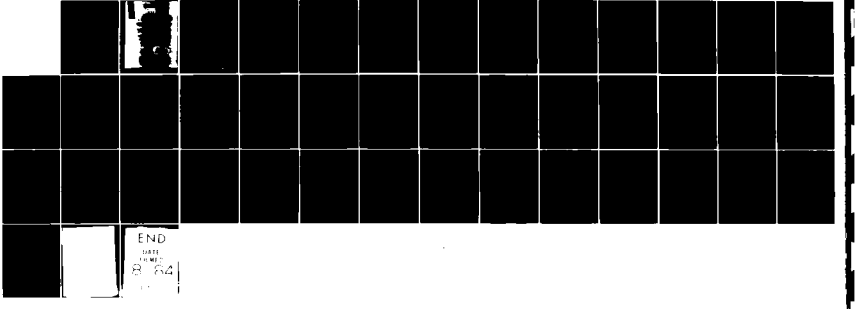
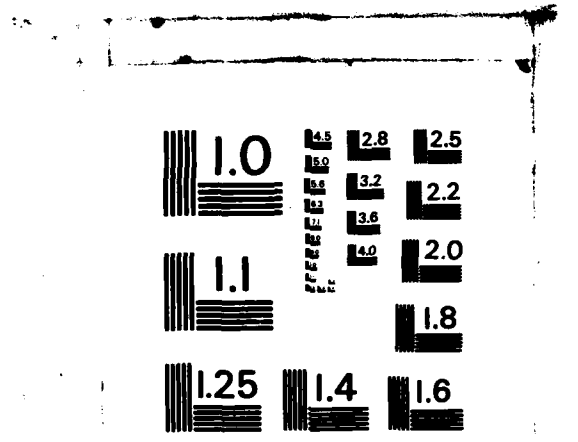
AD-A142 942 EFFECT OF THE CYBER 205 ON METHODS FOR COMPUTING 1/1
NATURAL FREQUENCIES OF S..(U) CALIFORNIA UNIV BERKELEY
CENTER FOR PURE AND APPLIED MATHEMAT.. J NATVIG ET AL.
UNCLASSIFIED MAR 84 PAM-218 N00014-76-C-0013 F/G 9/2 NL



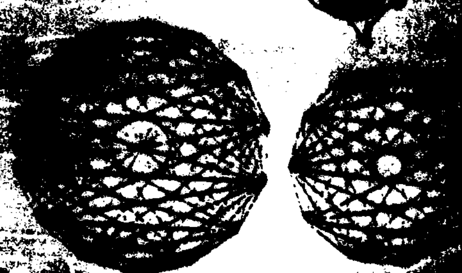


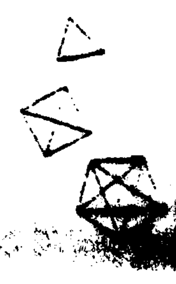
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

...MATICS

... CYBER 205 ... METHODS FOR COMPUTING ... NATURAL FREQUENCIES OF STRUCTURES†

..., B. NOUR-OMID, AND B.N. PARLETT





Effect of the CYBER 205 on Methods for computing Natural Frequencies of Structures†

J. Natvig‡, B. Nour-Omid, and B.N. Parlett

Center for Pure and Applied Mathematics
University of California, Berkeley, CA 94720, USA

ABSTRACT

We consider the generalized eigenvalue problem, $(A - \lambda M)x = 0$, where A and M are large, sparse, symmetric matrices. For large problems finding only a few eigenpairs involves a major computational task. In a typical example from structural dynamic analysis with matrices of order 8000, $O(10^9)$ operations are required to compute 50 eigenpairs. It is therefore interesting to examine the advantage that vector computers such as CYBER 205 can offer.

We adopted our best versions of the Subspace Iteration Method and the simple Lanczos Method in order to take advantage of the special vector processor of the CYBER 205. Both techniques lend themselves to vectorization. Our extensive comparisons support the following general statements. Both methods require the triangular factorization of the same large $n \times n$ matrix. This factorization dominates the total computation as $n \to \infty$ provided that the number of wanted eigenpairs, p, remains fixed (independent of n). However, simple Lanczos is at least an order of magnitude more efficient (in CPU-time) for the remainder of the computation. For $p=40$, $n=500$ the factorization time is not important and the full order of magnitude difference is seen in the total CPU-time. When $p=40$, $n=8000$ simple Lanczos is only 4 times faster than Subspace Iteration on the CYBER 205. This confirms experience on serial computers.

For problems that cannot fit into primary storage, input/output becomes increasingly important. We found that the cost of input/output dominated over the CPU-cost for a problem that required twice the available primary storage on our CYBER 205. However, this will depend on the billing algorithm of the computer center. We conclude that problems which have a substantial overhead in reading and writing the matrices, should not be solved by the simple Lanczos Method, but by a Block Lanczos Method.

March 13, 1984

†Work made possible by a grant of computer time from CDC (grant #82CSU22). Partial support by the U.S. Office of Naval Research under contract N00014-76-C-0013 is gratefully acknowledged.
‡On leave from Rogaland Regional College, N-4000 Stavanger, Norway.

Effect of the CYBER 205 on Methods for computing Natural Frequencies of Structures†

J. Natvig‡, B. Nour-Omid, and B.N. Parlett

Center for Pure and Applied Mathematics

University of California, Berkeley, CA 94720, USA

1. Introduction.

The purpose of this project was to examine the performance of two different methods for the solution of the eigenvalue problem

$$(A - \lambda M)\, x = 0 \tag{1}$$

when implemented on a CYBER 205 vector computer. We investigated the Subspace Iteration Method [13], [1] (called SUBIT hereafter), which is widely used on traditional serial computers for medium sized and large eigenproblems, and the Lanczos' Method [8], [10] (called LANC hereafter), which recently has been shown to be an order of magnitude faster than SUBIT [9]. It was of interest to see how the methods compare on a vector computer.

Standard in-core versions of the two algorithms were modified to take advantage of the special features of the CYBER 205 [4]. The algorithms were not redesigned, but wherever possible, a CYBER 205 vector function was substituted for the original FORTRAN code.

The test problems were derived from the dynamic analysis of idealized 3-dimensional structures, as modelled by the finite element program FEAP [16, Ch. 24]. For various problem sizes the two methods were timed extensively, both before and after the explicit vectorization took place.

---

†Work made possible by a grant of computer time from CDC (grant #82CSU22). Partial support by the U.S. Office of Naval Research under contract N00014-76-C-0013 is gratefully acknowledged.

‡On leave from Rogaland Regional College, N-4000 Stavanger, Norway.

2. The nature of the eigenvalue problem.

We are interested in the solution of eq. (1) when A and M are large, sparse, $n \times n$, symmetric matrices. In many applications the matrices have a banded form, i.e. $a_{ij} = 0$ for all $|i-j| > b_m$, where a_{ij} is the (i,j)-element of A and b_m is the half bandwidth. For typical problems in structural dynamics b_m is (5–10)% of n, cfr. Section 5. M must be positive semi-definite. In some cases M (or A) may be diagonal, which leads to a significant savings in the computational effort with either method.

"Large" today means $n > 10^5$, but 10 years ago 10^2 was considered large. In these large problems all eigenpairs (λ_i, x_i) in a given interval may be sought; these are usually quite few, perhaps between 10 to 50. The interval may be at either end of the eigenspectrum and may contain the origin. For best convergence properties it is best to perform a shift of origin to this interval before the actual eigen computation takes place. For more details, see Sections 4, 4.1, and 4.2 as well as [10].

3. Special features on CYBER 205.

The CYBER 205 is capable of attaining a rate of several hundred million floating point operations (add or multiply) per second, depending on the actual machine configuration and also on the precision of the arithmetic. We have used a 205 with 2 pipes in ordinary single precision (64 bits), and an asymptotic rate of 100 megaflops [4],[7]. One megaflop (mflop) is a rate of 1 million floating point operations per second.

3.1. Vectorisation.

The top performance can only be obtained by those parts of a program that operate on vectors, where a "vector" is a set of consecutive memory cells all treated in the same way.

In the following simple example there are three vectors, each corresponding to the n first elements of arrays B, A, and D. The vector in B is the Schur product of the vectors in A and D.

There are essentially two ways to achieve vector performance for this DO-loop on the CYBER 205. The FORTRAN code in Fig. 1 may be left unchanged and then be compiled with

```
      DO 100 I=1,N
         B(I)=A(I)*D(I)
100   CONTINUE
```

Fig. 1. DO-loop operating on vectors (Schur product)

special vector optimizers ("automatic vectorization"), or we may replace the DO-loop by a direct reference to the vector multiply instruction, see Fig. 2.

```
B(1;N)=A(1;N)*D(1;N)
```

Fig. 2. Vector multiply instruction corresponding to Fig. 1

In either case the vector instruction is composed of a start-up phase during which the operands are lined up and made ready, and the actual execution phase with two operations (because of two pipes) per CPU-cycle (one CPU-cycle was equal to 20 nanoseconds on the CYBER 205). Because the unproductive start-up phase has to be amortized over all operations, the longer the vector length, the higher the performance rate. (However, there is a maximum allowed vector length of 65535 elements.) We have timed the vector multiply instruction (Fig. 2) for various vector lengths, and we have found good agreement with the performance data given by CDC [7]; e.g. a rate of 50 mflops, or half the asymptotic rate, is achieved with vector lengths about 100, and 80 mflops is reached when the vector lengths are about 400. The results of our direct performance measurements are presented in Appendix A.

Consider next the example in Fig. 3, where there are two input vectors and one input scalar that are to be combined through an addition and a multiplication. We refer to this as a SAXPY (*s*ingle precision *a* $\times$ *x* *p*lus *y*), and on the CYBER 205 it may be realized as one vector operation, i.e. after start-up two output elements (because of two pipes) are computed per CPU-cycle. As is customary in numerical analysis we shall count the production of one output element as one floating point operation. CDC uses a different terminology and calls a SAXPY a linked triad, and also counts each pass within a SAXPY as two floating point operations (multiplication and addition). With vector lengths 100, we found an effective rate of about 37 mflops; 50 mflops is

reached for vector lengths about 160, 80 mflops for vector lengths about 650, and the asymptotic value is still 100 mflops (cfr. Appendix A).

```
      DO 200 I=1,M
         Y(I)=Y(I)+ A*X(I)
200   CONTINUE
```

Fig. 3. DO-loop for the SAXPY

Again, the vector operation will be invoked if automatic vectorization is specified during compilation, or if explicit vector code is substituted in the program, see Fig. 4.

```
Y(1;M)=Y(1;M)+ A*X(1;M)
```

Fig. 4. Vector code corresponding to Fig. 3

The dot product of two vectors may be coded as in Fig. 5.

```
      DOT=0.
      DO 100 I=1,N
         DOT=DOT+ R(I)*U(I)
100   CONTINUE
```

Fig. 5. DO-loop for the dot product

The dot product function on the CYBER 205, Q8SDOT, is not a vector function, but a simulation of one, see Fig.6. It has been implemented with scalar instructions in a very efficient way.

```
DOT=Q8SDOT(R(1;N),U(1;N))
```

Fig. 6. "Vector" dot product equivalent to Fig. 5

Q8SDOT has a relatively slow start-up phase, and then performs one partial product and accumulation per cycle. This operation is unrelated to the pipeline feature. On a CYBER 205 with two pipelines, as was the case in the present investigation, the dot product will therefore only reach half the speed of the SAXPY for long vectors. We shall count one partial product and accumulation as one floating point operation (thus Figs. 5 and 6 each contains n floating point

operations). In our direct measurements of the vector dot product (Fig. 6) we found an effective rate of about 20 mflops with vector lengths 100, and about 43 mflops with vector lengths 1000 (cfr. Appendix A). For all vector lengths it is to be seen from Appendix A that the SAXPY is almost twice as fast, and the Schur product more than twice as fast, as the dot product.

The ratios of mflop rates in Appendix A suggest strongly that programs for the CYBER 205 should be written using linear combinations of vectors (SAXPY's) rather than dot products. For example, if V is $n \times m$ and P is $m \times m$ then the product $Z = VP$ can be computed either as nm dot products of length m (requires V held by rows) or as m^2 SAXPY's, each of length n. The second form (see Fig. 7) is clearly preferable when $n > m$.

```
      DO 200 J=1,M
          Z(1,J;N)=P(1,J)*V(1,1;N)
          DO 200 I=2,M
              Z(1,J;N)=Z(1,J;N)+ P(I,J)*V(1,I;N)
200   CONTINUE
```

Fig. 7. SAXPY's used for matrix multiplication, $n > m$

When $n < m$, the first form (using dot products) may be faster than Fig. 7, because the vector lengths are longer. However, it is more efficient to compute the rows of Z as linear combinations of the rows of P (using SAXPY's), although then P and Z should be held by rows.

3.2. Memory management.

The CYBER 205 has virtual memory (theoretical upper bound 2×10^{12} words per user). The real memory on the machine we used was 2 million words, and there was a 36.5 Mbit/s link to a total of 450×10^6 words on disk. A program, with instructions and data, will be organized on pages, for which there are two choices, either small pages (equal to 512 words) or large pages (equal to 65536 words). The paging system will seek to keep the most recently used pages in the primary storage. When the program references data (or instructions) that do not at that moment reside in the primary storage, a "page fault" occurs. CYBER 205 will halt the execution of the program until the page that contains the requested data has been transferred from the secondary

storage, usually at the expense of another page being put out to the secondary storage. While this swapping takes place, the CYBER 205 may execute other programs that are allowed to occupy part of central memory. There is a small overhead in CPU-time when a vector crosses a page boundary, and also during a page fault. When a program references data in an "orderly" fashion, as when consecutive columns of a matrix are used consecutively and not at random, it is more efficient to use large pages. This is indeed the case with both our eigenvalue methods.

The accounting system assesses a cost penalty for a large page fault of .156 SBU (System Billing Units), equivalent to .156 CPU-seconds [12]. For large problems that cannot be fitted into primary storage, this penalty might actually be larger than the CPU time for the whole computation.

4. Description of the two eigenvalue methods.

We are interested in some of the eigenvalues closest to a specified value, σ. In each method we shall perform the same initial calculations:

- shift the **A** matrix by σ: $\overline{\mathbf{A}} = \mathbf{A} - \sigma\mathbf{M}$
- factor $\overline{\mathbf{A}}$ into $\mathbf{L}\,\Delta\,\mathbf{L}^T$

Here **L** is a lower triangular matrix with diagonal elements equal to one, Δ is a diagonal matrix, and $\mathbf{L}^T$ is the transpose of **L**. Incidentally the number of negative elements in Δ equals the number of eigenvalues that are smaller than σ.

We used a standard active column profile solver (called PROFIL). The upper triangular part of $\overline{\mathbf{A}}$ is stored and gradually overwritten with $\mathbf{L}^T$. Consider the computation of a typical column of $\mathbf{L}^T$ of "height" h (above the diagonal). Each element will require a dot product and the lengths of the vectors involved will vary from 1 to $h-1$. Altogether nb dot products are needed for $\mathbf{L}^T$ and the average length is $b/2$, where b is the average half bandwidth of $\overline{\mathbf{A}}$.

Of course PROFIL could be reorganised to compute **L** by columns and so replace dot products by SAXPY's. This helps, but not much. The significant fact is that $b/2$ is small compared to n in most structural problems and the factorization of narrow banded matrices cannot exploit

the full vector mflop rate of the CYBER 205 since it is manipulating vectors of (average) length $b/2$ rather than n.

As we shall see the factorization process dominates both methods to a greater extent on the CYBER 205 than in serial machines.

The remaining part of either method, SUBIT or LANC, is formulated in such a way as to solve the following transformed eigenvalue problem

$$(\mathbf{L}\Delta\mathbf{L}^T)^{-1}\mathbf{M} \tag{2}$$

The largest α's correspond to the eigenvalues λ closest to σ (these are the smallest λ's when σ=0) according to

$$\alpha_i = \frac{1}{\lambda_i - \sigma} \tag{3}$$

The eigenvectors, $\mathbf{x}$, in Eq. (2) are the same as in Eq. (1). See [5] for more on this transformation of the problem.

The following sections, 4.1 and 4.2, will describe in detail the two methods. At each stage we emphasize the vector operations.

4.1. Subspace Iteration (SUBIT).

The method works with m iteration vectors at a time. We say that the subspace dimension is m. At step k the current set of vectors, held as the columns of the $n \times m$ matrix $\mathbf{S}^{(k-1)}$, is replaced by another set, held as the columns of $\mathbf{S}^{(k)}$. The columns of $\mathbf{S}^{(k)}$ are kept mutually orthogonal.

During each major step k, (k=1,2, $\cdots$), the following tasks are performed.

4.1.1. M-operation.

(a) Compute

$$\mathbf{R}^{(k)} = \mathbf{M}\mathbf{S}^{(k-1)} \tag{4}$$

This involves a total of $(2b_M + 1)nm$ operations, where b_M is the average half

bandwidth of **M**. For a general **M** we would have an average vector length equal to b_M. But when **M** is diagonal, b_M=0 and only m vector Schur products of length n are performed.

(b) Compute the upper triangular part of the symmetric $m \times m$ projection matrix

$$\mathbf{M}^{(k)} = \mathbf{S}^{(k-1)T}\,\mathbf{R}^{(k)} \tag{5}$$

Here $m^2/2$ vector dot products of length n are performed.

4.1.2. A-projection.

(a) Solve $\overline{\mathbf{AS}}^{(k)}=\mathbf{R}^{(k)}$ for $\overline{\mathbf{S}}^{(k)}$. This is done in three phases:

(a1) Solve for **C**

$$\mathbf{L}\,\mathbf{C} = \mathbf{R}^{(k)} \tag{6}$$

nm dot products with average vector length b.

(a2) Compute

$$\mathbf{F} = \Delta^{-1}\,\mathbf{C} \tag{7}$$

m vector Schur products of length n.

(a3) Solve for $\overline{\mathbf{S}}^{(k)}$

$$\mathbf{L}^T\,\overline{\mathbf{S}}^{(k)} = \mathbf{F} \tag{8}$$

nm SAXPY's with average vector length b.

With care **C**, **F**, and $\overline{\mathbf{S}}_{(k)}$ can share the same storage area.

(b) Compute the upper triangular part of the symmetric $m \times m$ projection matrix

$$\mathbf{A}^{(k)} = \overline{\mathbf{S}}^{(k)o}\,\mathbf{R}^{(k)} \tag{9}$$

$m^2/2$ vector dot products with vectors of length n.

4.1.3. Small eigenproblem.

Solve the projected m by m eigenvalue problem

$$\left(\mathbf{A}^{(k)} - \theta^{(k)}\mathbf{B}^{(k)}\right) \tag{10}$$

for all m eigenvalues $\theta^{(i)}$ and (orthogonal) eigenvectors $G^{(i)}$.

Here we transformed the problem to a special eigenvalue problem [10, ch 15], which was solved by the EISPACK subroutine EISQL [6]. The transformations involve an $m \times m$ factorization and forward reduction and backward substitutions; a total of $4/3m^3$ operations. And it is our experience on scalar computers that EISQL contains twice as many operations, i.e. $8/3m^3$. The full eigensolution thus represents some $4m^3$ operations. For large typical eigenproblems this is negligible compared to the number of operations that are required in other parts of the program. Nevertheless, we replaced dot products and SAXPY's in the transformation subroutines with equivalent vector expressions. The EISQL subroutine cannot be vectorized.

4.1.4. Formation of new basis.

Compute

$$\mathbf{S}^{(i)} = \mathbf{\Sigma}^{(i)} \mathbf{G}^{(i)} \tag{11}$$

This can be written as m^2 SAXPY's of length n, cfr. Fig. 7.

Typically, both b and m are $\ll n$, and the number of necessary iterations, l, is about 20. E.g., when $n=2000$, $b=100$, then to find 40 eigenvalues we might use $m=50$, 60, 70, or 80, and expect convergence in $l=15$ to 30 iterations.

Under these circumstances the dominant parts of the algorithm, when the M matrix is diagonal (as in our test cases, cfr. Section 5), are

- Factorization

  $nb^2/2$ operations

- Linear operator (4.1.2 (a1), (a3))

  $l(2bnm)$ operations

- New basis (4.1.4)

  $l(nm^2)$ operations

The break-even between the factorization and the linear operator is reached after $l=b/(2m)$ iterations as far as the number of operations is concerned, but because of the short vector length during factorization we shall expect to need more iterations than this on the CYBER 205 before the linear operator takes as much CPU-time as the factorization.

The computation of the new basis will usually be far less expensive than the first two. m is usually small compared to $2b$, and the vector length is equal to the problem size n in this computation.

There is a potential for reduction of the number of operations in the fact that the first eigenvalues/eigenvectors will converge rather quickly. These eigenvectors may then be locked, and not participate e.g. in the time-consuming linear operator (Section 4.1.2(a)).

4.2. Lanczos' Method (LANC).

We used a simple Lanczos' algorithm. A single random starting vector, $\mathbf{r}_0$, is iterated according to the scheme presented in [9]. Initializations include setting $\mathbf{q}_0=0$, and computing $\mathbf{p}_0=\mathbf{M}\mathbf{r}_0$ and $\beta_1=\sqrt{\mathbf{r}_0^T\mathbf{p}_0}$.

In each step j, ($j=1,2,\cdots$), the following tasks are done:

(a) Orthogonalization

$\mathbf{r}_{j-1}$ will be orthogonalized against the previous Lanczos vectors when needed [15]. (This is called selective orthogonalization.)

A maximum of $(j-2)$ SAXPY's and dot products with vector length n.

(b) Compute $\mathbf{q}_j=\frac{\mathbf{r}_{j-1}}{\beta_j}$ and $\bar{\mathbf{p}}_{j-1}=\frac{\mathbf{p}_{j-1}}{\beta_j}$.

This is two vector operations, vector length n.

(c) Solve $\bar{\mathbf{A}}\mathbf{r}_j=\bar{\mathbf{p}}_{j-1}$ for $\mathbf{r}_j$.

(This is equivalent to Section 4.1.2(a), now with $m=1$.)

- "solve (A)", i.e. $(2b+1)n$ operations, n dot products and n SAXPY's with average vector length equal to b, and one vector Schur multiply with vector

length n.

(d) Compute $\mathbf{r}_j = \mathbf{r}_j - \mathbf{q}_{j-1}\beta_j$.

Single SAXPY, vector length n.

(e) Compute $\alpha_j = \mathbf{r}_j^T \overline{\mathbf{p}}_{j-1}$.

Single dot product, vector length n.

(f) Compute $\mathbf{r}_j = \mathbf{r}_j - \mathbf{q}_j \alpha_j$.

Single SAXPY, vector length n.

(g) Compute $\mathbf{p}_j = \mathbf{M}\mathbf{r}_j$ and $\beta_{j+1} = \sqrt{\mathbf{r}_j^T \mathbf{p}_j}$.

- "mult (M)", i.e. a vector operation (Schur product) of length n (when M is diagonal)
- a dot product, vector length n

If β_{j+1} is small compared to $|\alpha_j|$ and β_j, (e), (f), and (g) will be repeated once. This is found to take place in fewer than 1/4 of the steps.

(h) Analysis of the symmetric tri-diagonal matrix $\mathbf{T}_j$ which has the α's as diagonal elements and the β's as bidiagonal elements.

$\approx 80j$ scalar operations [11].

(i) For converged eigenvalues compute eigenvectors of $\mathbf{T}_j$ and then compute eigenvectors $\mathbf{x}$.

j SAXPY's with vector length n per computed $\mathbf{x}$.

Typically, the first eigenvalue will converge in 5 - 10 iterations, and 20 eigenvalues will converge in 40 - 50 iterations. For longer runs it is a good assumption that $l/2$ eigenvalues will have converged in l iterations.

The operations count for a LANC run, when the M matrix is diagonal, cfr. Section 5, includes

(1) Factorization

$nb^2/2$ operations; dot products, average vector length $b/2$

(2) Orthogonalization, total for l steps

For selective orthogonalization we have found that $\approx nl^2/5$ operations are required; equally divided between SAXPY's and dot products, each of vector length n.

For a full reorthogonalization $n(l^2-l)$ operations are required; equally divided between SAXPY's and dot products, each of vector length n

(3) l Lanczos' steps

(3a) $(2b+1)nl$ operations; dominated by SAXPY's and dot products, average vector length b (solve $(\overline{A})$)

(3b) $5/4\ nl$ operations, vector length n (mult (M))

(3c) $6nl$ operations; equally divided between dot products and scalar vector products, vector length n

(4) Analysis of tri-diagonal matrix, T, total for l steps

$\approx 40l^2$ operations; non-vectorizable

(5) Computation of $l/2$ eigenvectors

$\approx 3/8\ nl^2$ operations; estimated for the case that one eigenvalue-/eigenvector converges in each of the $l/2$ last steps. This is an overestimate, more typical would be $1/8\ nl^2$ to $1/4\ nl^2$ operations. These operations are SAXPY's with vector length n.

A comparison of operation counts shows that a LANC run with fewer than $b/4$ steps does not permit the initial cost of factorization, (1) above, to be amortized. For longer runs ($l > b/4$) the Lanczos' steps (3) will require more operations than the factorization, and then (selective) orthogonalization (2) and the analysis of T (4) become significant parts of a LANC step. Because of the poor vector length during factorization as compared with the other parts, we shall expect

to need even more steps than $b/4$ to amortize the factorization on the CYBER 205.

The operation count break-even between parts (4) and (5) above is for $n \approx 100$, but because (5) is vectorized, the two modules will have equal CPU-time on the CYBER 205 for a much higher value of n. Note also that (5) will always be less than half the cost of (2).

5. Test examples.

We used test examples that typically occur in structural dynamics. A version of the finite element program FEAP [16, Ch. 24] was converted to run on CYBER 205; this program was then used to generate stiffness matrices, A, and mass matrices, M, for the test examples. The eigenvalues, λ_i, are the squares of the frequencies of free vibration of the structures, ω_i, i.e. $\lambda_i = \omega_i^2$.

We generated a total of 4 sets of matrices, with the order of the matrices ranging from 150 to 7296 (i.e. $n = 150$ to 7296). In all 4 examples we chose to have a diagonal M. There is no loss of generality with this assumption. Our intention was only to keep the total computational cost down, and yet be able to examine large problems.

Example (i)
$n = 150$, $b = 17$
This is a simple truss structure.

Example (ii)
$n = 468$, $b = 60$
This is a structure, first presented in [2], which we have used extensively during previous testing [8].

Examples (iii) and (iv) were generated with a 3-dimensional beam element. The model is an idealization of a multistory structure, see Fig. 8. Each story had the same geometry, with $(N_x-1)N_y + (N_y-1)N_x$ elements parallel to the x- and y-axes, and also N_xN_y elements parallel to the z-axis (connecting the story to the next lower story). Of the total number of nodes, $N_xN_yN_z$, all that belonged to the bottom story were held fixed. Each node had 6 variables.

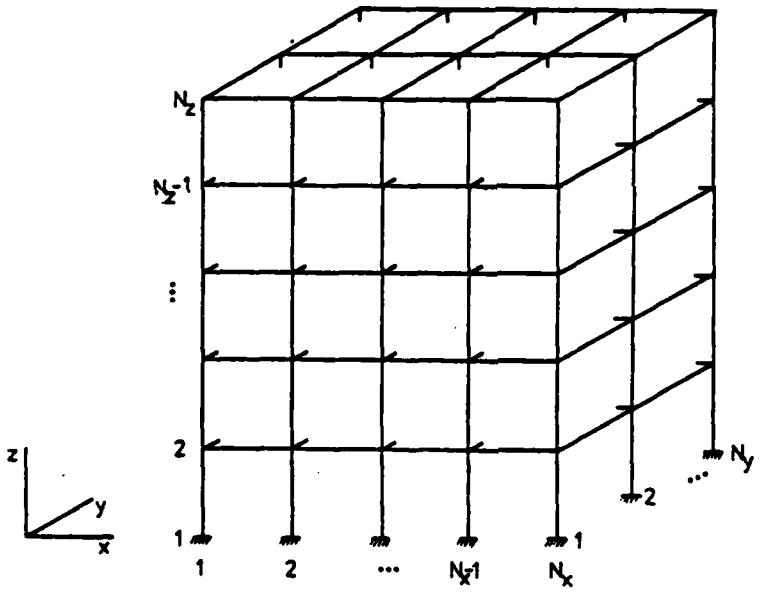


Fig. 8. Model used in examples (iii) and (iv)

Example (iii)
With $N_x = N_y = 4$, $N_z = 20$ we get $n = 1824$, $b = 95$.

Example (iv)
With $N_x = N_y = 8$, $N_z = 20$ we get $n = 7296$, $b = 370$.

The model was so constructed that whenever $N_x = N_y$, due to symmetry there was a set of double eigenvalues corresponding to the transverse vibration of the structure. Both Example (iii) and Example (iv), therefore, contain double eigenvalues.

6. Test runs.

While we ran our 4 test examples on both SUBIT and LANC, the CPU-time consumed by all major parts of the programs was measured by the CDC FORTRAN function SECOND [3], and we kept track of the step at which eigenvalues were accepted. In SUBIT we also varied the number of iteration vectors, m. In all cases a number of the smallest eigenvalues/eigenvectors were computed, i.e. those closest to the shift σ=0.

7. Results and comparisons.

7.1. Effects of vectorization.

7.1.1 Performance improvements through vectorization.

As pointed out in Section 3.1 vectorization may be achieved through an optimization option for the FORTRAN compiler or through the replacement of instructions by explicit vector instructions. Typically, the automatic vectorization will only affect quite obviously vectorizable code, e.g. "clean" DO-loops, as in Figs. 1, 3, and 5. More complicated sequences of instructions will not be vectorized through the compiler option, although they might well be vectorizable. We wanted to see how much various subroutines improved from the original scalar version to the explicit vectorized version, and made test runs under the following 4 regimes:

Condition (A): The code was as before vectorization, compiled with no optimization options.

Condition (B): The code was as before vectorization, and it was compiled with the scalar optimization option.

Condition (C): The code was as before vectorization, and it was compiled with scalar and vector optimization options.

Condition (D): The code was explicitly vectorized as outlined in Section 3, and it was compiled with scalar and vector optimization options.

First we show the CPU-times and mflop rates for the important factorization subroutine (PROFIL) as a function of the problem size, see Table 1.

Table 1. Performance of vectorized factorization subroutine PROFIL

Example	Average vector length	Operations [millions]	CPU-time [seconds]	Mflop rate
(i)	9	.022	.0148	1.49
(ii)	30	.84	.1689	5.0
(iii)	48	8.23	1.091	7.6
(iv)	185	499	24.59	20

Note that that these rates are well below the rates that we have presented in Appendix A. The reason is that PROFIL contains various conditional statements and also some scalar arithmetic (e.g. on indices) in addition to the vector operations, that will slow down the code accordingly. This is also the case for other subroutines.

The values in Table 1 are for condition D, i.e. explicitly vectorized code. Table 2 will show how much was gained in this subroutine by the various compiler options.

Table 2. Performance of factorization subroutine PROFIL under various optimization options

	Mflop rates			
Example	Cond A	Cond B	Cond C	Cond D
(i)	.53	.63	1.03	1.49
(ii)	1.04	1.64	3.5	5.0
(iii)	.94	1.70	4.4	7.6

In the following three tables we shall see better performance, due to longer vector lengths.

The M-operation, Section 4.1.1(a) and (b), was coded as one subroutine, MPROJ. For diagonal M the effort is dominated by dot products of length s. Table 3 shows that the automatic vectorization worked almost as well as our explicit vectorization for this subroutine. Note also that, based on the vector lengths given in Table 3, the performance rates for the vectorized dot

products alone are about 27, 38, 46, and 49.5 mflops, as obtained from Appendix A.

Table 3. MPROJ performance (diagonal M)

Example	Vector length	Mflop rate		
		Cond A	Cond C	Cond D
(i)	150	1.3	16	20
(ii)	468	1.4	29	32
(iii)	1824	1.4	38	39
(iv)	7296			47

Note that we could reformulate MPROJ to use SAXPY's instead of the dot products, but these SAXPY's would use much shorter vector length, i.e. m, and therefore would lead to poorer performance. From Appendix A we may infer that with m=43 the mflop rate for the vectorized MPROJ with SAXPY's would be less than 20 in all 4 examples (the SAXPY's alone would perform at about 22 mflops, but the additional work would be at scalar speed). It is the vast difference in vector lengths that makes the dot products superior to the SAXPY's in this subroutine.

The formation of the new basis in SUBIT, Section 4.1.4, was coded as a subroutine, VFORM. Table 4 shows that pure SAXPY expressions with long vectors are indeed very efficient. If VFORM were coded using dot products instead of SAXPY's, then the mflop rate would be about half (and the CPU-time about twice) the values given in Table 4.

Table 4. VFORM performance, Example (iv)

Analysis of first step, m=23 and m=63				
m	Vector length	Operations [millions]	CPU-time [seconds]	Mflop rate
23	7296	3.86	.0402	96
63	7296	28.96	.3016	96

In LANC the orthogonalization task, Section 4.2(a), may be coded as a full Gram-Schmidt orthogonalization, as in our subroutine GSORT, or as a selective orthogonalization, as in our

subroutine PRORT. We shall comment on the choice of method in Section 7.3, and at this point include performance data for GSORT, see Table 5.

Table 5. Orthogonalization performance GSORT

Example	Vector length	Mflop rate			
		Cond A	Cond B	Cond C	Cond D
(i)	150	1.9	2.1		13
(ii)	468	2.0	2.2	5.1	29
(iii)	1824	2.0		5.3	42
(iv)	7296				50

GSORT contains equal amounts of SAXPY's and dot products, all with vector length n; we should therefore expect improved performance compared with MPROJ in Table 3. It is surprising that we did not find this to be the case in Examples (i) and (ii). Table 5 also illustrates that the automatic vectorization did not vectorize perfectly vectorizable code (low performance for condition C); this is because the original GSORT contained an external reference to a general purpose dot product function, which was coded in a way the automatic vectorizer did not recognize.

7.1.2. The importance of vector length.

So far we have seen that the performance of a given subroutine improves dramatically with longer vectors. However, this improvement is not as big as the direct measurements of pure vector codes indicate. For subroutines like MPROJ, VFORM, and GSORT, that consist almost completely of vector instructions, we have observed a markedly lower performance than that presented in Appendix A. Typically, a subroutine - or a whole program for that matter - will include slow parts that will degrade the overall performance even further. Scalar operations, dot products, and/or operations with short vector lengths are examples of such slow parts. In order for the slow parts to have any significant influence on the overall performance, however, they must represent a significant fraction of the total work. Below we shall give a detailed discussion of two important subroutines that have certain operations on short vectors and other operations on long vectors. The overall performance will be inbetween what would have been expected for

the short vector length alone and for the long vector length alone.

Within SUBIT the A-projection, Section 4.1.2(a1), (a2), (a3), and (b), was coded as one subroutine, APROJ. The vector lengths in APROJ are both b (on average), during 4.1.2(a1) and (a3), and n, during 4.1.2(a2) and (b), so that for every $2nm$ vector operations with length b there are $(m^2/2 + m)$ vector operations with length n. Most of the long vector operations are dot products ($m^2/2$ as opposed to m SAXPY's). From Appendix A we may see that for large examples (i.e. large n) the mflop rate over the long vectors will be about 40 to 50, but never significantly over 50. Half of the short vector operations are efficient SAXPY's. When the short vector length b is greater than about 120, these SAXPY's will also execute at mflop rates of about 40 or more. The other half of the short vector operations are dot products which are significantly slower than the SAXPY's; e.g. with vector length 120 the dot product mflop rate is about 22.

To further illustrate the performance of APROJ we include some detailed timings of one iteration step of our largest example, see Table 6.

Table 6. Analysis of APROJ vector performance, Example (iv)

Breakdown of first iteration step, subspace size = 23				
Task	Average vector length	Operations [millions]	CPU-time [seconds]	Mflop rate
4.1.2(a1)	370	62.1	1.850	34
4.1.2(a2)	7296	.1678	.00171	98
4.1.2(a3)	370	62.1	1.067	58
4.1.2(b)	7296	1.93	.0414	47
OVERALL		126.3	2.96	43

Ignoring for a moment the slow-down effect of non-vectorized parts of APROJ we may extrapolate from Appendix A that the overall mflop rate for this subroutine cannot exceed 67, and the short vector length (b) has to be larger than 500 for the overall mflop rate to be better than 50.

The subspace size, m, will not greatly influence the performance rate, but of course the

CPU-time for tasks 4.1.2(a1), (a2), and (a3) will increase linearly with m, and the CPU-time for task 4.1.2(b) will increase quadratically with m. Table 7 contains a summary of overall performance rates for APROJ for all our examples.

Table 7. APROJ performance, vectorized code

Example	Vector lengths	Subspace size	Mflop rate
(i)	17/150	23	6.1
(i)	17/150	43	7.2
(ii)	60/468	23	15.4
(ii)	60/468	43	16.2
(ii)	60/468	63	16.9
(iii)	95/1824	23	21.2
(iii)	95/1824	43	21.9
(iii)	95/1824	63	22.4
(iv)	370/7296	23	42.4
(iv)	370/7296	43	42.6
(iv)	370/7296	63	42.0

Within LANC, see Section 4.2, the tasks (b) - (g) were coded as one subroutine, LANSIM. During task (c) there are vectors of average length b, but in the other tasks the vector length is equal to n. In an average step there are about 9 vector operations of length n (6.5 (fast) vector multiply's or SAXPY's, and 2.5 (slow) dot products) as opposed to $2n$ vector operations (n SAXPY's and n dot products) with average vector length b. Thus the shorter vector length occurs much more often than the longer vector length in typical examples, e.g. 100 times more often in Example (ii), and nearly 400 times more often in Example (iii). As a result the overall mflop rate for LANSIM will be determined by the shorter vector length, and we have in fact a situation that is quite similar to what we have described above for the APROJ subroutine in SUBIT. Again based on data in Appendix A the problem must be very large, with short vector length > 500 (if we ignore the effect of non-vectorized parts), to give an overall mflop rate in LANSIM that is higher than 50. Table 8 shows the performance that we measured for LANSIM for our more typical examples.

Table 8. Lanczos' step performance LANSIM

Example	Vector length Short b	Vector length Long n	No of steps l	Operations [millions]	CPU-time [seconds]	Mflop rate
(i)	17	150	80	.507	.1183	4.3
(ii)	60	468	100	6.00	.497	12
(iii)	95	1824	100	36.3	2.08	17
(iv)	370	7296	100	546	13.66	40

7.1.3. Modules that were not vectorised.

The solution of the small eigenproblem in SUBIT, Section 4.1.3, was written as one subroutine, GEIG, that called the set of transformation subroutines and also EISQL. The transformation modules, representing about 1/3 of the operations, were vectorized with vector lengths varying from 1 to m ($m/3$ on average). According to Appendix A dot products and SAXPY's perform equally well on CYBER 205 for the vector lengths that we used in these transformation modules in our test runs (with m = 23, 43, and 63). As a result of vectorization of the transformation modules, EISQL's fraction of the total time in GEIG was found to increase from 64% (scalar) to 74% (subspace size m=23), 80% (m=43), and 84% (m=63). The performance rate of GEIG came out between 1.9 mflops (m=23) and 2.9 mflops (m=63).

Within LANC there is also an unvectorized subroutine, ANALZT, that performs the analysis of the tridiagonal matrices, Section 4.2(h). The operations count for ANALZT that we have given there, is quite approximate. Taken literally it indicates a performance rate of about .9 - 1.5 mflops. We recorded an improvement of about 10% for this subroutine through the scalar optimization option for the compiler.

7.2. Overall SUBIT performance.

Because the performance rate is different for the different modules of SUBIT it is of interest to look at the overall performance of the method.

As explained in previous sections the SUBIT step consists mainly of 4 subroutines: MPROJ (Section 4.1.1), APROJ (Section 4.1.2), GEIG (Section 4.1.3), and VFORM (Section 4.1.4). With subspace size $m = 23$ the slowest of these modules, GEIG, represents from 14.7% (in Example (i)) to .04% (in Example (iv)) of the number of operations in each SUBIT step, but this amounts to from 45.9% (Example (i)) to .8% (Example (iv)) of the CYBER 205 CPU-time in each SUBIT step, as shown in Table 9.

Table 9. Floating point operations and CPU-times for modules in a SUBIT step [Given as percentages of whole step]

Example	m	MPROJ		APROJ		GEIG		VFORM	
		ops	time	ops	time	ops	time	ops	time
(i)									
n=150	23	13.0	3.3	48.3	46.6	14.7	45.9	23.9	4.2
b=17	43	13.1	3.2	33.0	26.5	28.8	66.0	25.1	4.3
(ii)	23	7.3	3.0	76.8	73.1	2.6	20.7	13.3	3.2
n=468	43	10.1	3.7	63.7	53.1	7.1	38.9	19.2	4.3
b=60	63	11.5	3.9	54.1	39.6	12.0	52.0	22.3	4.5
(iii)	23	5.2	2.6	84.3	89.1	.5	5.7	9.6	2.5
n=1824	43	8.0	4.1	75.3	79.0	1.4	12.9	15.3	4.1
b=95	63	9.9	5.0	68.1	69.3	2.7	20.6	19.3	5.1
(iv)	23	1.6	1.4	95.5	96.4	.04	.8	2.9	1.3
n=7296	43	2.7	2.5	92.0	93.2	.12	2.0	5.2	2.3
b=370	63	3.7	3.3	88.8	89.9	.25	3.6	7.2	3.2

Table 9 clearly shows how dominating APROJ is, and increasingly so with increasing problem size n, both in terms of number of operations and CPU-time. (Should M not be diagonal, but have a banded form similar to A, then MPROJ should be expected to take just as many operations and as much CPU-time as APROJ.)

The maxim that GEIG represents a negligible effort for medium size and large problems is no longer true on vector computers. From a coarse interpolation between the values of Table 9 we have found that while GEIG has less than 10% of the number of operations in a SUBIT step

for problem size n larger than about 200 (m=23) to 500 (m=63), its fraction of the step's CPU-time is still over 10% for much larger problems, until n is about 1000 (m=23) to 4000 (m=63).

Table 10. Floating point operations and CPU-times
Initial PROFIL vs. one SUBIT step

Example	m	PROFIL		SUBIT step	
		ops [millions]	CPU-time [seconds]	ops [millions]	CPU-time [seconds]
(i)					
n=150	23	.022	.0148	.331	.0562
b=17	43			1.104	.1914
(ii)	23	.842	.1689	1.859	.1263
n=468	43			4.504	.3327
b=60	63			8.311	.6729
(iii)	23	8.231	1.091	10.033	.4493
n=1824	43			22.12	.9647
b=95	63			37.54	1.645
(iv)	23	499.4	24.59	132.3	3.087
n=7296	43			260.1	6.027
b=370	63			400.0	9.411

In small problems the initial factorization (done by PROFIL) will be negligible compared to a SUBIT step in terms of number of operations as well as CPU-time. It is only in our largest example, Example (iv), that we find PROFIL to represent a significant portion of a typical run's number of operations and CPU-time, as is clearly seen from Table 10. Considering the fact that we usually will perform more than 10 steps of SUBIT, we therefore find this method relatively insensitive to how efficiently the factorization is done, unless the problem is very large and few eigenvalues are sought.

Our SUBIT program "accepts" an eigenvalue (in the sense that it will be counted as a converged eigenvalue) when the present approximation to the eigenvalue is closer to the previous approximation (i.e. from previous step) than 10^{-7}. The last few eigenvalues that are accepted might therefore not be very accurate, but they will continue to improve in the next iterations.

When the program finishes it is generally the case that more than 80% of the accepted eigenvalues have been found within 10^{-12} of the last previous approximation. If we are looking for a fixed number (p) of eigenvalues, the number of iterations that are required seems to be independent of problem size (n), but it is quite dependent on the number of iteration vectors (m). The following Table 11 summarizes our convergence experience for the 4 test examples using different subspace sizes. Both in Example (iii) and in Example (iv) to find 30 eigenvalues it takes 20 iterations with m=43, but 15 iterations with m=63.

Table 11. Number of converged eigenvalues in SUBIT

Subspace size	No. of steps	n=150	n=468	n=1824	n=7296
m=23	5	2	1	3	3
	10	9	12	7	11
	15	13	15	12	14
m=43	5	3	7	3	3
	10	13	25	20	14
	15	16	32	26	21
	20	25		31	30
	25	30		31	31
	30	30			
m=63	5		8	5	3
	10		32	26	15
	15		42	31	32
	20		44	38	42
	25		47	47	42
	30		47	47	50

Our results indicate that the formula $m=\min(2p, 2p+8)$, that is often used to determine the number of iteration vectors, leeds to a subspace size that is smaller than the optimum size, at least when $p>15$.

7.3. Overall LANC performance.

A LANC run consists mainly of the following subroutines: PROFIL and STPONE (Section 4.2, initializations), performed once; and GSORT (Section 4.2(a)), LANSIM (Section 4.2(b)-(g)), ANALZT (Section 4.2(h)), and VECT (Section 4.2(i)), performed repeatedly. GSORT, LANSIM,

and ANALZT are performed in each step; VECT only in those steps when an eigenvalue has converged and its eigenvector is to be computed.

Our intention was to run LANC with our latest version of selective orthogonalization. However, our selective code, PRORT, which worked perfectly on our serial computer, began to misbehave on the CYBER 205 on the large example (Example (iv)). For this reason we switched to full reorthogonalization, GSORT, for all our tests. Separately we compared the two techniques on Examples (i), (ii), and (iii) and found that PRORT required about 1/3 the CPU-time of GSORT. As we shall see (e.g. Table 12) this is not a significant advantage for PRORT since GSORT vectorizes so well and is a minor part of the LANC loop. This is in strong contrast to the situation with serial computers in which full orthogonalization comes to dominate unless the LANC runs are kept short.

Because all of GSORT, ANALZT, and VECT gradually involve more operations as the iteration progresses, whereas LANSIM has the same number of operations in every step, the relative importance of GSORT, ANALZT, and VECT will increase with the number of iterations.

Table 12 gives cumulative CPU-times for the subroutines within the LANC loop after some typical number of steps, l. The number of eigenvalues, c, that have converged to machine precision (1.42108×10^{-14}) is also given. The VECT column contains estimates on the CPU-time for the eigenvector computation, Section 4.2(5). The estimates are taken to be 1/2 of the CPU-time measured for GSORT; this is most certainly an over-estimation. The column for ANALZT in the largest example also contains estimates, taken to be the time for $40l^2$ operations at 1 mflops. The results are presented separately for Examples (i), (ii), (iii), and (iv).

In small problems the unvectorized ANALZT will take a significant portion of the CPU-time in the LANC loop. Even in Example (iii) ANALZT takes about as much time as the orthogonalization subroutine, GSORT, while there are about 45 times as many operations in GSORT. After 44 LANC steps (Example (iii)) ANALZT has consumed about 7% of the LANC loop time; its share rises to about 15% after 100 steps, and it will eventually take more time than LANSIM (after some 400 steps). If on a scalar computer we can achieve 1 mflops for all operations, we can

Table 12. Cumulative CPU-times for subroutines in the LANC loop
All times are in seconds

Example	l	c	GSORT	LANSIM	ANALZT	VECT
(i)	10	3	.0020	.0135	.0070	(.0010)
n = 150	20	5	.0062	.0287	.0177	(.0031)
b=17	40	17	.0282	.0613	.0623	(.0141)
	80	40	.0771	.1183	.2575	(.0386)
(ii)	10	1	.0013	.0453	.0060	(.0007)
n = 468	20	5	.0058	.0963	.0182	(.0029)
b=60	40	15	.024	.196	.054	(.012)
	80	47	.099	.396	.188	(.050)
	100	54	.156	.497	.271	(.078)
	150	69	.352	.748	.545	(.176)
(iii)	10	2	.0034	.1906	.0074	(.0017)
n = 1824	20	5	.0153	.4053	.0197	(.0077)
b=95	44	20	.081	.987	.082	(.040)
	100	46	.418	2.080	.362	(.209)
(iv)	10	2	.0115	1.241	(.0040)	(.0058)
n = 7296	20	3	.053	2.621	(.016)	(.027)
b=370	40	17	.226	5.382	(.064)	(.113)
	60	26	.520	8.142	(.144)	(.260)
	80	59	.934	10.902	(.256)	(.467)
	100	72	1.469	13.664	(.400)	(.735)

expect ANALZT to catch up with LANSIM at a rate that is about 17 times slower than this, cfr. Table 8. GSORT would have been slowed down by a factor of 42, and would have taken about half of the time taken by LANSIM after 100 LANC steps with Example (iii) on this fictitious computer.

The full LANC run also includes initializations, most of which is the factorization done by PROFIL. Table 13 shows that the inititialization cost may be quite substantial compared to the cost of the LANC loop; e.g. in Example (iii) first after some 60 steps does the LANC loop take more CPU-time than the initialization, and in Example (iv) this does not happen in 100 steps.

For this method as well the number of converged eigenvalues seems to be independent of problem size (n). For all examples about 30 steps were necessary for the first 10 eigenvalues to converge. Table 14 shows the exact step when each of the 10 first eigenvalues converged to machine precision. However, the eigenvalues will not necessarily be found in strictly increasing

Table 13. Initialization vs. cumulative LANC steps
Floating point operations and CPU-time

Example	No. of steps	Initialization ops [millions]	Initialization CPU-time [seconds]	LANC steps ops [millions]	LANC steps CPU-time [seconds]
(i)	10	.027	.0211	.090	.0235
n = 150	20			.233	.0557
b=17	40			.678	.166
	80			2.20	.492
(ii)	10	.900	.193	.675	.0533
n = 468	20			1.50	.159
b=60	40			3.59	.286
	80			9.55	.733
	100			13.42	1.00
	150			25.7	1.82
(iii)	10	8.58	1.50	3.90	.204
n = 1824	20			8.34	.448
b=95	44			21.3	1.19
	100			63.9	2.44
(iv)	10	505	26.6	55.7	1.26
n = 7296	20			114	2.72
b=370	40			236	5.79
	60			367	9.07
	80			507	12.56
	100			655	16.3

Table 14. Initial convergence in LANC

Eigenvalue no.	Iteration no. when eigenvalue converges Example (i)	Example (ii)	Example (iii)	Example (iv)
1	6	9	6	5
2	8	16	9	10
3	9	17	11	12
4	15	18	12	21
5	18	20	18	21
6	21	22	21	22
7	21	23	26	22
8	24	24	27	24
9	25	27	27	25
10	29	29	29	26

order. This is particularly the case when there are double eigenvalues, as in Example (iii) and Example (iv). E.g. in Example (iii), when 30 eigenvalues have converged, they are in fact found to be all of λ_1 to λ_{28}, then λ_{29} through λ_{30}, and λ_{34}. The "missing" eigenvalue, λ_{27}, is equal to λ_{28}, and it was found in a few more steps (along with λ_{31}, λ_{32}, λ_{33}, and λ_{37}). In practice it is reassuring to do one extra factorization simply to check that there are no missing eigenvalues among the ones which have been computed. This represents an added cost of $nb^2/2$ operations; that has not been included in our tables or figures.

7.4. Comparisons.

Examples (i), (ii), and (iii) could be solved within the available CYBER 205 primary storage, and a comparison of the two methods may then justly consider only the CPU-time that was needed for the computation. Fig. 9, 10, and 11 show a graph of the total CPU-time vs. the number of converged eigenvalues for these examples.

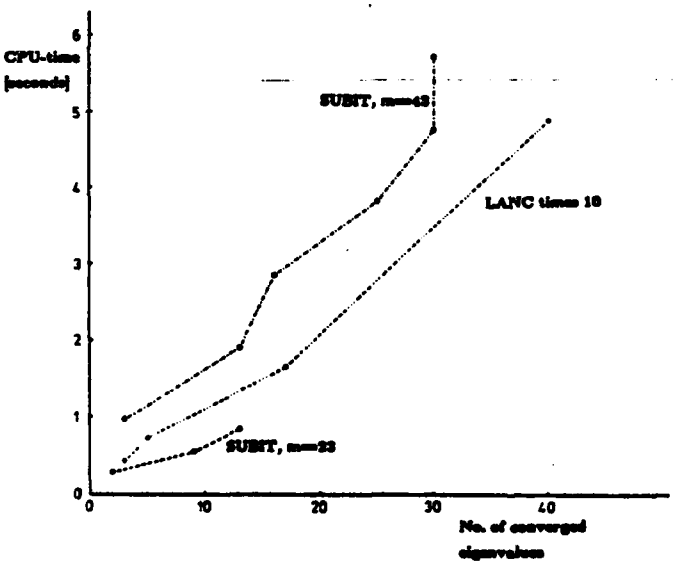


Fig. 9. SUBIT vs. LANC, Example (i), $n=150$

Note that the CPU-times for LANC have been multiplied by 10 in Figs. 9, 10, and 11. We feel it fair to say that LANC is an order of magnitude faster than SUBIT on these examples, but

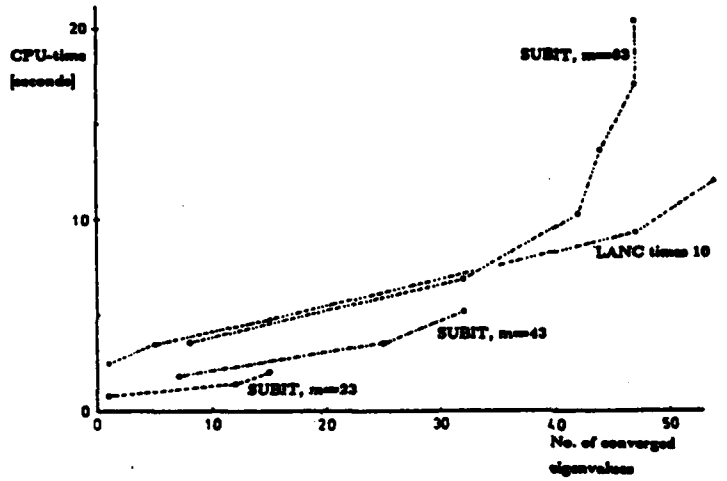


Fig. 10. SUBIT vs. LANC, Example (ii), $n=468$

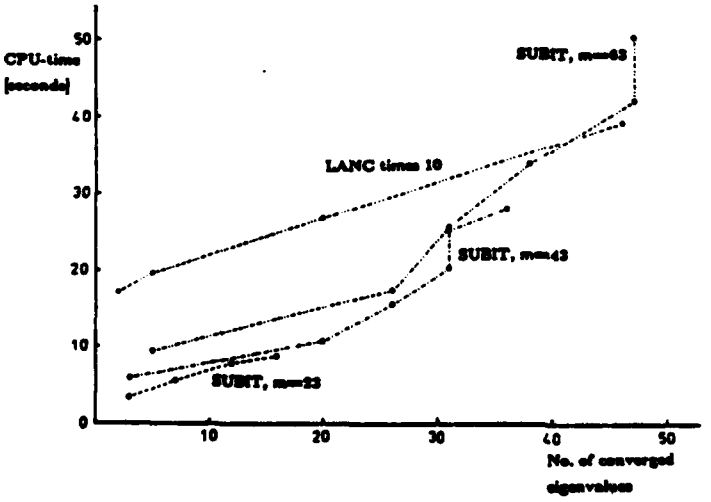


Fig. 11. SUBIT vs. LANC, Example (iii), $n=1824$

the trend is for SUBIT to gain relative to LANC as the problem size n increases. This is of course due to the fact that the cost of the initial factorization is more dominant in LANC. It is also to be seen that SUBIT is at its best when very few eigenpairs are sought; then a small

number of iteration vectors (m) is sufficient. LANC is definitely superior also in these cases, although not by an order of magnitude.

Note also that we are comparing the computation of the same fixed number of eigenpairs, up to about 50, irrespective of problem size n. But it may be more common to look for more eigenvalues in a larger example than in a small one. From the tendencies that we have observed, e.g. in Figs. 9 - 11, we may argue that when we are looking for a number of eigenpairs that is the same fixed fraction of the problem size, SUBIT is no longer seen to gain relative to LANC with increasing problem size. However, we do not want to stretch this argument too far, because the best way to compute a large number of eigenpairs (with either method) will ordinarily include the use of repeated shifts and factorizations, which is not being discussed in the present report.

Our largest example, Example (iv), required almost 3 million words to store the A (or L) matrix. Because a typical step both in SUBIT and in LANC involves 2 passes through the L matrix and there are less than 2 million words of primary storage, it is evident that extensive swapping did take place during the course of the programs. Although it is possible to insert commands in the program that may advise the paging system about pages that can be removed from the primary storage and pages that are needed in the primary storage, we relied solely on the standard scheduling.

Fig. 12 shows the comparison of SUBIT and LANC as far as CPU-time is concerned for Example (iv). To compute 30 eigenpairs in this case required about 4 times as much CPU-time in SUBIT as in LANC.

Table 15 summarizes some typical runs of Example (iv) and compares the cost of input/output to the cost of computation. l is the number of iteration steps, m is the subspace size (in SUBIT), LPF is the number of large page faults, "Penalty" is the SBU corresponding to the large page faults, and "Comp.time" is the total CPU-time for the computation. The load module for both programs was about 61 large pages.

When we are searching for quite few eigenvalues, SUBIT may be run with a small subspace (m) and get away with fewer page faults per iteration step. In LANC it is seen that the number

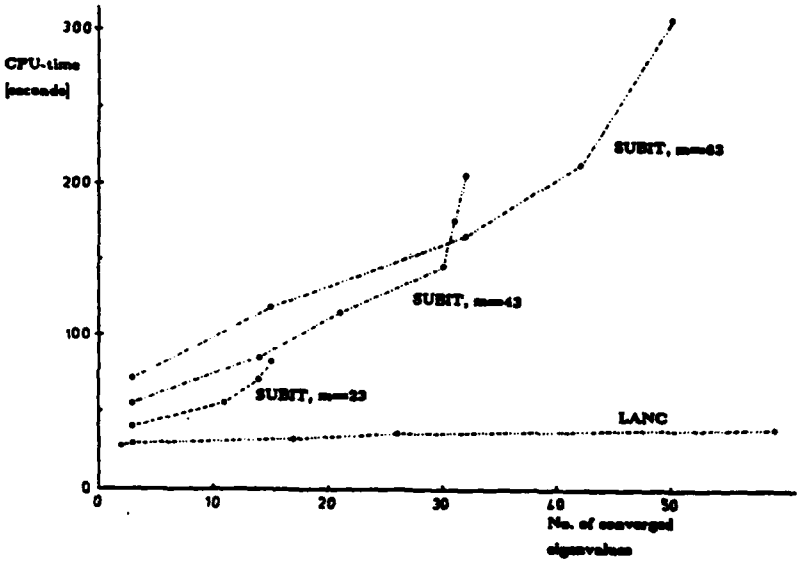


Fig. 12 SUBIT vs. LANC, Example (iv), n=7296

Table 15. Page fault penalties, Example (iv)

Method	l	m	LPF	LPF/l	Penalty [SBU]	Comp.time [CPU-sec]
SUBIT	10	23	1036	55	161.6	83.3
	30	43	2010	67	313.6	206.6
	30	63	2251	75	351.2	308.1
LANC	20		885	45	138.1	29.5
	40		1710	43	266.8	32.4
	60		2624	44	409.3	35.5
	80		3629	45	566.1	38.7
	100		4714	47	735.4	41.9

of page faults increases slightly in the later steps of an iteration. This is due to the fact that more vectors will take part in the orthogonalization in a later step. Because the first steps of LANC only involves a few vectors, whereas in SUBIT all vectors participate in each iteration, LANC requires fewer page faults than SUBIT.

Essentially it is the 2 passes through L that lead to the about 70 large page faults in each step of SUBIT and the about 45 large page faults in each step of LANC. The penalty for these

page faults is larger than the CPU-time for the compatation. (And when M is not diagonal, we shall expect a substantial increase in the number of page faults, because we shall also need to make a pass through M.) Because SUBIT works on several vectors at the same time and converges to a required number of eigenvalues and eigenvectors in fewer steps than does our simple LANC, SUBIT may perform better than (simple) LANC in cases like Example (iv), where the problem is too big for the primary storge and extensive paging takes place.

It is possible to work with more than one vector in each step of the Lanczos method, using Block Lanczos [14] (called BLANC hereafter). When the block size is s (i.e. s vectors), each BLANC step will involve about s times as many operations and s times as much CPU-time as a step of LANC. At the same time the number of steps required for a certain number of eigenpairs to converge in BLANC will be only a fraction (roughly $1/s$) of that in LANC, at least for fairly long runs. The net effect is that the computation's total CPU-time will remain about the same. However, the overall number of page faults with BLANC will be greatly reduced (by a factor of s) compared to LANC. This is because still only two passes through L are needed in each step of the algorithm. For the example in Table 15 it is seen that already with $s=2$ BLANC would have had page fault penalties comparable with SUBIT, and a larger block size (s) would have reduced the number of page faults even further.

When BLANC is used for finding very few eigenpairs and/or with a large block size, the number of steps is reduced by a factor that is less than s, and the total CPU-time will increase compared with simple LANC. Further, the saving in the number of page faults will be smaller because of the increase in the number of steps.

We conclude that for problems that are too big for the primary storage, BLANC would be more efficient than LANC because of the smaller penalties for page faults. Both LANC and BLANC have smaller CPU-time than SUBIT. The number of page faults occuring in LANC is greater than that for SUBIT. With a proper choice of block size BLANC will remain at the same low level for the CPU-time as simple LANC, and at the same time have fewer page faults than SUBIT.

At present we are experimenting with different ways of adapting ANALZT to block tridiagonal matrices.

8. Acknowledgement.

We want to thank Professor R.L. Taylor for valuable discussions and for making a copy of the FEAP program available to us.

References

[1] K.J. Bathe and E.L. Wilson, *Numerical Methods in Finite Element Analysis* (Prentice-Hall, Englewood Cliffs, N.J., 1977).

[2] K.J. Bathe and E.L. Wilson, Large Eigenvalue Problems in Dynamic Analysis, *A.S.C.E., Journal of Engineering Mechanics Division* **99**, (1973) 467-479.

[3] *CDC CYBER 200 FORTRAN Version 2 Reference Manual*, Publication No. 60485000 (CDC, 1981).

[4] *CDC CYBER 200 MODEL 205 Technical Description* (CDC, 1980).

[5] T. Ericsson and A. Ruhe, The Spectral Transformation Lanczos Method for the Numerical Solution of Large Sparse Generalized Symmetric Eigenvalue Problems, *Math. Comp.* **34**, (1980) 1251-1268.

[6] B.S. Garbow et al., *Lecture Notes in Computer Science, Volume 51* (Springer-Verlag, 1977).

[7] M.J. Kascic jr., *Vector Processing on the CYBER 205* (CDC CYBER 205 Workshop, Fort Collins, Col., February 1983).

[8] C. Lanczos, An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators, *J. Res. Nat. Bur. Standard* **45**, (1950) 255-282.

[9] B. Nour-Omid, B.N. Parlett, & R.L. Taylor, Lanczos versus Subspace Iteration for Solution of Eigenvalue Problems, *International Journal for Numerical Methods in Engineering* **19**, (1983) 859-871.

[10] B.N. Parlett, *The Symmetric Eigenvalue Problem* (Prentice-Hall, Englewood Cliffs, N.J., 1980).

[11] B.N. Parlett and B. Nour-Omid, The Use of Refined Error Bounds when updating Eigenvalues of Tridiagonals, *Tech. Rep. PAM-175* (Center for Pure and Applied Mathematics, University of California, Berkeley, 1983).

[12] *Price Schedule for CYBER 205 Services* (CSU, Computer Center, Fort Collins, Col., 1982).

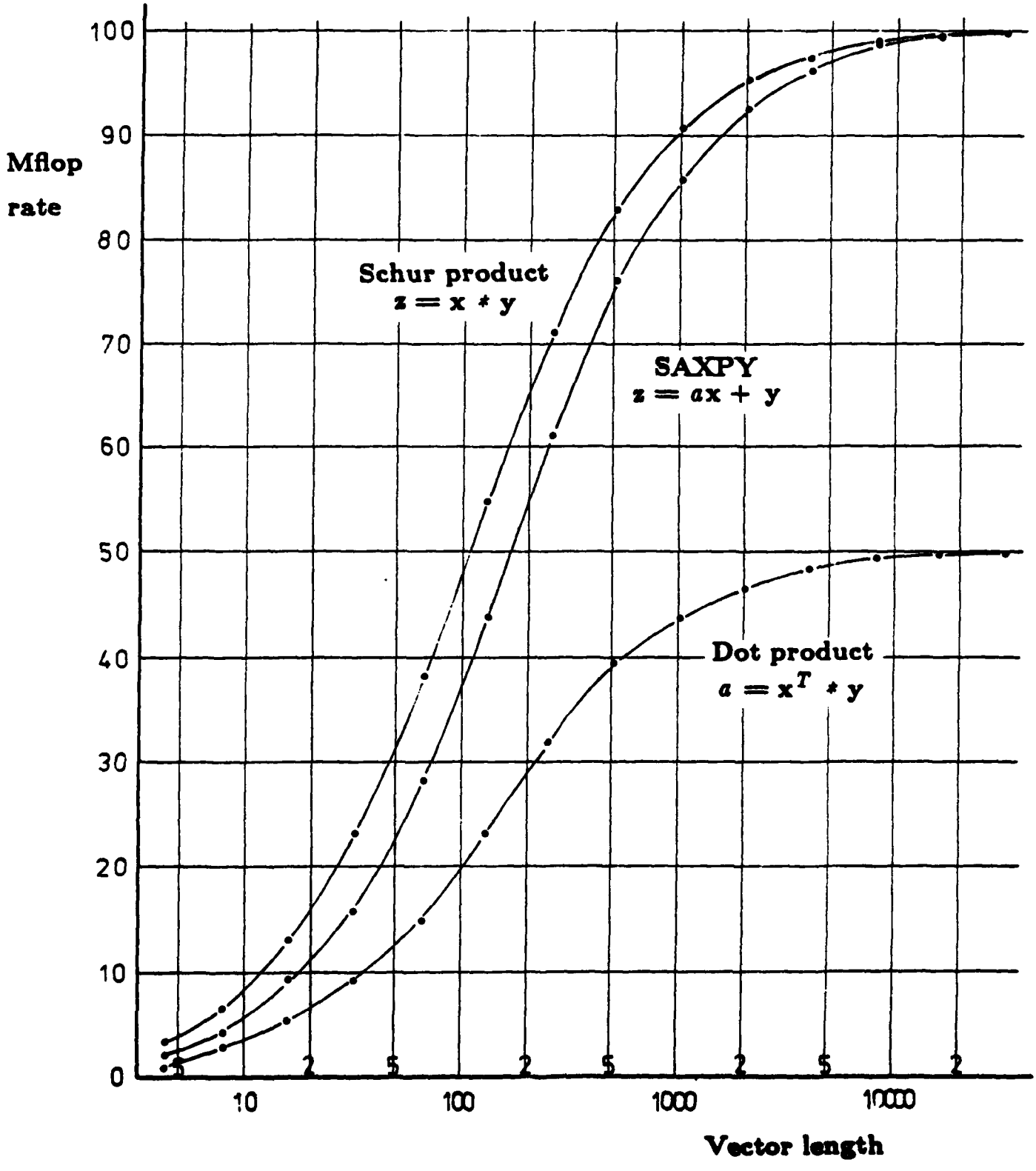
[13] H. Rutishauser, Simultaneous Iteration Method for Symmetric Matrices, in J.H. Wilkinson and C.H. Reinsch, Eds., *Handbook for Automatic Computation (Linear Algebra)* (Springer-Verlag, New York, 1971) 284-302.

[14] D.S. Scott, Block Lanczos Software for Symmetric Eigenvalue Problems, *Report ORNL/CSD-48,UC-32* (Union Carbide Corporation, 1979).

[15] H.D. Simon, The Lanczos Algorithm for Solving Symetric Linear Systems, *Tech. Rep. PAM-74* (Center for Pure and Applied Mathematics, University of California, Berkeley, 1982).

[16] O.C. Zienkiewics, *The Finite Element Method (3rd ed.)* (McGraw-Hill, London, 1977).

Appendix A-1


Mflop rate
100
90
80
70
60
50
40
30
20
10
0
Schur product
z = x * y
SAXPY
z = ax + y
Dot product
a = x^T * y
10
100
1000
10000
Vector length

Appendix A-2

The megaflop rates that have been plotted in the diagram in Appendix A-1, were obtained by the following program. Note that for each n 500 vector operations (dot product, Schur product, and SAXPY) were measured by the SECOND function.

```
      PROGRAM LOOP (TAPE6=OUTPUT)
      DIMENSION X(32768), Y(32768), Z(32768), T(3), P(3), TT(4)
      X(1;32768) = 1.
      Y(1;32768) = 2.
      N = 32768
C
100   T(1;6) = 0.
      TT(1) = SECOND()
      DO 300 I = 1, 500
          A = Q8SDOT (X(1;N),Y(1;N))
300   CONTINUE
      TT(2) = SECOND()
      DO 400 I = 1, 500
          Z(1;N) = X(1;N) * Y(1;N)
400   CONTINUE
      TT(3) = SECOND()
      DO 500 I = 1, 500
          Z(1;N) = A*X(1;N) +  Y(1;N)
500   CONTINUE
      TT(4) = SECOND()
      DO 600 J = 1, 3
          T(J) = TT(J+1) - TT(J)
600   CONTINUE
      P(1;3) = 5.E-4 * N / T(1;3)
      WRITE (6,1000) N,T,P
1000  FORMAT (1X,2HN=,I5,3F10.6,3F6.2)
      N = N/2
      IF (N.GT.1) GOTO 100
C
      STOP
      END
```

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: 218	2. GOVT ACCESSION NO.: AD-A142942
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Effect of Cyber 205 on Methods for computing Natural Frequencies of Structures	5. TYPE OF REPORT & PERIOD COVERED: Unclassified
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): J. Natvig, B. Nour-Omid, and B.N. Parlett	8. CONTRACT OR GRANT NUMBER(s): N00014-76-C-0013
9. PERFORMING ORGANIZATION NAME AND ADDRESS: University of California, Berkeley, CA 94720	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE: March, 1984
	13. NUMBER OF PAGES: 37
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We consider the generalized eigenvalue problem, $(A - \lambda M)x = 0$, where A and M are large, sparse, symmetric matrices. For large problems finding only a few eigenpairs involves a major computational task. In a typical example from structural dynamic analysis with matrices of order 8000, $O(10^9)$ operations are required to compute 50 eigenpairs. It is therefore interesting to examine the advantage that vector computers such as CYPBER 205 can offer.

We adopted our best versions of the Subspace Iteration Method and the simple

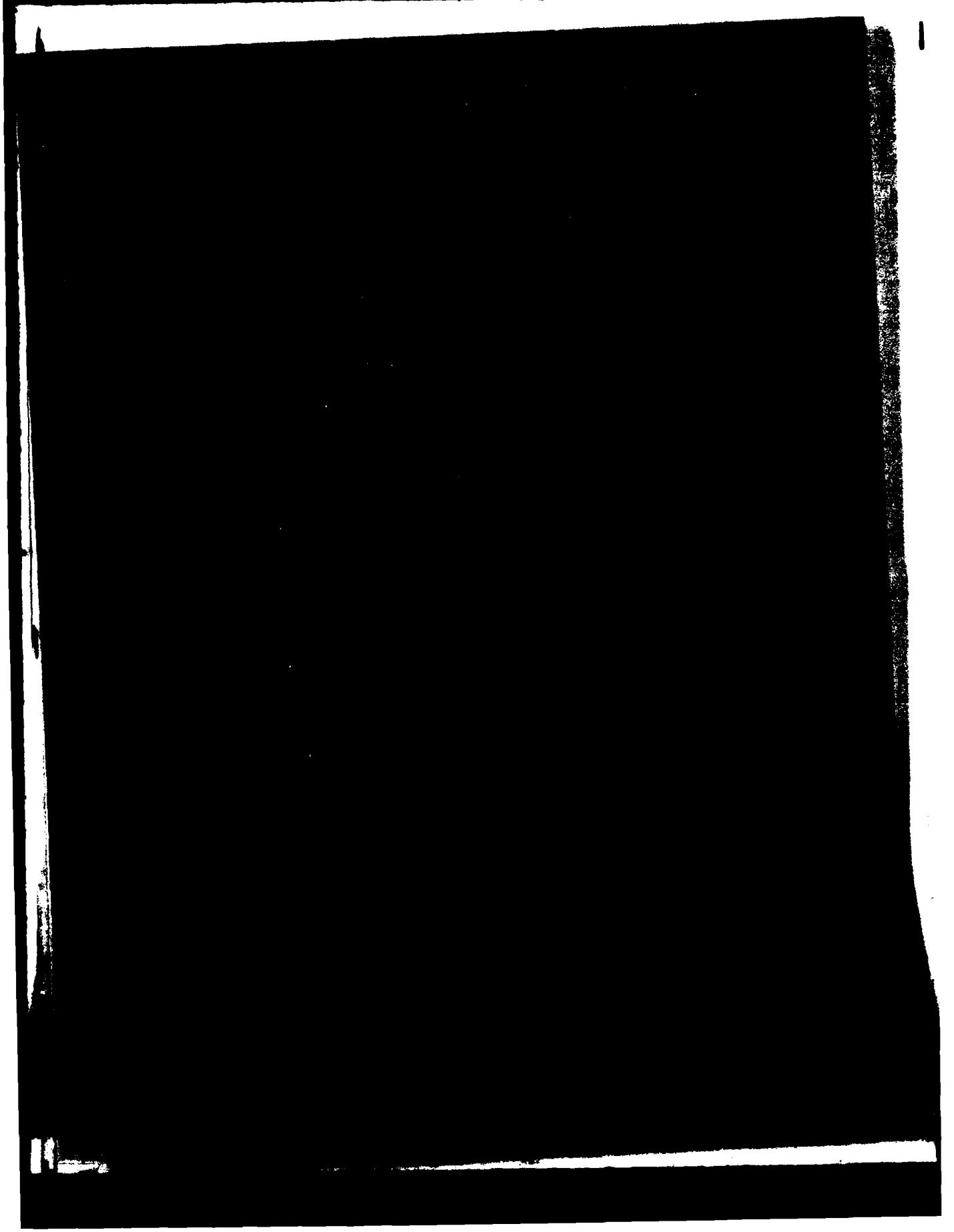
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Lanczos Method in order to take advantage of the special vector processor of the CYBER 205. Both techniques lend themselves to vectorization. Our extensive comparisons support the following general statements. Both methods require the triangular factorization of the same large n x n matrix. This factorizstion dominates the total computation as $n \to \infty$ provided that the number of wanted eigenpairs, p, remains fixed (independent of n). However, simple Lanczos is at least an order of magnitude more eifficient (in CPU-time) for the remainder of the computation. For p=40, n=500 the factorization time is not important and the full order of magnitude difference is seen in the total CPU-time. When p=40, n=8000 simple Lanczos is only 4 times faster than Subspace Iteration on the CYBER 205. This confirms experience on serial computers.

For problems that cannot fit into primary storage, input-output becomes increasingly important. We found that the cost of input/output dominated ober the CPU-cost for a problem that required twice the available primary storage on our CYBER 205. However, this will depend on the billing algorithm of the computer center. We conclude that problems which have a substantial overhead in reading and writing the matrices, should not be solved by the simple Lanczos Method, but by a Block Lanczos Method.

machine precision. However, the eigenvalues will not necessarily be found in strictly increasing




DAT
ILM