j = k, m
140         a(i,j) = a(i,j) - y * a(m,j)
160   continue
```

targeting this code for relatively small problems, we believe that estimation is inapplicable, and think that these operations should be vectorized. One possibility is that the compiler would interchange the two loops, effectively converting the row operations to column operations. Unfortunately, there is a dependency of the inner loop index, `j`, on the outer loop index, `i`, that prevents the interchange. To force the compiler to vectorize an eligible loop, one need simply insert a comment of a prespecified form just before that loop.

```
         do 400 i = 2, uk
            do 380 j = i, uk
               rm1(i,j) = rm1(i,j) - x * rm1(mp,j) + y * rm2(mp,j)
               rm2(i,j) = rm2(i,j) - x * rm2(mp,j) - y * rm1(mp,j)
  380       continue
  400    continue
```

The comment below instructs the compiler that we would like it to generate vector code for a loop that (in this case) it has estimated would run faster in scalar mode.

```
         do 400 i = 2, uk
c"    (prefer vector
            do 380 j = i, uk
               rm1(i,j) = rm1(i,j) - x * rm1(mp,j) + y * rm2(mp,j)
               rm2(i,j) = rm2(i,j) - x * rm2(mp,j) - y * rm1(mp,j)
  380       continue
  400    continue
```

4. **Replace references to the function `pythag` by an in-line equivalent using the `sqrt` function.** For example, the statement

```
      X = pythag(Y,Z)
```

would become the code segment:

```
      if (abs(Y) .gt. abs(Z)) then
         X = abs(Y)*sqrt(1.0e0 + (Z/Y)**2)
```

Code with the same functionality appears in the PVS version of `hqr`. However, that code (below) is much more readable and, since the array is traversed by column, it is much more efficient as well.

```
      do 50 j = 1, n
         do 50 i = 1,min0(n,j+1)
50          norm = norm + dabs(h(i,j))
```

3. **Force vectorization of operations on rows and diagonals.** When it is not possible to rearrange a computation (i.e. converting row operations to equivalent column operations) one should specify with a compiler directive in the FORTRAN source code that the row (or diagonal) operation should, if possible, be performed using the vector hardware. Here we take advantage of the assumption that N is small. When N is small enough that a large portion of the data to be accessed will fit in the 3090's cache, the disadvantages of row access are outweighed by the advantages of vectorization. When N is this small, the archetypical problem associated with row operations, page faulting, is insignificant. On the 3090-VF, there is generally a penalty for performing vector operations on rows because (with single precision) the attempt to access each element in at least one row out of four will cause a cache fault. The problem is even more pronounced when the data are double precision or complex, since only two elements are copied into the cache after each fault. Another problem with row vector operations arises when N is very small. If N is a sufficiently small fraction of the vector section length, the overhead of setting up for the vector operation is not offset by the few pipelined arithmetic operations.

   However, there is a middle ground, where N is large enough to gain more from vector operation than is lost to page and cache faults, and yet not so small that pipelining overhead dominates. For most routines that middle ground seems to be in a range from $N = 60$ to about 150.

   To illustrate this transformation, consider the following code taken from the NATS routine `cinvit`. The inner loop is eligible to be vectorized. That is, it has no vectorization-prohibiting characteristics (e.g. recursive dependencies, references to user functions or subroutines, etc.). However, the two arrays `rm1` and `rm2` are traversed by row, and that fact leads the compiler to estimate that the code would run faster in scalar mode than in vector mode. Since we are

an identity matrix. Another common instance of unnecessary row operations was found in the computation of matrix norms. Such small, simple loops are easily converted to a form more amenable to effective vectorization. More time-consuming is the task of correctly converting even so straightforward an algorithm as that for reducing a real matrix to Hessenberg form using stabilized elementary vector transformations (as is done in `elmhes`). The following example was taken verbatim from the NATS routine `minfit`.

```
      do 500 i = m1, n
         do 500 j = 1, ip
            b(i,j) = 0.0e0
  500 continue
```

This code is trivially transformed into its column-major counterpart which appears in the PVS version of `minfit`. Column-major traversal of the array `b` makes much more efficient use of the cache.

```
      do 500 j = 1, ip
         do 500 i = m1, n
            b(i,j) = 0.0e0
  500 continue
```

The following segment of code (here, slightly modified for clarity) is used in the NATS routine `hqr` to compute the norm of the Hessenberg matrix `h(n,n)`. Note that the elements of `h` are accessed by row, and that the auxiliary variable, `k`, is needed to conform to the ANSI 66 requirement that initial and final `do`-loop indices be simple variables or constants (expressions, like `max0(1,i-1)`, were not permitted).

```
      k = 1
      do 50 i = 1, n
         do 40 j = k, n
   40       norm = norm + dabs(h(i,j))
         k = i
   50 continue
```

```
         xxr = a(i, j)
         a(i, j) = a(i, k)
         a(i, k) = xxr
100   continue
       .
       .
      call cdiv(ar, ai, br, bi, xxr, xxi)
```

A solution that permits the desired vectorization is to replace the two occurrences of `xxr` in the loop with a variable (e. g. `foo`) that is never used as anything but a compiler-recognizable temporary.

```
      do 100 i = 1,n
         foo = a(i, j)
         a(i, j) = a(i, k)
         a(i, k) = foo
100   continue
       .
       .
      call cdiv(ar, ai, br, bi, xxr, xxi)
```

Another, slightly less time-efficient option would be simply to assign to `xxr` some constant value just after the loop, thus:

```
      do 100 i = 1,n
         xxr = a(i, j)
         a(i, j) = a(i, k)
         a(i, k) = xxr
100   continue
      xxr = 0.0e0
       .
       .
      call cdiv(ar, ai, br, bi, xxr, xxi)
```

2. **Reorder simple loops to traverse arrays by column.** For example, in a few EISPACK routines, an input array is traversed by row to initialize it to

an entire line (probably containing one or more additional elements) is copied from primary storage into the cache. Adding the optional VF hardware to a 3090 gives it an 128-element, off-chip vector processing unit and several general purpose vector registers. The vector processor pipelines arithmetic operations such as elementary vector transformations and inner products, and can achieve much higher throughput than a conventional, scalar processor. The efficient use of the vector processor depends heavily upon the compiler and upon effective use of the cache. For example, the vectorization of an isolated inner product involving a large stride will not normally be efficient, since memory accesses will be widely separated. A greater familiarity with the equipment and the computing environment can be obtained in [2][3][4][5][6].

4 Classes of Program Transformations

In this section, several classes of program transformations are presented. With each is an example or two illustrating how it is typically applied. Most of the transformations enable the 3090's vectorizing compiler to produce more efficient code. Using these techniques, the reader should be able to recognize how nearly all of PVS was obtained from the original code, NATS. Notably exceptional is the new eigenvalue convergence test used in both `bisect` and `tsturm`. The test used in NATS is architecture- and compiler-dependent when intermediate results may be stored in extended precision registers. The new test is more portable and generally yields results at least as accurate as the NATS version.

1. **Avoid the use of critical temporary variables as parameters to subroutines.** In the code fragment below, the variable `xxr` is used as a temporary in the column exchange and is also a parameter to the subroutine `cdiv`. The last two arguments to `cdiv` are not input parameters, however, and since the compiler does not perform inter-procedural analysis it is unable to determine that the last value computed in the loop is not used as input to the subroutine. The compiler must then assume that the value assigned to `xxr` in the last iteration of the loop may be used in the subroutine. This assumption prevents the vectorization of the `do 100` loop, since the vectorization of such a loop requires that the temporary be expanded to a vector, and there is no way to extract from the temporary vector the final value of `xxr`.

```
      do 100 i = 1,n
```

referred to as PVS.

2 Design Considerations

Several factors influenced the ultimate form of our version of single precision EISPACK (PVS). One of the most important of those was the requirement that each subroutine and function retain the calling sequence and functionality of the respective module in the original package. A related objective was to keep the code of PVS as close as possible to that of NATS, modifying NATS just enough to obtain the desired degree of vectorization. The intent was to produce a version of EISPACK that would take advantage of the environment (hardware, compiler, operating system) peculiar to the 3090-VF, and yet would remain portable. However, a single non-portable construct was introduced into the PVS code: many subroutines call a routine from the system library, `XUFLOW`, with a flag indicating that floating point underflows are silently to be set to zero. The default behavior, which involves the generation of a system interrupt and much diagnostic output, can significantly decrease performance. The effort to maintain portability should be kept in mind when comparing the times of our routines to those that were carefully tuned for the 3090. A key assumption was that since the package would use single precision, it would be used almost exclusively for small problems ($N \leq 150$). Accordingly, our aim was to produce software that would be fast for suitably small problems, recognizing that performance might degrade as N increased beyond the $N = 150$ threshold.

3 High Performance Hardware of the IBM 3090-VF

In the following discussion, it is important that the reader bear in mind some key features of the 3090-VF and its vector compiler. The 3090-VF has a fairly standard virtual memory hierarchy: secondary storage, primary storage, cache, vector and scalar registers. The cache and vectors registers deserve special attention, since the performance of any software run on this system depends most heavily on the efficiency with which these resources are used. The cache can be viewed simply as a very fast extension of primary memory, but for our purposes, we must elaborate. We note that the cache has a capacity of 64K bytes and is divided into 4K lines of 16 bytes each. A cache line is analogous to a page in primary storage. Whenever a memory access is performed and the desired element is not found in the cache (called a cache miss) then

Converting EISPACK to Run Efficiently on a Vector Processor

Alan Kaylor Cline
James Meyering

Pleasant Valley Software,
8603 Altus Cove,
Austin, Texas 78759

May 26, 1991

1 Introduction

This report documents the production, performance evaluation, and testing of a version of the single precision FORTRAN software package (known as EISPACK) that takes advantage of the high-speed vector registers and cache of the IBM 3090-VF. In addition, the results of testing and timing a carefully crafted and tuned double precision version of the same package help put the single precision results into perspective. We explain how design objectives shaped the single precision end-product. Timing data are then presented and we describe how various low-level modifications to the FORTRAN source affected the performance of individual routines. Our testing methodology is examined and the results of our tests are presented and analyzed.

To facilitate the discussion of the three versions of EISPACK mentioned above, we assign them names. NATS will refer to the original code (dated 1983) that was obtained from the netlib archive. Both single and double precision versions are referenced: if the precision of the version being discussed is not stated explicitly, it should be apparent from the context. The carefully crafted double precision code will be called PASC. It was written at the IBM Palo Alto Scientific Center by Augustin Dubrulle and is based on the collection of ALGOL procedures from which EISPACK evolved. The third version, the single precision code that is the major topic of this report, will be