Basic Transformations in Linear Algebra

for

Vector Computing

BASIC TRANSFORMATIONS IN LINEAR ALGEBRA FOR VECTOR COMPUTING

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof. dr. S. K. Thoden van Velzen,
in het openbaar te verdedigen in de Aula der Universiteit
(Oude Lutherse Kerk, ingang Singel 411, hoek Spui),
op vrijdag 19 mei 1989 te 15.00 uur

door

Walter Hoffmann

geboren te Haarlem

1989

Promotor: prof. dr. Th. J. Dekker

Faculteit: Wiskunde en Informatica

Printed at the Centrum voor Wiskunde en Informatica, Amsterdam.

Cover: part of arithmetic circuit board CDC CYBER 205, courtesy SARA.
Photograph: Gerhard Jaeger.
Graphic design: R. T. Baanders.

Aan de nagedachtenis van mijn vader

Morgen

Und morgen wird die Sonne wieder scheinen
Und auf dem Wege, den ich gehen werde,
Wird uns, die Glücklichen, sie wieder einen
inmitten dieser sonnenatmenden Erde...

Und zu dem Strand, dem weiten, wogenblauen,
Werden wir still und langsam niedersteigen,
Stumm werden wir uns in die Augen schauen,
Und auf uns sinkt des Glückes stummes Schweigen...

John Henry Mackay

Table of Contents

	Preface	viii
	Acknowledgement	ix
	Introduction	1
Chapter I;	Solving linear systems on a vector computer	15
Chapter II;	Rehabilitation of the Gauss-Jordan algorithm (coauthor: T.J. Dekker)	33
Chapter III;	An estimate for the spectral norm of the inverse of a matrix with the Gauss-Jordan Algorithm	43
Chapter IV;	A fast variant of the Gauss-Jordan Algorithm with partial pivoting	53
Chapter V;	Iterative Algorithms for Gram-Schmidt Orthogonalization	61
Chapter VI;	Definition and use of Householder reflections	77
Chapter VII;	NUMVEC FORTRAN Library manual Chapter: Simultaneous Linear Equations (NM-R8614) (coauthor: W.M. Lioen)	89
Chapter VIII;	NUMVEC FORTRAN Library manual Chapter: Simultaneous Linear Equations (NM-R8712)	127
Chapter IX;	NUMVEC FORTRAN Library manual Chapter: Simultaneous Linear Equations (NM-R8903) (coauthor: K. Potma)	137
	Samenvatting	158

PREFACE

The research reported in this thesis, was started in the fall of 1984. In that year a Cyber 205 vector computer was installed at the Academic Computing Centre, SARA, in Amsterdam. By this, the access to super computers became more easy for scientists in the Netherlands, so that 'vector computing' could be experienced in practice on a larger scale.

Some early experiments on the Cyber 205 made clear that existing software for linear algebra problems was not optimal and could be improved.
In that same period, the 'Centrum voor Wiskunde en Informatica', CWI, decided to build a numerical analysis subroutine library, NUMVEC, for use on super computers. Subroutines in NUMVEC are based on algorithms that are available in the open scientific literature or on algorithms that are especially developed for that purpose.

The research group guided by professor Th. J. Dekker has experience in the development of numerical software. This is illustrated by major contributions to CWI's Algol 60 library NUMAL and by contributions to the NAG libraries. The NUMVEC initiative of CWI stimulated the extension of the research effort to portable numerical software for supercomputers. With the Cyber 205 super computer in the vicinity, it is only natural that routines for use on this machine were developed initially.

The combined knowledge of numerical linear algebra and numerical software formed the basis for this thesis.

Amsterdam, February 1989

ACKNOWLEDGEMENTS

My interest in numerical analysis was aroused by the pioneer of Computer Science in the Netherlands, the late prof. A. van Wijngaarden. His courses at the University of Amsterdam are still vividly remembered.

My teacher and adviser in numerical linear algebra was my current promotor, prof. Th. J. Dekker. I wish to thank him for the things he taught to me; also otherwise I have learned to know him as a man of great integrity.

I have had the privilege of being a guest of prof. G. H. Golub at the Computer Science division of Stanford University during a full Academic year. I wish to express my thanks for all I learned of him in that period. The opportunity of spending almost a year with my family in California was also made possible by a grant of the Netherlands Organization for the Advancement of Pure Research (Z.W.O.) .

From more recent times, I wish to thank prof. P. J. van der Houwen who has acted as a stimulating force on specific essential moments; I also wish to thank prof. H. A. van der Vorst for his contributions to this research and dr. P. van Emde Boas for always showing interest in my work.

I wish to thank IBM Nederland for supporting a very much appreciated working visit to ECSEC, the IBM research Centre at 'Roma, la Città eterna' in October 1985. There, I developed a number of ideas which are included in this thesis. At ECSEC, which is headed by the stimulating science manager dr P. Sguazzero, I met three french scientists, prof. Y. Robert, prof. M. Cosnard and dr. D. Trystram. I gladly acknowledge their support by introducing me to the IBM computer system and I thank them for sharing some great moments.

I like to express my great appreciation to Peter de Rijk for his skill in exploring new available computing systems and more specifically for his support in the development of our Cyber 205 programs. Furthermore I like to thank Walter Lioen for his help in coding some of our earlier routines and Te Yung Fu for his work to facilitate data communication and data processing. I am grateful to D. Zwarst for fitting me in his tight printing schedule and to him and his colleagues of the Centrum for Wiskunde en Informatica for a fine result.

Finally, I like to thank Marianne and Niels for sustaining the harmonic surroundings which enabled me to continue this work.

Ja, mach nur einen Plan
Sei nur ein grosses Licht!
Und mach dann noch 'nen zweiten Plan
Gehn tun sie beide nicht.

Bertolt Brecht,
Dreigroschenroman.

And it never failed that during the
dry years the people forgot about the
rich years, and during the wet years
they lost all memory of the dry years.
It was always that way.

John Steinbeck,
East of Eden.

INTRODUCTION

This thesis contains nine chapters which can be divided into two parts of six and three chapters, respectively. The first six chapters are papers which either have been published, have been accepted for publication, or have been submitted for publication in scientific journals. They deal with problems in the area of numerical linear algebra.
The last three chapters are a part of the user documentation for the NUMVEC FORTRAN library [14] and have been published as CWI reports. The part of the documentation presented here, describes the use of routines which are based on algorithms presented in the first six chapters of this thesis.

The six papers in the first part have in common that algorithms for solving numerical linear algebra problems are presented which are especially designed to enable a translation into efficient routines for vector computers. This has been realized by analyzing the underlying mathematical structure in order to formulate these algorithms in terms of matrices and vectors, instead of scalars. Special care has been taken that no demands for numerical stability were violated. We think that these algorithms are also suitable to serve for an efficient implementation on parallel machines with shared memory.
We did not pursue the alternative way of algorithm design where an existing sequential algorithm is transformed, more or less automatically, into an algorithm with an optimal loop structure.

To furnish the reader with a proper context for appreciating the papers in this thesis, we present some background material from the area of numerical linear algebra and computer science; more details can be found in several text books [8,9,12,15]. This background material is covered in two sections; the first emphasizes basic matrix transformations and the second emphasizes vector computing for linear algebra.

1.1 Applicable elementary matrices in numerical linear algebra

The basic transformations as mentioned in the title of this thesis are, generally speaking, those transformations which are described by *elementary matrices*. A matrix is called elementary if it is the sum of the identity and a rank-one matrix.
So, for any two vectors $f = (\phi_1, \ldots, \phi_n)^T$ and $s = (\sigma_1, \ldots, \sigma_n)^T$ and a scalar μ, the matrix $E = (I + \mu fs^T)$ is elementary.

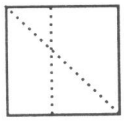
This type of matrices is used in many methods for solving matrix problems. For instance, all direct methods for solving square linear systems can be described in terms of elementary matrices; a QR factorization of a matrix (being itself an important tool for solving matrix problems) can be described in terms of elementary matrices; similarity transformations (used in the context of eigenvalue problems) can be constructed using elementary matrices.

Among basic transformations are also transformations that can be described by low-rank matrices, such as projections onto subspaces of low dimension, and planar rotations.

We consider several types of elementary matrices.

Type-I matrices

Matrices that for unit vector e_k and vector f with $\phi_k = 0$ are defined by:



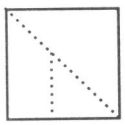
$$E = (I - fe_k^T).$$

This type of matrices is used in Gaussian elimination and in the Gauss-Jordan method; they have the following nice property:

$$E^{-1} = (I + fe_k^T).$$

The process of Gaussian elimination, as is taught to every science student in an early linear algebra course, is based on the fact that the solution of a linear system doesn't change if a multiple of an equation is subtracted from another equation. This is equivalent with subtracting corresponding rows in the coefficient matrix and can be expressed by a premultiplication of the coefficient matrix with an elementary matrix of the type considered here.

Elementary matrices in Gaussian Elimination
In Gaussian elimination, only multiples of rows are subtracted from rows that are located lower in the matrix (their row-number is higher). Consequently, the elementary matrices $(I - fe_k^T)$ are such that $\phi_i = 0$ for $i = 1, \ldots, k$.



Using this type of matrices, Gaussian elimination can be described by constructing $(I - f_1e_1^T)$, $(I - f_2e_2^T)$, ..., $(I - f_{n-1}e_{n-1}^T)$, in such a way that the matrix product

$$U = (I - f_{n-1}e_{n-1}^T)(I - f_{n-2}e_{n-2}^T) \ldots (I - f_1e_1^T)\, A$$

has upper triangular form. This relation is usually written as:

$$A = (I - f_1e_1^T)^{-1} \ldots (I - f_{n-2}e_{n-2}^T)^{-1}(I - f_{n-1}e_{n-1}^T)^{-1}\, U.$$

Because of the special structure of the elementary matrices under consideration, we observe that this identity can be rewritten as:

$$A = (I + f_1e_1^T + f_2e_2^T + \ldots + f_{n-2}e_{n-2}^T + f_{n-1}e_{n-1}^T)\, U,$$

and again from the structure of the vectors f_k , we see that the first factor at the right hand side equals a lower triangular matrix L (say) so that the identity

$$A = L\,U$$

holds.

Now we see that the problem of solving a linear system Ax = b, can be divided into the following steps:

a. Calculate a factorization A = LU of the coefficient matrix A.
b. Solve L y = b for vector y; this is called *forward substitution.*
c. Solve U x = y for vector x; this is called *backward substitution.*

It is well-known that almost always the given linear equations need to be reordered by interchanging selected equations. Such a reordering is for instance necessary if the first unknown in the first equation has a coefficient that is equal to zero. If the computations are carried out in finite precision arithmetic, then reordering of the equations is necessary for numerical stability; the error in the calculated solution depends on the ordering of the equations.

Our first paper summarizes variants of constructing an LU-factorization, including various ways of properly reordering the equations and/or unknowns. A strategy for finding a suitable reordering is called a *pivoting strategy.* If only rows or columns are interchanged, then we speak of *partial pivoting*; in *complete pivoting* column and row interchanges are involved.

The calculated solution by means of LU decomposition followed by forward and backward substitution, can be seen as the exact solution of a (slightly) perturbed problem. A bound on the perturbation depends on the matrix and the pivoting strategy. This is expressed in the following theorem, due to Wilkinson [16]:

Theorem:
Suppose that the linear system A x = b has been solved with floating-point arithmetic in finite precision, using LU decomposition with partial- or complete pivoting. The calculated solution is the exact solution of a problem

$$(A + E)\,\hat{x} = b \text{ , with } \| E \|_\infty \le 8\, n^3\, g(A)\, \| A \|_\infty\, \varepsilon + O(\varepsilon^2) \text{ ,}$$

where ε denotes the machine precision ($= \min_\rho \{ fl(1 + \rho) > 1 \}$) and g(A) the growth factor which indicates the maximal value during any stage of the process, defined by $g(A) = \max_{(i,j,k)} |a_{ij}^{(k)}| \,/\, \max_{(i,j)} |a_{ij}|$.
A proof can be found in [8,15]. □

If partial pivoting is used then g(A) is bounded by 2^{n-1} .
If complete pivoting is used then g(A) is bounded by a function ω, defined by $\omega(n) = n^{1/2}(2.3^{1/2} \ldots n^{1/(n-1)})^{1/2}$.
Compared with 2^{n-1}, ω is a slowly growing function. This can for instance be seen from the numerical values $\omega(50) \approx 570$, $\omega(100) \approx 3570$ and $2^{100} \approx 10^{30}$.
The following question arises:
In what way does the size of $\| E \|_\infty$ influence $(\hat{x} - x)$, i.e. the error in x ?
This question is answered in the following theorem:

Theorem:
Suppose that errors in A and b are given by ΔA and Δb, respectively, and assume that ΔA is such that the product r, defined by $r = \| \Delta A \| . \| A^{-1} \|$, satisfies $r < 1$. In that case, the perturbed matrix $(A + \Delta A)$ is not singular and for any $\delta > 0$, being a simultaneous upperbound in the relative errors such that:

$$\| \Delta A \| \leq \delta \| A \| \quad \text{and} \quad \| \Delta b \| \leq \delta \| b \|,$$

the solutions x and y of $Ax = b$ and $(A + \Delta A) y = b + \Delta b$, respectively, satisfy:

$$\frac{\|x - y\|}{\|x\|} \leq 2\delta \frac{\|A\| . \|A^{-1}\|}{1 - r}$$

A proof can again be found in [8,15].

□

Elementary matrices in Gauss-Jordan
The elimination process as globally described above, can also be carried out in parts of the columns that are above the diagonal. In that case, the result of the elimination will be a diagonal matrix D, instead of an upper triangular matrix U. The algorithm that arises is the so called Gauss-Jordan factorization.

For this algorithm, the elimination steps can again be expressed by premultiplication with elementary matrices of the form $(I - g_k e_k^T)$, where the k-th element of g_k equals 0. In this situation none of the other elements of g_k need to be zero. The effect of the Gauss-Jordan algorithm is equivalent with applying all relevant elementary matrix transformations. This is denoted by:

$$D = (I - g_n e_n^T)(I - g_{n-1} e_{n-1}^T) \ldots (I - g_1 e_1^T) A.$$

From this we find the factorization:

$$A = (I + g_1 e_1^T) \ldots (I + g_{n-1} e_{n-1}^T)(I + g_n e_n^T) D.$$

The Gauss-Jordan algorithm needs more floating-point operations than Gaussian elimination; $n^3/2 + O(n^2)$ multiplications for Gauss-Jordan against $n^3/3 + O(n^2)$ for Gaussian elimination. Moreover, if the Gauss-Jordan algorithm is combined with the pivoting strategy that is generally used in Gaussian elimination (i.e. the use of row interchanges), then it may be possible that a solution is calculated with a large residual vector $(b - A\hat{x})$. This has been shown by Peters and Wilkinson [13]. Our second paper goes into details of this subject and we show that the use of column interchanges in the Gauss-Jordan algorithm gives a much more satisfactory result.

The operation count shows that, on sequential computers, the Gauss-Jordan algorithm is slower than Gaussian elimination. On vector and parallel computers, however, the use of Gauss-Jordan may be advantageous, because of its simpler structure and the possibility of more efficient data access.

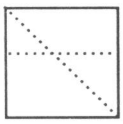
An extended version of the Gauss-Jordan algorithm can be used to calculate the explicit form of the inverse matrix. This algorithm uses the same number of operations as an inversion method based on Gaussian elimination, viz. $n^3 + O(n^2)$ multiplications and the same number of additions. It can easily be arranged such that the inverse overwrites the original matrix. The algorithm and results of experiments on a Cyber 205 vector computer are described in our contribution to the 'ICIAM 87' Conference in Paris [2].

An important application for matrix inversion by the Gauss-Jordan method is given by Gallivan, Jalby and Meier [7]. They communicate its use as an essential part of calculating a block LU factorization on a parallel machine with a hierarchical memory.

The inverse matrix, or to be precise, its norm only, is needed to compute an upper bound for the error in the calculated solution. Many algorithms do exist that calculate an estimate for this norm of the inverse during Gaussian elimination in $O(n^2)$ operations. In our third paper we present an algorithm which also calculates this norm in $O(n^2)$ operations, but is tailored to go with Gauss-Jordan.

Type-II matrices

Matrices that for unit vector e_k and vector f with $\phi_k = 0$ are defined by:



$$E = (I - e_k f^T).$$

This type of elementary matrix arises in a remarkable variant of the Gauss-Jordan algorithm [1,10] which has been known for some time. This method has attracted little attention, because it did not allow for the 'standard' pivoting strategy by row interchanges (which can't be proven to be safe anyway!). The interesting point is, that its operation count is equal to that of Gaussian elimination, $n^3/3 + O(n^2)$ multiplications. The latter algorithm and the Gauss-Jordan factorization have been known for a long time; the variant we are aiming at is only 10 years old.
We describe this algorithm in our fourth paper and show that it allows partial pivoting by column interchanges.

Type-III matrices

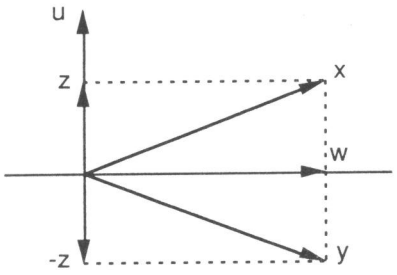
Matrices that for $v \neq \underline{0}$, and scalar γ are defined by:

$$P = (I - \gamma v v^T).$$

Certain projection matrices and reflection matrices belong to this class.

A. For $\gamma = 1/v^T v$, the matrix defines an orthogonal projection on the hyper-plane orthogonal to v. This type of matrices is used for Gram-Schmidt orthogonalization, where an orthogonal basis is to replace a non-orthogonal one. For instance, if a set of independent vectors is given, then a typical step in the construction of such an orthogonal basis is described by the premultiplication of the remaining vectors by a projection matrix of the type considered here.
An algorithm for this task is in fact straight-forward. However, the inexperienced user should be aware of a numerical instability in the *classical* Gram-Schmidt algorithm. It seems as if an efficient implementation of the Gram-Schmidt algorithm for a vector- or parallel machine should be based on precisely this unstable formulation. Our fifth paper is dealing with this subject.

B. For $\gamma = 2/v^Tv$, the matrix defines a reflection in the hyperplane orthogonal to v. Reflections can be described with the use of projection matrices.



Let u be given with $\|u\|_2 = 1$, then $(I - uu^T)$ describes the projection on the hyperplane orthogonal to u.
Consider an arbitrary vector $x \neq \underline{0}$; let z be the orthogonal projection of x onto u and define $w = x - z$. Now an orthogonal decomposition of x is given by:
$x = z + w = (uu^T)\,x + (I - uu^T)\,x$.
This decomposition can be used to construct the mirror image y of x by adding components w and -z to form $y = w - z$. Thus we have: $y = (I - 2uu^T)x$.

Reflection matrices are used for mapping a given vector onto a multiple of a suitable unit vector. This can be applied to factorize a given matrix in the product of a unitary matrix and an upper triangular matrix as an alternative to the construction by means of Gram-Schmidt orthogonalization. The way these reflections are constructed and how they should be applied, is described in the sixth paper. In that paper we also describe how these same reflection matrices can be used for a similarity transformation that is used in the context of calculating eigenvalues.

1.2 Essentials of vector computing for numerical linear algebra

From the time that electronic computers became commercially available for scientific calculations (shortly before 1950), there has been an increase in speed and memory capacity for successive most powerful machines by a factor of ten, roughly, every five years [9]. Until the early seventies, this growth has been achieved by improved technology of the arithmetical processor and of memory devices, without essential changes in the architecture of the computer itself.

One could speak of a break-through in the design of electronic computers, when the first large-scale *parallel* computers came on the market in the mid seventies. With the introduction of this new type of machines, the increase of computing power was served tremendously.

For a 'conventional' design, the speed of a single floating-point calculation had already increased until almost 10^{-7} sec., and the physical limitations for a further increase seemed within reach.

There exist several ways to exploit parallelism for scientific calculations. According to the classification of parallel machines [6] as standardly used, a vector computer as considered in our research, belongs to the SIMD type, which is an abbreviation of "Single Instruction, Multiple Data".
In this type of computers, a number of processing units is combined to a so called *vector processor*. This vector processor is capable of operating on one or two arrays of numbers, called the vector operands, in order to execute arithmetical operations with high speed.

We may think of calculating a vector which is the sum of two input vectors, or calculate the scalar which is the innerproduct of two vectors, or any type of arithmetical operations on arrays of numbers (vectors).
These so called vector operations can only be executed efficiently, if the operands are delivered to the processor in a constant 'stream', which means that prior to the operation to be executed, the numbers that do constitute the operands must physically be lined up, for this purpose, at a dedicated position (a *vector register* or a part of the central memory).

The actual execution of the vector operation can best be illustrated by the example of calculating the sum of two vectors.
So, assume that for i = 1, ..., n the numbers β_i and γ_i are given in suitable locations in the computer and that for each index the sum $\alpha_i = \beta_i + \gamma_i$ must be calculated.

The central idea is that the activity of adding two numbers is split up into a number of partial computations; the same holds for computing the difference, the product and so on.

A row of 'primitive' processors (typically in the order of ten) is lined up and each one is capable of performing a unique piece of the total arithmetical operation. The first processor receives two numbers β_i, γ_i, and performs its piece of the addition (corresponding with something in the order of 10 % of the total work).

After a fixed time step (a so-called *clock cycle*), the partial result and the operands β_i and γ_i are passed on to the next processor. The first processor then receives the numbers β_{i+1} and γ_{i+1} and repeats its part of the action on this new numbers.

Each clock cycle, an updated result is transfered to the next processor, so that after a number of clock cycles equal to the number of processors, the sum α_i is delivered at the end of the vector processor. It is clear why this type of processing is compared with working at a conveyor-belt.

For the addition of two single numbers in isolation, this way of operating is very inefficient, but for the addition of (long) arrays of numbers it will result in an improved performance compared with adding in a conventional way.

The efficiency of a piece of code on a vector computer is normally measured in millions of floating-point operations per second, called Mega-flops or Mflops for short. Each vector computer reaches its peak performance, if it can produce results on the vector processor in a constant stream. The number of Mflops in that case is the inverse of the clock cycle time. For present top of the market computers, the peak performance approaches 1000 Mflops, which is equal to 1 Giga-flop.

It is hardly surprising that most programs on a vector computer are running with a speed that is (far) below the peak performance.
The efficiency of a program on a vector computer is measured with two parameters, R_∞ and $n_{1/2}$ [9]. The parameter R_∞ stands for the Mflop rate in which the results would be produced if the occurring vectors would have

infinite length; $n_{1/2}$ stands for the length of a vector that admits the program to work with a speed that equals half the asymptotic speed R_∞ .

Especially programs in the area of linear algebra, admit efficient vector code. The occurring quantities are matrices and vectors so that arrays of numbers can be defined in a natural way. For a program to work on these vectors, it is advantageous that the vector elements are stored in contiguous memory locations. This is especially true for the Cyber 205 vector computer, where the operands for the vector processor are taken directly from memory. In most other vector computers, the operands are loaded firstly into vector registers from which they are supplied to the processors. For those computers, we should prevent the so called *memory bank conflicts* in the transport from main memory to vector register.

If matrices are used, then it is necessary to be aware of the way its elements are stored in memory. For FORTRAN programs this is realized per column of the matrix and for a PASCAL program per row of the matrix. From the fact that for vector computing the standard programming language appears to be FORTRAN, we concentrate on the FORTRAN mode. That means that we focus on storing matrices column-wise.

It should be appreciated that already the design of an algorithm highly influences the resulting efficiency of the code.
We like to illustrate this for the Cyber 205 vector computer with the example of calculating the product of two matrices.

Assume that matrices B and C are given; their elements β_{ij} and γ_{ij} are stored column-wise in the FORTRAN program that we are going to design after the proposed algorithm for calculating $A = B \times C$.
Matrix A is defined by n^2 scalars α_{ij} . For each α_{ij} we have:

$$\alpha_{ij} = \sum_{k=1}^{n} \beta_{ik}\gamma_{kj} .$$

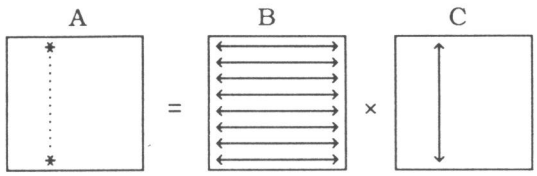
For a further presentation of alternative ways for calculating this matrix product, we introduce the following notation:

Notation: with $b_{\cdot j}$ we denote the j-th column of matrix B;
with $b_{i\cdot}$ we denote the i-th row of matrix B.
Moreover, we assume that a subroutine "`innerproduct(x,y)`" does exist, which calculates the innerproduct of two vectors x and y.

We like to visualize the alternative methods by pictures that were introduced by Dongarra, Gustavson and Karp [5]. A square will denote a matrix, an arrowed line a row or a column and an asterisk an element of the matrix.

The first alternative we present, is the 'column wise' calculation of the elements α_{ij} , using the above definition:

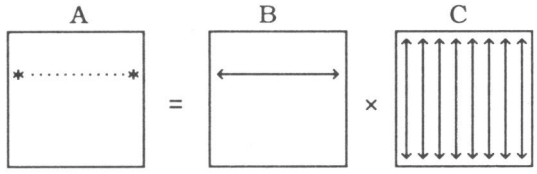
1. Column-wise *(jik):*




```
for j = 1 to n
    for i = 1 to n
        αij = innerproduct(bi., c.j )
```

The second alternative is the 'row-wise' calculation of elements α_{ij}, again using the above definition:

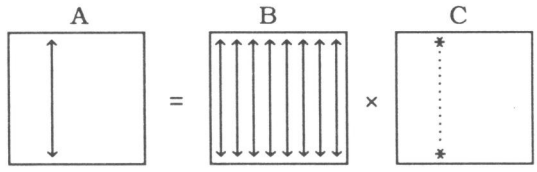
2. Row-wise *(ijk):*




```
for i = 1 to n
    for j = 1 to n
        αij = innerproduct(bi., c.j )
```

We may consider A as a row of n columns, each being a linear combination of all columns of B. This interpretation gives the next alternative:

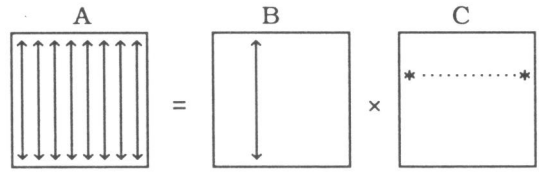
3. *(jki):*




```
for j = 1 to n
    a.j = Σ (k=1 to n) b.k γkj
```

Using the same interpretation of the matrix product, we can firstly compute all contributions of a column of B to all columns of A. This gives the following alternative:

4. *(kji):*



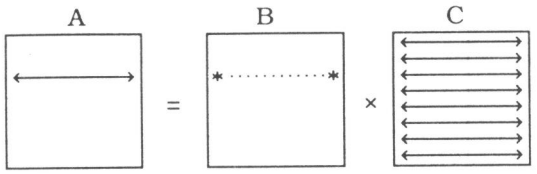

```
A = O {the zero matrix}
    for k = 1 to n
      for j = 1 to n
        a.j := a.j + b.k γkj
```

Finally we may consider A as a system of n rows. That will give us two row oriented versions which are analogue to column versions three and four respectively.

5. *(ikj):*

```
for i = 1 to n
```

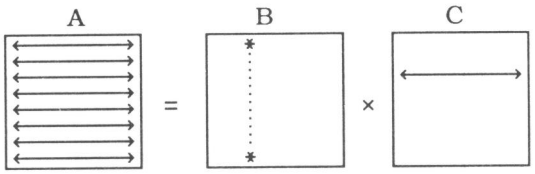
$$a_{i.} = \sum_{k=1}^{n} \beta_{ik} c_{k.}$$



6. *(kij):*

```
A = O {the zero matrix}
    for k = 1 to n
      for i = 1 to n
```

$$a_{i.} := a_{i.} + \beta_{ik} c_{k.}$$



The programs that can be constructed after these six alternatives, behave quite differently on the Cyber 205. The differences are even more pronounced because of the special design of the Cyber 205. For certain consecutive vector operations, so called *linked triads*, the result of an expression can be directly used as an operand for the next expression, without an intermediate storage in memory. The use of linked triads results in pieces of code that run twice as fast as one would expect otherwise. The following construction is an example of a linked triad:

$x := y + \beta \times z$, where x, y and z are vectors and β is scalar.

With respect to the alternative programs for matrix multiplication, we may expect that versions 3 and 4 give an efficient code, because they use linked triads.
Constructions 5 and 6 are expected to be inefficient, because the rows of the matrix that are needed as input vectors need to be lined up as proper vector operands.
Constructions 1 and 2 are expected to be moderately efficient, because the calculation of an innerproduct is less efficient than the calculation of a linked triad.

All matrix elements are supposed to be floating-point numbers. Their actual values do not influence the computing time (as opposed to human like computations). We experimented with the above six versions of matrix multiplication by having a program fill two matrices B and C successively for orders of n = 25, 50, 100, 200 with pseudo random numbers and calculate the matrix product.

In the next table we show the computing time in seconds for calculating the matrix product $A = B \times C$ on the Cyber 205.

method	n = 25	50	100	200
1. innerprod. col. wise (jik)	0.0033	0.0171	0.0956	0.5947
2. innerprod. row wise (ijk)	0.0033	0.0171	0.0956	0.5947
3. linear comb. cols. (jki)	0.0020	0.0091	0.0464	0.2659
4. contrib. each col. (kji)	0.0020	0.0091	0.0464	0.2659
5. linear comb. rows (ikj)	0.0043	0.0248	0.1549	1.0871
6. contrib. each row (kij)	0.0043	0.0248	0.1548	1.0878

Table 1. Processing time in seconds for A = B × C .

For each of the methods discussed, we will also calculate the actual performance of the computer. For the calculation of each element α_{ij} we need a total of n multiplications and n additions, independent of the method chosen. So, in total, we obtain $2n^3$ floating-point operations. The number of Mflops for any of the considered methods is thus calculated according to the formula:

$$\#\text{Mflops} = 2n^3 \ / \ \text{'time in micro seconds'} .$$

The same experiment is reported in this form by the following table:

method	n = 25	50	100	200
1. innerprod. col. wise (jik)	9.4	14.6	20.9	26.9
2. innerprod. row wise (ijk)	9.4	14.6	20.9	26.9
3. linear comb. cols. (jki)	15.7	27.4	43.1	60.2
4. contrib. each col. (kji)	15.7	27.4	43.1	60.2
5. linear comb. rows (ikj)	7.3	10.1	12.9	14.7
6. contrib. each row (kij)	7.3	10.1	12.9	14.7

Table 2. Performance in Mflops for A = B × C .

In the fall of 1988 the Cyber 205 of SARA was extended with a second vector processor; at the time these experiments were performed (summer 1988), the machine had a single vector processor. Its peak performance was 50 Mflops which could be raised to 100 Mflops if linked triads were used.

We would like to emphasize that the programs we used, were written in FORTRAN 77. For that language, most 'harmless' do-loops are automatically translated into vector code. In cases that the compiler can not perform this automatic translation, the piece of program is executed in the standard sequential way, using scalar code.

For constructions that do not admit this automatic translation and for useful constructions that are not defined in FORTRAN 77, an extension of this language has been defined for the Cyber 205. This 'local dialect' is called FORTRAN 200. With the use of this extension, some of the results as presented above can be improved. For instance, a 200 × 200 matrix multiplication using method 2, could be performed at a speed of 56 Mflops and the result of method 5 could be improved to a speed of 22 Mflops.

Methods 3 and 4 could not be improved. The FORTRAN 77 compiler produces the same vector operations in this case as the FORTRAN 200 compiler.

Concluding Remarks

We would like to end this introduction with some remarks concerning the implementation of linear algebra routines.
For a particular program to run at highest possible speed on a specific machine, it may be possible that certain pieces of code cannot be expressed in FORTRAN 77, as has been shown above for the Cyber 205. The use of what we have called 'local dialect', prevents exchanging (high quality) software for solving well defined, standard problems. If this software were completely programmed in FORTRAN 77, then it would be *portable*, (i.e. easily transportable to other machines) but it might be (very) slow.

The construction of good, portable codes has been facilitated by the introduction of well defined sets of 'Basic Linear Algebra Subprograms', BLAS [4,11].

For many scientific programs, most of the computing time is spent in matrix and vector calculations. These calculations can be expressed by calls to suitable routines in BLAS. From the fact that BLAS has received international recognition, we could say that these calls more or less belong to the standard environment, which also includes FORTRAN 77. For a specific machine, the BLAS routines are implemented with use of as much local dialect as necessary. In this way, both portability and efficiency of programs are served.

In the original BLAS level-1 version [11], dating from 1979, routines are included for all kinds of operations on one or two vector operands. A matrix times vector calculation, for instance, can be coded by n successive calls to an applicable subroutine. The speed of modern computers has become so high, that the time needed for these n routine calls is no longer negligible compared with the total time for the floating-point calculations.

This has been one of the reasons for a BLAS level-2 proposal [5], where matrix-vector operations are addressed.
It shows that for so called MIMD machines, having parallel processors and with a hierarchical memory, an even higher operation level is advantageous. For this reason, BLAS level-3 has been proposed [4].

The subroutines we have implemented for the NUMVEC library are structured in such a way that calls to either BLAS-2 or BLAS-1 routines can be clearly identified. The explicit use of FORTRAN 200, however, yielded much more efficient subroutines for the Cyber 205 almost always.
Therefore, we have indicated, with comment statements in our program texts, which calls to BLAS-2 or BLAS-1 routines should replace the relevant FORTRAN 200 statements.

References

1. M. Cosnard, Y. Robert, D. Trystram, Résolution parallèlle de systèmes linéaires denses par diagonalisation; Bulletin E.D.F, série C, 2(1986)67-88.
2. T.J. Dekker and W. Hoffmann, Numerical Improvement of the Gauss-Jordan algorithm; Proceedings ICIAM 87, Paris-La Villette, june 29-july 3 1987; Contributions from the Netherlands; ed.: A.H.P. van der Burgh, R.M.M. Mattheij.
3. J.J. Dongarra, J. du Croz, I. Duff and S. Hammarling, A set of level 3 Basic Linear Algebra Subprograms; Technical Memorandum No. 88 (Revision 1); Argonne National Laboratory 1988
4. J.J. Dongarra, J. du Croz, S. Hammarling and R.J. Hanson, An extended set of FORTRAN Basic Linear Algebra Subprograms; ACM Trans. Math. Softw. 14, (1988), pp. 1-17.
5. J.J. Dongarra, F.G. Gustavson and A. Karp, Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine; SIAM Review 26, (1984), pp.91-112.
6. M.J. Flynn, Some computer organisations and their effectiveness; IEEE Trans. Comput.. C-21, (1972), pp. 119-136.
7. K. Gallivan, W. Jalby and U. Meier, The use of BLAS3 in linear algabra on a parallel processor with a hierarchical memory; SIAM J.Sci. Stat. Comput. 8, (1987), 1079-1084
8. G. H. Golub and C.F. van Loan, Matrix Computations; North Oxford Academic, Oxford 1983.
9. R.W. Hockney and C.R. Jesshope, Parallel Computers,Adam Hilger Ltd, Bristol, 1981.
10. P. Huard, La méthode du Simplexe sans inverse explicite; bulletin E.D.F. Série C n.2, (1979).
11. C.L. Lawson, R.J. Hanson, R.J. Kincaid and F.T. Krogh, Basic Linear Algebra Subprograms for Fortran Usage, ACM Trans. Math. Softw. 5, 308-323 (1979)
12. B.N. Parlett, The Symmetric Eigenvalue Problem, Prentice Hall, Englewood Cliffs N.J. 1980.
13. G. Peters and J.H. Wilkinson, On the Stability of Gauss-Jordan Elimination with Pivoting; Communications of the ACM 18, (1975), pp. 20-24 .
14. H.J.J. te Riele (ed.), NUMVEC, a library of numerical software for Vector and parallel computers in FORTRAN; Centre for Mathematics and Computer Science, Amsterdam.
15. G.W. Stewart, Introduction to matrix computations; Academic Press 1973.
16. J.H. Wilkinson, Error Analysis of Direct Methods of Matrix Inversion; J. Assoc. Comp. Mach. 8, (1961), pp. 281-330.

The annals of scientific discovery are full of errors that opened new worlds: Bell was working on an apparatus to aid the deaf when he invented the telephone; Edison was tinkering with the telephone when he invented the phonograph. If a man can keep alert and imaginative, an error is a possibility, a chance at something new; to him, wandering and wondering are part of the same process, and he is most mistaken, most in error, whenever he quits exploring.

William Least Heat Moon,
Blue Highways.

CHAPTER I

Solving linear systems
on a vector computer

reprint of:
Journal of Computational and Applied Mathematics 18(1987) 353-367

Journal of Computational and Applied Mathematics 18 (1987) 353–367
North-Holland

Solving linear systems on a vector computer

W. HOFFMANN
Mathematical Institute, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands

Received 7 April 1986

Abstract: This paper gives a classification for the triangular factorization of square matrices. These factorizations are used for solving linear systems. Efficient algorithms for vector computers are presented on basis of criteria for optimal algorithms. Moveover, the Gauss–Jordan elimination algorithm in a version which admits efficient implementation on a vector computer is described. Comparative experiments in FORTRAN 77 with FORTRAN 200 extensions for the Cyber 205 are reported.

Keywords: Gaussian elimination, LU-decomposition, Gauss–Jordan algorithm, linear equations, vector computing.

1. Introduction

The use of vector processors for solving large linear systems necessitates reassessment and redesign of numerical algorithms. In view of this we describe several variants of the Gaussian elimination algorithm and results of performance measurements on a Cyber 205.

In [4] Dongarra, Gustavson and Karp investigated the performance of Gaussian elimination by reorganizing the algorithm. Their reorganization does only affect the loop structure; we show that also choices with respect to the pivoting strategy, the normalization of the diagonal elements in the resulting factorization and the ordering of the calculation contribute to the stability and the efficiency of the resulting algorithm. This efficiency, expressed in Mflops, measures the number of floating-point operations divided by CPU-time and does not take into account the I/O-time or the number of large pages used.

Throughout the paper we fix our notation to standard conventions: lower case greek letters for scalars (all real here), lower case roman letters for vectors and indices, upper case roman letters for matrices. For the ith rowvector of matrix A we use the notation $a_{i.}$ and for the jth column of the same matrix we use $a_{.j}$. By $\boldsymbol{a}_{.j}$ we denote the vector that is defined by the last $n-j$ elements of $a_{.j}$; the use of $\boldsymbol{a}_{i.}$ is analogous. An element of matrix A is α_{ij} and the order of a matrix is always denoted by n. A discussion with respect to the various algorithms appears in Section 2 and a description of our preferred algorithm in section 2.3. This algorithm performs an LDU decomposition of matrix A with partial pivoting by column interchanges and normalization of the diagonals according to $\lambda_{ii} = \upsilon_{ii} = \delta_i^{-1}$.

In section 2.4 we recall an idea of Businger, concerning the growth factor of a matrix [1] and demonstrate how this can be efficiently combined with the choices in our algorithm.

In Section 3 we give a description of the well known Gauss–Jordan elimination algorithm which appeared to do extremely well for matrices that are not very large. In our version we

implemented a partial pivoting strategy without actually interchanging any rows or columns in the matrix.

In Section 4 we give a comparison with respect to CPU-times required on a Cyber 205 between our routines and some well-known routines from program libraries.

In an appendix we give a detailed description of our preferred algorithms and some examples of others.

2. Triangular factorization

For the description of algorithms that are variants of the well-known Gaussian elimination algorithm for solving linear systems of equations, we distinguish three essential choices (degrees of freedom) that determine the formulation of such an algorithm.

These choices are with respect to:

(a) normalization of the diagonals,

(b) ordering of the calculation, and

(c) pivoting strategy.

In the sequel we deal with these three matters in detail. (Only (b) has been covered partially in [4].)

2.1. Degrees of freedom

Normalization of the diagonals

The action of Gaussian elimination on a given matrix A is equivalent with factoring that matrix in a lower-triangular matrix L $(=(\lambda_{ij}))$ and an upper-triangular matrix U $(=(\upsilon_{ij}))$ such that (apart from pivoting) $A = LU$ with $\lambda_{ii} = 1$, $i = 1, \ldots, n$. The elements in the resulting decomposition can be modified for the normalization $\upsilon_{ii} = 1$, $i = 1, \ldots, n$, while still $A = LU$ holds. This is a consequence of the fact that the diagonal elements of L and U are not uniquely defined by a factorization of A in triangular factors, a so-called LU-decomposition. In the sequel we consider the more general factorization

$$A = LDU, \tag{1}$$

where L and U are lower- and upper-triangular respectively and $D = \operatorname{diag}(\delta_1, \ldots, \delta_n)$. Such a factorization is completely defined by the values that are given to the diagonals of L, U and D; for all choices with $\delta_i = 1$ (i.e. $D = \mathrm{I}$) an LU-decomposition like considered before is defined.

Usefull and well known choices are:

(i) $\lambda_{ii} = \delta_i = 1$: this choice holds for the standard Gaussian elimination algorithm and for the Doolittle factorization (see for instance [6] or [13]);

(ii) $\upsilon_{ii} = \delta_i = 1$: this choice is made in the Crout factorization [6,13];

(iii) $\lambda_{ii} = \upsilon_{ii} = 1$: this defines the standard LDU factorization [6];

(iv) $\lambda_{ii} = \upsilon_{ii} = \delta_i^{-1}$: this choice belongs to the 'folklore' of the numerical analysts.

For the case of positive definite matrices:

(v) $\lambda_{ii} = \upsilon_{ii}$, $\delta_i = 1$: this choice is made in the Cholesky factorization.

These choices all have the advantage that for an implementation one needs only a single memory location to store the triple $(\lambda_{ii}, \upsilon_{ii}, \delta_i)$. Choice (iv) appears to be advantageous on a vector computer as will be explained in the sequel. In [4] only choice (i) has been considered.

Ordering of the calculation

Starting from (1), with a chosen normalization for the diagonals, the elements of L, D and U can be calculated by equating left- and right-hand side of that equation. This can be done in several ways. In order to investigate various possibilities, we distinguish between three interpretations of matrix multiplication. The first one is the scalar interpretation (denoted by S--) in which we interpret (1) as

$$\alpha_{ij} = \sum_{k=1}^{\min(i,j)} \lambda_{ik}\delta_k v_{kj}, \quad i, j \in \{1, \ldots, n\}$$

which corresponds with schemes *ijk* and *jik* for matrix multiplication as introduced by Dongarra et al. in [4].

If we use this formula for the calculation of the elements of L, D and U, we still can choose to calculate the elements of both L and U either columnwise or rowwise which gives four possible combinations. Consequently we distinguish four calculation schemes denoted by Scc ($\approx jik$), Scr, Srr ($\approx ijk$) and Src respectively. The denotation Scr, for example, stands for *S*calar interpretation, L *c*olumnwise, U *r*owwise. The other denotations have analogous meaning.

The next interpretation of the matrix product in (1) is the columninterpretation (denoted by C--) which follows from the point of view that columns of matrix A are linear combinations of the columns of matrix L, as is expressed in the formula

$$a_{\cdot j} = \sum_{k=1}^{j} l_{\cdot k}\delta_k v_{kj}, \quad j \in \{1, \ldots, n\}.$$

If we use this formula for the calculation of L, D and U, we observe that matrix L is calculated by columns, but we still can choose between the elements of U being calculated either columnwise or rowwise. The two resulting calculating schemes are denoted by Ccr ($\approx kji$) and Ccc ($\approx jki$).

The analogous viewpoint on rows of the matrix for the rowinterpretation (denoted by R--) of the matrixproduct in (1) yields the formula:

$$a_{i\cdot} = \sum_{k=1}^{i} \lambda_{ik}\delta_k u_{k\cdot}, \quad i \in \{1, \ldots, n\}.$$

In the schemes based on this formula we observe that U is calculated by rows, but here we can choose between the elements of L being calculated either columnwise or rowwise. The resulting schemes are denoted by Rcr ($\approx kij$) and Rrr ($\approx ikj$).

In the pictures in Fig. 1 the eight calculation schemes are visualized. Calculated elements of L and U are indicated by dots or lines in the lower and upper triangular part respectively of each square.

Pivoting strategy

To obtain numerical stability in the factorization of a general matrix, a pivoting strategy must be applied. In most algorithms a choice is made for interchanging rows during the process of factorization such that in each column of the resulting matrix L an element of maximal size (i.e., absolute value) is found on the diagonal. This implies that the decomposition is made of the original matrix with permuted rows; in formula:

$$PA = LDU,$$

where P is a permutation matrix.

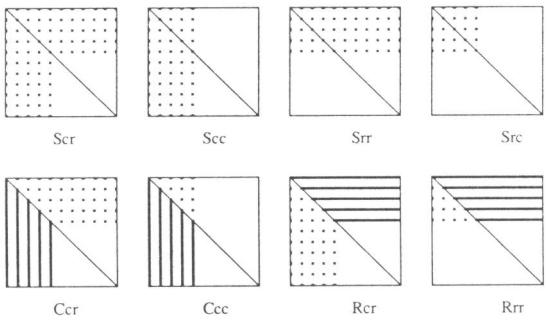


Fig. 1.

Interchanging rows during the elimination process can only be done in a meaningful way, if in each step we can dispose of the entire next column of L. This is only the case in the -cr and -cc type schemes so that only these schemes are suitable for row interchanging.

The stability of Gaussian elimination can likewise be obtained by performing column interchanges. The final result will be a decomposition of a matrix with permuted columns; so we have:

$$AQ = LDU,$$

for a permutation matrix Q. The criterium in this pivoting strategy is such that in each row of the resulting matrix U an element of maximal size is found on the diagonal. This strategy can only be applied if matrix U is calculated rowwise, which only holds for the -cr and -rr type schemes. Observe that the -rc type scheme doesn't admit any kind of interchanging. The -rc type scheme, which only exists in the Src form, is often used in the Cholesky decomposition for the symmetrical positive definite case where indeed no pivoting is necessary, see e.g. [13]. In the sequel, we will no longer consider the Src scheme.

2.2. Inventarization

To summarize our inventarization of triangular factorization algorithms, we present an overview of the schemes that can be combined with row or column interchanges respectively. Moreover, we indicate where well-known factorization algorithms fit in our display.

From the ten schemes in Tables 1 and 2 in combination with the four suggestions for normalization of the diagonals, we can construct forty different algorithms for the triangular factorization of a matrix, each showing different performance in different environments. The choice between algorithms of the S-- type, C-- type and R-- type depends on the machine architecture in combination with the programming language used.

On a machine where vectors can be handled as entire quantities the C-- and R-- type algorithms are to be preferred; C-- type algorithms if the matrix is stored by columns (as in FORTRAN) and R-- type if the matrix is stored by rows (as in PASCAL and ADA).

The choice of the pivoting strategy depends likewise on both the machine architecture and the programming language used so that storage by columns (rows) is to be combined with column (row) interchanges.

Table 1
Schemes that can be implemented with row interchanges

Scheme	Normalization	Published as
Scr	$\lambda_{ii} = \delta_i = 1$	Doolittle, e.g. [6,13]
	$\upsilon_{ii} = \delta_i = 1$	Crout, e.g. [6,13]
Scc	$\lambda_{ii} = \delta_i = 1$	SDOT version (*jik*) [4]
Ccr	$\lambda_{ii} = \delta_i = 1$	SAXPY version (*kji*) [4]
Ccc	$\lambda_{ii} = \delta_i = 1$	GAXPY version (*jki*) [4]
Rcr	$\lambda_{ii} = \delta_i = 1$	standard Gaussian elimination; version (*kij*) [4]

Table 2
Schemes that can be implemented with column interchanges

Scheme	Normalization	Published as
Scr		–
Srr	$\lambda_{ii} = \delta_i = 1$	version (*ijk*) [4] [a]
Ccr	$\lambda_{ii} = \upsilon_{ii} = \delta_i^{-1}$	routine CCRPCF (this report)
Rcr		–
Rrr	$\lambda_{ii} = \delta_i = 1$	version (*ikj*) [4] [a]

[a] The versions (*ijk*) and (*ikj*) were not published with column interchanges; they can not be implemented with row interchanges as is suggested in [4].

At this point the conclusion can be drawn that on a vector machine the code for a FORTRAN program which is optimal with respect to CP-time is likely to be based on the Ccr scheme with column interchanges and the code for a PASCAL or an ADA program on the Rcr scheme with row interchanges.

The normalization of the diagonals should be selected such that the resulting algorithm has optimal performance. For all four normalization variants in a C-- type or R-- type algorithm the number of multiplications and divisions equals $\frac{1}{3}(n^3 - n)$. The number of array accesses, however, is minimal for the normalization $\lambda_{ii} = \upsilon_{ii} = \delta_i^{-1}$ as is illustrated for the Ccr scheme in the next section.

2.3. Description of triangular factorization algorithms

Two algorithms of S-- type and two of C-- type will be described in detail in the appendix. Here we give a description of our preferred algorithm CCRPC, which stands for Ccr-type with *p*artial pivoting by *c*olumn interchanges (in the sequel we discuss another type of pivoting, so that the letter P supplies information):

For $k = 1, \ldots, n$

$$(1) \quad \begin{cases} \text{Determine } p \in \{k,\ k+1, \ldots, n\} : |\alpha_{kp}| = \max\limits_{k \leqslant j \leqslant n} |\alpha_{kj}| & (* \text{ select pivot } *) \\ a_{\cdot k} \leftrightarrow a_{\cdot p} & (* \text{ interchange columns } *) \end{cases}$$

(2) $\alpha_{jj} \leftarrow \left\{ \begin{matrix} \lambda_{jj} \\ \delta_j \\ \upsilon_{jj} \end{matrix} \right\} = ?$ (* choose normalization *)

(3) $\boldsymbol{a}_{\cdot k} \leftarrow \boldsymbol{l}_{\cdot k} = \boldsymbol{a}_{\cdot k}/(\delta_k \upsilon_{kk})$ (* update column of L *)

For $j = k+1, \ldots, n$

(4) $\alpha_{kj} \leftarrow \upsilon_{kj} = \alpha_{kj}/(\lambda_{kk}\delta_k)$ (* next elm. in row of U *)

(5) $\boldsymbol{a}_{\cdot j} \leftarrow \boldsymbol{a}_{\cdot j} - \boldsymbol{l}_{\cdot k}\delta_k \upsilon_{kj}$ (* update jth column of A *)

(* $(k+1)$st provisional column of L has been calculated in $\boldsymbol{a}_{\cdot k+1}$ *)

◇ CCRPC

With the choice $\lambda_{kk} = \upsilon_{kk} = \delta_k^{-1}$, statements (2), (3) and (4) become trivial. In that case the amount of work still remains $\frac{1}{3}(n^3 - n)$ multiplications and divisions, but the number of array-accesses is minimized. With this choice the description of the algorithm reduces to:

For $k = 1, \ldots, n$

Perform pivoting as in statement (1) above

For $j = k+1, \ldots, n$

$a_{\cdot j} \leftarrow a_{\cdot j} - \boldsymbol{a}_{\cdot k}(\alpha_{kj}/\alpha_{kk})$ (* update jth column of A *)

◇ CCRPC

which can easily be recognized as successively updating the original matrix with a specific rank-one matrix as follows:

For $k = 1, \ldots, n$

Perform pivoting as in statement (1) above

$A \leftarrow A - \alpha_{kk}^{-1}\boldsymbol{a}_{\cdot k}\boldsymbol{a}_{k\cdot}$ ◇ CCRPC

2.4. Monitoring the growth factor

With respect to the pivoting strategy, it is generally accepted that partial pivoting is used in almost all practical situations. With the introduction of vector computers it became possible to solve very large full systems of equations ($n > \approx 1000$).

For these systems insufficient experience exists to declare partial pivoting still reliable. The application of complete pivoting, which yields a stable algorithm, has the disadvantage of expensive code. In 1971 Businger [1] published an idea for calculating, with little extra cost, an upperbound for the growth factor during Gaussian elimination with partial pivoting.

This idea can be applied if an update of the matrix is calculated in each step, as is the case in schemes Ccr and Rcr. Scheme Scr similarly admits efficient monitoring of the growth factor, but because of the fact that in successive steps the remaining part of the coefficient matrix has not

been updated, a switch to complete pivoting is not feasible there. If this upperbound for the growth factor becomes too large, the pivoting strategy can be switched to complete pivoting from then on. This so-called mixed pivoting strategy provides the user with a reasonable upperbound for the growth factor so that an estimate of the error matrix [6.13] can be calculated. As a consequence, it reveals the situations where complete pivoting is likely to be necessary.

An Algol 60 procedure implementing the mixed pivoting strategy, written by Bus and Dekker in 1973, has been included in the NUMAL library [7].

3. Gauss–Jordan factorization

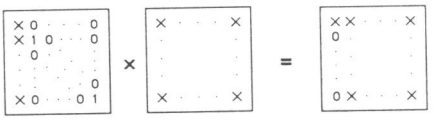
While experimenting with several implementations of the schemes as mentioned in Section 2 on a Cyber 205 vector computer, it appeared that for solving linear systems with not too big a coefficient-matrix ($n < \approx 50$) our implementation of the Gauss–Jordan algorithm used the least CP-time. For that reason we discuss it here.

3.1. Description of the algorithm

In the well-known Gauss–Jordan elimination algorithm, the coefficientmatrix of a given linear system is reduced to diagonal form in n steps by successively subtracting a suitable multiple of the ith row from all other rows, such that in the ith column the elements outside the diagonal become zero. If in these steps the right-hand side is considered as the $(n+1)$st column of the matrix and treated accordingly, the resulting linear system has the same solution as the original one.

For stability reasons, also in this algorithm, a pivoting strategy must be applied. We will treat this subject later.

The effect of the ith step in the algorithm can be described in matrix calculus by computing A^i from $A^{(i-1)}$ by leftmultiplication with an elementary matrix of the form $(I + g_i e_i^{\mathrm{T}})$ for some vector g_i. For the first step, with $A^0 = A$, we have in this way:



where g_1 has to be chosen such that the first column of A^1 is a multiple of the first unit vector. The effect of the multiplication on the first column of A^0 is given in the following formula:

$$\left(I + g_1 e_1^{\mathrm{T}}\right) a_{\cdot 1} = \delta_1 e_1,$$

for some value of scalar δ_1. This yields:

$$\alpha_{11} g_1 = \delta_1 e_1 - a_{\cdot 1}.$$

If δ_1 is given the value α_{11}, the subtraction causes no cancellation. In that case we have:

$$g_1 = e_1 - (1/\alpha_{11}) a_{\cdot 1}.$$

All remaining columns of the matrix are updated according to the following formula:

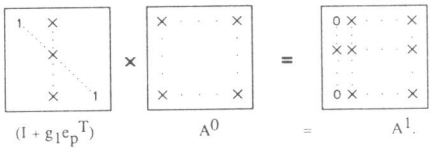
$$a_{\cdot k} \leftarrow (I + g_1 e_1^{\mathrm{T}}) a_{\cdot k},$$

which yields

$$a_{\cdot k} \leftarrow a_{\cdot k} + \alpha_{1k} g_1.$$

3.2. Partial pivoting

For numerical stability, row- or columninterchanges should be implemented in such a way that in each step the element of maximal modulus in the current column or row of the remaining $(n - i + 1)$st order submatrix is placed in the diagonal position. As an alternative to interchanging rows p and i $(p \geqslant i)$ in the ith elimination step, the leftmultiplication can be carried out with an elementary matrix of the form $(I + g_i e_p^{\mathrm{T}})$. In case the maximal element in modulus in the first column was found in row p (the pivotal row), the first step can be modified into



With the notation $\pi(i)$ (or πi for short) for the pivotal row in step i, the effect of all elimination-steps is described by

$$(I + g_n e_{\pi n}^{\mathrm{T}}) \cdots (I + g_1 e_{\pi 1}^{\mathrm{T}})(A \mid b) = (\delta_1 e_{\pi 1}, \ldots, \delta_n e_{\pi n} \mid b^n).$$

Introducing D for diagonal$(\delta_1, \ldots, \delta_n)$ and P for the permutation matrix $(e_{\pi 1}, \ldots, e_{\pi n})$, the right hand side of this equation can be described by $(PD \mid b^n)$. The resulting equivalent permuted diagonal system reads:

$$\delta_i \xi_i = \beta_{\pi i}^n, \quad i = 1, \ldots, n$$

from which the solution is easily calculated. The complete algorithm is found in the appendix. The technique for monitoring the growth factor as mentioned in section 2.4. is also applicable for this algorithm and a switch-over to complete pivoting is feasible too. (Note added in proof: Recent research shows that partial pivoting by column interchanges is numerically to be preferred.)

4. Numerical experiments

Experiments were carried out on the Cyber 205 computer (one vector pipe) of the Academic computer centre SARA in Amsterdam. The vector arithmetic in this computer is accessible through FORTRAN 200, which is a Control Data extension of FORTRAN 77.

We compared implementations of several algorithms from the overview in section 2.2. The timing showed that for our language/machine combination an algorithm based on a Ccr scheme with column interchanges gives optimal performance as was already suggested in section 2.2. The

Table 3
CP time in seconds for various n

	$n = 25$	$n = 50$	$n = 100$	$n = 200$	$n = 400$
(a) LINPACK (SGEFA + SGESL)	0.0028	0.0107	0.0441	0.1965	0.9974
(only SGESL)	(0.0003)	(0.0007)	(0.0015)	(0.0034)	(0.0083)
(b) NAG (F03AFF + F04AJF)	0.0030	0.0109	0.0445	0.2010	1.0042
(only F04AJF)	(0.0003)	(0.0005)	(0.0011)	(0.0027)	(0.0069)
(c) QQLIB (QQGEL)	0.0017	0.0065	0.0280	0.1367	0.7519
(d) GAUJOR	0.0012	0.0051	0.0249	0.1369	0.8649
(e) CCRPCF (decomp. and sol.)	0.0014	0.0051	0.0232	0.1154	0.6705
(only solution)	(0.0003)	(0.0005)	(0.0011)	(0.0027)	(0.0069)
(f) CCRMCF (decomp. and sol.) (partial pivoting throughout)	0.0016	0.0057	0.0243	0.1181	0.6779
(g) CCRMCF (decomp. and sol.) (complete pivoting throughout)	0.0031	0.0122	0.0550	0.2171	1.3704

algorithm we used is described in section 2.3. and in the appendix; our implementations are included in the NUMVEC library [10].

Subroutines from Extended Blas [3] can be combined with our algorithm so that an optimized and transportable code can be constructed. We compared our routines with routines from LINPACK [5], NAG [11], QQLIB [12], which are all (manufacturer) optimized for the Cyber 205. The results are reported in section 4.1. We also compared our implementation of the Gauss–Jordan elimination algorithm [10]. As is well known, Gauss–Jordan requires for the solution of a system of linear equations 1.5 times as many operations as standard Gaussian elimination. Nevertheless, Gauss–Jordan used less CP-time for matrices up to order ≈ 50 and performed rather well (with a high Mflop rate) for all matrices. This is due to the fact that throughout the Gauss–Jordan routine all 'active columns' remain vectors of full length as opposed to the Ccr scheme where the active columns become shorter in successive steps, so that the overhead for the vector calculations takes relatively more time. Moreover, the partial pivoting strategy in Gauss–Jordan is implemented such that no column or row interchanges are actually performed. The results of our experiments are reported hereafter.

4.1. Timing and efficiency

In this section we report the result of a number of experiments. We solved a linear system of order n having one right-hand side for five different values of n.

The experiments are described shortly; tables giving CP time (Table 3) and Mflop rate (Table 4) are presented separately. The experiments concern the following routines:

(a) LINPACK routines SGEFA and SGESL for the decomposition and solution respectively.
(b) NAG routines F03AFF for the decomposition and F04AJF for the solution.
(c) QQLIB routine QQGEL for both the decomposition and the solution.
(d) NUMVEC routine GAUJOR, our implementation of the Gauss–Jordan elimination algorithm with partial pivoting by 'virtual row interchanges' as described in section 3.2. Decomposition and solution are combined.
(e) NUMVEC routine CCRPCF, our implementation of algorithm CCRPC with normalization $\lambda_{kk} = v_{kk} = \delta_k^{-1}$; decomposition and solution are combined.

Table 4
Mflops for various n

	$n = 25$	$n = 50$	$n = 100$	$n = 200$	$n = 400$
(a) LINPACK (SGEFA, SGESL)	4.2	8.3	15.6	27.5	43.1
(b) NAG (F03AFF, F04AJF)	3.9	8.1	15.4	26.9	42.8
(c) QQLIB (QQGEL)	6.9	13.6	24.5	39.6	57.2
(e) CCRPCF (sol. included)	8.3	17.3	29.6	46.9	64.1
(f) CCRMCF (sol. included) (partial pivoting throughout)	7.3	15.5	28.3	45.8	63.4
(d) Gauss-Jordan $\{2\times(n^3/2+n^2)\}$	14.0	25.5	41.0	59.0	74.4

(f) NUMVEC routine CCRMCF. Routine CCRMCF is an extension of CCRPCF; it adds an implementation of the mixed pivoting strategy as explained in section 2.4. This strategy depends on a steering parameter, a sort of 'confidence measure'. As long as the estimated value of the growth factor is smaller than the confidence measure, the subroutine sticks to partial pivoting and the effect of CCRMCF is the same as that of CCRPCF. In this experiment we gave the confidence measure a very 'liberal' value such that partial pivoting was used throughout and the overhead for calculating the upperbound for the growth factor could be measured.

(g) In this experiment CCRMCF was used with an extremely 'conservative' value for the confidence measure so that complete pivoting was performed from the very first step.

4.2. Discussion

The experiments were tested under equal conditions; with the same optimization parameters for the compiler, the same size of the arrays, the same number of 'large pages'. The timings proved to be reproducible with some fluctuations in the last decimal given.

The need for Extended Blas [3] in stead of the original Blas, as used until now in LINPACK, is illustrated by the fact that LINPACK's SGESL takes more time than NAG's F04AJF or our routine for the solution.

It shows that for values of n up to $n \approx 50$, our Gauss–Jordan implementation gives the fastest routine; for larger values of n CCRPCF is the fastest.

Experiments (e) and (f) show that monitoring the growth factor while performing partial pivoting is not expensive; the use of complete pivoting takes roughly twice as much time as is shown in experiment (g). It should be emphasized however, that an eventual application of complete pivoting occurs only when a result produced by partial pivoting is probably totally unreliable.

In Table 4 we show the efficiency of the code expressed in 10^6 floating-point operations per second: Mflops. For the routines listed in experiments (a), (b), (c), (e), (f) we used the formula $2\times(n^3/3+n^2)$ for the number of floating point operations required, based on decomposition and solution for one right-hand side. This formula is also used in Dongarra [2]. For the routine in experiment d we used the formula $2\times(n^3/2+n^2)$ for the number of floating point operations required. For experiment (g) no Mflop number is given, because a considerable amount of time was spent in maximum search and row and column interchanging.

The Cyber 205 machine used for our experiments, being a one-pipe machine, has an optimal performance of 50 Mflops. For the use of linked triads (constructions of the form $x \leftarrow \alpha y + z$) the optimal performance on this machine is 100 Mflops with $n_{1/2} = 80$ ($n_{1/2}$ is the vectorlength for which 50 Mflops is reached [9])

The value of 50 Mflops in experiment (e) was reached for $n \approx 230$ and in experiment (d) for $n \approx 140$.

5. Conclusion

On basis of a theoretical analysis of Gaussian elimination (inspired by the work of Dongarra, Gustavson and Karp [4]) we propose an algorithm which is to our believe optimal for the Cyber 205. Our FORTRAN 200 implementation yielded the fastest subroutine for the solution of a linear system on basis of triangular decomposition that we know. The subroutine can still be speeded up somewhat (as has been confirmed by recent experiments of ours) by techniques such as loopunrolling and the use of scalar code for short vectors. Subroutine CCRMCF, with the upperbound for the growth factor, is still one of the fastest subroutines while adding the facility of delivering a realistic bound on the error matrix.

The analysis we carried out can be helpfull in selecting the optimal algorithm for other supercomputers, depending on machinearchitecture and programming language used.

Acknowledgements

The author wants to thank professors T.J. Dekker and H.A. van der Vorst for their valuable contributions to the ideas presented in this paper. He thanks P.P.M. de Rijk for his help in carrying out the experiments and W.M. Lioen for coding the final implementation.

Appendix

In this appendix we present the description of several algorithms from the overview in section 2.2 and also the Gauss-Jordan algorithm. The algorithms from section 2.2 are described in terms of elements of the matrix. We have chosen to describe the following algorithms:

(i) SCRPC which is scheme Scr with *p*artial pivoting by *c*olumn interchanges;
(ii) SCCPR which is scheme Scc with partial pivoting by row interchanges;
(iii) CCCPR which is scheme Ccc with partial pivoting by row interchanges;
(iv) CCRPC which was already described (using entire columns) in section 2.3;
(v) GAUSSJ which is the Gauss–Jordan algorithm with 'virtual pivoting' as described in section 4.2.

SCRPC

With a choice of $\delta_i \neq 1$, the code for an optimal routine does not follow directly from a straightforward description of the algorithm. For that reason we leave out the diagonal matrix D here.

For $k = 1, \ldots, n$

  For $j = k, \ldots, n$

(1) $\alpha_{kj} \leftarrow \tilde{v}_{kj} = \alpha_{kj} - \sum_{h=1}^{k-1} \lambda_{kh} v_{hj}$ (* provisional row of U *)

(2) $\begin{cases} \text{Determine } p \in \{k, k+1, \ldots, n\} : |\tilde{v}_{kp}| = \max_{k \le i \le n} |\tilde{v}_{kj}| & (* \text{ select pivot } *) \\ a_{\cdot k} \leftrightarrow a_{\cdot p} & (* \text{ interchange columns } *) \end{cases}$

(3) $\alpha_{kk} \leftarrow \begin{cases} \lambda_{kk} \\ v_{kk} \end{cases} = \alpha_{kk}$ (* choose normalization *)

  For $j = k+1, \ldots, n$

(4) $\alpha_{kj} \leftarrow v_{kj} = \tilde{v}_{kj} / \lambda_{kk}$ (* update row of U *)

  For $i = k+1, \ldots, n$

(5) $\alpha_{ik} \leftarrow \lambda_{ik} = \left(\alpha_{ik} - \sum_{h=1}^{k-1} \lambda_{ih} v_{hk} \right) / v_{kk}$ (* update column of L*)

◇ SCRPC

The number of multiplications and divisions in SCRPC equals $(n^3 - n)/3$, regardless the choice for the normalization in statement (3). With the choice $\lambda_{kk} = 1$, statement (4) need not be executed, which saves some array-accesses. In that case this algorithm is numerically equivalent with the Crout factorization algorithm (with normalization $v_{kk} = 1$ and row interchanging, [6,13]) applied on A^T. Algorithm Scr with partial pivoting by rowinterchanges (SCRPR) is strictly analogous, in the sense that firstly the provisional column of L is calculated which is updated after the selection of the pivotal row and that the next row of U is calculated after that. The choice of $v_{kk} = 1$ in SCRPR gives an analogous saving in array-accesses and defines Crout's factorization on the original matrix.

SCCPR

For $j = 1, \ldots, n$

  For $i = 1, \ldots, j-1$

(1) $\alpha_{ij} \leftarrow v_{ij} = \left(\alpha_{ij} - \sum_{k=1}^{i-1} \lambda_{ik} v_{kj} \right) / \lambda_{ii}$ (* column of U *)

  For $i = j, \ldots, n$

(2) $\alpha_{ij} \leftarrow \tilde{\lambda}_{ij} = \alpha_{ij} - \sum_{k=1}^{j-1} \lambda_{ik} v_{kj}$ (* provisional column of L *)

(3) $\begin{cases} \text{Determine } p \in \{j, j+1, \ldots, n\} : |\tilde{\lambda}_{pj}| = \max_{j \le i \le n} |\tilde{\lambda}_{ij}| & (* \text{ select pivot } *) \\ a_{j\cdot} \leftrightarrow a_{p\cdot} & (* \text{ interchange rows } *) \end{cases}$

(4) $\alpha_{jj} \leftarrow \begin{Bmatrix} \lambda_{jj} \\ v_{jj} \end{Bmatrix} = \alpha_{jj}$ (* choose normalization *)

For $i = j + 1, \ldots, n$

(5) $\alpha_{ij} \leftarrow \lambda_{jj} = \tilde{\lambda}_{ij} / v_{jj}$ (* update column of L *)

◇ SCCPR

The number of multiplications and divisions in this algorithm is again $(n^3 - n)/3$, regardless the choice made in (4). With the choice $v_{jj} = 1$, statement (5) becomes superfluous, which saves some work; the resulting decomposition in that case is mathematically equivalent with the result of Crout's algorithm. With choice $\lambda_{jj} = 1$, the algorithm defines version (*jik*) in [4] and gives as result of the factorization a decomposition of A which is equivalent with Doolittle's factorization.

CCCPR

For $j = 1, \ldots, n$

For $k = 1, \ldots, j - 1$

(1) $\alpha_{kj} \leftarrow v_{kj} = \alpha_{kj} / (\lambda_{kk} \delta_k)$ (* next elm. in column of U *)

For $i = k + 1, \ldots, n$

(2) $\alpha_{ij} \leftarrow \alpha_{ij} - \lambda_{ik} (\delta_k v_{kj})$ (* update jth column of A*)

(* jth provisional column of L has been calculated in $\boldsymbol{a}_{\cdot j}$ *)

(3) $\begin{cases} \text{Determine } p \in \{ j, j+1, \ldots, n \} : |\alpha_{pj}| = \max\limits_{j \leq i \leq n} |\alpha_{ij}| & (* \text{ select pivot } *) \\ a_{j\cdot} \leftrightarrow a_{p\cdot} & (* \text{ interchange rows } *) \end{cases}$

(4) $\alpha_{jj} \leftarrow \begin{Bmatrix} \lambda_{jj} \\ \delta_j \\ v_{jj} \end{Bmatrix} = ?$ (* choose normalization *)

For $i = j + 1, \ldots, n$

(5) $\alpha_{ij} \leftarrow \lambda_{ij} = \alpha_{ij} / (\delta_j v_{jj})$ (* update column of L *)

◇ CCCPR

As in the algorithms presented before, the number of multiplications and divisions is $(n^3 - n)/3$, regardless most choices for the normalizations made in (4). With the normalization choice $v_{jj} = \lambda_{jj} = \delta_j^{-1}$, it is clear that fewer array-accesses are required because of the fact that statements (1) and (5) become trivial.

CCRPC

Here we describe the essential part of the algorithm on element level (in section 2.3 it is described using entire columns throughout).

For $k = 1, \ldots, n$

(1) $\begin{cases} \text{Determine } p \in \{k, k+1, \ldots, n\} : |\alpha_{kp}| = \max\limits_{k \leqslant j \leqslant n} |\alpha_{kj}| & (* \text{ interchange columns } *) \\ a_{\cdot k} \leftrightarrow a_{\cdot p} & (* \text{ select pivot } *) \end{cases}$

(2) $\alpha_{jj} \leftarrow \left\{ \begin{matrix} \lambda_{jj} \\ \delta_j \\ \upsilon_{jj} \end{matrix} \right\} = ?$ (* choose normalization *)

For $i = k+1, \ldots, n$

(3) $\alpha_{ik} \leftarrow \lambda_{ik} = \alpha_{ik}/(\delta_k \upsilon_{kk})$ (* update column of L *)

For $j = k+1, \ldots, n$

(4) $\alpha_{kj} \leftarrow \upsilon_{kj} = \alpha_{kj}/(\lambda_{kk}\delta_k)$ (* next elm. in row of U *)

For $i = k+1, \ldots, n$

(5) $\alpha_{ij} \leftarrow \alpha_{ij} - \lambda_{ik}(\delta_k \upsilon_{kj})$ (* update jth column of A *)

(* $(k+1)$st provisional column of L has been calculated in $a_{\cdot k+1}$ *)

◇ CCRPC

With the choice $\lambda_{kk} = \upsilon_{kk} = \delta_k^{-1}$, statements (2), (3) and (4) become trivial. The amount of work still remains $(n^3 - n)/3$, but the number of array-accesses is minimized.
With this choice CCRPC reduces to:

For $k = 1, \ldots, n$

$\begin{cases} \text{Determine } p \in \{k, k+1, \ldots, n\} : |\alpha_{kp}| = \max\limits_{k \leqslant j \leqslant n} |\alpha_{kj}| & (* \text{ interchange columns } *) \\ a_{\cdot k} \leftrightarrow a_{\cdot p} & (* \text{ select pivot } *) \end{cases}$

For $j = k+1, \ldots, n$
  For $i = k+1, \ldots, n$
    $\alpha_{ij} \leftarrow \alpha_{ij} - \alpha_{ik}(\alpha_{kj}/\alpha_{kk})$

◇ CCRPC

Next we present GAUSSJ which implements the Gauss–Jordan algorithm with partial pivoting by row 'interchanges', without actually interchanging rows.

GAUSSJ

For $i = 1, \ldots, n$, $\alpha_{i,n+1} = \beta_i$
For $i = 1, \ldots, n$

$\text{Ind}_i = \{1, 2, \ldots, n\} \setminus \{\pi 1, \ldots, \pi(i-1)\}$

Determine $p \in \text{Ind}_i$: $|\alpha_{pi}| = \max_{(k \in \text{Ind}_i)} |\alpha_{ki}|$

$\pi i = p$

$\delta_i = \alpha_{pi}$

$g_i = e_p - (1/\delta_i) a_{\cdot i}$

For $k = i+1, \ldots, n+1$
$a_{\cdot k} \leftarrow a_{\cdot k} + \alpha_{pk} g_i$

For $i = 1, \ldots, n$, $\xi_i = \alpha_{\pi i, n+1}/\delta_i$

◇ GAUSSJ

For the implementation of this algorithm, we use an array diag (say) to store both the inverses of the elements δ_i and the information for the set Ind_i. At start all elements of this array are set to zero; in the ith step the element $\text{diag}(\pi i)$ is set equal to δ_i^{-1}; in that case the truth-value of $(p \in \text{Ind}_i)$ is equivalent to $(\text{diag}(p) = 0)$.

References

[1] P.A. Businger, Monitoring the numerical stability of Gaussian elimination, *Numer. Math.* **16** (1971) 360–361.
[2] J.J. Dongarra, Performance of various computers using standard linear equations software, Argonne, IL, 1983.
[3] J.J. Dongarra, J. Du Croz, S. Hammarling and R.J. Hanson, A proposal for an extended set of FORTRAN Basic Linear Algebra Subprograms, Technical Memorandum 41, Argonne Nat. Lab., 1984.
[4] J.J. Dongarra, F.G. Gustavson and A. Karp, Implementing linear algebra algorithms for dense matrices on a vector pipeline machine, *SIAM Rev.* **26** (1984) 91–112.
[5] J.J. Dongarra, C.B. Moler, J.R. Bunch and G.W. Stewart, *LINPACK User's Guide* (SIAM, Philadelphia 1979).
[6] G.H. Golub and C.F. Van Loan, *Matrix Computations* (North Oxford Academic, Oxford, 1983).
[7] P.W. Hemker, Ed., Numal, Numerical procedures in Algol 60, MC Syllabus 47.1, Math. Centrum, Amsterdam, 1981.
[8] R.W. Hockney, The $n_{1/2}$ method of algorithm analysis, Computer Sci. Dept., Reading University, Reading, U.K., 1983.
[9] R.W. Hockney and C.R. Jesshope, *Parallel Computers — Architecture, Programming and Algorithms* (Adam Hilger, Bristol/Boston, 1981) 423.
[10] W. Hoffmann and W.M. Lioen, Chapter simultaneous linear equations, Report NM-R8614, in: NUMVEC FORTRAN library manual, Centre for Mathematics and Computer Science, Amsterdam, 1986.
[11] NAG Library Manual, Numerical algorithms group, Oxford 1982.
[12] QQLIB, A library of utility routines and math. algorithms on the Cyber 200, Cyber 200 support, Roseville, MN., 1983.
[13] G.W. Stewart, *Introduction to Matrix Computations* (Academic Press, New York/London, 1973).

Those who device numerical methods must be aware that any successful development will, at best, only placate the demon Progress for a short time. One is reminded of the many-headed hydra that menaced the ancient Greeks; no sooner was one head lopped off than several more grew in its place.

J.R. Rice et all.
Numerical Computation, its Nature and Research Directions;
Special Issue of ACM SIGNUM Newsletter, February 1979.

CHAPTER II

Rehabilitation of the
Gauss-Jordan algorithm

jointly written with T.J. Dekker
accepted for publication in:
Numerische Mathematik, 1989

REHABILITATION OF THE GAUSS-JORDAN ALGORITHM

by

T. J. Dekker and W. Hoffmann

Summary. In this paper a Gauss-Jordan algorithm with column interchanges is presented and analysed. We show that, in contrast with Gaussian elimination, the Gauss-Jordan algorithm has essentially differing properties when using column interchanges instead of row interchanges for improving the numerical stability. For solutions obtained by Gauss-Jordan with column interchanges, a more satisfactory bound for the residual norm can be given. The analysis gives theoretical evidence that the algorithm yields numerical solutions as good as those obtained by Gaussian elimination and that, in most practical situations, the residuals are equally small. This is confirmed by numerical experiments. Moreover, timing experiments on a Cyber 205 vector computer show that the algorithm presented has good vectorization properties.

Subject classification: AMS(MOS): 65F05, 65G05, 15A06; CR: G1.3 .

1. INTRODUCTION

With the advent of vector and parallel computers, the Gauss-Jordan algorithm has received renewed interest because of its supposedly good properties with respect to vectorization and parallelization.

The stability of the Gauss-Jordan algorithm with partial pivoting has been analysed by Peters and Wilkinson [6] who came to the following conclusion: "*in general the absolute error in the solution is strictly comparable with that corresponding to Gaussian elimination with partial pivoting plus back substitution; however, when the matrix is ill conditioned, the residual corresponding to the Gauss-Jordan solution will often be much greater than that corresponding to the Gaussian elimination solution.*" These results hold true for the standard column pivoting strategy, where at each stage a pivot is selected in a certain column and correspondingly rows are interchanged to bring the pivot in diagonal position.

In this paper we show that Gauss-Jordan with row pivoting, and correspondingly interchanging of columns, is much more satisfactory. In most practical situations, the residual corresponding to the solution obtained by Gauss-Jordan with row pivoting is not larger than that corresponding to the Gaussian elimination solution.

The Gauss-Jordan algorithm with any pivoting strategy is equivalent to Gaussian elimination - with the same pivoting strategy - followed by a further reduction of the resulting upper triangular system to a diagonal system. With column pivoting this further reduction may yield arbitrarily large elements and, hence, a large residual, as is shown by Peters and Wilkinson. With row pivoting, however, the elements of the resulting upper triangular matrix are bounded by the diagonal elements in the corresponding rows.

For an error analysis it is convenient to consider Gauss-Jordan's algorithm with row scaling, i.e. at each stage the equation corresponding to the pivotal row is divided by the pivot. Then the resulting matrix U is unit upper triangular and its elements are bounded by 1. It follows that the growth of the elements in the further reduction to diagonal form is not much larger than the norm of the inverse of U. Consequently, the residual of the calculated solution is not much larger than that corresponding to the Gaussian elimination solution, except in those rare cases where U is ill conditioned.

In section 2 we consider the Gauss-Jordan algorithm in more detail and present an error analysis. In section 3 we give some numerical results, showing that the error and mostly also the residual are satisfactory, and some results of experiments on a Cyber 205 vector computer, showing that Gauss-Jordan is not slower than Gaussian elimination for systems of order up to 25, although it requires about 1.5 times more work.

2. Error Analysis of Gauss-Jordan with Row Pivoting

Let A be a given matrix of order n and b a given right-hand side vector. The application of the Gauss-Jordan elimination on the given system is equivalent with performing n successive transformations, starting from the original matrix $A^{(1)} = A$ and right-hand side $b^{(1)} = b$. The total effect is the transformation of $A^{(1)}$ with permuted columns into the identity matrix. This is described in the following algorithm.

```
For k = 1 (1) n do
   Determine p such that k≤p≤n and |A(k)kp| = max_{k≤j≤n} |A(k)kj|
   Pk   :=  I - (ek-ep)(ek-ep)^T
            {permutation matrix for interchanging columns p and k}
   δk   :=  A(k)kp
   Dk   :=  I + (δk - 1) ek ek^T    {= diag(1,...,1,δk,1,...,1) }
   gk   :=  δk ek - A(k) ep
   Gk   :=  gk ek^T
   A(k+1) := (I + Gk) Dk^-1 A(k) Pk
   b(k+1) := (I + Gk) Dk^-1 b(k)
enddo
```

Figure 1. Gauss-Jordan algorithm with column interchanges

The application of this algorithm results in $A^{(n+1)} = I$ and $b^{(n+1)} = P_n^{-1} \ldots P_1^{-1} x$.

Summarizing, with the use of $P = P_1 \ldots P_n$, the effect of all elimination-steps is given by:

$$(I + G_n)D_n^{-1} \ldots (I + G_1)D_1^{-1}(AP \mid b) = (I \mid P^{-1}x) .$$

For our theoretical analysis we introduce the following notation.

Define m_k to be equal to the lower part of g_k (below the k-th element) and v_k equal to the upper part of g_k such that matrix $[m_1, m_2, \ldots, m_n]$ is strictly lower triangular and matrix $[v_1, v_2, \ldots, v_n]$ strictly upper triangular.

Furthermore

$$M_k := m_k e_k^T \quad \text{and} \quad V_k := v_k e_k^T .$$

Then we have

$$g_k = m_k + v_k , \quad G_k = M_k + V_k ,$$

and

$$(I + G_k) = (I + V_k)(I + M_k) .$$

We observe that

$$(I + M_i)D_i^{-1}(I + V_j) = (I + V_j)(I + M_i)D_i^{-1}, \quad \text{for } j < i ,$$

so that

$$(I + G_n)D_n^{-1} \ldots (I + G_1)D_1^{-1} = (I + V_n) \ldots (I + V_1)(I + M_n)D_n^{-1} \ldots (I + M_1)D_1^{-1}.$$

If L and V are defined by

$$L := [(I + M_n)D_n^{-1} \ldots (I + M_1)D_1^{-1}]^{-1} = D_1(I - M_1) \ldots D_n(I - M_n) = (D_1 \ldots D_n - M_1 - \ldots - M_n),$$

$$V := (I + V_n) \ldots (I + V_1) = I + V_1 + \ldots + V_n ,$$

then the Gauss-Jordan elimination is symbolically given by:

$$V L^{-1}(AP \mid b) = (I \mid P^{-1}x).$$

Let the upper triangular matrix U and vector y, which are intermediate results during the calculation, be defined by

$$U := L^{-1} A P \; ; \quad y := L^{-1} b,$$

then the error analysis of (standard) Gaussian elimination shows that these calculated L, U and y satisfy:

$$L U = AP + E_1, \text{ with } \|E_1\| \leq \phi_1(n)\, g\, \|A\|\, \mu, \tag{2.1}$$

and

$$(L + E_2)\, y = b, \text{ with } \|E_2\| \leq \phi_2(n)\, \|L\|\, \mu, \tag{2.2}$$

where $\phi_1(n)$ and $\phi_2(n)$ are low-degree polynomials in n, g is the growth factor and μ is a small, arithmetic-dependent, constant times the machineprecision. (See e.g. [3,7]).

For the rest of our rounding error analysis we have to examine the remaining part of the algorithm. This is the part where V is calculated such that

$$V\ (U \mid y) = (I \mid P^{-1}x)\ .$$

The pivoting strategy and row-scaling in the first part of the algorithm have ensured that $|U_{ij}| \leq 1$ for $j > i$ and $U_{ii} = 1$.
Defining $U^{(1)} = U$ and $y^{(1)} = y$, the calculation is carried out according to the following rules. Note that these rules are part of the algorithm described in figure 1.

```
For k = 1 (1) n do
   v_k   :=  e_k - U^(k) e_k
   V_k   :=  v_k e_k^T
   U^(k+1) := (I + V_k) U^(k)
   y^(k+1) := (I + V_k) y^(k)
enddo
```

Figure 2. Inversion and solution of triangular system

The result of this calculation is $U^{(n+1)} = I$ and $y^{(n+1)} = P^{-1}x$.
For the calculated quantities we observe that for each k an error matrix $F^{(k)}$ exists such that

$$U^{(k+1)} = U^{(k)} + V_k\, U^{(k)} + F^{(k)}, \text{ with } F^{(k)}_{ij} = 0 \text{ for } j \leq k \text{ and } i \geq k\ .$$

A simple rounding error analysis yields:

$$\max |F^{(k)}_{ij}| \leq 3 \max_{i<k} |U^{(k)}_{ij}|\, \mu.$$

Since $(I + V_i)\, F^{(k)} = F^{(k)}$ for $i \geq k$, we obtain

$$I = U^{(n+1)} = (I + V_n) \ldots (I + V_1)\, U + F^{(1)} + F^{(2)} + \ldots + F^{(n)}\ .$$

For V this implies

$$VU + E_3 = I, \text{ with } E_3 = F^{(1)} + F^{(2)} + \ldots + F^{(n)}\ . \tag{2.3.a}$$

For an estimate of E_3 we need a bound for $\max_{i<k} |U^{(k)}_{ij}|$.
From

$$U^{(k)} = (I + V_1 + \ldots + V_{k-1})U,$$

we find

$$\max_{i<k} |U^{(k)}_{ij}| \leq \max_{i<k} |U_{ij}| \left\{1 + \sum_{h=1}^{k-1} \max_{ij} |(V_h)_{ij}|\right\} \leq$$

$$\{1 + (k-1)\ .\ \max |V_{ij}|\} \leq \{1 + (k-1)\, \|V\|\}\ .$$

Using $\|F^{(k)}\| \leq n \max_{ij}(|F^{(k)}_{ij}|)$, we find for $\|E_3\|$

$$\|E_3\| \le \sum_{k=1}^{n} \|F^{(k)}\| \le 3\,n \sum_{k=1}^{n} \{1 + (k-1)\,\|V\|\}\,\mu$$

which gives

$$\|E_3\| \le \phi_3(n)\,\|V\|\,\mu \qquad (2.3.b)$$

for a low-degree polynomial ϕ_3 in n.

With respect to the error in the calculated solution $z = P^{-1}x$, we notice that this calculation is numerically equivalent with multiplying y from the left by V, hence

$$(V + E_4)\,y = z, \quad \text{with} \quad \|E_4\| \le \phi_4(n)\,\|V\|\,\mu\,, \qquad (2.4)$$

for a low-degree polynomial ϕ_4 in n.

The combination of formulae (2.3 and 2.4) yields:

$$y = U\,(I - E_3 + E_4U)^{-1}\,z,$$

which in combination with (2.1 and 2.2) gives

$$b = (A + E_1 + E_2U)(I - E_3 + E_4U)^{-1}z\,. \qquad (2.5)$$

If we put

$$w = (I - E_3 + E_4U)^{-1}z\,, \qquad (2.6.a)$$

then this results in

$$b = (A + E_1 + E_2U)w\,. \qquad (2.6.b)$$

If we furthermore use E_5 for $(E_3 - E_4U)$ then the distance between z and w satisfies

$$\|z - w\| / \|z\| \le \|E_5\| / (1 - \|E_5\|) \text{ provided that } \|E_5\| < 1\,. \qquad (2.7)$$

For $\|E_5\|$ we find the following bound

$$\|E_5\| \le \|E_3\| + \|E_4\|\,\|U\| \le (\phi_3(n) + \phi_4(n)\,\|U\|)\,\|V\|\,\mu \le \phi_5(n)\,\|V\|\,\mu \qquad (2.8)$$

for a low-degree polynomial ϕ_5.

Using 2.3.a for a bound on $\|V\|$ this can be written as

$$\|E_5\| \le \phi_5(n)\,\|U^{-1}\|\,\mu\,/\,\{1 - \phi_3(n)\,\|U^{-1}\|\,\mu\}, \qquad (2.9)$$

provided that the denominator is positive.

Summarizing, the calculated solution $z = P^{-1}x$ is close to a vector w, which is the exact solution of a nearby problem as specified in formulae 2.6.a and 2.6.b.

For the residual $r := b - Az$ we have according to these formulae

$$r = (A + E_1 + E_2U)\,w - A(I - E_5)\,w\,,$$

which can be bounded by

$$\|r\| \le (\|E_1\| + \|E_2U\| + \|A\|\,\|E_5\|)\,\|z\| / (1 - \|E_5\|). \qquad (2.10)$$

In this bound the contribution $\|A\|\,\|E_5\| / (1 - \|E_5\|)$ creates the essential difference with the formula for the residual bound for Gaussian elimination.

As long as $\|E_5\| \ll 1$, this term has order of magnitude $\|A\|\,\|U^{-1}\|\,\mu$. As a consequence of our pivoting strategy, U will mostly be well-conditioned, even in cases where A itself is ill-conditioned, so that the contribution of this term is harmless. However, a well known example of an ill-conditioned unit triangular matrix is given in the next section in experiments series d.

3. NUMERICAL EXPERIMENTS

Experiments on accuracy and timing were carried out on the Cyber 205 computer (one vector pipe) of the Academic computer centre SARA in Amsterdam; the arithmetic precision of this machine is about 10^{-14}.
For a large number of linear systems we compared the solution obtained via Gauss-Jordan with row pivoting with the solution from Gaussian elimination. These experiments are described hereafter and listed in Table 1.
For timing results we compared the CP time for our Gauss-Jordan algorithm with the CP time for LINPACK-routines SGESL and SGEFA [2] and with the CP time for the NUMVEC implementation of $LD^{-1}U$ factorization with row pivoting (which is equivalent with Gaussian elimination) followed by forward and backward substitution [4,5]. An overview of these results is given in Table 2.

Our implementation of the Gauss-Jordan algorithm is a slight modification of the algorithm described above. In each step the factor D_k^{-1} is omitted so that the resulting matrix is given by $A^{(n+1)} = D = \mathrm{diag}(\delta_1, \ldots, \delta_n)$. In this form the algorithm is more efficiënt on the Cyber 205 vector computer, because no extra updating of the pivotal row in each step is required. The error analysis remains essentially the same.

Experiments on accuracy and residuals.

a) In test series a1 - a4 we use linear systems with prescribed condition. The matrices are constructed from a given diagonal matrix (the singular values chosen) which is pre- and post- multiplied by random orthogonal matrices. These left and right orthogonal factors are the product of $\sqrt{n}$ random Householder reflections. The singular values are chosen in various ways; the largest always +1, the smallest 10^{-6} or smaller and the remaining ones either distributed equally, or clustered on one end of the spectrum, or on the other end.

In series a1 we use very ill-conditioned matrices of order 25; the right-hand side vector b is constructed by taking the product of the coefficient-matrix A and a random vector x.

In series a2 we use matrices of order 50 of the same type and with the same type of right-hand side vector .

In series a3 we use a different type of right-hand side vector. Firstly the linear system is solved with a random right-hand side vector. With the solution x_0 of this system, the vector $b_0 = Ax_0$ is calculated. This vector b_0 serves as right-hand side vector in the test system. For ill-conditioned matrices the right-hand side vector constructed in this way is in general "rich" in the least singular vector of the matrix, so that the solution is very sensitive for perturbations.

In series a4 the left singular vector corresponding to the least singular value is taken as right-hand side vector.

All these series yield solutions with accuracy as expected in view of the condition number of the matrix, and small residuals both for Gaussian elimination and Gauss-Jordan factorization with no significant difference.

b) In series b1 - b2 we use upper triangular matrices. The diagonal elements have the value +1 except A_{33} and A_{44} which have the value 10^{-7}. The elements in the strictly upper triangular part have random values between -1 and +1. This type of matrices is used by Peters and Wilkinson [6] to show that Gauss-Jordan with column pivoting can produce larger residual vectors than Gaussian elimination. The choices for the right-hand sides in b1 and b2 are made in the same way as in series a1 and a3 respectively. The results of Gauss-Jordan factorization with row pivoting and Gaussian elimination are fully comparable and as accurate as can be expected in view of the condition of the matrices.

We also tested Gauss-Jordan with column pivoting on these matrices. The accuracy of the solution is comparable with the accuracy in the other solutions, but the residual is much larger (of the same size as the error in the solution), which confirms the analysis in [6].

c) In series c1 - c2 we use a matrix W for which maximal growth in its elements is obtained during Gaussian elimination with partial pivoting. For our situation where row pivoting is performed, this matrix is given by

$W_{ij} = -1$ for $j > i$; $W_{jj} = W_{nj} = 1$ for all j and $W_{ij} = 0$ elsewhere.

$$W = \begin{pmatrix} 1 & -1 & \cdots & -1 \\ 0 & 1 & & -1 \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -1 \\ 1 & \cdots & & 1 & 1 \end{pmatrix}$$

For n = 50, as used in series c1, the conditionnumber of W roughly equals 1700 and the element growth is 2^{49} . As right-hand side vector we have chosen the product Wx for a random vector x.

In series c2 we use the same experiment but now for the order n = 30.
The results of Gaussian elimination are for some cases slightly better than with the Gauss-Jordan factorization.

Note that for this matrix, a result obtained with column pivoting is correct to almost full working accuracy; this is true for both Gaussian elimination and Gauss-Jordan factorization. An implementation of a variant of Gaussian elimination where this dangerous element-growth is detected and can be cured is given by Hoffmann and Lioen [5]. The technique used is presented in a paper by Businger [1] and can also be applied to the Gauss-Jordan algorithm.

d) In series d1 - d2 we use a unit upper triangular matrix D having all elements in the strictly upper triangular part equal to -1.

$$\Delta = \begin{pmatrix} 1 & -1 & \cdots & -1 \\ 0 & 1 & & \vdots \\ \vdots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & 1 \end{pmatrix}$$

The least singular value of Δ is less than $2^{-n}\sqrt{3}$. As right-hand side vector, we take matrix times random vector. For this type of matrices, which have an increasing bad condition for growing values of the order n, the Gauss-Jordan factorization for large values of n produces a solution with a much larger residual vector than the solution produced by Gaussian elimination.

In all our experiments we calculated the number of correct digits in the solution and in the residual.
For a system with right-hand side b, exact solution x_0 and calculated solution x these numbers are given by $^{-10}\log(\|x - x_0\| / \|x_0\|)$ and $^{-10}\log(\|b - Ax\| / \|x_0\|)$ respectively. In fact the latter quotient between brackets should also be divided by $\|A\|$ for a homogeneous result. However, for all matrices in series a we have $\|A\|_2 = 1$ and for all other matrices the norms are of order n, so the omission of this factor is harmless.

							#correct digits (rel. to x)			
							solution		residual	
series	order	#syst.	A	x	b	cond. nr	Gauss	G.-J.	Gauss	G.-J.
a1	25	200	$U\Sigma V^T$	random	Ax	10^{15}	< 2	same	> 12	same
a2	50	1800	$U\Sigma V^T$	random	Ax	10^6-10^{15}	< 8	same	> 12	same
a3	50	200	$U\Sigma V^T$	A-1b	random	10^{10}	3 - 7	same	> 12	same
a4	50	200	$U\Sigma V^T$	v_{50}	$\sigma_{50}u_{50}$	10^{10}	3 - 7	same	> 12	same
b1	25	200	upper	random	Ax	> 10^7	< 2	same	> 12	same
b2	25	200	upper	A-1b	random	> 10^7	< 2	same	> 12	same
c1	50	200	W	random	Ax	1700	0 - 4	0 - 1	0 - 4	0 - 1
c2	30	200	W	random	Ax	<1700	5 - 10	5 - 7	5 - 9	5 - 6
d1	50	200	Δ	random	Ax	> 10^{14}	0 - 4	0 - 1	> 12	0 - 1
d2	30	200	Δ	random	Ax	≈ 10^{10}	5 - 10	5 - 7	> 12	6 - 7

Table 1. Overview of experiments on accuracy

Experiments on timing.
Our implementation off the Gauss Jordan algorithm with row pivoting (column interchanges) GJPCF was compared with routines from LINPACK and with the NUMVEC implementation of Gaussian elimination, CCRPCF [4,5]. The timing results are as in the following table.

n =	25	50	100	200
LINPACK (SGEFA + SGESL)	0.0028	0.0107	0.0441	0.1965
CCRPCF	0.0014	0.0051	0.0232	0.1154
GJPCF	0.0014	0.0054	0.0256	0.1394

Table 2. Overview of timing experiments

These results show that the Gauss-Jordan algorithm is rather efficiënt on a vector computer and that the processing time is competitive with Gaussian elimination for order up to 25.
As is well known, the number of floating-point operations equals, apart from lower order terms, n^3 for Gauss-Jordan and $(2/3)n^3$ for Gaussian elimination. The time needed for these algorithms, however, is not only determined by this order n^3 term, but also by a significant contribution of order n^2, needed for pivot search and interchanges and, on a vector machine, also for the start-up of the vector iterations. This contribution of order n^2 is (nearly) equal for CCRPCF and GJPCF, as also appears from this table.

ACKNOWLEDGEMENTS.
We want to thank professors G.H. Golub, A. Ruhe and G.W. Stewart for mentioning the second author the key reference [6] at the conference at Loen, Norway.

REFERENCES

1. Businger, P.A.: Monitoring the numerical stability of Gaussian elimination. Numer. Math. **16**, 360-361 (1971)
2. Dongarra, J.J.,Moler, C.B., Bunch, J.R. and Stewart, G.W.:LINPACK User's guide: Philadelphia: SIAM 1979
3. Golub, G.H. and Van Loan, C.F.: Matrix Computations: Oxford: North Oxford Academic 1983
4. Hoffmann, W.: Solving Linear Systems on a Vector Computer: to appear in Journal of Computational and Applied Mathematics
5. Hoffmann, W. and Lioen, W.M.: Chapter Simultaneous Linear Equations. Report NM-R8614. In: NUMVEC FORTRAN Library Manual. Amsterdam: Centre for Mathematics and Computer Science, 1986
6. Peters, G. and Wilkinson, J.H.: On the Stability of Gauss-Jordan Elimination with Pivoting. Comm. of the ACM **18**, 20-24 (1975)
7. Stewart, G.W.: Introduction to matrix computations. New York and London: Academic Press 1973

CHAPTER III

An Estimate for the Spectral Norm of the Inverse of a Matrix with the Gauss-Jordan Algorithm

reworked version of:
report 87-06, Dept. of Mathematics, Univ. of Amsterdam (1987)

An Estimate for the Spectral Norm of the Inverse of a Matrix with the Gauss-Jordan Algorithm

by

W. Hoffmann

Abstract. In this paper an algorithm is presented for calculating an estimate for the spectral norm of the inverse of a matrix. This algorithm is to be used in combination with solving a linear system by means of the Gauss-Jordan algorithm. The norm of the inverse is needed for the condition number of that matrix. The algorithm exploits the effect that Gauss-Jordan elimination is equivalent with writing the matrix as a product of n elementary matrices. These elementary matrices are sequentially used to maximize (locally) the norm of a solution vector that matches right-hand side vector under construction. In n steps this produces a satisfactory estimate. Our algorithm uses $5n^2 + O(n)$ extra floating-point multiplications for the calculation of the required estimate and is tested for a multitude of matrices on the Cyber 205 vector computer of the Academic Computer Centre, SARA, in Amsterdam.

Keywords and phrases:
Gauss-Jordan algorithm, condition estimation, vector algorithms.

AMS subject classification:
65F05, 65G05, 15A06.

1. INTRODUCTION

In this paper we present an algorithm for estimating the spectral norm of the inverse matrix of a linear system. A norm of the inverse matrix is used to calculate the condition number with respect to inversion or solving linear systems. Many articles on condition number estimation and related problems have already been published [1,2,3,8,9,14,17,18]. In these publications an LU factorization or QR decomposition of the coefficient matrix A is considered and all algorithms use forward and/or backward substitution for the solution of triangular systems. Our estimator is designed for the Gauss-Jordan algorithm which can be viewed as an algorithm for writing a matrix as the product of n elementary matrices [11].

In spite of the fact that solving a linear system by Gauss-Jordan uses approximately 1.5 times as many operations compared to Gaussian elimination, the interest in the Gauss-Jordan algorithm has not diminished in recent years [16,19,23]. On the contrary, the introduction of parallel and vector computers induced a revival of the Gauss-Jordan algorithm; its application has some advantages in the parallel case.

The Gauss-Jordan algorithm in its standard form was shown to be unsafe [16], but it can be made practically as stable as Gaussian elimination with partial pivoting by using suitable column interchanges instead of row interchanges as has been shown in [4].

In section 2 we briefly overview the stabilized version of the Gauss-Jordan algorithm and extend it with our rules for estimating $\|A^{-1}\|_2$. Our estimator uses the factorization of A into the product of elementary matrices $(I + g_k e_k^T)$ for the construction of two vectors, t and f, for which $At = f$ and $\|t\|_2 / \|f\|_2$ is 'large'. We recall that the spectral norm (or l_2-norm) of matrix A^{-1} is defined by the supremum of $\|A^{-1}f\|_2 / \|f\|_2$, taken over all non-zero vectors f.

In section 3 we present the results of computer experiments. For a large number of matrices we compared our estimate for $\|A^{-1}\|_2$ with the correct value. For a specific selection of test matrices we present our results in a form that makes them comparable with results of estimating the norm of the inverse matrix that have been presented in the literature. In this respect we especially mention the work of Higham [9], who reviewed a number of estimators for $\|A^{-1}\|_p$ for various values of p, (including p = 2), but none of which were especially suited to work with the Gauss-Jordan factorization.

In section 4 we present our conclusions.

2. A SUPPLEMENT TO THE GAUSS-JORDAN ALGORITHM

We consider the Gauss-Jordan algorithm described in[6] using stabilizing column interchanges instead of the traditionally used row interchanges.
Let A be a given matrix of order n. The Gauss-Jordan algorithm consists of n consecutive transformation steps reducing the matrix to a diagonal matrix. The basic form of the algorithm we use is repeated here:

```
For k = 1 (1) n do
   Determine p such that k≤p≤n and |A(k)kp| = max_{k≤j≤n} |A(k)kj|
   Pk   :=  I - (ek-ep)(ek-ep)^T
            {permutation matrix for interchanging columns p and k}
   δk   :=  A(k)kp
   Dk   :=  I + (δk - 1) ekek^T    {= diag(1,...,1,δk,1,...,1) }
   gk   :=  δk ek - A(k) ep
   A(k+1) := (I + gk ek^T )Dk^-1 A(k) Pk
   b(k+1) := (I + gk ek^T )Dk^-1 b(k)
enddo
```

For the description of our algorithm to estimate $\|A^{-1}\|_2$, we recall that Gauss-Jordan can be used for the explicit calculation of A^{-1}. For that purpose it should be extended with the calculation of the iterated product of matrices $(I + g_k e_k^T)D_k^{-1}$ as described in the following rules:

$$B^{(0)} = I, \text{ and}$$
$$B^{(k)} = (I + g_k e_k^T)D_k^{-1}B^{(k-1)} \quad ,k = 1, 2, \ldots, n. \tag{2.1.}$$

If P denotes the product of all permutation matrices applied, i.e. $P = P_1 \cdots P_n$, then one can verify for the matrix $B^{(n)}$:

$$A^{-1} = PB^{(n)}. \tag{2.2.}$$

We would like to remark that in a practical implementation to calculate A^{-1}, one likely uses the fact that A^{-1} can overwrite A and can be calculated in the memory location that has been occupied by A itself. An implementation using this observation combined with the stabilized Gauss-Jordan algorithm is presented in [5].

We here consider, however, the calculation of $\|A^{-1}\|_2$ in a situation where A^{-1} itself is not needed. We require that computing the estimate of the norm should be feasible in $O(n^2)$ operations so that it can be seen as only a slight overhead in the total cost for the factorization of A itself.
The l_2-norm of A^{-1}, which is equal to the l_2-norm of $B^{(n)}$, is defined as the maximal value of
$\|A^{-1}f\|_2$ taken over all vectors f with $\|f\|_2 = 1$. Our algorithm constructs a row of estimates for $\|B^{(k)}\|_2$, k = 1, 2, ..., n.
For this purpose we construct vectors $f_1, f_2, \ldots, f_n$, with $\|f_k\|_2 = 1$, k = 1, 2, ..., n, such that $\|B^{(k)}f_k\|_2$ approximates $\|B^{(k)}\|_2$. An optimal vector f_k is searched for in the 2-dimensional subspace span$\{f_{k-1}, e_k\}$.

This choice is motivated as follows. We would like to deal correctly with the case that $g_k e_k^T$ wipes out all other information in the calculation of $B^{(k)}$ and with the case that $g_k e_k^T$ is negligible compared to $B^{(k-1)}$. In the first case the optimal value for f_k equals e_k and in the other case we observe that f_{k-1} is still optimal.
Our algorithm starts with $f_1 = e_1$; consequently, $f_k \in$ span$\{e_1, \ldots, e_k\}$, k = 1, ... n.
The construction of $f_2, \ldots, f_n$ and $t_1, t_2, \ldots, t_n$, with $t_k = B^{(k)} f_k$ is as follows.
Obviously,

$$t_1 = \delta_1^{-1}(e_1 + g_1) \ . \tag{2.3.}$$

Suppose that for $k > 1$ the unit length vector f_{k-1} is given and that $\|t_{k-1}\|_2 = \|B^{(k-1)}f_{k-1}\|_2$ approximates $\|B^{(k-1)}\|_2$.
In step k, the following optimization problem is to be solved.
Determine λ and μ with $\lambda^2 + \mu^2 = 1$ such that the vector t_k defined by

$$t_k = B^{(k)} f_k = B^{(k)} (\lambda f_{k-1} + \mu e_k), \tag{2.4.}$$

has maximal l_2-norm . We stress that matrix $B^{(k)}$ appearing in this expression is not needed explicitly, which follows from the observations:

i) $B^{(k-1)} e_k = e_k$

and

ii) $B^{(k-1)}f_{k-1} = t_{k-1}$ (by definition). (2.5.)

From (2.3.) we deduce

$$t_k = (I + g_k e_k^T)D_k^{-1} B^{(k-1)}(\lambda f_{k-1} + \mu e_k) = (I + g_k e_k^T)D_k^{-1} (\lambda t_{k-1} + \mu e_k).$$

The l_2-norm of vector t_k yields a homogeneous quadratic equation in λ and μ, the coefficients of which are denoted by α, β and γ:

$$\|t_k\|_2^2 = \alpha\mu^2 + 2\beta\mu\lambda + \gamma\lambda^2. \tag{2.6.}$$

With the introduction of quantities $\tilde{t}_{k-1}$, τ , ζ and δ, defined by

$$\tilde{t}_{k-1} = D_k^{-1}t_{k-1} \text{ , } \tau = e_k^T \tilde{t}_{k-1} \text{ , } \zeta = g_k^T \tilde{t}_{k-1} \text{ and } \delta = \delta_k,$$

the values of α, β and γ are given by:

$$\alpha = \delta^{-2} (1 + \|g_k\|^2), \tag{2.7.}$$

$$\beta = \delta^{-1} \{\tau (1 + \|g_k\|^2) + \zeta\}, \tag{2.8.}$$

$$\gamma = \|\tilde{t}_{k-1}\|^2 + \tau^2 \|g_k\|^2 + 2\tau\zeta. \tag{2.9.}$$

The optimal λ and μ, under the constraint $\lambda^2 + \mu^2 = 1$, can be calculated by means of standard techniques.

a) For $\beta = 0$ we have $\|t_k\|_2^2 = \alpha + (\gamma - \alpha)\lambda^2$ which yields

a1) for $(\gamma - \alpha) \geq 0$: $\lambda = 1$; $\mu = 0$;

a2) for $(\gamma - \alpha) < 0$: $\lambda = 0$; $\mu = 1$.

b) For $\beta \neq 0$ the maximal value for $\|t_k\|_2$ is attained for:

$$\frac{\mu}{\lambda} = \frac{\alpha - \gamma}{2\beta} \pm \sqrt{\left(\frac{\alpha - \gamma}{2\beta}\right)^2 + 1}, \tag{2.10.}$$

where the plus-sign applies if $\beta > 0$ and the minus-sign if $\beta < 0$.
The values for λ and μ follow from the relations

$$\lambda = 1/\sqrt{1 + (\mu/\lambda)^2} \quad \text{and} \quad \mu = (\mu/\lambda) \times \lambda.$$

The vector t_k can be constructed via

$$t_k = \lambda \tilde{t}_{k-1} + \delta^{-1}\mu e_k + (\lambda\tau + \delta^{-1}\mu) g_k . \tag{2.11.}$$

Note that the vector f_k is not needed explicitly, so that finally the calculation of t_n can be realized in $4n^2 + O(n)$ multiplications and $3n^2 + O(n)$ additions.
The resulting vector t_n can be interpreted as a single inverse-iteration step applied to the vector f_n; the fact that f_n itself is constructed during this process is merely a technical detail. Denoting the singular value decomposition of A by $AV = U\Sigma$, it follows that f_n approximates u_n and t_n approximates $\sigma_n^{-1} v_n$. From the theory of inverse iteration applied to the calculation of singular vectors[13], we find that a second iteration step yields a better approximation to the required singular vector. In the general case where the matrix is not symmetric, this second step should be applied with matrix A^T. In our situation this means that $(A^T)^{-1} \frac{t_n}{\|t_n\|}$ approximates $\sigma_n^{-1} u_n$.

Especially when σ_n is small, this can give a considerable improvement in the estimation of $\|A^{-1}\|_2 = \sigma_n^{-1}$.
The Gauss-Jordan algorithm calculates scalars δ_k and vectors g_k, k=1,...,n, for which we have

$$(AP)^{-1} = (I + g_n e_n^T)D_n^{-1} \ldots (I + g_1 e_1^T)D_1^{-1}, \text{ (cf. 2.1. and 2.2.).} \qquad (2.12.)$$

The quantities δ_k and g_k also define the following factorization of $(A^T)^{-1}$:

$$(A^T)^{-1} = D_1^{-1}(I + e_1 g_1^T) \ldots D_n^{-1}(I + e_n g_n^T)\, P^T ; \qquad (2.13.)$$

so that the calculation of $(A^T)^{-1}\frac{t_n}{\|t_n\|}$ can be performed in $n^2 + O(n)$ multiplications and additions.

3. NUMERICAL EXPERIMENTS

The technique as explained in section 2 was built into routine GJPCF [10], our linear system solver based on the Gauss-Jordan algorithm with column interchanges as explained in [4]. In GJPCF we have implemented the stable algorithm without explicit row-scaling in each step. For the application of the algorithm to estimate the norm of the inverse as described in section 2, this explicit row-scaling cannot be avoided, however. This gives an overhead of $\frac{1}{2}n^2 + O(n)$ multiplications in the optimal Gauss-Jordan routine. The effect on the execution time is shown in table 1.
The algorithm as described in section 2 (without the extra inverse iteration step) has been implemented in our routine GJE_NRM_INV1. In this routine, the vector t_n is calculated for an estimate of σ_n^{-1} with an extra amount of work of $4n^2 + O(n)$ multiplications and $3n^2 + O(n)$ additions. The Gauss-Jordan algorithm with explicit row scaling for solving a linear system with one right-hand side vector requires a total amount of work of $\frac{1}{2}n^3 + \frac{1}{2}n^2 + O(n)$ multiplications and the same number of additions.
For a large number of matrices we estimated the l_2-norm of the inverse matrix and compared it with the correct value. In all our experiments with various matrices having orders varying between 10 and 100, we never observed an estimate that was wrong by a factor of more than 10.
The experiments were carried out on the Cyber 205 vectorcomputer of SARA, the Academic Computer Centre in Amsterdam.
In the following table the CPU-time for our implementation of the stabilized Gauss-Jordan algorithm is compared with the total time if only the concept of explicit row scaling is included and with the total time if also the extension for the calculation of $\|t_n\|_2$ is included.

n =	25	50	100	200
GJPCF	0.0014	0.0054	0.0256	0.1394
GJPCF, with expl. rowscaling	0.0014	0.0056	0.0260	0.1404
GJE_NRM_INV1; (estimates also $\|A^{-1}\|_2$)	0.0019	0.0067	0.0285	0.1479

Table 1. Solution of a linear system having one right-hand side for various orders.

For two classes of matrices we report the behaviour of our algorithm more extensively. These are classes of matrices that have been used by Higham in

his overview on various algorithms for estimating a norm of A^{-1} [9]. We present our results in the same form that Higham used, in order to enable easy comparison.

The matrices we use in our overview are constructed with prescribed singular values. A given diagonal-matrix, containing the singular values, is pre- and postmultiplied with pseudo-random (products of) orthogonal Householder matrices. This way of constructing pseudo-random test matrices is described in [17].

For various values of the order n, varying from 10 to 100 and for various values of the condition number κ_2, varying from 10 to 10^9, we used groups of one hundred pseudo-random matrices. For each matrix we calculate the quotient (< 1) of the estimated norm of the inverse and the true value of that norm. For each group of one hundred matrices we calculate the arithmetic mean of these quotients and we report the minimal value. The results of these tests are given in the following tables.

Firstly we use a group of matrices having their singular values distributed exponentially.
The singular values σ_i are defined by $\sigma_i = \alpha^{i-1}$, i = 1, ..., n for an appropriate constant α such that $\kappa_2 = \alpha^{-(n-1)}$.

Test results with $\sigma_i = \alpha^{i-1}$, i = 1,...,n; GJE_NRM_INV1.

κ_2	n = 10	25	50	100
10	.50/.77	.62/.81	.67/.86	.84/.90
10^3	.27/.69	.45/.73	.61/.80	.78/.87
10^6	.19/.62	.35/.68	.58/.78	.73/.83
10^9	.19/.64	.29/.69	.58/.77	.71/.83

Minimum and mean over 'estimate' / $\|A^{-1}\|_2$ for groups of 100 matrices.
Table 2.

Secondly we use a group of matrices with a sharp break in the distribution of their singular values. They are defined by $\sigma_i = 1$, i = 1, ... n-1 and $\sigma_n = (\kappa_2)^{-1}$. The test matrices are again constructed via pre- and post multiplication with pseudo-random orthogonal matrices.

Test results with $\sigma_i = 1$, i = 1,...,n - 1; $\sigma_n = \kappa_2^{-1}$; GJE_NRM_INV1.

κ_2	n = 10	25	50	100
10	.56/.81	.43/.73	.59/.77	.69/.83
10^3	.50/.82	.36/.71	.53/.76	.71/.83
10^6	.47/.80	.46/.71	.51/.77	.68/.81
10^9	.53/.80	.41/.72	.57/.77	.70/.82

Minimum and mean over 'estimate' / $\|A^{-1}\|_2$ for groups of 100 matrices.
Table 3.

We also experimented with a different choice for the starting vector f_1. The iteration as defined in section 2 was started with $f_1 = e_1$; the matching vector t_1 is given by (2.3.). This vector is not in the direction of the appropriate left

singular vector of $B^{(1)}$ so that $\|B^{(1)}\|_2$ is underestimated already. It is easy to verify that this left singular vector is a linear combination of e_1 and g_1.

The iteration may be started with the correct right and left singular vectors of $B^{(1)}$, f_1 and t_1 respectively, so that the estimate for $\|B^{(1)}\|_2$ matches its correct value. The implementation of this idea showed a change in the calculated estimate of at most two units in the second digit, while requiring the explicit calculation of f_k in the k-th step, which requires an extra n^2 multiplications and additions.

In GJE_NRM_INV2 we implemented the extended version of our algorithm where the extra inverse iteration step is finally applied. It uses $5n^2 + O(n)$ extra multiplications and $4n^2 + O(n)$ additions.

This routine never produced estimates that were wrong by a factor of more than 3. The estimates are much sharper than the estimates of GJE_NRM_INV1 as can also be judged from the following tables.

Test results with $\sigma_i = \alpha^{i-1}$, i = 1,...,n; GJE_NRM_INV2.

κ_2 n =	10	25	50	100
10	.60/.92	.75/.92	.85/.94	.92/.96
10^3	.49/.96	.66/.93	.75/.95	.91/.97
10^6	.34/.97	.84/.97	.74/.97	.80/.97
10^9	.98/1.0	.85/.98	.91/.98	.89/.98

Minimum and mean over 'estimate' / $\|A^{-1}\|_2$ for groups of 100 matrices.

Table 4.

Test results with $\sigma_i = 1$, i = 1,...,n - 1; $\sigma_n = \kappa_2^{-1}$; GJE_NRM_INV2.

κ_2 n =	10	25	50	100
10	.99/1.0	.98/1.0	1.0/1.0	.99/1.0
10^3	1.0/1.0	1.0/1.0	1.0/1.0	1.0/1.0
10^6	1.0/1.0	1.0/1.0	1.0/1.0	1.0/1.0
10^9	1.0/1.0	1.0/1.0	1.0/1.0	1.0/1.0

Minimum and mean over 'estimate' / $\|A^{-1}\|_2$ for groups of 100 matrices.

Table 5.

4. CONCLUSION

The algorithms we present to calculate an estimate for the spectral norm of the inverse matrix produce reliable estimates. In extensive tests on matrices of order up to 100, our favourite estimator, implemented as GJE_NRM_INV2, was never wrong by more than a factor of 3 and almost always correct within a factor of 2. The somewhat faster routine GJE_NRM_INV1 was never wrong by a factor of more than 10, which, in the context of estimating the condition-number of a matrix, is mostly good enough.
Comparing our results with the test results by Higham shows that the behaviour of our estimator is good; it is comparable with the l_2-norm estimator SIGMAN [9,18] and the well known l_∞ -norm estimator which is implemented in LINPACK [6]. Our estimator, however, fits the Gauss-Jordan algorithm while the others fit Gaussian elimination or the QR decomposition.

REFERENCES

[1] A.K. CLINE, A.R. CONN, C.F. VAN LOAN; Generalizing the LINPACK condition estimator; pp. 73-83 in: J.P. HENNART, ed., Numerical Analysis, Mexico 1981, Lecture Notes in Mathematics 909, Springer-Verlag, Berlin.

[2] A.K. CLINE, C.B. MOLER, G.W. STEWART and J.H. WILKINSON; An estimate for the condition number of a matrix; SIAM J. Numer. Anal. 16(1979) pp.368-375.

[3] A.K. CLINE, R.K. REW; A set of counter-examples to three condition number estimators, SIAM J. Sci. Stat. Comput. 4(1983) pp. 602-611.

[4] T.J. DEKKER and W. HOFFMANN; Rehabilitation of the Gauss-Jordan algorithm; Report 86-28, Department of Mathematics, University of Amsterdam 1986.

[5] T.J. DEKKER and W. HOFFMANN; Numerical improvement of the Gauss-Jordan algorithm; in: Proceedings ICIAM 87, Paris-La Villette, june 29-july 3 1987; Contributions from the Netherlands; ed.: A.H.P. van der Burgh, R.M.M. Mattheij; Mathematisch Centrum, Amsterdam 1987

[6] J.J. DONGARRA, J.R. BUNCH, C.B. MOLER and G.W. STEWART; LINPACK User's Guide; SIAM, Philadelphia 1979.

[7] G.H. GOLUB and C.F. VAN LOAN; Matrix Computations; North Oxford Academic, Oxford 1983.

[8] W.W. HAGER; Condition estimators; SIAM J. Sci. Stat. Comput. 5(1984),.pp. 311-316.

[9] N.J. HIGHAM; A survey of condition number estimation for triangular matrices; SIAM Rev. 29 (1987) pp. 575-596 .

[10] W. HOFFMANN; Chapter simultaneous equations, Report NM-R8712 in: NUMVEC FORTRAN library manual update #1; Centre for Mathematics and Computer Science, Amsterdam 1987.

[11] A.S. HOUSEHOLDER; The theory of matrices in numerical analysis; Blaisdell Publ. Cy.; New York, Toronto, London 1964.

[12] T. KIMURA; Gauss-Jordan elimination by VLSI mesh-connected processors; in: Jesshope and Hockney (eds.) Infotech State of the Art Report: Supercomputers, Vol. 2, pp271-290, (1979) Infotech, Maidenhead, England

[13] C. LANCZOS; Linear differential operators; Van Nostrand Comp. Ltd.; London etc. 1961.

[14] D.P. O'LEARY; Estimating matrix condition numbers, SIAM J. Sci. Statist. Comput. 1(1980)pp.205-209.

[15] D. PARKINSON; Experience in exploiting large scale parallelism; in: Kowalik (ed.) Proceedings of the NATO workshop on high speed computations; West Germany NATO ASI Series, Vol. F-7, (1984) Springer Verlag, Berlin.

[16] G. PETERS and J.H. WILKINSON; On the stability of Gauss-Jordan elimination with pivoting; Comm. ACM 18(1975), pp. 20-24.

[17] G.W. STEWART; The efficient generation of random orthogonal matrices with an application to condition estimators, SIAM J. Numer. Anal. 17(1980) pp.403-409.

[18] C.F. VAN LOAN; On estimating the condition of eigenvalues and eigenvectors, Linear Algebra Appl. 88/89(1987) pp.715-732.

[19] D. ZOIS; Parfes, a parallel finite element system; Supercomputer 17(1987)pp. 34-43.

Nessun dorma!

Nessun dorma! Nessun dorma!
Tu pure, o Principessa,
nella tua fredda stanza
guardi le stelle che tremano
d' amore e di speranza!
Ma il mio mistero è chiuso in me,
il nome mio nessun saprà!
No, no sulla tua bocca lo dirò,
quando la luce splenderà!
Ed il mio bacio scioglierà
il silenzio che ti fa mia!

G. Adami / R. Simoni
G. Puccini
Turandot

CHAPTER IV

A fast variant of the Gauss-Jordan
Algorithm with partial pivoting

report on recent research dating from
december 1988 and january 1989

A FAST VARIANT OF THE GAUSS-JORDAN ALGORITHM WITH PARTIAL PIVOTING

by

W. Hoffmann

Summary. In this paper we deal with a variant of the Gauss-Jordan algorithm, introduced by Huard. This algorithm uses the same amount of floating-point operations as Gaussian elimination. We show that it can be made of practical interest by including the relatively unknown strategy of partial pivoting by column interchanges. This pivoting strategy, when combined with standard Gauss-Jordan, yields a fully satisfactory algorithm. The algorithm discussed in this paper was compared with Gaussian elimination on an Alliant FX/4 parallel vector. Using BLAS level 2 routines, it turns out to be very efficient.

Subject classification: AMS: 65F05, 15A06 .

1. INTRODUCTION

We consider an algorithm, originally presented by Huard [8], that has been described by Cosnard, Robert and Trystram [1] in a paper on performance analysis of parallel algorithms; a pivoting strategy was not included.

In section 2 we describe the algorithm and give the operation count. The transportability of the algorithm is served by indicating parts that can be implemented by applicable calls to BLAS routines [4]. These calls will take care of all operations apart from those regarding pivoting and row scaling.

From inspecting the original algorithm, it is evident that the usual strategy of partial pivoting by row interchanges is impossible. In section 3 we give an analysis which shows that an intermediate result in the upper triangular part of the matrix is equal to the pivotal row in the upper triangle as produced with Gaussian elimination. This observation shows that an alternative pivoting strategy by column interchanges can be applied. This pivoting strategy was introduced in [7] and in combination with the Gauss-Jordan algorithm it turned out to be fully satisfactory.

An implementation using BLAS level 2 routines for an Alliant FX/4 machine was compared with Gaussian elimination using BLAS level 2. The results were encouraging and are reported in section 4.

2. DESCRIPTION OF THE ALGORITHM

The Huard algorithm is presented using the scheme in Fig. 1. We suppose that the right-hand side vector of the linear system is added to the matrix as an extra column.
The coefficient matrix is transformed to the identity matrix and similarly the right-hand side vector is transformed to the solution vector. This is performed in n stages as follows:

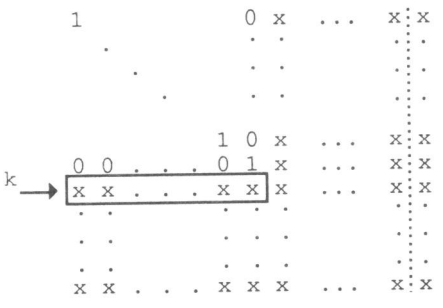


Fig.1

Assume that at the beginning of stage k, the leading (k-1)×(k-1) submatrix equals the identity matrix as indicated.
In step k, the first k-1 elements in row k (in the box) are eliminated, using rows 1 through k-1.
The total number of multiplications and additions equals {n-(k-1)}×(k-1) for updating the k-th row of the matrix and (k-1) for the right-hand side vector (RHS).
Next, the k-th row is divided by its diagonal element to enter the value "one" on the diagonal. This takes one division and (n-k) multiplications for the matrix and one multiplication for the RHS. Finally, the elements in the k-th column above the diagonal are eliminated with the k-th diagonal element, as is done in the Gauss-Jordan algorithm. This takes (n-k)×(k-1) multiplications and additions for updating the submatrix in the upper right-hand corner and (k-1) for the RHS.

At the end of step k, the leading (k×k) submatrix equals the identity and the n×n transformed matrix combined with the updated right-hand side vector still defines an equivalent linear system, having the same solution.
Hence, after n steps, the solution is found in the last column.

Operation count

The total number of floating-point operations for the matrix equals:

multiplications: $\sum_{k=1}^{n} \{(n-k)(2k-1) + k-1\} = \frac{1}{3} n^3 - \frac{1}{3} n$;

additions: $\sum_{k=1}^{n} (2n-2k+1)(k-1) = \frac{1}{3} n^3 - \frac{1}{2} n^2 + \frac{1}{6} n$;

divisions: n .

For the right-hand side vector we need a total of n^2 multiplications and $n^2 - n$ additions.

Algorithmic denotation

In a formal description of the algorithm we use the following notation: row k of the matrix is denoted by $a_{k\bullet}$ and matrix element (i,j) by α_{ij}.

```
 1 For k = 1 to n do
 2  For i = 1 to k-1 do
 3       a_k. := a_k. - α_ki × a_i.
 4  enddo
 5  perform pivoting as described in section 3
 6  a_k. := (1/α_kk) × a_k.
 7  For i = 1 to k-1 do
 8       a_i. := a_i. - α_ik × a_k.
 9  enddo
10 enddo
```

Fig.2

This algorithm can also be expressed by premultiplications with elementary matrices. Lines 2 through 4 correspond with a row update. This can be seen as a premultiplication of the matrix under consideration by $(I - e_k g_k^T)$; the k-1 non-zero elements of g_k^T are copied from the elements in row k as indicated by "the box" in Figure 1.

Line 6 describes a row scaling which can be expressed by premultiplication with a suitable diagonal matrix Δ_k. Lines 7 through 9 correspond with a rank-one update of the upper right-hand corner submatrix. This can be described by a premultiplication of the matrix under consideration by $(I - u_k e_k^T)$; the elements of u_k are copied from the elements in the k-th column above the diagonal.

The changes as a result of the premultiplications with the above elementary matrices can be indicated as changes on submatrices as a whole. This leads us to a description of the algorithm at matrix level. The notation that we use, should be clear from the context. We only denote the non trivial changes in the matrix; the data needed for the elimination process is saved, as usual, in the corresponding locations of the matrix elements.

```
For k = 1 to n do
  A[k,k:n+1]        := A[k,k:n+1] - A[k,1:k-1] × A[1:k-1,k:n+1]
                                                  {rowvector × matrix}
  A[k,k+1:n+1]      := A[k,k+1:n+1] × (1/α_kk)    {rowscaling}
  perform pivoting as described in section 3
  A[1:k-1,k+1:n+1]  := A[1:k-1,k+1:n+1] - A[1:k-1,k] × A[k,k+1:n+1]
                                                  {rank-one update}
enddo
```

Fig.3

3. PIVOTING STRATEGY

Without a suitable pivoting strategy, the above algorithm is hardly interesting for practical purposes.

For an explanation of our proposed strategy, we focus on the situation that arises when the first "half" of the p-th elimination step has been executed. We suppose that the algorithm in either Fig.2 or Fig.3 is in progress with k having the value p such that the update of the p-th row has been completed and the statement that describes the row scaling has not been executed yet.

The situation that now exists can be described by suitable premultiplications with elementary matrices as follows:

$$A^{(p')} = (I - e_p g_p^T)(I - u_{p-1} e_{p-1}^T)\Delta_{p-1}(I - e_{p-1} g_{p-1}^T)\ldots \\ \ldots (I - u_2 e_2^T)\Delta_2 (I - e_2 g_2^T)\Delta_1 A^{(0)}; \qquad (3.1)$$

the use of p' as a superscript stems from the fact that the p-th stage has not been completed yet.

Let $\begin{pmatrix} B & C \\ D & E \end{pmatrix}$ denote a partition of $A^{(0)}$ where the leading submatrix B has order (p-1) and the other submatrices have sizes that match the partition. Assume that the product of the elementary matrices has been accumulated in a single matrix, then relation (3.1) for the partitioned matrices is given by:

$$\begin{pmatrix} I_{p-1} & F \\ G & H \end{pmatrix} = \begin{pmatrix} X & 0 \\ Y & I_{n-p+1} \end{pmatrix} \times \begin{pmatrix} B & C \\ D & E \end{pmatrix}. \qquad (3.2)$$

Here I_{p-1} denotes the $(p-1)^{st}$ order identity matrix that has been formed in the indicated position in matrix $A^{(p')}$ and I_{n-p+1} denotes the $(n-p+1)^{st}$ order identity matrix that is in the indicated position in the accumulated product.
From the fact that the premultiplication with $(I - e_p g_p^T)$ has been performed, we know that the first row of G is zero; the rest of matrix G still equals the corresponding part of D. We also know that Y, apart from its first row, which equals $-g_p^T$, is a zero matrix of a size that matches the partition.
By comparing equivalent rows at both sides of (3.2), we have:

$$\underline{0}^T = e_1^T G = e_1^T (YB + D) = - g_p^T B + e_1^T D,$$

from which we see that g_p is determined by:

$$g_p^T \ (= - e_1^T Y) = e_1^T D B^{-1} \qquad (3.3)$$

If we substitute this result in the equation for the first row of H we find:

$$e_1^T H = e_1^T (E - D B^{-1} C). \qquad (3.4)$$

The expression between brackets is the Schur complement [2], which would have been created in the lower right-hand submatrix if Gaussian elimination were performed on $A^{(0)}$ [6, Problem 4.2-3]. The first row of that submatrix would remain unchanged in Gaussian elimination and would have been a row of the resulting upper triangular matrix U.

Consequently, the first row of H can be searched for an element that is maximal in size, so that by interchanging suitable columns in $A^{(p')}$ this element can be put in the leading diagonal position of H. This pivoting strategy is presented in [7] and its combination with standard Gauss-Jordan has shown to give such an improvement in that algorithm that it has become practically as stable as Gaussian elimination [3].

With the application of column interchanges, the algorithm under consideration will perform the same calculations in the upper triangular part of the matrix as the standard Gauss Jordan algorithm with column interchanges.

We conclude this section with describing the statements that perform the applicable partial pivoting; they must be included in the algorithm in Figures 2 and 3 at the indicated positions:

```
Determine q such that k ≤ q ≤ n and |α_kq| = max_{k≤j≤n} |α_kj|
if q > k then interchange columns q and k
```

Note that the calculated solution vector must undergo the same interchanges in reverse order.

4. NUMERICAL EXPERIMENTS

We implemented the algorithm with the pivoting strategy as described above on the Alliant FX/4 computer of the CWI at Amsterdam.
This machine has four parallel vector processors. Its precision with FORTRAN 77 is $\approx 10^{-7}$ for single length computations and $\approx 10^{-15}$ for double length computations.

The results of HGJPC (Huard-Gauss-Jordan-Partial pivoting by Columns) with respect to accuracy and processing time were compared with our Gaussian elimination variant CCRPCS [7], both in single and double precision. We also made a comparison of the double precision result with the results from LINPACK routines DGEFA and DGESL [5]. We should mention that the single precision routines SGEFA and SGESL take much more processing time on the Alliant; obviously these routines have not been optimized for this machine.

For all matrices tested, the accuracy of CCRPCS and HGJPC were similarly good and in their double length version also similar to the LINPACK results.

An overview of our timing results is presented in the next tables

n =	25	50	100	200	400
HGJPC-F77	0.008	0.025	0.111	0.737	5.462
HGJPC-F77+BLAS2	0.015	0.032	0.098	0.505	3.641
CCRPCS-F77	0.005	0.018	0.094	0.676	6.704
CCRPCS-F77+BLAS2	0.007	0.019	0.078	0.523	5.539

Table 1. Processing time in seconds - Single precision

n =	25	50	100	200	400
HGJPC-F77	0.008	0.025	0.129	0.820	7.719
HGJPC-F77+BLAS2	0.013	0.030	0.120	0.620	6.771
CCRPCS-F77	0.006	0.018	0.096	0.977	9.830
CCRPCS-F77+BLAS2	0.007	0.021	0.090	0.725	9.164
DGEFA-LINPACK	0.006	0.021	0.126	0.415	2.655

Table 2. Processing time in seconds - Double precision

Discussion

The FORTRAN 77 code of HGJPC follows closely the algorithmic structure as given in Fig. 2 section 2. As a consequence, the elements of the matrix are accessed by rows. This is far from optimal, as the elements are stored columnwise.
By comparing the results of HGJPC and CCRPCS, we observe that a further research on this algorithm may yield efficient codes for various architectures.
In this respect we think of operating on the transposed of the matrix, so that column operations are replacing row operations.
Another improvement will come from designing a block-variant of the algorithm in such a way that less transport to cache memory is necessary.

If BLAS level 2 routines are included, following the structure from Fig. 3 of section 2, then it can be observed that for the larger matrices routine HGJPC is (much) more efficient than CCRPCS.

The LINPACK double length routine has been optimized to yield a good performance on the Alliant. For this purpose, routine DGEFA is designed to exploit a block structure of the matrix, which results in less transport between cache and main memory. The timings of DGEFA are correspondingly good.

ACKNOWLEDGEMENTS

The author wants to thank the Centre for Mathematics and Computer Science, CWI, Amsterdam, for giving him access to their Alliant FX/4 computer; he also wants to thank Miss Kitty Potma for performing the experiments.

LITERATURE

1. M. Cosnard, Y. Robert, D. Trystram, Résolution parallèlle de systèmes linéaires denses par diagonalisation; Bulletin E.D.F, série C, 2(1986)67-88
2. R.W. Cottle, Manifestations of the Schur Complement; Linear Algebra Appl. 8(1974)189-211
3. T.J. Dekker and W. Hoffmann, Rehabilitation of the Gauss-Jordan algorithm; to appear in Numer. Mathematik 1989.
4. J.J. Dongarra, J. du Croz, S. Hammarling and R.J. Hanson, An extended set of FORTRAN Basic Linear Algebra Subprograms; ACM Trans. Math. Softw. 14, 1-17 (1988)
5. J.J. Dongarra, C.B. Moler, J.R. Bunch and G.W. Stewart, LINPACK User's guide; Philadelphia: SIAM 1979
6. G.H. Golub and C.F. Van Loan, Matrix Computations; North Oxford Academic, Oxford 1983.
7. W. Hoffmann, Solving linear systems on a vector computer; Journal of Computational and Applied Mathematics 18(1987)353-367
8. P. Huard, La méthode du Simplexe sans inverse explicite; bulletin E.D.F. Série C n.2, (1979)

CHAPTER V

Iterative Algorithms for
Gram-Schmidt Orthogonalization

accepted for publication in:
Computing, 1989

ITERATIVE ALGORITHMS FOR GRAM-SCHMIDT ORTHOGONALIZATION

by

W. Hoffmann, Amsterdam

Abstract - Zusammenfassung

The algorithms that are treated in this paper are based on the classical and the modified Gram-Schmidt algorithms. It is shown that Gram-Schmidt orthogonalization for constructing a QR factorization should be carried out iteratively to obtain a matrix Q that is orthogonal in almost full working precision. In the formulation of the algorithms, the parts that express manipulations with matrices or vectors are clearly identified to enable an optimal implementation of the algorithms on parallel and/or vector machines. An extensive error analysis is presented. It shows, for instance, that the iterative classical algorithm is not inferior to the iterative modified algorithm when full precision of Q is required. Experiments are reported to support the outcomes of the analysis.

Keywords: Gram-Schmidt orthogonalization, QR factorization, vector algorithms.
AMS subject classification: 65F25, 65G05, 15A23.

In diesem Artikel werden verschiedene Varianten der klassischen und der modifizierten Gram-Schmidt Methode präsentiert. Wir zeigen, dass man für die Konstruktion der QR-Zerlegung die Gram-Schmidt Orthogonalisation iterativ anwenden muss, falls man die Matrix Q ungefähr bis auf Maschinengenauigkeit orthogonal haben will. Die Algorithmen sind so formuliert, dass man alle Operationen mit Matrizen oder Vektoren deutlich identifizieren kann und eine Implementierung auf einem Parallel- oder Vektorcomputer keine Schwierigkeiten bietet. Eine ausführliche Fehleranalyse wird gegeben. Daraus folgt zum Beispiel, dass der iterative klassische Algorithmus nicht schlechter ist als der iterative modifizierte Algorithmus, wenn die Matrix Q so genau wie möglich orthogonal sein muss. Verschiedene Experimente auf einem Vektorcomputer werden beschrieben, welche die Resultate der Fehleranalyse bestätigen.

1. INTRODUCTION

We consider variants of Gram-Schmidt orthogonalization and their suitability for use on super computers. Algorithms for super computers must exploit the parallel and/or vector facilities of the machine to admit for an optimal performance. The numerical stability of the algorithm, however, may require that a formulation which seems to be particularly favourable for use on super computers should be avoided. Some well known variants of the Gram-Schmidt algorithm are good examples of this type of conflict.
The goal of Gram-Schmidt orthogonalization is to construct a QR factorization. This factorization is defined as follows.
Consider an $m \times n$ matrix $A = [a_1,\ldots,a_n]$ with $a_j \in \mathbf{R}^m$ and $m \geq n$. Let $k(j) = \dim(\text{span}(a_1,\ldots,a_j))$ for $j = 1,\ldots,n$ and let $p = k(n)$ {= rank(A)}.
An orthogonal basis $[q_1,\ldots,q_p]$ for $\text{span}(a_1,\ldots,a_n)$ is to be constructed such that $a_i \in \text{span}(q_1,\ldots,q_{k(i)})$, $i = 1,\ldots, n$. In terms of matrix calculation this is equivalent with: construct an orthogonal $m \times p$ matrix Q such that $A = QR$ for a $p \times n$ upper trapezoidal matrix R. If $p = n$, then the problem is called a full-rank problem .
If matrix Q is not used, or is only needed to calculate the product Qv for several vectors v, then the Householder algorithm is to be preferred. If the individual column vectors of matrix Q are wanted (the so called "orthogonal basis" problem), then, in case of a full-rank matrix A, the Gram-Schmidt algorithm is advantageous.
This paper deals with Gram-Schmidt orthogonalization for the case that the matrix has (numerically) full rank, i.e. $p = n$.
For the case $p < n$, the Gram-Schmidt algorithm has been extended with the application of column pivoting; see Businger and Golub [2]. This addition gives quite satisfactory results in most practical cases, but may not detect the right degree of rank deficiency. An adaption of the Gram-Schmidt algorithm which is presented by Chan [3], yields correct results in the general situation with $p \leq n$ and calculates the correct rank of the matrix.

It has become well known that various so called "block QR" algorithms admit efficient performance on super computers. Some of these algorithms, however, appear to be variants of the classical Gram-Schmidt algorithm and it has been acknowledged that the classical algorithm may produce a matrix Q that is far from orthogonal.
The method known as "the modified Gram-Schmidt algorithm" is numerically to be preferred over the classical algorithm; the orthogonality of Q is of the order of machine precision times condition number of the matrix. This may be insufficient for matrices that are ill conditioned. To overcome this shortcoming, the modified algorithm can be applied iteratively, so that almost full machine precision is reached.
We show that the classical algorithm in an iterative fashion can attain that same accuracy in an equal number of iterations. Consequently, constructions that were banned for the sake of accuracy can be accepted in an iterative algorithm.
In section 2 we give definitions of the classical and the modified Gram-Schmidt algorithm through an algorithmic formulation. In section 3 we

present the iterative versions of these algorithms. Iterative versions of the Gram-Schmidt algorithm are also presented by Daniel, Gragg, Kaufman and Stewart [4] and by Ruhe[7]. In section 4 we present the results of numerical experiments which are discussed in section 5. In section 6 we draw our conclusions.

2. ONE-STEP GRAM-SCHMIDT ALGORITHMS.

For ease of formulation we use the normalizing operator N defined by:
$N(x) = x / \|x\|$, for vectors $x \neq \underline{0}$.
For a full-rank rectangular $m \times n$ matrix A, $m \geq n$, the orthogonal $m \times n$ matrix Q whose columns form an orthogonal basis for the subspace $\text{span}(a_1,\ldots,a_n)$ can be defined with the use of projections as follows:

```
1.   Q := [0,..., 0]                          {the m × n zero-matrix}
2.   For j = 1,..., n do
     1.  Q := Q + N ((I - QQ^T) a_j) e_j^T
```

If the elements of the triangular matrix R ($=Q^TA$) are wanted too, then the description turns into the Classical Gram-Schmidt algorithm, CGS.
Algorithm CGS is given by :

```
1.   Q := [0,..., 0]                          {the m × n zero-matrix}
2.   For j = 1,..., n do
     1.  r_j  := Q^T a_j
     2.  t    := a_j  -  Q r_j               {t = (I - QQ^T) a_j}
     3.  r_jj := || t ||_2
     4.  q_j  := t / r_jj                    {q_j = N (t)}
```

The numerical behaviour of this algorithm is very poor in a sense that in many cases the constructed matrix Q is far from orthogonal. This well known result has been shown by Björck [1].
An improved algorithm is the Modified Gram-Schmidt algorithm .
This algorithm exists in two versions; MGSC and MGSR, constructing the matrix R column by column or row by row, respectively. The difference shows only in the way the data is accessed.
Algorithm MGSC has the same structure as algorithm CGS; the difference is that individual elements of vector Q^Ta_j, which is calculated in line 2.1 of CGS, must be calculated sequentially by taking innerproducts with successive columns q_i of Q so that the appropriate multiple of q_i can be subtracted from a_j as soon as its coefficient is available. This repeated modification of column a_j is the crux of the algorithm.

Algorithm MGSC is given by:

```
For j = 1,..., n do
1.    t   :=  a_j
2.    For i = 1,..., j-1 do
      1.  r_ij  :=  q_i^T t
      2.  t   :=  t  -  q_i r_ij
3.    r_jj  :=  || t ||_2
4.    q_j   :=  t / r_jj                          { q_j = N (t) }
```

The update-rule as given in lines 2.1 and 2.2. of MGSC can as well be applied for each q_i on all columns a_k with $k \geq i$. In that case the i-th row of R is computed as a whole; its elements can be calculated in parallel. This gives rise to algorithm MGSR.
For the description we use the following notation. With $r_{i\bullet}$ we denote the i-th row of R and with $\underline{r}_{i\bullet}$ we denote the part of $r_{i\bullet}$ that is strictly to the right of the diagonal: $(r_{ii+1},\ldots,r_{in})$.

Algorithm MGSR is described by:

```
For i = 1,..., n do
1.    r_ii  :=  || a_i ||_2
2.    q_i   :=  a_i / r_ii
3.    r_i•  :=  q_i^T [a_i+1,..., a_n]
4.    [a_i+1,..., a_n] := [a_i+1,..., a_n] - q_i r_i• {rank-one matrix update}
```

Although the results of both MGS algorithms are an improvement over the results obtained by CGS in the sense that the orthogonality of matrix $\underline{Q}$ is much better, in many cases the orthogonality is still not good enough. This is reflected in the bounds for $\|\underline{Q}^T\underline{Q} - I\|_2$ which is of the order of the product of the machineprecision ε and the condition number of the original matrix, as has been shown by Björck [1].

3. Iterative Gram-Schmidt algorithms

Iterative Gram-Schmidt algorithms with improved orthogonality have been presented and analysed by Daniel et al. [4] and Ruhe [7].
We here describe the iterative versions of both the classical and the modified Gram-Schmidt algorithms; the modified algorithm only in the MGSC form. A corresponding iterative version of the modified algorithm in its MGSR form is not possible.

Algorithm CGSI reads :

```
Q  :=  [0,…, 0]                                   {the m × n zero-matrix}
For j = 1,…, n do
1.    rj  :=  0
2.    t   :=  aj
3.    Repeat
      1.  p   :=  t
      2.  s   := Q^T p
      3.  v   := Q s
      4.  t   := p - v
      5.  rj  := rj + s
4.    Until  < t perpendicular span(q1,…, qj-1) >
5.    rjj := ‖ t ‖2
6.    qj  :=  t / rjj                             { qj = N (t) }
```

Algorithm MGSCI reads :

```
Q  :=  [0,…, 0]                                   {the m × n zero-matrix}
For j = 1,…, n do
1.    rj  :=  0
2.    t   :=  aj
3.    Repeat
      1.  p   := t
      2.  For i = 1,…, j-1 do
          1.  si  :=  qi^T t
          2.  t   := t  -  qi si
      3.  rj  := rj + (s1,…, sj-1,0,…, 0)^T
4.    Until < t perpendicular to span(q1,…, qj-1) >
5.    rjj := ‖ t ‖2
6.    qj  :=  t / rjj
```

It has been demonstrated by Ruhe [7] that the resulting r_j in the j-th step corresponds with the solution of the equation $Q^TQr_j = Q^Ta_j$ with $Q = [q_1,\ldots,q_{j-1}]$. The CGSI variant corresponds with Gauss-Jacobi iteration for solving that equation and the MGSCI variant with Gauss-Seidel iteration. The resulting accuracy depends on the number of iteration-steps performed.
We would like to emphasize that in Ruhe's analysis the (almost) orthogonality of matrix Q is not used; the goal in the j-th step is to find r_j such that $(a_j - Qr_j)$ is orthogonal to $\text{span}(q_1,\ldots,q_{j-1})$.
The new column q_j is obtained from $q_j := N(a_j - Qr_j)$. An implementation of the stopping criterion "t perpendicular to $\text{span}(q_1,\ldots,q_{j-1})$" was not suggested by Ruhe.
For a useful stopping criterion we are inspired by Parlett [6], who analyses Gram-Schmidt orthogonalization for two vectors. He presents an "iterative" orthogonalization algorithm which he attributes to W. Kahan; iterative has been put between quotes because a single reorthogonalization step is sufficient in practice. It provides us with an efficient stopping criterion for algorithm CGSI and it shows to be adequate for MGSCI too.

In the j-th major step of CGSI, vectors s = fl(Q^Tp) and v = fl(Qs) are calculated (lines 3.2 and 3.3). A backward error analysis, using one of the customary matrix norms, shows:

s = (Q + $\delta_1 Q$)^{T}p with $\|\delta_1 Q\| \le \phi_1(m,j)\ \|Q\|\varepsilon$ and

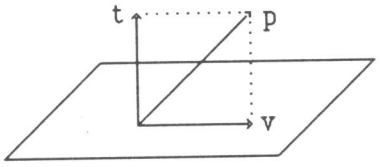
v = (Q + $\delta_2 Q$)s with $\|\delta_2 Q\| \le \phi_2(m,j)\ \|Q\|\ \varepsilon$

where ϕ_1 and ϕ_2 stand for low degree polynomials in m and j and ε stands for the effective machine precision (effective means that effects of arithmetic are taken into account).

If v were calculated without error (i.e. $\delta_2 Q$ = 0), it would be contained in the column space of Q, regardless the accuracy of s. If s were also calculated without error and Q were exactly orthogonal, then v would be equal to p's projection onto Q's column space.

The vector t = fl(p - v) (line 3.4) is calculated as an approximation to p's component orthogonal to span(q_1, ... ,q_{j-1}) . For t we have:

t = p + δp - (v + dv) with $\|\delta p\| \le m\ \|p\|\ \varepsilon$ and $\|\delta v\| \le m\ \|v\|\ \varepsilon$.



span(q_1, ... ,q_{j-1})

In the following theorem we show that t is close to a vector orthogonal to span(q_1, ... ,q_{j-1}).

Theorem 1.

Let vectors p and q_1,...,q_{j-1} be given in $\mathbf{R}^m$ with p $\notin$ span(q_1,...,q_{j-1}).

Suppose Q = [q_1,...,q_{j-1}] and a $\in$ (0,1) are such that $\|Q^TQ - I\|_2 \le \alpha$.

Let scalar μ (> 0), vectors δp and δv and matrices $\delta_1 Q$ and $\delta_2 Q$ be such that

$\|\delta_1 Q\|_2 \le \mu\|Q\|_2$, $\|\delta_2 Q\|_2 \le \mu\|Q\|_2$, $\|\delta p\|_2 \le \frac{1}{2}\mu\|p\|_2$ and $\|\delta v\|_2 \le \frac{1}{2}\mu\|v\|_2$.

Let s = (Q + $\delta_1 Q$)^{T}p, v = (Q + $\delta_2 Q$)s , t = p + δp - (v + δv) and let u denote the orthogonal projection of p onto the space perpendicular to span(q_1,...,q_{j-1}).

Then the difference between t and u (i.e. the error in t), is bounded as follows:

$$\|u - t\|_2 \le \left\{ \alpha\frac{1+\alpha}{1-\alpha} + \frac{1}{2}\mu\{\{(2+\mu)^2 + 1\}(1+\alpha) + 1\}\right\} \|p\|_2 .$$

For instance, for a and m $\le$ 0.1, this implies $\|u - t\|_2 \le (1.3\,\alpha + 3.5\,\mu)\,\|p\|_2$.

Proof of Theorem 1.

For the proof of this theorem we need the following results:

Lemma 1. If $\|Q^TQ - I\|_2 \le \alpha$ then $\|Q\|_2^2 \le 1 + \alpha$.

Proof: This result is a direct consequence of the triangle inequality.

Lemma 2. If $\|Q^TQ - I\|_2 \leq \alpha < 1$ then $\| I - (Q^TQ)^{-1}\|_2 \leq \frac{\alpha}{(1 - \alpha)}$.

Proof: From the identity

$I - (Q^TQ)^{-1} = \{\{(Q^TQ)^{-1} - I\} + I\} (Q^TQ - I)$,

it follows

$\|I - (Q^TQ)^{-1}\|_2 \leq \|I - (Q^TQ)^{-1}\|_2 \|Q^TQ - I\|_2 + \|Q^TQ - I\|_2$.

From this, the result follows immediately .

For u we have, using the appropriate projection operator

$u = (I - Q (Q^TQ)^{-1}Q^T) p$,

so that for the difference u - t we have

$u - t = (Q + \delta_2Q)(Q + \delta_1Q)^T p - Q (Q^TQ)^{-1}Q^T p + (\delta v - \delta p)$.

This yields

$u - t = \{Q \{I - (Q^TQ)^{-1}\}Q^T + Q\delta_1Q^T + \delta_2QQ^T + \delta_2Q\delta_1Q^T \} p + (\delta v - \delta p)$.

A straightforward calculation yields

$$\| u - t \|_2 \leq \{\{ \frac{\alpha}{(1 - \alpha)} + 2\mu + \mu^2 \}(1 + \alpha) + \tfrac{1}{2} \mu(1 + \frac{\|v\|_2}{\|p\|_2}) \} \|p\|_2 .$$

From the definition of v we find:

$$\frac{\|v\|_2}{\|p\|_2} \leq (1 + \mu)^2 (1 + \alpha), \qquad (3.1)$$

from which the rest of the proof follows immediately.

□

Consider for certain $j \leq m$ an orthonormal $m \times (j-1)$ matrix $Q = [q_1,\ldots,q_{j-1}]$ and an m-vector p not in the column space of Q. Suppose that for this vector p a single step in algorithm CGSI is carried out to construct an orthonormal basis for span$(q_1,\ldots,q_{j-1},p)$, then the conditions of the theorem are fulfilled with a small value for a and (possibly) a very small value for m, depending on the sizes of m and j and the effective machine precision. Although the calculated vector t is close to a vector that is perpendicular to the column space of Q, it is not true that consequently t itself is almost orthogonal to that column space. For example, the orthogonality may be (very) bad if t is of the order of the error and is therefore small compared to p.

In the Parlett-Kahan algorithm a reorthogonalization of t against v is prescribed in cases that $\|t\|_2$ is smaller than $\|p\|_2$ divided by a selected accuracy factor κ. This factor must be chosen larger than a constant $\beta > 1$. (They use the value $\beta = (0.83 - \varepsilon)^{-1}$.)

In our situation we conclude that if t is suspected of being not orthogonal to v, it can certainly be suspected of being not orthogonal to columns of Q.

So the decision that a reorthogonalization is required can be made on the same grounds.

The reverse, being the acceptance of t if it is large enough is not evident, but will follow from theorem 2.

The situation that $t = \underline{0}$ does not occur if the given matrix A has (numerically) full rank.

If t is large enough relative to p, then the orthogonality of t with respect to the column space of Q can be estimated as expressed in the following theorem.

Theorem 2.
Let the conditions of theorem 1 be fulfilled and let moreover k be such that $\|t\|_2 \geq \|p\|_2 / \kappa$.
The orthogonality between t and the column space of Q satisfies:

$$\frac{\|Q^T t\|_2}{\|t\|_2} \leq (1 + \tfrac{1}{2}\alpha)\{\tfrac{3}{2}\mu + (1+\mu)[\alpha + \mu(1+\alpha) + \tfrac{1}{2}\mu(1+\mu)(1+\alpha)]\}\kappa .$$

For instance: for α and $\mu < 0.1$ this implies $\frac{\|Q^T t\|_2}{\|t\|_2} \leq (1.2\,\alpha + 3.6\,\mu)\,\kappa$.

Proof of Theorem 2
Next to lemma 1 from theorem 1 we need the following lemma:

Lemma
If $\|Q^T Q - I\|_2 \leq \alpha$ then $\|Q\|_2 \leq 1 + \frac{1}{2}\alpha$.
Proof:
This result follows directly from the inequality $\sqrt{1+x} \leq 1 + x/2$ for $x \geq 0$.

Substitution of the expressions for t and v yields:

$$\begin{aligned} Q^T t &= Q^T (p - v + \delta p - \delta v) \\ &= (Q + \delta_1 Q)^T p - Q^T v + Q^T(\delta p - \delta v) - \delta_1 Q^T p . \end{aligned}$$

This implies

$$Q^T t = (I - Q^T Q)s - Q^T \delta_2 Q s + Q^T(\delta p - \delta v) - \delta_1 Q^T p,$$

which yields

$$\|Q^T t\|_2 \leq (\alpha + \mu\|Q\|_2^2)\|s\|_2 + \tfrac{1}{2}\mu\|Q\|_2\left(1 + \frac{\|v\|_2}{\|p\|_2}\right)\|p\|_2 + \mu\|Q\|_2\|p\|_2 .$$

With the use of formula 3.1, the observation $\|p\|_2 \leq \|t\|_2\kappa$ and
the inequality $\|s\|_2 \leq (1+\mu)\|Q\|_2\|p\|_2$, the result follows directly.

□

According to this theorem we use the following stopping criterion:
In line 4 of CGSI the condition on orthogonality can be implemented as:

$\|t\|_2 > \|p\|_2 / \kappa$ {for some positive $\kappa \geq \beta$ } .

The reliability on the orthogonality of Q diminishes with larger values of κ .
Conclusions from extensive experimenting on the choice of κ are reported in the next section; a provisional statement is that algorithm CGSI gives good results with the choice $\kappa = 2$.
From Ruhe's observation that the calculation of r_j is equivalent with Gauss-Jacobi iteration (c.q. Gauss-Seidel iteration) on the linear system $Q^T Q r_j = Q^T a_j$ and from the fact that the iteration matrix $Q^T Q$ is close to a diagonal matrix, we may conclude that in general the calculated solution is very accurate. In practice we can assume that the calculated r_j gives a small residue which says that for a constant γ that is not too big the following bound holds :

$$\|Q^T a_j - Q^T Q r_j\|_2 \leq \gamma.\varepsilon\,\|a_j\|_2 .$$

Let us focus on algorithm CGSI. For vector t we have theoretically:

$$t = a_j - Q r_j .$$

so that for the quantity $\|Q^T t\|_2$ we find

$$\|Q^T t\|_2 \leq \gamma.\varepsilon\,\|a_j\|_2 .$$

If a single iteration step is sufficient for the current value of j, we have $p = a_j$ so that for the quotient $\frac{\|Q^T t\|_2}{\|t\|_2}$ we have

$$\frac{\|Q^T t\|_2}{\|t\|_2} \leq \gamma.\varepsilon\,\kappa\,\frac{\|a_j\|_2}{\|p\|_2} = \gamma.\varepsilon\,\kappa\,. \qquad (3.2)$$

If an extra iteration step is needed, we are in the situation that $a_j \approx Qr_j$ i.e. $\|t\| << \|a_j\|$ (in all our experiments we never observed the need for more than one extra iteration step). The extra iteration can be interpreted as a calculation of t and of the residual vector $Q^T t$ in extended precision. Also in this situation we will find a good relative accuracy. In either situation, the bound in theorem 2, which depends on $\|Q^T Q - I\|_2$, may be therefore (much) too pessimistic.

Suppose that $\tilde{Q}$ denotes the matrix that follows from adding the calculated column q_j (i.e. $N(t)$) to columns $(q_1,\ldots,q_{j-1})$. In successive steps of the algorithm, matrix $\tilde{Q}$ replaces Q.

So we are concerned with a bound for $\|\tilde{Q}^T\tilde{Q} - I\|_2$ in relation to the bound for $\|Q^T Q - I\|_2$. This is settled in the following theorems.

In theorems 3 and 4 we treat the case that the added column has a spectral norm that is exactly equal to one; in theorem 5 we cover the effect of rounding errors.

Theorem 3.

Let $Q = [q_1,\ldots,q_{j-1}]$ and $\alpha \in (0,1)$ be such that $\|Q^T Q - I\|_2 \leq \alpha$.

Let q_j with $\|q_j\|_2 = 1$ be such that $\|Q^T q_j\|_2 \leq \omega$. Define $\tilde{Q} = [Q:q_j]$ then

$$\|\tilde{Q}^T\tilde{Q} - I\|_2 \leq \tfrac{1}{2}\,(\alpha + \sqrt{\alpha^2 + 4\omega^2}\,)\,.$$

Proof of Theorem 3.

If τ is an eigenvalue with maximal modulus of the $(j \times j)$ symmetric matrix $([Q:q_j]^T[Q:q_j] - I)$ then $\|\tilde{Q}^T\tilde{Q} - I\|_2 = |\tau|$.

Suppose that $(x:\delta)^T$ defines a partitioning of an eigenvector corresponding with τ and define $V = Q^T Q - I$ and $w = Q^T q_j$ then the following equations hold:

$$Vx + \delta w = \tau x \quad \text{and} \quad w^T x = \tau\delta\,. \qquad (3.3)$$

If $\delta = 0$ then $Vx = \tau x$ from which we find $|\tau| \leq \alpha$ which ends the proof for this case.

For $\delta \neq 0$ we may assume $x \neq \underline{0}$; the case $x = \underline{0}$ can only occur for $\tau = 0$ which satisfies the inequality to be proven in a trivial way.

Eliminating δ from equations (3.3) yields

$$Vx + \tau^{-1}ww^T x = \tau x.$$

This leads to a quadratic equation in τ by taking innerproducts with x and multiplication with τ.

Through the observation that the Rayleigh quotient $\frac{x^T V x}{x^T x}$ is bounded by α we find from this quadratic:

$$|\tau| \leq \tfrac{1}{2}\,(\alpha + \sqrt{\alpha^2 + 4\omega^2}\,)\,,$$

which ends the proof.

□

Using a fixed upperbound for $\|Q^T q_j\|$, independently of j, the departure from orthogonality of matrix Q can be expressed as in the following theorem:

Theorem 4.
Let $Q_j = [q_1,\ldots,q_j]$., for $j = 1,\ldots,n$. If $\|q_k\|_2 = 1$, for $k = 1,\ldots,j$ and ω is such that $\| Q_{k-1}^T q_k \|_2 \le \omega$ for $k = 2,\ldots,j$ then $\|Q_j^T Q_j - I \|_2 \le \omega\sqrt{2j}$.

Proof of theorem 4:
For the proof we use the following lemma
Lemma:
The elements of the row $(a_1, a_2, \ldots, a_k, \ldots)$, defined by the recurrence relation
$a_1 = 1;\ a_{k+1} = \frac{1}{2}(a_k + \sqrt{a_k^2 + 4})$, $k = 1, 2, \ldots$
satisfy $a_k < \sqrt{2k}$.
Proof:
The proof is by mathematical induction.
For $k = 1$ the result holds.
Assume that $a_k = f\sqrt{2k}$ for some value of $f < 1$. Using the definition of a_{k+1} we find

$$\left(\frac{a_{k+1}}{\sqrt{2k+2}}\right)^2 = \frac{f^2k + 1 + \sqrt{f^4k^2 + 2f^2k}}{2k + 2} < \frac{f^2k + 1}{k + 1}$$

From this we conclude $\frac{a_{k+1}}{\sqrt{2k+2}} < 1$ which ends the proof of this lemma.
We like to comment that the given bound is rather sharp for relatively small values of k already, as can be concluded from simple calculations which yield for example $\frac{a_{50}}{\sqrt{100}} \approx 0.99$.

The final proof of theorem 4 follows directly from the application of theorem 3 and the above lemma.

□

In practice, the norms of the columns of Q are not exactly equal to one. The consequences of this are considered in the next theorem.

Theorem 5.
Assume that $Q_j = [q_1,\ldots,q_j]$ is calculated in floating-point arithmetic.
Let $\delta_k = \|q_k\|_2$, for $k = 1,\ldots,j$ and $D_j = \mathrm{diag}(\delta_1,\ldots,\delta_j)$.
Let $\omega > 0$ be such that $\| Q_{k-1}^T q_k \|_2 \le \omega\, \delta_k$ for $k = 2,\ldots,j$
and let $\sigma \in (0,1)$ be such that $|\delta_k^2 - 1| \le \sigma$ for $k = 1,\ldots, j$.
Then $\|Q_j^T Q_j - I\|_2 \le \sigma + \omega(1 + \sigma)\sqrt{2j}$.

Proof of theorem 5:
Observe that theorem 4 is applicable for matrix $Q_j D_j^{-1}$.
If W_j is defined by $W_j = D_j^{-1} Q_j^T Q_j D_j^{-1}$ we find accordingly
$\|W_j - I\|_2 \le \omega\sqrt{2j}$.
For the spectrum of W_j, $\lambda(W_j)$, we have consequently
$\lambda(W_j) \in [1 - \omega\sqrt{2j}\ ,\ 1 + \omega\sqrt{2j}]$.
Combining this with $\lambda(D_j) \in [\sqrt{1 - \sigma}\ ,\ \sqrt{1 + \sigma}]$ yields
$(1 - \omega\sqrt{2j})(1 - \sigma) \le \|D_j W_j D_j\|_2 \le (1 + \omega\sqrt{2j})(1 + \sigma)$,
so that for the spectrum of $(Q_j^T Q_j - I)$ we have

$\lambda(Q_j^T Q_j - I) \in [-\sigma - \omega (1 - \sigma) \sqrt{2j}\ ,\ \sigma + \omega (1 + \sigma) \sqrt{2j}\ ]$
from which the desired result directly follows.

□

4. NUMERICAL EXPERIMENTS

All experiments were carried out on the CYBER 205 computer of SARA, the Academic Computer Centre in Amsterdam. For this machine the value of e is about 5_{10}-14. We carried out experiments with algorithms CGSI and MGSCI on a large number of matrices having various numbers of rows and columns and different sorts of distributions for their singular values. The smallest matrices consisted of 50 rows and 25 columns; the largest matrices of 210 rows and 200 columns.
The matrices are constructed by multiplying a given diagonal matrix (singular values) from both sides by random orthogonal matrices. The maximal singular value is always equal to 1 and the smallest varies between 0.1 and $_{10}$-12 so that the condition number of the matrices is between 10 and $_{10}12$.
We have observed that the distribution of the singular values within the interval $[\sigma_{min},\ldots,\sigma_{max}]$ is of little importance for the resulting orthogonality of Q.

The number of iterations performed depends on parameter κ; for all matrices used, κ has been given the values 2, 10, $_{10}2$, $_{10}3$, ... , $_{10}10$, in successive experiments. The effect of taking a smaller value for κ is that in some cases a second iteration is necessary to calculate the next column of Q; a third iteration never occurred.

In table 1 we show a representative selection of our test results; it shows the typical behaviour of algorithms CGSI and MGSCI for various values of parameter κ.
The average number of iterations per column is denoted by ν; the departure from orthogonality is measured in the l_1-norm, and given by $\| Q^T Q - I \|_1$.
All matrices used in the selection described in table 1 have m = 210 and n= 100; the singular values are distributed equally over the interval [(conditionnumber)$^{-1}$, 1].

We also carried out a number of experiments with matrices that are close to a matrix of rank one.
A representative result is described in table 2; the matrix that is used has m = 50 and n = 25 ,the largest singular value is equal to 1 and the remaining 24 singular values are distributed equally in [1.0_{10}-11 , 1.0_{10}-10].

cond. nr.	k	ν (= avg. nr. iter. per col)		error in $(Q^TQ - I)$	
		CGSI	MGSCI	CGSI	MGSCI
10	2	1.1	1.1	2.6_{10}-13	1.3_{10}-13
	10	1	1	3.0_{10}-13	1.6_{10}-13
$_{10}$+4	2	1.78	1.78	1.8_{10}-13	8.9_{10}-14
	10	1.57	1.57	3.3_{10}-12	3.1_{10}-13
	$_{10}$+2	1.26	1.26	3.1_{10}-10	3.4_{10}-12
	$_{10}$+3	1.02	1.02	7.8_{10}- 9	1.3_{10}-11
	$_{10}$+4	1	1	9.8_{10}- 9	1.7_{10}-11
$_{10}$+7	2	1.86	1.86	2.1_{10}-13	7.7_{10}-14
	10	1.76	1.76	1.1_{10}-12	2.7_{10}-13
	$_{10}$+2	1.58	1.58	5.9_{10}-10	4.6_{10}-12
	$_{10}$+3	1.42	1.42	6.5_{10}- 8	1.3_{10}-10
	$_{10}$+4	1.27	1.27	5.3_{10}- 6	7.0_{10}-10
	$_{10}$+5	1.13	1.13	1.8_{10}- 4	3.0_{10}- 9
	$_{10}$+6	1.01	1.01	6.5_{10}- 3	1.0_{10}- 8
	$_{10}$+7	1	1	6.5_{10}- 3	1.0_{10}- 8
$_{10}$+10	2	1.89	1.89	2.1_{10}-13	7.8_{10}-14
	10	1.81	1.81	7.6_{10}-12	2.1_{10}-13
	$_{10}$+2	1.71	1.71	3.6_{10}-10	4.9_{10}-12
	$_{10}$+3	1.6	1.6	8.4_{10}- 8	3.8_{10}-11
	$_{10}$+4	1.51	1.51	3.1_{10}- 6	4.0_{10}-10
	$_{10}$+5	1.29	1.39	1.0_{10}+0	1.2_{10}- 8
	$_{10}$+6	1.06	1.28	"	8.5_{10}- 8
	$_{10}$+7	1	1.20	"	2.1_{10}- 7
	$_{10}$+8	1	1.09	"	2.8_{10}- 6
	$_{10}$+9	1	1.01	"	1.8_{10}- 5
	$_{10}$+10	1	1	"	1.8_{10}- 5

table 1.

cond. nr.	k	ν (= avg. nr. iter. per col)		error in (QTQ - I)	
		CGSI	MGSCI	CGSI	MGSCI
$_{10}$+11	$_{10}$+8	1.96	1.96	6.1_{10}-14	3.0_{10}-14
	$_{10}$+9	1	1.48	1.0_{10}+0	1.7_{10}- 5
	$_{10}$+10	1	1.08	"	2.2_{10}- 4
	$_{10}$+11	1	1	"	2.3_{10}- 4

table 2.

5. Discussion

We observed that for all matrices in all experiments the decomposition is accurate, which means that we always find matrices Q and R such that the norm of the residue, $\| A - QR \|_2$, is of the order of magnitude of the machineprecision relatively to $\| A \|_2$, even in cases where Q is far from orthogonal.

For all matrices, with both the modified and the classical iterative Gram-Schmidt algorithm, the choice $\kappa = 2$ results in a matrix Q that is orthogonal to almost full precision; the condition of the matrix and the distribution of the

singular values is only reflected in the number of columns that needs a second iteration to yield this good orthogonality.
For matrices that are not well conditioned, the orthogonality becomes worse with larger values of k for both CGSI and MGSCI.
If κ is given so large a value that no column of Q needs a second iteration, the results of the one-step classical and the one-step modified Gram-Schmidt respectively are produced. In this one-step situation we observe for the modified Gram-Schmidt algorithm that the orthogonality of Q (i.e. $\| Q^TQ - I \|_2$) is roughly equal to ε times the conditionnumber of the matrix, which confirms the bounds for the modified algorithm as given by Björck (cf. remark at the end of section 2). With the classical algorithm in that situation the results are bad; for every matrix with a conditionnumber larger than $1/\sqrt{\varepsilon}$ the result was $\| Q^TQ - I \|_2 \approx 1$.
Using the modified Gram-Schmidt algorithm we observe that for all values of κ the orthogonality is roughly bounded by $\kappa\,\varepsilon$.
This is according to theorem 5 applied with $\omega \approx \kappa\,\varepsilon$ (discarding the factor $\sqrt{n}$).
In view of formula 3.2, we conjecture that the iterated modified Gram-Schmidt algorithm produces columns q_j that satisfy $\| Q_{j-1}^T q_j \|_2 \lesssim \kappa\,\varepsilon$, also for large values of κ. This inspires us to the following conjecture:

CONJECTURE: If the iterated modified Gram-Schmidt algorithm is used with a value of $\kappa \geq 2$, then the resulting matrix Q satisfies: $\| Q^TQ - I \|_2 \lesssim \kappa\,\varepsilon\,\sqrt{n}$.

For all matrices we tested, no matter its condition, this relation was fulfilled; we did not observe a relation of a similar sort for the classical Gram-Schmidt algorithm. This conjecture has the following application. For larger values of κ, the number of columns that need a second iteration may diminish. So, if the wanted accuracy is denoted by η (η should not be chosen smaller than $\approx 2 \times \varepsilon$), the factor κ may be chosen according to $\kappa = \text{maximum}\left(\frac{\eta}{\varepsilon\sqrt{n}}, 2\right)$.
Both CGSI and MGSCI can be used to solve the orthogonal basis problem. The operation count for these algorithms is νmn^2 flops.
The solution of the orthogonal basis problem with Householders method requires $2 \times (mn^2 - n^3 / 3)$ flops (see for instance Golub & Van Loan [5] p.152), so that for $\nu < 2 - (2n) / (3m)$ the iterative Gram-Schmidt algorithms require less operations.

6. CONCLUSIONS

Regarding the use of algorithms CGSI and MGSCI we come to the following conclusions.
For any small value of parameter κ ($\kappa = 2$, for example, will do) the orthogonality of the resulting matrix Q is of the order ε for both the modified and the classical iterative Gram-Schmidt algorithm.
Consequently, from a numerical point of view, there is no reason to prefer the iterative modified algorithm over the iterative classical algorithm **when full precision is wanted**.
Hence, a choice between these two algorithms can be made on considerations regarding efficient execution of the resulting code. For instance, we can use a

fast matrix - vector multiplication routine in CGSI but not in MGSCI. This consideration clearly favours CGSI.

On the other hand, when **less than full precision is sufficient,** then the use of algorithm MGSCI may be advantageous; according to the conjecture stated in the previous section, the parameter κ can be given an optimal value to cut down on iterations.
In cases that the wanted accuracy η is much larger than ε, for example $\eta = \sqrt{\varepsilon}$, this strategy yields a rather large value for κ, which may result in considerable savings on the number of 'reorthogonalizations'. This is especially valuable for matrices with a bad condition.

ACKNOWLEDGEMENTS
The author wants to thank professor T.J. Dekker for his contribution to the proof of theorem 5.

LITERATURE
1. A. Björck, Solving linear least squares problems by Gram-Schmidt orthogonalization, *BIT* 7, 1-21 (1967).
2. P. Businger and G.H. Golub, Linear least squares solutions by Householder transformations, *Numer. Math.* 1, 269-276 (1965).
3. T.F. Chan, Rank revealing QR factorizations, *Linear Algebra Appl.* 88/89, 67-82 (1987).
4. J.W. Daniel, W.B. Gragg, L. Kaufman and G.W. Stewart, Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization, *Math. Comp.* 30,772-795 (1976).
5. G.H. Golub and C.F. van Loan, *Matrix Computations*, North Oxford Academic, Oxford 1983.
6. B.N. Parlett, *The Symmetric Eigenvalue Problem*, Prentice Hall, Englewood Cliffs N.J. 1980.
7. A. Ruhe, Numerical aspects of Gram-Schmidt orthogonalization of vectors, *Linear Algebra Appl.* 52/53, 591-601 (1983).

When you notice a cat in profound meditation,
The reason, I tell you, is always the same:
His mind is engaged in a rapt contemplation
Of the thought, of the thought, of the thought of his name:
His ineffable effable
Effaneffable
Deep and inscrutable singular Name.

T.S. Eliot,
The Naming of Cats.

The Sage falls asleep not because he ought to. Nor even because he wants to.
But because he is sleepy.

Raymond M. Smullyan, The Tao is silent.

CHAPTER VI

Definition and use of
Householder reflections

reprint of:
CS-88-05 Dept. of Comp. Syst.
Univ. of Amsterdam, 1988

DEFINITION AND USE OF HOUSEHOLDER REFLECTIONS

by

W. Hoffmann

Abstract: In this paper we present various ways to define and calculate Householder reflections for QR-factorization and for orthogonal similarity transformation to construct a tridiagonal matrix that is similar with a given symmetric matrix. The latter is a subtask for the solution of the eigenproblem. The algorithms presented are used for routines on a CYBER 205 vectorcomputer. The performance of these routines is compared with routines based on Householder reflections from existing program libraries and with routines using other orthogonal methods.

Keywords and phrases: Householder reflection, QR-factorization, similarity transformation.
AMS subject classification: 65F15, 65F25.

1. INTRODUCTION

Orthogonal transformations play an important role in numerical linear algebra, because their application does not change the condition of the problem at hand. They are used for the construction of a QR-factorization of a matrix and for orthogonal similarity transformation in the solution of the eigenvalue-problem.

For QR-factorizations, the suitable methods are described geometrically by projections (Gram-Schmidt orthogonalization), rotations (Givens transformations) or reflections (Householder transformations)

In the area of vector computing for full matrices, it apears that the use of Givens or fast Givens doesn't yield efficient routines. This is explained by the fact that methods of Givens type operate on rows of a matrix, in contrast with Householder and Gram-Schmidt type methods, which operate on columns of a matrix. Even if the matrices that are operated upon can be efficiently accessed rowwise, one should realise that in general, for this type of problems, the column vectors are (much) longer than the row vectors.

In this paper we concentrate on Householder reflections. When the orthogonal matrix from the QR-factorization is needed explicitly (solution of the so called orthogonal basis problem), the application of iterated Gram-Schmidt orthogonalization [4,6] is advised, as it gives the fastest code; when the product form of the orthogonal matrix can be used, then routines based on Householder's method are to be prefered.

In section 2 we give a description of various ways to define a Householder reflection and in section 3 we demonstrate how these reflections can be used for a QR-factorization. Also in section 3 we report experiments on the CYBER 205 vector computer of the Academic Computing Centre SARA at Amsterdam.

If orthogonal transformations are needed for the construction of a similarity transformation, as is the case in methods for solving the eigenproblem for full matrices, then methods based on projections can not be used, as the defining matrices are not invertable. Consequently, only rotations and reflections are applicable; the methods based on Householder reflections give the most efficient routines in this area too.

In this paper we deal with similarity transformation for symmetrical matrices; in section 4 we describe the use of Householder reflections to calculate a tridiagonal matrix which is similar to a given symmetrical matrix. Also for this task, we report numerical experiments on the CYBER 205.

2. DEFINITION AND CONSTRUCTION OF HOUSEHOLDER REFLECTIONS

Unless stated otherwise, the vectornorm used will be the l_2-norm .
Let $x = (\xi_1, \ldots \xi_n)^T$ and $y = (\eta_1, \ldots \eta_n)^T$ denote two non-zero vectors in $\mathbf{R}^n$ such that x is not a multiple of y. It is well known [4], that a reflection H (being an orthogonal operator of $\mathbf{R}^n$) can be constructed such that x is transformed into a multiple of y.

The operator defining this transformation, named after Householder [7,8,16], is of the form

$$H = (I - \kappa v v^T), \tag{2.1.}$$

where v is either given by

$$v = \|y\|_2 \, x + \|x\|_2 \, y \, , \tag{2.2.}$$

or by

$$v = \|y\|_2 \, x - \|x\|_2 \, y \tag{2.3.}$$

and κ is given by

$$\kappa = \frac{2}{v^T v} \, . \tag{2.4.}$$

Note that vector v from formula 2.2 corresponds with the inner bisector of the angle between x and y and vector v from formula 2.3 with the outer bisector . The computation of v according to formula 2.2 will most probably suffer from numerical cancelation if x and y are (almost) in opposite direction; formula 2.3 is not to be preferred if x and y are (almost) in the same direction.

A numerically stable algorithm comes from the formula which produces the larger vector v. This is the vector that corresponds with the bisector of the acute angle between x and the span of y. From this we see that formula 2.2 may be used in the case $x^T y > 0$ and formula 2.3 otherwise.

To prevent arithmetical overflow or underflow in the calculation of $\|x\|_2$ and $\|y\|_2$, it is advised that x and y are properly scaled initially. From the formulae above it follows that a multiplication of x or y by any nonzero factor results in a vector v that is multiplied by the same factor and that matrix H remains unaffected under that operation.
In practical applications of Householder matrices, y has a length of order unity, so the proper scaling of x and y applies only to x. A suitable and easy to calculate scaling factor is for instance $1 / \|x\|_\infty$.

An important special case is $y = e_1$. The choice for the proper formula can be based on the sign of the innerproduct ξ_1 . The required algorithm now consists of constructing $v = (\phi_1,\ldots,\phi_n)^T$ such that a given arbitrary vector x is mapped on a multiple of e_1. An algorithm for this task belongs to the standard tools of numerical linear algebra.

In the situation that x is a multiple of e_1 already, the transform of x may be defined to be x itself.
This can be interpreted as a replacement of a reflection by the identity, which is justified if the distance of x to the span of e_1 (i.e. $\|x - \xi_1 e_1\|_2$), is less than a relative tolerance 'eps'. By doing so, however, we introduce a discontinuity in the linear operator H. A vector x for which this distance is slightly larger than 'eps' will have its image close to -x, while a vector at a smaller distance remains unchanged. This drawback can be avoided by choosing vector v to correspond with the outer bisector of the acute angle between x and λe_1, instead of the inner bisector as described above. The idea to consider the outer bisector for a Householder reflection and its numerically stable computation are due to Parlett [11].

For the constuction of the required reflection we may also consider to use a vector v that is scaled to a predefined length. In that case the value of k is independent of v so that it needs no longer be stored explicitly. For instance, if v is scaled such that v^Tv equals 2, then κ becomes 1.
Summarizing, we observe at three points a possibility of choice in the construction of a Householder reflection:

- The possibility of explicit scaling input vector x to protect against overflow; we denote the alternatives by the use of letter p or letter u respectively.
- Consider the outer bisector instead of the (usual) inner bisector to define v; we denote the alternatives by the use of letter o or letter i respectively.
- Use the scaling factor k or deliver v at predefined length; we denote the alternatives by the use of letter k or letter v respectively.

In the following algorithm we present all alternatives for computing the data of a Householder reflection that maps a given vector x on a vector $z = (\zeta_1, 0, \ldots, 0)^T$ which is a multiple of e_1. If x is close to a multiple of $e_1$already (this includes the case $x = \underline{0}$), then the transformation is skipped. Only the first component of vector z is calculated explicitly.
The algorithm is as follows:

{p; overflow protected }	{u; unprotected}
1. $\mu := \max_i \lvert\xi_i\rvert$	1. $\phi_i := \xi_i$, i = 1,…,n
2. if $\mu \neq 0$ then $\phi_i := \xi_i / \mu$, i = 1,…,n	2. $\mu := 1$

3. $\tau 2 := \sum_{i=2}^{n} \phi_i^2$
4. if $\tau 2 \leq (\text{eps} \times \phi_1)^2$ skip transformation
5. $\sigma := \sqrt{\phi_1^2 + \tau 2}$

{o; outer bisector}	{i; inner bisector}
6. if $\phi_1 \geq 0$ then $\sigma := -\sigma$	6. if $\phi_1 < 0$ then $\sigma := -\sigma$
7. $\phi_1 := -\frac{\tau 2}{\phi_1 - \sigma}$	7. $\phi_1 := \phi_1 + \sigma$

8. $\kappa := 1 / (\sigma\, \phi_1)$

{v; scale v}	{k; use of κ}
9. $\phi_i := \phi_i$ / sqrt(κ), i = 1,…,n	9. store κ

10. $\zeta_1 := -\sigma\mu$

We advocate the version that uses the outer bisector and scaling of vector v. In the case that the transformation is skipped, i.e. is replaced by the identity, vector v is set equal to the zero vector. With respect to explicit scaling of the

input vector x, we observe a substantial speed up of the resulting QR-routine if this scaling is omitted; this is especially true if the length of the columns is (much) larger then the total number of columns. More details on this are reported in the section on numerical experiments.

3. The use of Householder reflections to calculate a QR factorization

Consider an $m \times n$ matrix $A = [a_1,\ldots,a_n]$ with $a_j \in \mathbf{R}^m$ and $m \geq n$. It is well known that an algorithm for constructing a QR-factorization of A can be designed with the use of Householder's transformation [1,2,6,7,15]. This construction is via successively premultiplying matrix A by selected Householder matrices H_j, $j = 1,\ldots, n$. A premultiplication by H_j introduces zeroes in the last n-j positions of the j-th column. It leaves the first j-1 columns of the matrix unchanged and the effect on the remaining n-j columns is that of a rank-one matrix update.
For a given Householder matrix H (=H_j) and one of the last n-j columns a_p, say, the product Ha_p is calculated via

$$Ha_p = (I - \kappa vv^T)a_p = a_p - \kappa(v^T a_p)v , \qquad (3.1.)$$

which requires a vector-update and the calculation of an innerproduct.
After n premultiplications the result is

$$H_n H_{n-1}\cdots H_1 A = R, \qquad (3.2.)$$

where the non-zero elements of R form an upper triangular matrix.
Defining Q by $Q^T = H_n H_{n-1}\cdots H_1$, this latter relation can also be written as

$$A = QR .$$

From the fact that each Householder matrix is orthogonal, it follows that Q is an orthogonal matrix.

In step j of the QR-factorization, the appropriate Householder matrix H_j that introduces the required zero elements in the j-th columnin is constructed as described in section 2.
Here we consider the premultiplication by matrix H_j, which can be done in two different ways.

i) each multiplication with H_j is directly effectuated on all remaining columns a_p , p = j+1,...,n:

```
For j = 1,..., n do
1. construct vector v_j and scalar κ_j to define H_j
2.   For k = j+1,..., n do
         a_k := H_j a_k
```

ii) before constructing v_j, accumulate the effect of all earlier premultiplications on the j-th column:

```
For j = 1,..., n do
1.  For k = 1,..., j-1 do
        a_j := H_k a_j
2.  construct vector v_j and scalar κ_j to define H_j
```

The vectors v_j can be stored in the lower trapezoidal part of matrix A. The diagonal of R can be stored in a one-dimensional array, while the strictly upper triangular part of R may be stored in the corresponding part of A. If the variant is used that doesn't scale v_j, the scalars κ_j must be stored in an extra one-dimensional array.
From the available data it is possible to calculate Qx, for any vector x, without calculating matrix Q. If Q is wanted explicitly, all premultiplications $H_1 \ldots H_n$ must be accumulated starting from the identity matrix. For that purpose, however, Gram-Schmidt orthogonalisation is faster. This is confirmed by experiments that are reported below.

Numerical Experiments
All our experiments were carried out on the CYBER 205 vector computer of SARA, the Academic Computing Centre at Amsterdam. We investigated various implementations of the QR factorisation using Householder reflections. For the definition of the Householder reflections we used the variants that follow from our algorithmic description in section 2. In all situations we used the variant that skips a transformation if x is close to a multiple of e_1.
Implementations of the choices i or o (inner or outer bisector) and k or v (store scalar κ or explicitly scale v) resulted in subroutines with negligeable differences in execution time. This can be explained from the fact that only n different Householder reflections need to be constructed where $O(n^2)$ multiplications Ha_p are to be executed. The choice between versions p and u (scaling of the input vector or not) resulted in small differences in the execution times as reported in table 1.
With respect to choice i versus o, we are in favour of the method using the outer bisector, because it is less depending on the size of the relative tolerance that rules the replacement of a reflection by the identity. With respect to the choice v versus k, we are in favour of version v, because it saves the use of an extra one-dimensional array and its implementation on a vectorcomputer gives a code which is by no means slower.

We experimented with our algorithm on a number of matrices, varying in size and condition. The spectral condition number varied between 1 and 10^{10} .
In all cases, we checked the residual $\|A - QR\| / \|A\|$ and found it always to be about 10^{-13}, which is roughly the machine precision. We also checked the measure for orthogonality $\|Q^TQ - I\|$ and found it to be equally small in all cases.
In the next table we show the processing time for the calculation of a QR factorization for various $m \times n$ matrices. We show the total time needed (in seconds) for a QR factorisation with our routine written in FORTRAN 200 which is a superset of FORTRAN 77 with vector extensions for the CYBER 205. On each line we present in upright font the time for version u that does

not perform explicit scaling of the input vector and in italics the time for version p with explicit scaling.

n =	25	50	100	200
m = 50	0.0019	0.0070	*	*
	0.0021	*0.0073*		
m = 100	0.0024	0.0089	0.0321	*
	0.0027	*0.0093*	*0.0328*	
m = 200	0.0034	0.0129	0.0478	0.1684
	0.0038	*0.0135*	*0.0490*	*0.1703*

Table 1.

QR factorisation of m × n matrix ('outer bisector', scaling of v) - time in sec.'s

If Q is to be calculated explicitly, the total time needed is roughly twice as much.

In the next table we present the efficiency rate of the algorithm for the same values of m and n expressed in Megaflops (milion floating-point operations per second). For the number of floating point operations we only considered the contribution of the higher order terms: $2\,n^2(m - n/3)$.

n =	25	50	100	200
m = 50	27.4	23.8	*	*
m = 100	47.7	46.8	41.6	*
m = 200	70.5	71.1	69.7	63.3

Table 2.

QR factorisation (outer bisector , scaling of v) - performance in Mflops

For the asymptotical efficiency [5], we found for this algorithm $r_\infty = 125$. With respect to the performance parameter $n_{1/2}$ we computed that half of this efficiency is reached for $m \approx 157$.

We also experimented with some well known other methods and compare the results for a 200 × 100 matrix in the next table.

Givens (R and rotations):	0.261
Fast Givens (R and scaled rotations):	0.191
Modified Gram-Schmidt (R and Q):	0.064
Householder's method (R and Hh. vectors):	0.048

Table 3.

QR factorization for a 200 × 100 matrix - time in sec.'s

With respect to the calculation of a QR factorisation of a full rectangular matrix on a vector computer we come to the following conclusions:

- If only the triangular matrix R is needed then Householder's method yields the fastest routine;
- If Q is needed explicitly then, in view of the statement following table1, modified Gram-Schmidt is fastest;
- The use of Givens or fast Givens (for full matrices) is not profitable on a vectorcomputer.

4. ORTHOGONAL SIMILARITY TRANSFORMATIONS ON A SYMMETRIC MATRIX

In the context of eigenvalue calculations, Householder matrices are used for the construction of a similarity transformation to produce a matrix that admits eigenvalue calculation in a simpler way. In the symmetric case, a similar tridiagonal matrix is constructed for which the eigenvalues can be calculated by means of the QR-method or a root finding method which is based on evaluation of the characteristic polynomial or based on information from an LU factorisation [7,12].

The construction of a tridiagonal matrix that is similar to a given $n \times n$ symmetric matrix using Householder reflections is carried out in n-2 stages as follows. In stage j, a Householder reflection is constructed that introduces zeroes in the j-th column in positions j+2, ..., n . To retain a similar matrix, the postmultiplication with the inverse of the premultiplier must also be applied. For Householder matrices, this inverse is equal to the matrix itself.
The algorithm for constructing the required tridiagonal matrix has the following structure:

```
For j = 1, ..., n-2 do
1.construct v_j and κ_j to define apropriate Householder matrix H_j
2.A:= H_j A H_j
```

Execution of the pre- and post multiplication with the Householder matrices can be performed in two ways, which are described hereafter.
In the sequel, we drop the subscripts j.

i) For a given Householdermatrix $H = (I - \kappa vv^T)$ the transformation $A := H A H$ can be performed by calculating $A := (I - \kappa vv^T)A$ followed by $A := A (I - \kappa vv^T)$:

```
1.  For k = j+1, ..., n do
1.1.    ω_k := v^T a_k
1.2.    a_k := a_k - κ ω_k v
2.  enddo
3.  w := A v
4.  A := A - κ w v^T
```

If we assume that vector v consists of (n-j) elements, then the number of multiplications for this piece of algorithm is equal to $4(n-j)^2 + O(n)$. The algorithm for constructing the required similarity transformation in this way will cost $\frac{4}{3}n^3 + O(n^2)$ multiplications.
On a vector machine such as the CYBER 205, vector w in line 3 is calculated efficiently by adding appropriate multiples of the columns of A, as is well known. Line 4 denotes a rank-one matrix update and is implemented in a standard way. Both statements can be evaluated using BLAS [3,9].
On machines with parallel processors, full advantage can be taken of efficient matrix-vector calculation. The algorithm has been implemented for the CYBER 205 and is reported in table 4 as TFSYMT .

ii) Observe the following relation:

$$H A H = (I - \kappa v v^T) A (I - \kappa v v^T)$$
$$= A - \kappa v v^T A - \kappa A v v^T + \kappa^2 v v^T A v v^T. \qquad (4.1.)$$

Defining $u = \kappa A v$, this can be written as:

$$H A H = A - v (u^T - \tfrac{1}{2}\kappa(v^T u) v^T) - (u - \tfrac{1}{2}\kappa(v^T u) v) v^T \qquad (4.2.)$$

which can be recognized as a rank-two matrix update. The algorithm is given below.

```
1.  u := κA v
2.  ω := κ(v^T u)/2
3.  y := u - ω v
4.  A := A - v y^T - y v^T
```

If we do not use the fact that the given matrix is symmetrical, then the number of multiplications for this piece of algorithm is equal to $3(k)^2 + O(k)$, where k is the order of the matrix. If the symmetry of the matrix is used, then $2(k)^2 + O(k)$ multiplications are sufficient. The complete tridiagonalization algorithm uses $n^3 + O(n^2)$ multiplications, or $\frac{2}{3}n^3 + O(n^2)$ if symmetry is exploited. The algorithm is implemented for the CYBER 205 and is reported as TFSY2U .

Numerical Experiments

In the next table we compare our two implementations of the Householder similarity transformation, TFSYMT ans TFSY2U, with implementations from the NAG library [10], from QQLIB [13] and from EISPACK [14]. Our routines are implemented to use all elements of the symmetrical input matrix. Versions that use only elements from a triangular part of the matrix result in slower subroutines on the CYBER 205; although only half of the matrix need to be updated in each step, a matrix-vector multiplication is much more inefficient. This will be reported in table 5.

In table 4 we also show the execution time for the calculation of all eigenvalues of the resulting tridiagonal matrix. This was done by NAG routine F02AVF, which is an implementation of the QR algorithm. Moreover we show the processing time to calculate the transformationmatrix Q explicitly, i.e. the time to accumulate all Householder reflections.

In the calculation of the values for r_∞ and $n_{1/2}$ [5], we used $2 \times \frac{2}{3}n^3$ for the number of floating-point operations needed.

n =	50	100	200	400	r_∞	$n_{1/2}$
TFSY2U	0.0117	0.0507	0.2430	1.2792	72	172
TFSYMT	0.0142	0.0646	0.3281	1.8316	50	167
NAG (F01AGF)	0.0229	0.0873	0.3659	1.6989	56	194
QQLIB(QQTRED2)	0.0240	0.1093	0.5397	2.9543	31	167
EISPACK(TRED1)	0.0451	0.3287	2.5016	-	4	-
EIGENVAL. CALC.	0.0191	0.0729	0.2795	1.0650		
ACCUMULATE Q	0.0066	0.0314	0.1675	0.9889		

Table 4. Similarity-transformation

From this table we draw the following conclusions.

- The ratio 4 : 3 of the numbers of multiplications in the two alternative implementations of the simlarity transformation using Householder's method, is almost exactly reflected in the ratio of the execution times.
- At the time of our experiments (fall 1988), the FORTRAN subroutine TRED1 from EISPACK was clearly not available in a vectorizeable form in contrast to F01AGF from NAG
- The concept of the $n_{1/2}$ parameter was not applicable for TRED1.

In table 5 we report on our efforts to implement our subroutines to be efficient with respect to usage of memory. At the first line the results of our routine TFSY2U are repeated; at the next line we present the results if relevant pieces of code are replaced by calls to BLAS level 2 routines. Following this, we present the results if only the triangular part of a 2-dimensional array is used and finally the results for the case that this triangular matrix is stored in linear array.

n =	50	100	200	400	r_∞	$n_{1/2}$
full mat f200	0.0117	0.0507	0.2430	1.2792	72	172
full mat blas	0.0217	0.0775	0.3353	1.7585	53	181
triangle f200	0.0157	0.0656	0.2958	1.4481	65	187
triangle blas	0.0228	0.0812	0.3297	1.5284	62	192
packed array f200	0.0158	0.0664	0.2959	1.4481	65	187
packed array blas	0.0231	0.0803	0.3234	1.5167	63	197

Table 5. Variants of TFSY2U (Similarity-transformation)

The conclusions are:

- The use of BLAS subroutines, contributes to (trans)portability of the codes; for small to moderately large values of n the respective subroutines are not the most efficiënt ones on the CYBER 205.
- Using only a triangle of a symmetric matrix will not result in faster routines on the CYBER 205, although only half of the elements of the matrix need to be updated. If the triangular part is packed in a linear array, only about half of the memory is necessary to store the matrix.

ACKNOWLEDGEMENTS

The author wants to thank Nico Koning for providing him with most of the data from table 3.

LITERATURE

1. A. Björck, Solving linear least squares problems by Gram-Schmidt orthogonalization, *BIT* 7, 1-21 (1967).
2. P. Businger and G.H. Golub, Linear least squares solutions by Householder transformations, *Numer. Math.* 1, 269-276 (1965).
3. J.J. Dongarra, J. du Croz, S. Hammarling and R.J. Hanson, An extended set of FORTRAN Basic Linear Algebra Subprograms, *ACM Trans. Math. Softw.* 14, 1-17 (1988)
4. G.H. Golub and C.F. van Loan, *Matrix Computations*, North Oxford Academic, Oxford 1983.
5. R.W. Hockney & C.R. Jesshope, *Parallel Computers - Architecture, Programming and Algorithms*, Adam Hilger, Bristol, 1981.
6. W. Hoffmann, Iterative algorithms for Gram-Schmidt orthogonalization; *report 87-22, Dept. of Mathematics*, Univ. of Amsterdam 1987, to apear in Computing (1989).
7. A.S. Householder, Unitary triangularization of a nonsymmetric matrix, *j. Assoc. Comp. Mach.* 5, 339-342 (1958)
8. A.S. Householder & F.L. Bauer, On certain methods for expanding the characteristic polynomial, *Numer. Math.* 1, 29-37 (1959)
9. C.L. Lawson, R.J. Hanson, R.J. Kincaid and F.T. Krogh, Basic Linear Algebra Subprograms for Fortran Usage, *ACM Trans. Math. Softw.* 5, 308-323 (1979)
10. NAG Library Manual; Numerical Algorithms Group, Oxford 1982.
11. B.N. Parlett, Analysis of algorithms for reflections in bisectors, *SIAM Review* , 13, 197-208, (1971)
12. B.N. Parlett, *The Symmetric Eigenvalue Problem*, Prentice Hall, Englewood Cliffs N.J. 1980.
13. QQLIB, A library of utility toutines and math. algorithms on the Cyber 200; Cyber 200 support, Roseville, MN. 1983.
14. B.T. Smith, J.M. Boyle, B.S. Garbow, Y. Ikebe, V.C. Klema and C.B. Moler, Matrix Eigensystem Routines - EISPACK Guide, *Lecture Notes in Computer Science 6*, Springer Verlag, Berlin, Heidelberg, New York (1974)
15. G.W. Stewart, Introduction to matrix computations, Academic Press, New York (1973)
16. J.H. Wilkinson, Householder's method for the solution of the algebraic eigenproblem, *Computer J.* 3, 23-27 (1960)

☆ ★ ☆ ★ ☆

CHAPTER VII

NUMVEC FORTRAN Library manual
Chapter: Simultaneous Linear Equations

jointly written with: W.M. Lioen
reprint of NM-R8614
CWI, Amsterdam, 1986

NUMVEC FORTRAN Library manual

Chapter: Simultaneous Linear Equations

W. Hoffmann
Department of Mathematics, University of Amsterdam
Roetersstraat 15, 1018 WB Amsterdam, The Netherlands

W.M. Lioen
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

This document describes a set of NUMVEC FORTRAN Library routines, dealing with the unique solution of real linear systems. Presently, only highly optimized non-portable implementations for the CYBER 200 series computer systems are included in the Library.

1980 Mathematics subject classification (1985 revision): 65V05, 65F05, 15A06
1982 CR Categories: 5.14.

Keywords & Phrases: Gaussian elimination, LDU-decomposition, linear equations, software.

Note: The implementations are available in FORTRAN 200 (the CYBER 200 series FORTRAN, a superset of standard FORTRAN including vector extensions).

Report NM-R8614
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Chapter ***

Simultaneous Linear Equations

1. Scope of the Chapter

This chapter is concerned with the solution of the matrix equation $AX = B$, where B may be a single vector or a matrix of multiple right hand sides.

2. Background to the problem

A set of linear equations may be written in the form

$$Ax = b$$

where the known matrix A is of shape $(m \times n)$, the known right hand vector b has m components, and the required solution vector x has n components. There may also be p vectors b_i, $i = 1,2, \cdots, p$ on the right hand side and the equation may then be written as

$$AX = B$$

the required matrix X having as its p columns the solutions of $Ax_i = b_i$, $i = 1,2, \cdots, p$.

In the following we assume A to be a real square non singular matrix (i.e. rank$(A) = m = n$), so $Ax = b$ has a unique solution.

In the Gaussian elimination case, the computation starts with the LDU decomposition $A = PLDUQ^{-1}$, where L and U are lower and upper triangular matrices, respectively, D is a diagonal matrix and P and Q are row and column permutation matrices, respectively, chosen so as to ensure that the decomposition is numerically stable. The solution is then obtained by solving in succession the simpler equations

$$LDz = P^{-1}b$$

$$Uy = z$$

$$x = Qy$$

the first by forward substitution, the second by back substitution and the third simply by

permutation of the elements of the solution vector.

The application of complete pivoting (both row and column interchanges), which yields a stable algorithm, has the disadvantage of expensive code. In 1971 Businger published an idea for calculating, with little extra cost, an upperbound for the growth factor during Gaussian elimination with partial pivoting (see [1]). If this upperbound for the growth factor becomes too large, the pivoting strategy can be switched to complete pivoting, so the efficiency of partial pivoting is combined with the stability of complete pivoting. This so-called mixed pivoting strategy provides the user with a reasonable upperbound for the growth factor, so that an estimate of the error matrix can be made (see [4]).

In the Gauß-Jordan elimination case, the computation starts with the diagonalization of the system $AX = B$ by elementary transformations, using a partial pivoting scheme.
The p columns x of the solution matrix X are found by dividing by the diagonal of the diagonalized matrix and permuting the elements of the solution vector.

Due to rounding errors the computed solution x_0, say, will always be an approximation to the true solution x both for Gaussian and Gauß-Jordan elimination. This approximation can be satisfactory, agreeing with x to several figures, but if the problem is ill-conditioned then x and x_0 may have few or even no figures in common.

A detailed description on the algorithmic choices concerning the normalization of D and the pivoting strategy can be found in [3].

2.1. References

[1] Businger, P.A., Monitoring the numerical stability of Gaussian elimination. In: Numerische Mathematik 16, 1971, pp. 360-361.

[2] Golub, G.H. and Van Loan, C.F., Matrix computations, North Oxford Academic, 1983.

[3] Hoffmann, W., Solving linear systems on a vector computer, to appear.

[4] Stewart, G.W., Introduction to matrix computations, Academic Press, 1973.

3. Recommendations on Choice and Use of Available Routines

Since in exceptional cases partial pivoting may yield useless results, even for well-conditioned matrices, the user is advised to use CCRMCF. If it is known beforehand, that partial pivoting will suffice, one can use CCRPCF.

GAUJOR should never be used just because of its capability of dealing with multiple right hand sides: if the problem is large enough the user should use a factorization routine (LURMMP or LURMPC) followed by a call to SLDUCP or SLDUPC, respectively, to solve the equations for each right hand side.
Only for 'small' problems (up to order about 50 for the CYBER 205) the user might want to use GAUJOR which, despite its somewhat greater complexity compared to Gaussian elimination, performs rather well due to the fact that all vectors remain of full length during every stage of the algorithm.

The Black Box routines CCRMCF and CCRPCF both simply call two general purpose routines and are provided for convenience.

Warning: No particular paging scheme is implemented, the matrices A and C have to fit in the working set in order to avoid thrashing (mortal paging).

Chapter Contents - *

Routine name	Purpose
Black Box Routines	
These routines solve the matrix equation $AX = B$	
CCRMCF	CCRMCF calculates the approximate solution of a set of real linear equations with a single right hand side, $Ax = b$, by an LDU factorization method with mixed pivoting.
CCRPCF	CCRPCF calculates the approximate solution of a set of real linear equations with a single right hand side, $Ax = b$, by an LDU factorization method with partial pivoting.
GAUJOR	GAUJOR calculates the approximate solution of a set of real linear equations with multiple right hand sides, $AX = B$, by Gauß-Jordan elimination with partial pivoting.
General Purpose Routines	
These routines solve the matrix equation $AX = B$	
CPIND	CPIND searches the element of maximum modulus (complete pivot) in the $(n-j+1) \times (n-j+1)$ submatrix of the $n \times n$ matrix A with the upper left corner of the submatrix corresponding with A_{jj}.
LURMMP	LURMMP decomposes a real matrix into one diagonal and two triangular matrices by an LDU factorization method with mixed pivoting.
LURMPC	LURMPC decomposes a real matrix into one diagonal and two triangular matrices by an LDU factorization method with partial pivoting.
SLDUCP	SLDUCP calculates the approximate solution of a set of real linear equations with a single right hand side, $Ax = b$, where A has been decomposed using LURMMP.
SLDUPC	SLDUPC calculates the approximate solution of a set of real linear equations with a single right hand side, $Ax = b$, where A has been decomposed using LURMPC.

CCRMCF - NUMVEC FORTRAN Library Routine Document

1. Purpose

CCRMCF calculates the approximate solution of a set of real linear equations with a single right hand side, $Ax = b$, by an LDU factorization method with mixed pivoting (see [2]).

2. Specification

```
      SUBROUTINE CCRMCF(A, IA, B, N, C, CPIV, RPIV,
     +                  EPS, GRWLIM, MAXNRM, UPBGRW, IFAIL)
C     INTEGER IA, N, CPIV(N), RPIV(N), IFAIL
C     REAL A(IA,N), B(N), C(N), EPS, GRWLIM, MAXNRM, UPBGRW
```

3. Description

Given a set of linear equations, $Ax = b$, the routine first decomposes A using an LDU factorization with mixed pivoting: $P^{-1}AQ = LDU$, where P and Q are permutation matrices, L is lower triangular, D is diagonal and U is upper triangular. The diagonals are normalized according to: $L_{ii} = U_{ii} = D_{ii}^{-1}$

Partial pivoting by column interchanges will be used as long as the calculated upper bound for the growth (see [1]) is less than a critical value, that equals GRWLIM $\times$ N $\times$ $||A||_\infty$. In practice, the upper bound for the growth rarely exceeds the critical value if we choose GRWLIM properly (see [1] and [4]), hence we will usually take advantage of the greater speed of partial pivoting (maximum search order N$-k+1$ in the k-th step), while in doubtful cases numerical difficulties will be recognized and the process will switch to complete pivoting (maximum search order $(N-k+1)^2$ in the k-th step). The process will also switch to complete pivoting if the modulus of the pivot obtained with partial pivoting is less than EPS.

The choice GRWLIM$<N^{-1}$ will result in complete pivoting only, while partial pivoting will be used in every step if we choose GRWLIM$>2^{(N-1)} \times N^{-1}$.

Usually, GRWLIM = 8 will give good results (see [4]).

The approximate solution x is found by forward substitution and backward substitution in $LDz = P^{-1}b$, $Uy = z$ and $x = Qy$, where b is the right hand side.

4. References

[1] Businger, P.A., Monitoring the numerical stability of Gaussian elimination. In: Numerische Mathematik 16, 1971, pp. 360-361.

[2] Hoffmann, W., Gaussian elimination algorithms on a vector computer. Report 85-10, University of Amsterdam, Department of Mathematics, 1985.

[3] Numerical Algorithms Group, NAG FORTRAN library manual - mark 11, 1984.

[4] Wilkinson, J.H., Rounding errors in algebraic processing. Prentice Hall, 1963, Chapter 3, Section 16.

[5] Wilkinson, J.H. and Reinsch, C., Handbook for Automatic Computation. Volume II, Linear Algebra. Springer-Verlag, 1971, pp. 93-110.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geq$ N.
Before entry, A should contain the elements of the real matrix.
On successful exit, it will contain the LDU factorization with the diagonals of L and U understood.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geq$ N).
Unchanged on exit.

B - REAL array of DIMENSION at least (N).
Before entry, B should contain the elements of the right hand side.
Unchanged on exit, but see Section 11.

N - INTEGER.
On entry, N specifies the order of matrix A.
Unchanged on exit.

C - REAL array of DIMENSION at least (N).
On successful exit, C will contain the solution vector.

CPIV - INTEGER array of DIMENSION at least (N).
RPIV - INTEGER array of DIMENSION at least (N).
Used as working space.
(On successful exit, CPIV and RPIV will contain the pivotal column indices and the pivotal row indices respectively.)

EPS - REAL.
On entry, EPS must specify a small positive value used as threshold for singularity tests (e.g. the machine precision: X02AAF).
Unchanged on exit.

GRWLIM - REAL.
On entry, GRWLIM must specify a value which is used for controlling the pivoting strategy. Usually GRWLIM = 8 will give good results. See also Section 3.
Unchanged on exit.

MAXNRM - REAL.
On exit, MAXNRM contains the maximum norm $||A||_\infty$ of the matrix argument.

UPBGRW - REAL.
On exit, UPBGRW contains an upper bound for the growth factor. Usually, however, this will be a crude overestimate.

IFAIL - INTEGER.
Before entry, IFAIL must be assigned a value. For users not familiar with this parameter (described in Chapter P01 of [3]) the recommended value is 0.
Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error indicators and warnings

Errors detected by the routine:-

IFAIL = 1
Failure in LURMMP, the matrix A is singular, possibly due to rounding errors.

7. Auxiliary routines

This routine calls the NUMVEC Library routines CPIND, LURMMP, SLDUCP and P01AAF.

8. Timing

The time taken is approximately proportional to N^3.
T(ime) in seconds for various N on a CYBER 205 (1-pipe):

N	25	50	100	200	400
T (partial pivoting throughout)	0.0016	0.0057	0.0243	0.1181	0.6779
T (complete pivoting from the beginning)	0.0031	0.0122	0.0550	0.2171	1.3704

9. Storage

There are no internally declared arrays, but see Section 11.1.

10. Accuracy

The accuracy of the computed solution depends on the conditioning of the original matrix. For a detailed error analysis see [4], page 107.

11. Further comments

If the routine is called with the same name for parameters B and C then the solution vector will overwrite the right hand side.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions. Special call statements are used where appropriate (e.g. the pivot search is done with sign control).
The routine uses at most $2N+\max\{2N, \left\lceil \frac{1}{64}\min\{65535, IA \times N\}\right\rceil\}$ words dynamic storage.

12. Keywords

Approximate Solution of Linear Equations.
LDU Factorization.
Real Matrix.
Single Right Hand Side.

13. Example

To solve the set of linear equations $Ax = b$ where

$$A = \begin{bmatrix} 33 & 16 & 72 \\ -24 & -10 & -57 \\ -8 & -4 & -17 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} -359 \\ 281 \\ 85 \end{bmatrix}$$

13.1. Program text

```
C
C       CCRMCF EXAMPLE PROGRAM TEXT
C       NUMVEC 1986
C       MARK 1
C
        PROGRAMME
        REAL A(4,4), B(6), C(6), WKS1(18), WKS2(18),
     +      EPS, GRWLIM, MAXNRM, UPBGRW
        INTEGER NIN, NOUT, I, N, J, IA, IFAIL
        DATA NIN /5/, NOUT /6/
        OPEN(UNIT=NIN, FILE='INPUT')
        OPEN(UNIT=NOUT, FILE='OUTPUT')
        READ (NIN,99999) (WKS1(I),I=1,7)
        WRITE (NOUT,99997) (WKS1(I),I=1,6)
        N = 3
        READ (NIN,99998) ((A(I,J),J=1,N),I=1,N), (B(I),I=1,N)
        IA = 4
        EPS = X02AAF(XXXX)
        GRWLIM = 8.
        IFAIL = 1
        CALL CCRMCF(A,IA,B,N,C,WKS1,WKS2,EPS,GRWLIM,MAXNRM,UPBGRW,IFAIL)
        WRITE (NOUT,99996) MAXNRM
        IF (IFAIL.NE.0) THEN
          WRITE (NOUT,99995) IFAIL
          STOP
        ENDIF
        WRITE (NOUT,99994) (C(I),I=1,N)
        WRITE (NOUT,99993) UPBGRW
        STOP
99999   FORMAT (6A4, 1A3)
99998   FORMAT (3F5.0)
99997   FORMAT (4(1X/), 1X, 5A4, 1A3, 'RESULTS'/1X)
99996   FORMAT ('0MAXIMUM NORM OF MATRIX: ', F5.0)
99995   FORMAT ('0ERROR IN CCRMCF IFAIL = ', I2)
99994   FORMAT ('0SOLUTIONS'/(1X, F4.1))
99993   FORMAT ('0UPPER BOUND FOR THE GROWTH FACTOR: ', F7.4)
        END
```

13.2. Program data

```
CCRMCF EXAMPLE PROGRAM DATA
   33   16   72
  -24  -10  -57
   -8   -4  -17
 -359  281   85
```

13.3. Program results

```
CCRMCF EXAMPLE PROGRAM RESULTS


MAXIMUM NORM OF MATRIX:   72.

SOLUTIONS
 1.0
-2.0
-5.0

UPPER BOUND FOR THE GROWTH FACTOR:  2.0370
```

CCRPCF - NUMVEC FORTRAN Library Routine Document

1. Purpose

CCRPCF calculates the approximate solution of a set of real linear equations with a single right hand side, $Ax = b$, by an LDU factorization method with partial pivoting (see [1]).

2. Specification

```
      SUBROUTINE CCRPCF(A, IA, B, N, C, PIV, EPS, IFAIL)
C     INTEGER IA, N, PIV(N), IFAIL
C     REAL A(IA,N), B(N), C(N), EPS
```

3. Description

Given a set of linear equations, $Ax = b$, the routine first decomposes A using an LDU factorization with partial pivoting by column interchanges: $AP = LDU$, where P is a permutation matrix, L is lower triangular, D is diagonal and U is upper triangular. The diagonals are normalized according to: $L_{ii} = U_{ii} = D_{ii}^{-1}$. The approximate solution x is found by forward substitution and backward substitution in $LDz = b$, $Uy = z$ and $x = Py$, where b is the right hand side.

4. References

[1] Hoffmann, W., Gaussian elimination algorithms on a vector computer. Report 85-10, University of Amsterdam, Department of Mathematics, 1985.

[2] Numerical Algorithms Group, NAG FORTRAN library manual - mark 11, 1984.

[3] Wilkinson, J.H. and Reinsch, C., Handbook for Automatic Computation. Volume II, Linear Algebra. Springer-Verlag, 1971, pp. 93-110.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A should contain the elements of the real matrix.
On successful exit, it will contain the LDU factorization with the diagonals of L and U understood.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geqslant$ N).
Unchanged on exit.

B - REAL array of DIMENSION at least (N).
Before entry, B should contain the elements of the right hand side.
Unchanged on exit, but see Section 11.

N - INTEGER.
On entry, N specifies the order of matrix A.
Unchanged on exit.

C - REAL array of DIMENSION at least (N).
On successful exit, C will contain the solution vector.

PIV - INTEGER array of DIMENSION at least (N).
Used as working space.
(On successful exit, PIV will contain the pivotal column indices.)

EPS - REAL.
On entry, EPS must specify a small positive value used as threshold for singularity tests (e.g. the machine precision: X02AAF).
Unchanged on exit.

IFAIL - INTEGER.
Before entry, IFAIL must be assigned a value. For users not familiar with this parameter (described in Chapter P01 of [2]) the recommended value is 0.
Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error indicators and warnings

Errors detected by the routine:-

IFAIL = 1
Failure in LURMPC, the matrix A is singular, possibly due to rounding errors.

7. Auxiliary routines

This routine calls the NUMVEC Library routines LURMPC, SLDUPC and P01AAF.

8. Timing

The time taken is approximately proportional to N^3.
T(ime) in seconds for various N on a CYBER 205 (1-pipe):

n	25	50	100	200	400
T	0.0014	0.0051	0.0232	0.1154	0.6705

9. Storage

There are no internally declared arrays, but see Section 11.1.

10. Accuracy

The accuracy of the computed solution depends on the conditioning of the original matrix. For a detailed error analysis see [3], page 107.

11. Further comments

If the routine is called with the same name for parameters B and C then the solution vector will overwrite the right hand side.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions. Special call statements are used where appropriate (e.g. the pivotal column search is done with sign control).
The routine explicitly uses N words dynamic storage.

12. Keywords

Approximate Solution of Linear Equations.
LDU Factorization.
Real Matrix.
Single Right Hand Side.

13. Example

To solve the set of linear equations $Ax = b$ where

$$A = \begin{bmatrix} 33 & 16 & 72 \\ -24 & -10 & -57 \\ -8 & -4 & -17 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} -359 \\ 281 \\ 85 \end{bmatrix}$$

13.1. Program text

```
C
C       CCRPCF EXAMPLE PROGRAM TEXT
C       NUMVEC 1986
C       MARK 1
C
        PROGRAMME
        REAL A(4,4), B(6), C(6), WKS(18), EPS
        INTEGER NIN, NOUT, I, N, J, IA, IFAIL
        DATA NIN /5/, NOUT /6/
        OPEN(UNIT=NIN, FILE='INPUT')
        OPEN(UNIT=NOUT, FILE='OUTPUT')
        READ (NIN,99999) (WKS(I),I=1,7)
        WRITE (NOUT,99997) (WKS(I),I=1,6)
        N = 3
        READ (NIN,99998) ((A(I,J),J=1,N),I=1,N), (B(I),I=1,N)
        IA = 4
        EPS = X02AAF(XXXX)
        IFAIL = 1
        CALL CCRPCF(A, IA, B, N, C, WKS, EPS, IFAIL)
        IF (IFAIL.NE.0) THEN
          WRITE (NOUT,99996) IFAIL
          STOP
        END IF
        WRITE (NOUT,99995) (C(I),I=1,N)
        STOP
99999   FORMAT (6A4, 1A3)
99998   FORMAT (3F5.0)
99997   FORMAT (4(1X/), 1X, 5A4, 1A3, 'RESULTS'/1X)
99996   FORMAT ('0ERROR IN CCRPCF IFAIL = ', I2)
99995   FORMAT ('0SOLUTIONS'/(1X, F4.1))
        END
```

13.2. Program data

```
CCRPCF EXAMPLE PROGRAM DATA
   33   16   72
  -24  -10  -57
   -8   -4  -17
 -359  281   85
```

13.3. Program results

```
CCRPCF EXAMPLE PROGRAM RESULTS


SOLUTIONS
 1.0
-2.0
-5.0
```

CPIND - NUMVEC FORTRAN Library Routine Document

1. Purpose

CPIND searches the element of maximum modulus (complete pivot) in the $(n-j+1) \times (n-j+1)$ submatrix of the $n \times n$ matrix A with the upper left corner of the submatrix corresponding with A_{jj}.

2. Specification

```
      SUBROUTINE CPIND(A, IA, N, J, CIND, RIND, ACR, IFAIL)
C     INTEGER IA, N, J, CIND, RIND, IFAIL
C     REAL A(IA,N), ACR
```

3. Description

CPIND searches the (n-j+1)*(n-j+1) submatrix starting in the (j,j)-th element of array A for the element of maximum modulus (complete pivot). The most efficient of the two following alternatives is chosen depending on an estimate of the timings: the first version works on vectors of maximal length: a bit-vector is used to skip those elements of A that do not belong to the submatrix; the second version first searches every column of the submatrix for the element of maximum modulus, and afterwards searches these elements for their maximum.

4. References

None.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A should contain the elements of the real matrix.
Unchanged on exit.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geqslant$ N).
Unchanged on exit.

N - INTEGER.
On entry, N specifies the order of matrix A.
Unchanged on exit.

J - INTEGER.
On entry, J specifies the upper left corner of the submatrix: A(J,J).
Unchanged on exit.

CIND - INTEGER.
RIND - INTEGER.
On exit, CIND and RIND will contain the column and row indices of the complete pivot element.

ACR - REAL
On exit, ACR will contain the value of the complete pivot element.

IFAIL - INTEGER.
Unused.
IFAIL contains 0 on exit.

6. Error indicators and warnings

None.

7. Auxiliary routines

None.

8. Timing

The time taken is approximately proportional to $(N-J+1)^2$.

9. Storage

There are no internally declared arrays, but see Section 11.1.

10. Accuracy

Not applicable.

11. Further comments

None.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions. Special call statements are used where appropriate.
The routine uses at most $\max\{2(N-J+1), \left\lceil \frac{1}{64}\min\{65535, IA \times (N-J+1)\}\right\rceil\}$ words dynamic storage.

12. Keywords

Real Matrix.
Complete Pivoting.

13. Example

To search the element of maximum modulus in the 2×2 submatrix with the upper left corner corresponding with A_{22}, where

$$A = \begin{pmatrix} 33 & 16 & 72 \\ -24 & -10 & -57 \\ -8 & -4 & -17 \end{pmatrix}$$

13.1. Program text

```
C
C       CPIND  EXAMPLE PROGRAM TEXT
C       NUMVEC 1986
C       MARK 1
C
        PROGRAMME
        REAL A(4,4), WKS1(18), ACR
        INTEGER NIN, NOUT, I, N, J, IA, IFAIL, CIND, RIND
        DATA NIN /5/, NOUT /6/
        OPEN(UNIT=NIN, FILE='INPUT')
        OPEN(UNIT=NOUT, FILE='OUTPUT')
        READ (NIN,99999) (WKS1(I),I=1,7)
        WRITE (NOUT,99997) (WKS1(I),I=1,6)
        N = 3
        READ (NIN,99998) ((A(I,J),J=1,N),I=1,N)
        IA = 4
        J = 2
        IFAIL = 1
        CALL CPIND(A, IA, N, J, CIND, RIND, ACR, IFAIL)
        IF (IFAIL.NE.0) THEN
          WRITE (NOUT,99996) IFAIL
          STOP
        ENDIF
        WRITE (NOUT,99995) CIND, RIND, ACR
        STOP
99999   FORMAT (6A4, 1A3)
99998   FORMAT (3F5.0)
99997   FORMAT (4(1X/), 1X, 5A4, 1A3, 'RESULTS'/1X)
99996   FORMAT ('0ERROR IN CPIND IFAIL = ', I2)
99995   FORMAT ('0A(', I2, ',', I2, ') = ', F5.0)
        END
```

13.2. Program data

```
CPIND  EXAMPLE PROGRAM DATA
   33   16   72
  -24  -10  -57
   -8   -4  -17
```

13.3. Program results

```
CPIND  EXAMPLE PROGRAM RESULTS


A( 2, 3) =  -57.
```

GAUJOR - NUMVEC FORTRAN Library Routine Document

1. Purpose

GAUJOR calculates the approximate solution of a set of real linear equations with multiple right hand sides, $AX = B$, by Gauß-Jordan elimination with partial pivoting (see [1]).

2. Specification

```
      SUBROUTINE GAUJOR(A, IA, B, IB, N, M, C, IC, DIAG, PIV, EPS,IFAIL)
C     INTEGER IA, N, IB, IC, M, PIV(N), IFAIL
C     REAL A(IA,N), B(IB,M), C(IC,M), DIAG(N), EPS
```

3. Description

Given a set of linear equations, $AX = B$, the routine first diagonalizes A using elementary transformations with partial pivoting by rows. The partial pivoting is implemented without actually interchanging the rows.
The columns x of the solution X are found by dividing by the diagonal and permuting the rows of X.

4. References

[1] Hoffmann, W., Solving linear systems on a vector computer, to appear.

[2] Numerical Algorithms Group, NAG FORTRAN library manual - mark 11, 1984.

[3] Wilkinson, J.H. and Reinsch, C., Handbook for Automatic Computation. Volume II, Linear Algebra. Springer-Verlag, 1971, pp. 93-110.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A should contain the elements of the real matrix.
The contents of A are altered on exit.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geqslant$ N).
Unchanged on exit.

B - REAL array of DIMENSION (IB,p) where p $\geqslant$ M.
Before entry, B should contain the elements of the M right hand sides stored in columns.
Unchanged on exit, but see Section 11.

IB - INTEGER.
On entry, IB specifies the first dimension of array B as declared in the calling (sub)program (IB $\geqslant$ N).
Unchanged on exit.

N - INTEGER.
On entry, N specifies the order of matrix A.
Unchanged on exit.

M - INTEGER.
On entry, M specifies the number of right hand sides.
Unchanged on exit.

C - REAL array of DIMENSION (IC,p) where p $\geqslant$ M.
On successful exit, C will contain the M solution vectors.

IC - INTEGER.
On entry, IC specifies the first dimension of array C as declared in the calling (sub)program (IC $\geqslant$ N).
Unchanged on exit.

DIAG - INTEGER array of DIMENSION at least (N).
Used as working space.
(On successful exit, DIAG will contain the non permuted elements of the diagonalized matrix A.)

PIV - INTEGER array of DIMENSION at least (N).
Used as working space.
(On successful exit, PIV will contain the pivotal row indices.)

EPS - REAL.
On entry, EPS must specify a small positive value used as threshold for singularity tests (e.g. the machine precision: X02AAF).
Unchanged on exit.

IFAIL - INTEGER.
Before entry, IFAIL must be assigned a value. For users not familiar with this parameter (described in Chapter P01 of [2]) the recommended value is 0.
Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error indicators and warnings

Errors detected by the routine:-

IFAIL = 1
The matrix A is singular, possibly due to rounding errors.

7. Auxiliary routines

This routine calls the NUMVEC Library routine P01AAF.

8. Timing

The time taken is approximately proportional to N^3.
T(ime) in seconds for various N on a CYBER 205 (1-pipe):

n	25	50	100	200	400
T	0.0012	0.0051	0.0249	0.1369	0.8649

9. Storage

There are no internally declared arrays, but see Section 11.1.

10. Accuracy

The accuracy of the computed solution depends on the conditioning of the original matrix. For a detailed error analysis see [3]. page 107.

11. Further comments

If the routine is called with the same name for parameters B and C then the solution vectors will overwrite the right hand sides.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions. Special call statements are used where appropriate (e.g. the pivotal row search is done with sign control).
The routine explicitly uses N words dynamic storage.

12. Keywords

Approximate Solution of Linear Equations.
Gauß-Jordan elimination.
Multiple Right Hand Sides.
Real Matrix.

13. Example

To solve the set of linear equations $AX = B$ where

$$A = \begin{bmatrix} 33 & 16 & 72 \\ -24 & -10 & -57 \\ -8 & -4 & -17 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} -359 \\ 281 \\ 85 \end{bmatrix}$$

13.1. Program text

```
C
C       GAUJOR EXAMPLE PROGRAM TEXT
C       NUMVEC 1986
C       MARK 1
C
        PROGRAMME
        REAL A(4,4), B(6), C(6), WKS1(18), WKS2(18), EPS
        INTEGER NIN, NOUT, I, N, J, IA, IFAIL
        DATA NIN /5/, NOUT /6/
        OPEN(UNIT=NIN, FILE='INPUT')
        OPEN(UNIT=NOUT, FILE='OUTPUT')
        READ (NIN,99999) (WKS1(I),I=1,7)
        WRITE (NOUT,99997) (WKS1(I),I=1,6)
        N = 3
        M = 1
        READ (NIN,99998) ((A(I,J),J=1,N),I=1,N), (B(I),I=1,N)
        IA = 4
        IB = 6
        IC = 6
```

```
      EPS = X02AAF(XXXX)
      IFAIL = 1
      CALL GAUJOR(A, IA, B, IB, N, M, C, IC, WKS1, WKS2, EPS, IFAIL)
      IF (IFAIL.NE.0) THEN
        WRITE (NOUT,99996) IFAIL
        STOP
      END IF
      WRITE (NOUT,99995) (C(I),I = 1,N)
      STOP
99999 FORMAT (6A4, 1A3)
99998 FORMAT (3F5.0)
99997 FORMAT (4(1X/), 1X, 5A4, 1A3, 'RESULTS'/1X)
99996 FORMAT ('0ERROR IN GAUJOR IFAIL = ', I2)
99995 FORMAT ('0SOLUTIONS'/(1X, F4.1))
      END
```

13.2. Program data

```
GAUJOR EXAMPLE PROGRAM DATA
   33   16   72
  -24  -10  -57
   -8   -4  -17
 -359  281   85
```

13.3. Program results

```
GAUJOR EXAMPLE PROGRAM RESULTS


SOLUTIONS
 1.0
-2.0
-5.0
```

LURMMP - NUMVEC FORTRAN Library Routine Document

1. Purpose

LURMMP decomposes a real matrix into one diagonal and two triangular matrices by an LDU factorization method with mixed pivoting (see [1]).

2. Specification

```
      SUBROUTINE LURMMP(A, IA, N, CPIV, RPIV, EPS, GRWLIM, MAXNRM, UPBGRW, IFAIL)
C     INTEGER IA, N, CPIV(N), RPIV(N), IFAIL
C     REAL A(IA,N), EPS, GRWLIM, MAXNRM, UPBGRW
```

3. Description

The routine decomposes A using an LDU factorization with mixed pivoting: $P^{-1}AQ = LDU$, where P and Q are permutation matrices, L is lower triangular, D is diagonal and U is upper triangular. The diagonals are normalized according to: $L_{ii} = U_{ii} = D_{ii}^{-1}$

Partial pivoting by column interchanges will be used as long as the calculated upper bound for the growth (see [1]) is less than a critical value, that equals GRWLIM $\times$ N $\times$ $||A||_\infty$. In practice, the upper bound for the growth rarely exceeds the critical value if we choose GRWLIM properly (see [1] and [4]), hence we will usually take advantage of the greater speed of partial pivoting (maximum search order N$-k+1$ in the k-th step), while in doubtful cases numerical difficulties will be recognized and the process will switch to complete pivoting (maximum search order (N$-k+1)^2$ in the k-th step). The process will also switch to complete pivoting if the modulus of the pivot obtained with partial pivoting is less than EPS.

The choice GRWLIM$<$N^{-1} will result in complete pivoting only, while partial pivoting will be used in every step if we choose GRWLIM$>2^{(N-1)} \times$ N^{-1}.

Usually, GRWLIM = 8 will give good results (see [4]).

4. References

[1] Businger, P.A., Monitoring the numerical stability of Gaussian elimination. In: Numerische Mathematik 16, 1971, pp. 360-361.

[2] Hoffmann, W., Gaussian elimination algorithms on a vector computer. Report 85-10, University of Amsterdam, Department of Mathematics, 1985.

[3] Numerical Algorithms Group, NAG FORTRAN library manual - mark 11, 1984.

[4] Wilkinson, J.H., Rounding errors in algebraic processing. Prentice Hall, 1963, Chapter 3, Section 16.

[5] Wilkinson, J.H. and Reinsch, C., Handbook for Automatic Computation. Volume II, Linear Algebra. Springer-Verlag, 1971, pp. 93-110.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geq$ N.
Before entry, A should contain the elements of the real matrix.
On successful exit, it will contain the LDU factorization with the diagonals of L and U understood.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geq$ N).
Unchanged on exit.

N - INTEGER.
On entry, N specifies the order of matrix A.
Unchanged on exit.

CPIV - INTEGER array of DIMENSION at least (N).
RPIV - INTEGER array of DIMENSION at least (N).
On successful exit, CPIV and RPIV will contain the pivotal column indices and the pivotal row indices respectively.

EPS - REAL.
On entry, EPS must specify a small positive value used as threshold for singularity tests (e.g. the machine precision: X02AAF).
Unchanged on exit.

GRWLIM - REAL.
On entry, GRWLIM must specify a value which is used for controlling the pivoting strategy. Usually GRWLIM = 8 will give good results. See also Section 3.
Unchanged on exit.

MAXNRM - REAL.
On exit, MAXNRM contains the maximum norm $||A||_\infty$ of the matrix argument.

UPBGRW - REAL.
On exit, UPBGRW contains an upper bound for the growth factor. Usually, however, this will be a crude overestimate.

IFAIL - INTEGER.
Before entry, IFAIL must be assigned a value. For users not familiar with this parameter (described in Chapter P01 of [3]) the recommended value is 0.
Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error indicators and warnings

Errors detected by the routine:-

IFAIL = 1
The matrix A is singular, possibly due to rounding errors.

7. Auxiliary routines

This routine calls the NUMVEC Library routines CPIND and P01AAF.

8. Timing

The time taken is approximately proportional to N^3.

9. Storage

There are no internally declared arrays, but see Section 11.1.

10. Accuracy

The accuracy of the computed solution depends on the conditioning of the original matrix. For a detailed error analysis see [4], page 107.

11. Further comments

None.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions. Special call statements are used where appropriate (e.g. the pivot search is done with sign control).
The routine uses at most $2N+\max\{2N, \left\lceil \frac{1}{64}\min\{65535, IA \times N\} \right\rceil\}$ words dynamic storage.

12. Keywords

LDU Factorization.
Real Matrix.

13. Example

See SLDUCP.

LURMPC - NUMVEC FORTRAN Library Routine Document

1. Purpose

LURMPC decomposes a real matrix into one diagonal and two triangular matrices by an LDU factorization method with partial pivoting (see [1]).

2. Specification

```
      SUBROUTINE LURMPC(A, IA, N, PIV, EPS, IFAIL)
C     INTEGER IA, N, PIV(N), IFAIL
C     REAL A(IA,N), EPS
```

3. Description

The routine decomposes A using an LDU factorization with partial pivoting by column interchanges: $AP = LDU$, where P is a permutation matrix, L is lower triangular, D is diagonal and U is upper triangular. The diagonals are normalized according to: $L_{ii} = U_{ii} = D_{ii}^{-1}$.

4. References

[1] Hoffmann, W., Gaussian elimination algorithms on a vector computer. Report 85-10, University of Amsterdam, Department of Mathematics, 1985.

[2] Numerical Algorithms Group, NAG FORTRAN library manual - mark 11, 1984.

[3] Wilkinson, J.H. and Reinsch, C., Handbook for Automatic Computation. Volume II, Linear Algebra. Springer-Verlag, 1971, pp. 93-110.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A should contain the elements of the real matrix.
On successful exit, it will contain the LDU factorization with the diagonals of L and U understood.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geqslant$ N).
Unchanged on exit.

N - INTEGER.
On entry, N specifies the order of matrix A.
Unchanged on exit.

PIV - INTEGER array of DIMENSION at least (N).
On successful exit, PIV will contain the pivotal column indices.

EPS - REAL.
On entry, EPS must specify a small positive value used as threshold for singularity tests (e.g. the machine precision: X02AAF).
Unchanged on exit.

IFAIL - INTEGER.
Before entry, IFAIL must be assigned a value. For users not familiar with this parameter (described in Chapter P01 of [2]) the recommended value is 0.
Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error indicators and warnings

Errors detected by the routine:-

IFAIL = 1
The matrix A is singular, possibly due to rounding errors.

7. Auxiliary routines

This routine calls the NUMVEC Library routine P01AAF.

8. Timing

The time taken is approximately proportional to N^3.

9. Storage

There are no internally declared arrays, but see Section 11.1.

10. Accuracy

The accuracy of the computed solution depends on the conditioning of the original matrix. For a detailed error analysis see [3], page 107.

11. Further comments

None.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions. Special call statements are used where appropriate (e.g. the pivotal column search is done with sign control).
The routine explicitly uses N words dynamic storage.

12. Keywords

LDU Factorization.
Real Matrix.

13. Example

See SLDUPC.

SLDUCP - NUMVEC FORTRAN Library Routine Document

1. Purpose

SLDUCP calculates the approximate solution of a set of real linear equations with a single right hand side, $Ax = b$, where A has been decomposed using LURMMP (see [2]).

2. Specification

```
      SUBROUTINE SLDUCP(A, IA, B, N, C, CPIV, RPIV, IFAIL)
C     INTEGER IA, N, CPIV(N), RPIV(N), IFAIL
C     REAL A(IA,N), B(N), C(N)
```

3. Description

The routine solves $Ax = b$ where A is real and b is the right hand side. The routine must be preceded by a call to LURMMP which calculates a permutation of the LDU factorization: $P^{-1}AQ = LDU$.
The approximate solution x is found by forward substitution and backward substitution in $LDz = P^{-1}b$, $Uy = z$ and $x = Qy$, where b is the right hand side.

4. References

[1] Businger, P.A., Monitoring the numerical stability of Gaussian elimination. In: Numerische Mathematik 16, 1971, pp. 360-361.

[2] Hoffmann, W., Gaussian elimination algorithms on a vector computer. Report 85-10, University of Amsterdam, Department of Mathematics, 1985.

[3] Numerical Algorithms Group, NAG FORTRAN library manual - mark 11, 1984.

[4] Wilkinson, J.H., Rounding errors in algebraic processing. Prentice Hall, 1963, Chapter 3, Section 16.

[5] Wilkinson, J.H. and Reinsch, C., Handbook for Automatic Computation. Volume II, Linear Algebra. Springer-Verlag, 1971, pp. 93-110.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A must contain the LDU decomposition, as given by LURMMP.
Unchanged on exit.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geqslant$ N).
Unchanged on exit.

B - REAL array of DIMENSION at least (N).
Before entry, B should contain the elements of the right hand side.
Unchanged on exit, but see Section 11.

N - INTEGER.
On entry, N specifies the order of matrix A.
Unchanged on exit.

C - REAL array of DIMENSION at least (N).
On successful exit, C will contain the solution vector.

CPIV - INTEGER array of DIMENSION at least (N).
RPIV - INTEGER array of DIMENSION at least (N).
Before entry CPIV and RPIV must contain the details of the column and row interchanges in the LDU factorization as given by LURMMP.
Unchanged on exit.

IFAIL - INTEGER.
Unused.
IFAIL contains 0 on exit.

6. Error indicators and warnings

None.

7. Auxiliary routines

None.

8. Timing

The time taken is approximately proportional to N^2.

9. Storage

There are no internally declared arrays, but see Section 11.1.

10. Accuracy

The accuracy of the computed solution depends on the conditioning of the original matrix. For a detailed error analysis see [4], page 107.

11. Further comments

If the routine is called with the same name for parameters B and C then the solution vector will overwrite the right hand side.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions.
The routine explicitly uses N words dynamic storage.

12. Keywords

Approximate Solution of Linear Equations.
LDU Factorization.
Real Matrix.
Single Right Hand Side.

13. Example

To solve the set of linear equations $Ax = b$ where

$$A = \begin{pmatrix} 33 & 16 & 72 \\ -24 & -10 & -57 \\ -8 & -4 & -17 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} -359 \\ 281 \\ 85 \end{pmatrix}$$

13.1. Program text

```
C
C       SLDUCP EXAMPLE PROGRAM TEXT
C       NUMVEC 1986
C       MARK 1
C
        PROGRAMME
        REAL A(4,4), B(6), C(6), WKS1(18), WKS2(18),
     +     EPS, GRWLIM, MAXNRM, UPBGRW
        INTEGER NIN, NOUT, I, N, J, IA, IFAIL
        DATA NIN /5/, NOUT /6/
        OPEN(UNIT=NIN, FILE='INPUT')
        OPEN(UNIT=NOUT, FILE='OUTPUT')
        READ (NIN,99999) (WKS1(I),I=1,7)
        WRITE (NOUT,99997) (WKS1(I),I=1,6)
        N = 3
        READ (NIN,99998) ((A(I,J),J=1,N),I=1,N), (B(I),I=1,N)
        IA = 4
        EPS = X02AAF(XXXX)
        GRWLIM = 8.
        IFAIL = 1
        CALL LURMMP(A,IA,N,WKS1,WKS2,EPS,GRWLIM,MAXNRM,UPBGRW,IFAIL)
        WRITE (NOUT,99996) MAXNRM
        IF (IFAIL.NE.0) THEN
          WRITE (NOUT,99995) IFAIL
          STOP
        ENDIF
        CALL SLDUCP(A, IA, B, N, C, WKS1, WKS2, IFAIL)
        WRITE (NOUT,99994) (C(I),I=1,N)
        WRITE (NOUT,99993) UPBGRW
        STOP
99999   FORMAT (6A4, 1A3)
99998   FORMAT (3F5.0)
99997   FORMAT (4(1X/), 1X, 5A4, 1A3, 'RESULTS'/1X)
99996   FORMAT ('0MAXIMUM NORM OF MATRIX: ', F5.0)
99995   FORMAT ('0ERROR IN SLDUCP IFAIL = ', I2)
99994   FORMAT ('0SOLUTIONS'/(1X, F4.1))
99993   FORMAT ('0UPPER BOUND FOR THE GROWTH FACTOR: ', F7.4)
        END
```

13.2. Program data

```
SLDUCP EXAMPLE PROGRAM DATA
   33   16   72
  -24  -10  -57
   -8   -4  -17
 -359  281   85
```

13.3. Program results

```
SLDUCP EXAMPLE PROGRAM RESULTS


MAXIMUM NORM OF MATRIX:   72.

SOLUTIONS
 1.0
-2.0
-5.0

UPPER BOUND FOR THE GROWTH FACTOR:  2.0370
```

SLDUPC - NUMVEC FORTRAN Library Routine Document

1. Purpose

SLDUPC calculates the approximate solution of a set of real linear equations with a single right hand side, $Ax = b$, where A has been decomposed using LURMPC (see [1]).

2. Specification

```
      SUBROUTINE SLDUPC(A, IA, B, N, C, PIV, IFAIL)
C     INTEGER IA, N, PIV(N), IFAIL
C     REAL A(IA,N), B(N), C(N)
```

3. Description

The routine solves $Ax = b$ where A is real and b is the right hand side. The routine must be preceded by a call to LURMPC which calculates a column permutation of the LDU factorization: $AP = LDU$.
The approximate solution x is found by forward substitution and backward substitution in $LDz = b$, $Uy = z$ and $x = Py$, where b is the right hand side.

4. References

[1] Hoffmann, W., Gaussian elimination algorithms on a vector computer. Report 85-10, University of Amsterdam, Department of Mathematics, 1985.

[2] Numerical Algorithms Group, NAG FORTRAN library manual - mark 11, 1984.

[3] Wilkinson, J.H. and Reinsch, C., Handbook for Automatic Computation. Volume II, Linear Algebra. Springer-Verlag, 1971, pp. 93-110.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A must contain the LDU decomposition, as given by LURMPC
Unchanged on exit.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geqslant$ N).
Unchanged on exit.

B - REAL array of DIMENSION at least (N).
Before entry, B should contain the elements of the right hand side.
Unchanged on exit, but see Section 11.

N - INTEGER.
On entry, N specifies the order of matrix A.
Unchanged on exit.

C - REAL array of DIMENSION at least (N).
On successful exit, C will contain the solution vector.

PIV - INTEGER array of DIMENSION at least (N).
Before entry, PIV must contain details of the column interchanges in the LDU factorization as given by LURMPC.
Unchanged on exit.

IFAIL - INTEGER.
Unused.
IFAIL contains 0 on exit.

6. Error indicators and warnings

None.

7. Auxiliary routines

None.

8. Timing

The time taken is approximately proportional to N^2.

9. Storage

There are no internally declared arrays, but see Section 11.1.

10. Accuracy

The accuracy of the computed solution depends on the conditioning of the original matrix. For a detailed error analysis see [3], page 107.

11. Further comments

If the routine is called with the same name for parameters B and C then the solution vector will overwrite the right hand side.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions.
The routine explicitly uses N words dynamic storage.

12. Keywords

Approximate Solution of Linear Equations.
LDU Factorization.
Real Matrix.
Single Right Hand Side.

13. Example

To solve the set of linear equations $Ax = b$ where

$$A = \begin{bmatrix} 33 & 16 & 72 \\ -24 & -10 & -57 \\ -8 & -4 & -17 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} -359 \\ 281 \\ 85 \end{bmatrix}$$

13.1. Program text

```
C
C       SLDUPC EXAMPLE PROGRAM TEXT
C       NUMVEC 1986
C       MARK 1
C
        PROGRAMME
        REAL A(4,4), B(6), C(6), WKS(18), EPS
        INTEGER NIN, NOUT, I, N, J, IA, IFAIL
        DATA NIN /5/, NOUT /6/
        OPEN(UNIT=NIN, FILE='INPUT')
        OPEN(UNIT=NOUT, FILE='OUTPUT')
        READ (NIN,99999) (WKS(I),I=1,7)
        WRITE (NOUT,99997) (WKS(I),I=1,6)
        N = 3
        READ (NIN,99998) ((A(I,J),J=1,N),I=1,N), (B(I),I=1,N)
        IA = 4
        EPS = X02AAF(XXXX)
        IFAIL = 1
        CALL LURMPC(A, IA, N, WKS, EPS, IFAIL)
        IF (IFAIL.NE.0) THEN
          WRITE (NOUT,99996) IFAIL
          STOP
        END IF
        CALL SLDUPC(A, IA, B, N, C, WKS, IFAIL)
        WRITE (NOUT,99995) (C(I),I=1,N)
        STOP
99999   FORMAT (6A4, 1A3)
99998   FORMAT (3F5.0)
99997   FORMAT (4(1X/), 1X, 5A4, 1A3, 'RESULTS'/1X)
99996   FORMAT ('0ERROR IN SLDUPC IFAIL = ', I2)
99995   FORMAT ('0SOLUTIONS'/(1X, F4.1))
        END
```

13.2. Program data

```
SLDUPC EXAMPLE PROGRAM DATA
   33   16   72
  -24  -10  -57
   -8   -4  -17
 -359  281   85
```

13.3. Program results

```
SLDUPC EXAMPLE PROGRAM RESULTS


SOLUTIONS
 1.0
-2.0
-5.0
```

CHAPTER VIII

NUMVEC FORTRAN Library manual
Chapter: Simultaneous Linear Equations

reprint of NM-R8712
CWI, Amsterdam, 1987

NUMVEC FORTRAN Library manual

Chapter: Simultaneous Linear Equations

Update #1

W. Hoffmann

Department of Mathematics, University of Amsterdam
Roetersstraat 15, 1018 WB Amsterdam, The Netherlands

This document describes two NUMVEC FORTRAN Library routines, INVGJ and GJPCF.
INVGJ calculates the approximate inverse of a real square matrix by Gauß-Jordan elimination with partial pivoting using column interchanges.
GJPCF calculates the approximate solution of a set of real linear equations with multiple right hand sides, $AX = B$, by Gauß-Jordan elimination with partial pivoting using column interchanges. GJPCF replaces routine GAUJOR, documented in Report NM-R8614, May 1986.

1980 Mathematics subject classification (1985 revision): 65V05, 65F05, 15A06
1982 CR Categories: 5.14.

Key Words & Phrases: Gauß-Jordan elimination, linear equations, software.

Note: The implementations are available in FORTRAN 200 (the CYBER 200 series FORTRAN, a superset of standard FORTRAN including vector extensions).

Report NM-R8712
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

GJPCF - NUMVEC FORTRAN Library Routine Document

1. Purpose

GJPCF calculates the approximate solution of a set of real linear equations with multiple right hand sides, $AX = B$, by Gauß-Jordan elimination with partial pivoting using column interchanges (see [1]).

2. Specification

```
      SUBROUTINE GJPCF(A, IA, B, IB, N, M, C, IC, DIAG, PIV, EPS,IFAIL)
C     INTEGER IA, N, IB, IC, M, PIV(N), IFAIL
C     REAL A(IA,N), B(IB,M), C(IC,M), DIAG(N), EPS
```

3. Description

Given a set of linear equations, $AX = B$, the routine firstly diagonalizes A using elementary transformations with partial pivoting by column-interchanges. The columns of X are found by dividing by the diagonal and permuting the rows of X.
The columns x of the solution X are found by dividing by the diagonal and permuting the rows of X.

4. References

[1] Dekker, T.J. and Hoffmann, W., Rehabilitation of the Gauss-Jordan algorithm; Report 86-28, Math Inst., Univ of Amsterdam, 1986.

[2] Numerical Algorithms Group, NAG FORTRAN library manual - mark 11, 1984.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A should contain the elements of the real matrix.
The contents of A are altered on exit.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geqslant$ N).
Unchanged on exit.

B - REAL array of DIMENSION (IB,p) where p $\geqslant$ M.
Before entry, B should contain the elements of the M right hand sides stored in columns.
Unchanged on exit, but see Section 11.

IB - INTEGER.
On entry, IB specifies the first dimension of array B as declared in the calling (sub)program (IB $\geqslant$ N).
Unchanged on exit.

N - INTEGER.
On entry, N specifies the order of matrix A.
Unchanged on exit.

M - INTEGER.
On entry, M specifies the number of right hand sides.
Unchanged on exit.

C - REAL array of DIMENSION (IC,p) where p $\geqslant$ M.
On successful exit, C will contain the M solution vectors.

IC - INTEGER.
On entry, IC specifies the first dimension of array C as declared in the calling (sub)program (IC $\geqslant$ N).
Unchanged on exit.

DIAG - INTEGER array of DIMENSION at least (N).
Used as working space.
(On successful exit, DIAG will contain the permuted elements of the diagonalized matrix A.)

PIV - INTEGER array of DIMENSION at least (N).
Used as working space.
(On successful exit, PIV will contain the pivotal column indices.)

EPS - REAL.
On entry, EPS must specify a small positive value used as threshold for singularity tests (e.g. the machine precision (X02AAF) times a quantity that reflects the order of magnitude of the matrix, like a norm of the matrix).
Unchanged on exit.

IFAIL - INTEGER.
Before entry, IFAIL must be assigned a value. For users not familiar with this parameter (described in Chapter P01 of [2]) the recommended value is 0.
Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error indicators and warnings

Errors detected by the routine:-

IFAIL = k
The matrix A is singular, possibly due to rounding errors; in the k-th stage of the elimination process the pivot was less than EPS.

7. Auxiliary routines

This routine calls the NUMVEC Library routine P01AAF.

8. Timing

The time taken is approximately proportional to N^3.
T(ime) in seconds for various N on a CYBER 205 (1-pipe):

N	25	50	100	200	400
T	0.0013	0.0053	0.0252	0.1394	0.8739

9. Storage

There are no internally declared arrays, but see Section 11.1.

10. Accuracy

The accuracy of the computed solution depends on the conditioning of the original matrix. For a detailed error analysis see [1].

11. Further comments

If the routine is called with the same name for parameters B and C then the solution vectors will overwrite the right hand sides.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions. Special call statements are used where appropriate
The routine explicitly uses N words dynamic storage.

12. Keywords

Approximate Solution of Linear Equations.
Gauß-Jordan elimination.
Multiple Right Hand Sides.
Real Matrix.

13. Example

To solve the set of linear equations $AX = B$ where

$$A = \begin{bmatrix} 33 & 16 & 72 \\ -24 & -10 & -57 \\ -8 & -4 & -17 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} -359 \\ 281 \\ 85 \end{bmatrix}$$

13.1. Program text

```
C
C       GJPCF EXAMPLE PROGRAM TEXT
C       NUMVEC 1987
C       MARK 1
C
        PROGRAMME
        REAL A(4,4), B(6), C(6), WKS1(18), WKS2(18), EPS
        INTEGER NIN, NOUT, I, N, J, IA, IFAIL
        DATA NIN /5/, NOUT /6/
        OPEN(UNIT=NIN, FILE='INPUT')
        OPEN(UNIT=NOUT, FILE='OUTPUT')
        READ (NIN,99999) (WKS1(I),I=1,7)
        WRITE (NOUT,99997) (WKS1(I),I=1,6)
        N = 3
        M = 1
        READ (NIN,99998) ((A(I,J),J=1,N),I=1,N), (B(I),I=1,N)
        IA = 4
        IB = 6
        IC = 6
        EPS = X02AAF(XXXX)
```

```
      IFAIL = 1
      CALL GJPCF(A, IA, B, IB, N, M, C, IC, WKS1, WKS2, EPS, IFAIL)
      IF (IFAIL.NE.0) THEN
        WRITE (NOUT,99996) IFAIL
        STOP
      END IF
      WRITE (NOUT,99995) (C(I),I = 1,N)
      STOP
99999 FORMAT (6A4, 1A3)
99998 FORMAT (3F5.0)
99997 FORMAT (4(1X/), 1X, 5A4, 1A3, 'RESULTS'/1X)
99996 FORMAT ('0ERROR IN GJPCF IFAIL = ', I2)
99995 FORMAT ('0SOLUTIONS'/(1X, F4.1))
      END
```

13.2. Program data

```
GJPCF EXAMPLE PROGRAM DATA
   33   16   72
  -24  -10  -57
   -8   -4  -17
 -359  281   85
```

13.3. Program results

```
GJPCF EXAMPLE PROGRAM RESULTS


SOLUTIONS
 1.0
-2.0
-5.0
```

INVGJ - NUMVEC FORTRAN Library Routine Document

1. Purpose

INVGJ calculates the approximate inverse of a square real matrix 'in situ' by Gauß-Jordan elimination with partial pivoting using column interchanges (see [1]).

2. Specification

```
      SUBROUTINE INVGJ(A, IA, N, DIAG, PIV, EPS,IFAIL)
C     INTEGER IA, N, PIV(N), IFAIL
C     REAL A(IA,N), DIAG(N), EPS
```

3. Description

The inverse of the given real matrix A is constructed by accumulating the Gauß-Jordan transformations that diagonalize A. Partial pivoting by column interchanges is used throughout.
The result is divided by the calculated diagonal matrix and by the inverse of the permutation matrix, that corresponds with the column interchanges.

4. References

[1] Dekker, T.J. and Hoffmann, W., Rehabilitation of the Gauss-Jordan algorithm; Report 86-28, Math Inst., Univ of Amsterdam, 1986.

[2] Numerical Algorithms Group, NAG FORTRAN library manual - mark 11, 1984.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A should contain the elements of the real matrix.
On successful exit, A contains the inverse matrix.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geqslant$ N).
Unchanged on exit.

N - INTEGER.
On entry, N specifies the order of matrix A.
Unchanged on exit.

DIAG - INTEGER array of DIMENSION at least (N).
Used as working space.
(On successful exit, DIAG will contain the permu.ed elements of the diagonalized matrix A.)

PIV - INTEGER array of DIMENSION at least (N).
Used as working space.
(On successful exit, PIV will contain the pivotal column indices.)

EPS - REAL.
On entry, EPS must specify a small positive value used as threshold for singularity tests (e.g. the machine precision (X02AAF) times a quantity that reflects the order of magnitude of the matrix, like a norm of the matrix).

Unchanged on exit.

IFAIL - INTEGER.
Before entry, IFAIL must be assigned a value. For users not familiar with this parameter (described in Chapter P01 of [2]) the recommended value is 0.
Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error indicators and warnings

Errors detected by the routine:-

IFAIL = k
The matrix A is singular, possibly due to rounding errors; in the k-th stage of the elimination process the pivot was less than EPS.

7. Auxiliary routines

This routine calls the NUMVEC Library routine P01AAF.

8. Timing

The time taken is approximately proportional to N^3.
T(ime) in seconds for various N on a CYBER 205 (1-pipe):

N	25	50	100	200	400
T	0.0023	0.0098	0.0484	0.2719	1.7230

9. Storage

There are no internally declared arrays, but see Section 11.1.

10. Accuracy

The accuracy of the computed solution depends on the conditioning of the original matrix. For a detailed error analysis see [1].

11. Further comments

None.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions. Special call statements are used where appropriate.
The routine explicitly uses N words dynamic storage.

12. Keywords

Inversion
Gauß-Jordan elimination.
Real Matrix.

13. Example

To find the inverse of the 3 × 3 matrix:

$$\begin{bmatrix} 33 & 16 & 72 \\ -24 & -10 & -57 \\ -8 & -4 & -17 \end{bmatrix}$$

13.1. Program text

```
C
C       INVGJ EXAMPLE PROGRAM TEXT
C       NUMVEC 1987
C       MARK 1
C
        PROGRAMME
        REAL A(5,5), WKS1(7), WKS2(7), EPS
        INTEGER NIN, NOUT, I, N, J, IA, IFAIL
        DATA NIN /5/, NOUT/6/
        OPEN(UNIT=NIN, FILE='INPUT')
        OPEN(UNIT=NOUT, FILE='OUTPUT')
        READ (NIN,99999) (WKS1(I),I=1,7)
        WRITE (NOUT,99997) (WKS1(I),I=1,6)
        N = 3
        READ (NIN,99998) ((A(I,J),J=1,N),I=1,N)
        IA = 5
        EPS = X02AAF(XXXX)
        IFAIL = 1
        CALL INVGJ (A, IA, N, WKS1, WKS2, EPS, IFAIL)
        IF (IFAIL.NE.0) THEN
          WRITE (NOUT,99996) IFAIL
          STOP
        END IF
        WRITE (NOUT,99995) ((A(I,J),J=1,N),I=1,N)
        STOP
99999   FORMAT (6A4, 1A3)
99998   FORMAT (3F5.0)
99997   FORMAT (4(1X/), 1X, 5A4, 1A3, 'RESULTS'/1X)
99996   FORMAT ('0ERROR IN INVGJ IFAIL = ', I2)
99995   FORMAT ('0INVERSE'/(1X, 3F10.4))
        END
```

13.2. Program data

```
INVGJ  EXAMPLE PROGRAM DATA
   33   16   72
  -24  -10  -57
   -8   -4  -17
```

13.3. Program results

```
INVGJ  EXAMPLE PROGRAM RESULTS


INVERSE
   -9.6667   -2.6667  -32.0000
    8.0000    2.5000   25.5000
    2.6667    0.6667    9.0000
```

CHAPTER IX

NUMVEC FORTRAN Library manual
Chapter: Simultaneous Linear Equations

jointly written with: K. Potma
reprint of NM-R8903
CWI, Amsterdam, 1989

NUMVEC FORTRAN Library manual

Chapter: Simultaneous Linear Equations

W. Hoffmann, K. Potma,
Department of Computer Systems
Faculty of Mathematics and Computer Science
University of Amsterdam
Kruislaan 409, 1098 SJ Amsterdam, The Netherlands

This document describes a set of NUMVEC FORTRAN Library routines, dealing with the full-rank linear least-squares problem and the orthogonal basis problem. In particular, it contains a subroutine for calculating the factors of a QR decomposition with a compactly stored orthogonal factor. This storage scheme is customary when using Householder reflections. Moreover, it contains subroutines for calculating the product of such a coded orthogonal matrix (or its transposed) with a vector and a subroutine for calculating the explicit form of this orthogonal matrix.

1980 Mathematics subject classification (1985 revision): 65F20, 65V05, 15A23.
1982 CR Categories: 5.14.

Keywords & Phrases: least squares problems, orthogonal basis problem, QR Decomposition, Householder reflections.

Note: The implementations are available in FORTRAN 200 (the CYBER 200 series FORTRAN, a superset of standard FORTRAN including vector extensions).

Report NM-R8903
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

EXPLQ - NUMVEC FORTRAN Library Routine Document

1. Purpose

EXPLQ calculates the $m \times n$ matrix Q from the QR factorization as calculated by HSHVOX, in its explicit form.

2. Specifications

```
      SUBROUTINE EXPLQ(A, Q, IA, M, N, W)
C     INTEGER IA, M, N
C     REAL A(IA, N), Q(IA, N), W(N)
```

3. Description

The sequence of Householder reflections that defines matrix Q is applied, in reverse order, to the identity matrix of appropriate size to form matrix Q.

4. References

None.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A should contain in its lower trapezoidal part the Householder vectors as calculated by HSHVOX.
Unchanged on exit.

Q - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
On exit Q contains the $m \times n$ orthogonal matrix defining the QR factorization of A as calculated by HSHVOX.

IA - INTEGER.
On entry, IA specifies the first dimension of arrays A and Q as declared in the calling (sub)program (IA $\geqslant$ M).
Unchanged on exit.

M - INTEGER.
On entry, M specifies the number of rows of matrices A and Q.
Unchanged on exit.

N - INTEGER.
On entry, N specifies the number of columns of matrices A and Q.
Unchanged on exit.

W - REAL array of DIMENSION at least (M).
Used as work space.

6. Error indicators and warnings

None.

7. Auxiliary routines

No auxiliary routines are used.

8. Timing

The time taken is proportional to MN^2 and is approximately equal to the time of HSHVOX for the same sizes of M and N.

9. Storage

There are no internally declared arrays.

10. Accuracy

The measure of orthogonality, $\|Q^TQ - I\|_2$, is roughly equal to machine-precision.

11. Further comments

None.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions.

12. Keywords

Orthogonal basis.
QR factorization.

13. Example

To calculate an orthogonal basis for the columnspace of matrix A where

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 1 & 0 & -1 \\ 1 & 2 & 3 \\ 1 & 0 & 1 \end{pmatrix}$$

13.1. Program text

```
C
C       ORTBAS EXAMPLE PROGRAM TEXT
C       NUMVEC 1988
C       MARK 1
C
        PROGRAM ORTBAS(OUTPUT,TAPE6=OUTPUT)
        REAL A(4,3), Q(4,3), DIAGR(3), W1(3), W2(4)
        INTEGER M, N, I, J, NZER, NMAX
        DATA ((A(I,J),J=1,3),I=1,4)
     + / 1., 2., 1.,
     +   1., 0.,-1.,
     +   1., 2., 3.,
```

```
     +    1., 0., 1. /
          M = 4
          N = 3
          NMAX = 4
          WRITE(6,99996)
          DO 40 I = 1, M
            WRITE(6,99997)(A(I,J),J = 1, 3)
       40 CONTINUE
          CALL HSHVOX(A, NMAX, M, N, DIAGR, W2, 0, NZER)
          CALL EXPLQ(A, Q, NMAX, M, N, W1)
          WRITE(6,99998)
          DO 50 I = 1, M
            WRITE(6,99999)(Q(I,J),J = 1, 3)
       50 CONTINUE
          STOP
    99996 FORMAT ('1ORTHOGONAL BASIS PROGRAM'/'0MATRIX A')
    99997 FORMAT (3(1X,F7.3))
    99998 FORMAT ('0CALCULATED BASIS: ')
    99999 FORMAT (3(1X,F7.3))
          END
```

13.2. Program results

```
ORTHOGONAL BASIS PROGRAM

MATRIX A
  1.000   2.000   1.000
  1.000   0.000  -1.000
  1.000   2.000   3.000
  1.000   0.000   1.000

CALCULATED BASIS:
  0.500   0.500   0.500
  0.500  -0.500   0.500
  0.500   0.500  -0.500
  0.500  -0.500  -0.500
```

HSHVOX - NUMVEC FORTRAN Library Routine Document

1. Purpose

HSHVOX calculates a QR-factorization of a matrix by means of Householder-reflections.

2. Specifications

```
      SUBROUTINE HSHVOX(A, IA, M, N, DIAGR, W, IP, NZER)
C     INTEGER  IA, M, N, IP, NZER
C     REAL  A(IA, N), DIAGR(N), W(M)
```

3. Description

Given an $m \times n$ matrix A, $m \geqslant n$, a factorization $A = QR$ is calculated where Q is $m \times n$ orthogonal and R is $n \times n$ upper triangular. The routine uses Householder's method with optional scaling of the columnvectors for protection against overflow.
Matrix Q is delivered in factorized form, each factor being defined by the appropriate 'Householder vector'. If the diagonal of R contains p (say) entries equal to zero, then the rank of matrix A is at most $n-p$.

4. References

[1] Golub, G.H., Van Loan, C.F., Matrix Computations, North Oxford Academic, Oxford, 1983.

[2] Hoffmann, W., Definition and use of Householder reflections, Report CS-88-05, University of Amsterdam, Department of Computer Systems, 1988.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A should contain the elements of the real matrix.
On successful exit, it will contain, in its lower trapezoidal part, the Householder vectors defining the reflections applied and, in its strictly upper triangular part the elements of the strict upper triangle of the calculated matrix R.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geqslant$ M).
Unchanged on exit.

M - INTEGER.
On entry, M specifies the number of rows of matrix A.
Unchanged on exit.

N - INTEGER.
On entry, N specifies the number of columns of matrix A.
Unchanged on exit.

DIAGR - REAL array of DIMENSION at least (N).
On successful exit DIAGR will contain the diagonal of R.

W - REAL array of DIMENSION at least (M).
Used as working space.

IP - INTEGER value.
If IP has the value -1, then Euclidean norms are calculated without overflow protection. This causes the subroutine to be less robust, but may save some processing-time.
If IP has a value different from -1, then Euclidean norms are calculated with overflow protection by appropriate intermediate scaling.

NZER - INTEGER.
If IP is different from -1, then on output NZER is the number of zeroes in DIAGR; if IP equals -1 then NZER becomes zero.

6. Error indicators and warnings

If the routine is used in its robust version, i.e. IP $\neq -1$, then the number of zeroes on R's diagonal is calculated. If this number is greater than 0 then matrix R is singular. It should be stressed that the reverse is not true; for example, a matrix R with exclusively ones on the diagonal and a small norm may be close to a singular matrix.

7. Auxiliary routines

No auxiliary routines are used.

8. Timing

The time taken is approximately proportional to MN^2.
Time in seconds for various M and N on a CYBER 205 (2-pipe) with IP $= -1$:

	N = 25	N = 50	N = 100	N = 200
M = 50	0.0019	0.0070	*	*
M = 100	0.0024	0.0089	0.0321	*
M = 200	0.0034	0.0129	0.0478	0.1684

Time in seconds for various M and N on a CYBER 205 (2-pipe) with IP $\neq -1$:

	N = 25	N = 50	N = 100	N = 200
M = 50	0.0021	0.0073	*	*
M = 100	0.0027	0.0093	0.0328	*
M = 200	0.0038	0.0135	0.0490	0.1703

9. Storage

There are no internally declared arrays.

10. Accuracy

The measurement for orthogonality of Q, $\|Q^TQ-I\|_2$, is small within working precision and the residual $\|A-QR\|_2 / \|A\|_2$ is equally small.

11. Further comments

None.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions.

12. Keywords

Householder reflection.
QR factorization.
Overdetermined systems.

13. Example

See EXPLQ.

LINLSQ - NUMVEC FORTRAN Library Routine Document

1. Purpose

LINLSQ calculates the least-squares solution of a full-rank overdetermined linear system by means of a QR-factorization and the solution of a triangular system.

2. Specifications

```
      SUBROUTINE  LINLSQ(A, IA, M, N, B, X, W, RES, NZER)
C     INTEGER  IA, M, N, NZER
C     REAL  A(IA, N), B(M), X(M), W(M+N), RES
```

3. Description

Given a linear system with not more unknowns than equations, an attempt is made to calculate its least-squares solution. First the QR factorization of the coefficient matrix A is calculated using Householder reflections.
If none of the diagonal elements of R is zero, then the unique solution is calculated by means of back substitution with the triangular matrix R.
Moreover, the norm of the residual vector is calculated. For a well-posed problem, this norm should be considerably less than the norm of the original right-hand side.
If one or more diagonal elements of R are zero, then no solution is delivered and the number of zeroes is reported; in that case the user is advised to calculate a minimal-norm solution by means of singular value decomposition as is performed by NUMVEC routine LSQMNS; this chapter.

4. References

[1] Golub, G.H., Van Loan, C.F., Matrix Computations, North Oxford Academic, Oxford, 1983.

[2] Hoffmann, W., Definition and use of Householder reflections, Report CS-88-05, University of Amsterdam, Department of Computer Systems, 1988.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geq$ N.
Before entry, A should contain the elements of the real matrix.
On exit, it will contain information for the QR factorization as calculated by HSHVOX.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geq$ M).
Unchanged on exit.

M - INTEGER.
On entry, M specifies the number of rows of matrix A.
Unchanged on exit.

N - INTEGER.
On entry, N specifies the number of columns of matrix A.
Unchanged on exit.

B - REAL array of DIMENSION at least (M).
Before entry, B should contain the elements of the right hand side.

Unchanged on exit, but see section 11.

X - REAL array of DIMENSION at least (M).
On exit it will contain the solution vector X in its first N elements.

W - REAL array of DIMENSION at least (M+N).
Used as working space.

RES - REAL.
On exit, RES contains the norm of the residual: $\|B-AX\|_2$.

NZER - INTEGER variable.
On output NZER contains the number of zero diagonal elements that has been detected during the *QR* factorization.

6. Error indicators and warnings

If NZER is larger then zero, then no solution is calculated; the rank of the matrix is less than or equal to N−NZER. The use of LSQMNS is advised.

7. Auxiliary routines

This routine uses the NUMVEC Library routines HSHVOX, MULQTX, and LSSOLU.

8. Timing

The time taken is approximately proportional to MN^2.
Time in seconds for various M and N on a CYBER 205 (2-pipe):

	N = 25	N = 50	N = 100	N = 200
M = 50	0.0024	0.0078	*	*
M = 100	0.0029	0.0099	0.0340	*
M = 200	0.0041	0.0142	0.0505	0.1735

9. Storage

There are no internally declared arrays.

10. Accuracy

The accuracy of the computed solution depends on the condition of the matrix and on the angle between the right hand side vector and the columnspace of the matrix.

11. Further comments

If the routine is called with the same name for parameters B and X then the solution vector will overwrite the right hand side vector.

11.1. Vectorization information

The routine uses routines which are written in FORTRAN 200, making use of its vector syntax extensions.

12. Keywords

Linear least-squares solution.
Householder reflection.
QR factorization.
Overdetermined systems.

13. Example

To solve the linear least-squares problem for $Ax \approx b$ where

$$A = \begin{bmatrix} -2 & 0 & 0 & 0 \\ 1 & -2 & 0 & 0 \\ 0 & 1 & -2 & 0 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} 2.01 \\ 3.02 \\ 4.04 \\ 5.08 \\ -3.84 \end{bmatrix}$$

13.1. Program text

```
C
C       LSTSQR EXAMPLE PROGRAM TEXT
C       NUMVEC 1988
C       MARK 1
C
        PROGRAM LSTSQR(OUTPUT,TAPE6=OUTPUT)
        REAL A(5,4), A1(5,4), B(5), X(5), W1(10), W3(5), EPS, RES, SOM
        INTEGER M, N, I, J, NZER, NMAX
        DATA ((A1(I,J),J=1,4),I=1,5),(B(I),I=1,5)
     + / -2., 0., 0., 0.,
     +    1.,-2., 0., 0.,
     +    0., 1.,-2., 0.,
     +    0., 0., 1.,-2.,
     +    0., 0., 0., 1.,
     +    2.01, 3.02, 4.04, 5.08,-3.84 /
        M = 5
        N = 4
        NMAX = 5
        DO 20 J = 1, N
          DO 19 I = 1, M
            A(I,J) = A1(I,J)
   19     CONTINUE
   20   CONTINUE
        WRITE(6,99994)
        DO 40 I = 1, M
          WRITE(6,99995)(A(I,J),J = 1, 4)
   40   CONTINUE
        WRITE(6,99996)(B(I),I = 1, 5)
        CALL LINLSQ(A, NMAX, M, N, B, X, W1, RES, NZER)
        WRITE(6,99997)(X(I),I = 1, 4)
        WRITE(6,99998) RES
        DO 45 I = 1, M
```

```
      W3(I) = B(I)
   45 CONTINUE
      DO 50 J = 1, N
        DO 49 I = 1, M
          W3(I) = W3(I) - A1(I,J) * X(J)
   49   CONTINUE
   50 CONTINUE
      CALL MULQTX(A, NMAX, M, N, W3, W3)
      SOM = 0.
      DO 60 I = 1, N
        SOM = SOM + W3(I) * W3(I)
   60 CONTINUE
      F1 = SQRT(SOM)
      WRITE(6,99999) F1
  997 CONTINUE
      STOP
99994 FORMAT ('1LSTSQR EXAMPLE PROGRAM RESULTS'/'0MATRIX A')
99995 FORMAT (4(1X,F6.2))
99996 FORMAT ('0RIGHT-HANDSIDE VECTOR:'/(1X,F6.2))
99997 FORMAT ('0SOLUTION VECTOR:'/(1X,F6.2))
99998 FORMAT ('0RESIDUAL NORM OF (B - AX) = ',1PE10.3)
99999 FORMAT ('0INNER PRODUCT Q**T(B-AX)  = ',1PE10.3)
      END
```

13.2. Program results

```
LSTSQR EXAMPLE PROGRAM RESULTS

MATRIX A
 -2.00   0.00   0.00   0.00
  1.00  -2.00   0.00   0.00
  0.00   1.00  -2.00   0.00
  0.00   0.00   1.00  -2.00
  0.00   0.00   0.00   1.00

RIGHT-HANDSIDE VECTOR:
  2.01
  3.02
  4.04
  5.08
 -3.84

SOLUTION VECTOR:
 -1.00
 -2.00
 -3.00
 -4.00

RESIDUAL NORM OF (B - AX) =  1.847E-01

INNER PRODUCT Q**T(B-AX)  =  3.047E-13
```

LSSOLU - NUMVEC FORTRAN Library Routine Document

1. Purpose

LSSOLU calculates the solution of a triangular system where the data is delivered as in routines HSHVOX and MULQTX respectively.

2. Specifications

```
      SUBROUTINE LSSOLU(A, IA, N, DIAG, C, X)
C     INTEGER IA, N
C     REAL A(IA, N), DIAG(N), C(N), X(N)
```

3. Description

The solution of a linear system $Rx = c$ with an upper triangular coefficient matrix R and right hand side vector c is solved for x. The strictly upper triangular part of R is given in the corresponding part of A and the diagonal of R is given in DIAG.

4. References

None.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A should contain in its strictly upper triangle the corresponding elements of matrix R.
Unchanged on exit.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geqslant$ N).
Unchanged on exit.

N - INTEGER.
On entry, N specifies the number of columns of matrix A.
Unchanged on exit.

DIAG - REAL array of DIMENSION at least (N).
Before entry, DIAG should contain the diagonal elements of matrix R.
Unchanged on exit.

C - REAL array of DIMENSION at least (N).
Before entry, C should contain the right hand side vector.
Unchanged on exit, but see section 11.

X - REAL array of DIMENSION at least (N).
On exit, X contains the solution of this system.

6. Error indicators and warnings

None.

7. Auxiliary routines

No auxiliary routines are used.

8. Timing

The time taken is approximately proportional to N^2.

9. Storage

There are no internally declared arrays.

10. Accuracy

The accuracy of the solution depends on the condition of matrix *R*.

11. Further comments

If the routine is called with the same name for parameters B and X then the solution vector will overwrite the right hand side vector.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions.

12. Keywords

Triangular system.

MULQTX - NUMVEC FORTRAN Library Routine Document

1. Purpose

MULQTX calculates the product $Q^T x$ for a given m-vector x where Q is defined by a sequence of Householder reflections, each one defined by an appropriate Householder vector as calculated by HSHVOX.

2. Specifications

```
      SUBROUTINE MULQTX(A, IA, M, N, VECIN, VECOUT)
C     INTEGER IA, M, N
C     REAL A(IA, N), VECIN(M), VECOUT(M)
```

3. Description

The sequence of Householder matrices which are defined by the columns of the lower trapezoidal part of matrix A are applied to and accumulated in vector x.

4. References

None.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A should contain in its lower trapezoidal part the Householder vectors as calculated by HSHVOX.
Unchanged on exit.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geqslant$ M).
Unchanged on exit.

M - INTEGER.
On entry, M specifies the number of rows of matrix A.
Unchanged on exit.

N - INTEGER.
On entry, N specifies the number of columns of matrix A.
Unchanged on exit.

VECIN - REAL array of DIMENSION at least (M).
On entry VECIN should contain the given M-vector which is to be multiplied by Q transposed.
Unchanged on exit, but see section 11.

VECOUT - REAL array of DIMENSION at least (M).
On exit, VECOUT contains in its first N elements the result of the matrix-vector multiplication; the remaining M$-$N elements are used for working space.

6. Error indicators and warnings

None.

7. Auxiliary routines

No auxiliary routines are used.

8. Timing

The time taken is approximately proportional to MN.

9. Storage

There are no internally declared arrays.

10. Accuracy

The accuracy is up to working precision.

11. Further comments

If the routine is called with the same name for parameters VECIN and VECOUT then the output vector will overwrite the input vector.

11.1. Vectorization information

The routine is written in FORTRAN 200, making use of its vector syntax extensions.

12. Keywords

Householder matrices.

13. Example

See LINLSQ.

MULQX - NUMVEC FORTRAN Library Routine Document

1. Purpose

MULQX calculates the product Qx for a given n-vector x where Q is defined by a sequence of Householder reflections, each one defined by an appropriate Householder vector as calculated by HSHVOX.

2. Specifications

```
      SUBROUTINE MULQX(A, IA, M, N, VECIN, VECOUT)
C     INTEGER IA, M, N
C     REAL A(IA, N), VECIN(M), VECOUT(M)
```

3. Description

The sequence of Householder matrices which are defined by the columns of the lower trapezoidal part of matrix A are backward applied to and accumulated in vector x.

4. References

None.

5. Parameters

A - REAL array of DIMENSION (IA,p) where p $\geqslant$ N.
Before entry, A should contain in its lower trapezoidal part the Householder vectors as calculated by HSHVOX.
Unchanged on exit.

IA - INTEGER.
On entry, IA specifies the first dimension of array A as declared in the calling (sub)program (IA $\geqslant$ M).
Unchanged on exit.

M - INTEGER.
On entry, M specifies the number of rows of matrix A.
Unchanged on exit.

N - INTEGER.
On entry, N specifies the number of columns of matrix A.
Unchanged on exit.

VECIN - REAL array of DIMENSION at least (M).
On entry VECIN should contain in its first N elements the vector which is to be multiplied by Q; the remaining M$-$N elements are used for working space.
Unchanged on exit, but see section 11.

VECOUT - REAL array of DIMENSION at least (M).
On exit, VECOUT contains the result of the matrix-vector multiplication.

6. Error indicators and warnings

None.

7. Auxiliary routines

No auxiliary routines are used.

8. Timing

The time taken is approximately proportional to MN.

9. Storage

There are no internally declared arrays.

10. Accuracy

The accuracy is up to working precision.

11. Further comments

If the routine is called with the same name for parameters VECIN and VECOUT then the output vector will overwrite the input vector.

11.1. Vectorization information

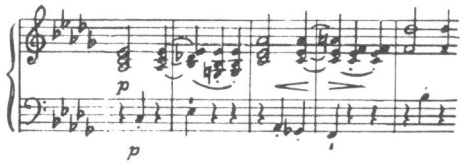
The routine is written in FORTRAN 200, making use of its vector syntax extensions.

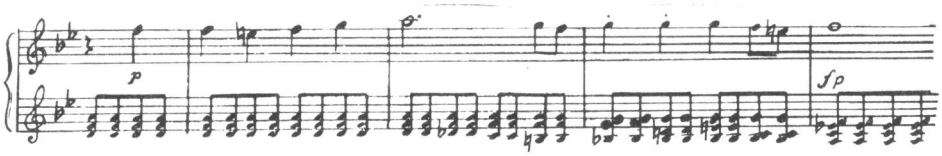
12. Keywords

Householder matrices.

13. Example

Analogously to the use of MULQTX; see example in section LINLSQ.


p
p


p
fp

SAMENVATTING

Dit proefschrift, waarvan de titel in vertaling luidt: "Fundamentele Transformaties in de Lineaire Algebra voor gebruik op Vector Computers", bevat na een korte inleiding een aantal artikelen met zowel theoretische als practische resultaten. De indeling in hoofdstukken volgt de onderverdeling in artikelen.

De introductie geeft als achtergrondinformatie een overzicht van de benodigde theorie op het gebied van de numerieke lineaire algebra en van het verrichten van wetenschappelijk rekenwerk op een vectorcomputer.

Het artikel in het eerste hoofdstuk geeft een klassificatie van methoden om een *LU-ontbinding* van een matrix te bepalen. Een LU-ontbinding is equivalent met het proces van Gauss eliminatie en wordt gebruikt bij het oplossen van lineaire stelsels op een computer. De door ons aangegeven klassificatie onderscheidt de methoden naar hun geschiktheid om geprogrammeerd te worden voor een vectorcomputer.

In het tweede hoofdstuk wordt aangetoond dat het voor de numerieke stabiliteit bij gebruik van de Gauss-Jordan methode voor het oplossen van een stelsel lineaire vergelijkingen, een groot verschil maakt of pivot selectie met rijverwisselingen dan wel kolomverwisselingen wordt uitgevoerd; bij Gauss eliminatie is het gedrag t.a.v. de numerieke stabiliteit niet wezenlijk verschillend. In het artikel wordt een bovengrens voor de norm van het residu bij de gevonden oplossing aangegeven, die alleen geldig is als pivot selectie met kolomverwisselingen wordt uitgevoerd.

Het derde artikel geeft aan, hoe bij gebruik van de Gauss-Jordan methode een schatting gemaakt kan worden voor de norm van de inverse van de coefficientenmatrix in slechts $O(n^2)$ bewerkingen. Met deze norm bepaalt men d.m.v. het conditiegetal de nauwkeurigheid van de berekende oplossing.

Het vierde artikel behandelt een snelle variant van de Gauss-Jordan methode; door het uitstellen van rijoperaties wordt een cumulatief effect bereikt, zodat het aantal arithmetische operaties van $n^3 - \frac{3}{2}n^2 + O(n)$ tot $\frac{2}{3}n^3 - \frac{1}{2}n^2 + O(n)$ wordt teruggebracht. Zolang niet bekend was dat de methode ook een goede pivotstrategie toeliet, leek zij hoofdzakelijk alleen van theoretisch belang. In ons artikel tonen wij aan dat de pivotstrategie uit het tweede hoofdstuk ook doelmatig is voor de hier beschouwde variant. De verbeterde numerieke betrouwbaarheid maakt de methode nu ook practisch van groot belang.

Het vijfde hoofdstuk beschrijft een onderzoek naar de implementatie op een supercomputer van de orthogonalisatiemethode volgens 'Gram-Schmidt'. Er is aangetoond hoe de numeriek verwerpelijke doch implementatie-technisch aantrekkelijke 'klassieke' Gram-Schmidt methode in ere hersteld kan worden, door toepassing van geselecteerde na-iteratie. Indien het aantal benodige iteraties niet te groot wordt (dit hangt af van de conditie van de matrix) kan de verbeterde iteratieve klassieke Gram-Schmidt methode concurreren met de methode van Householder.

In het zesde artikel wordt uitvoerig ingegaan op de verschillende mogelijkheden om een spiegeling te construeren die gebruikt wordt als deelalgoritme in diverse bekende algoritmen op het gebied van numerieke algebra. Het gebruik van dergelijke spiegelingen werd voor het eerst aangegeven door A. S. Householder, daarom dragen de betreffende matrices nu zijn naam. De toepassing van deze matrices in het proces van QR-factorisatie wordt bediscussieerd alsmede het gebruik er van bij de constructie van een gelijkvormigheidstransformatie voor eigenwaarden berekeningen.

De hoofdstukken zeven, acht en negen maken deel uit van het gebruikers handboek voor NUMVEC routines dat uitgegeven wordt door het CWI. Hier zijn de beschrijvingen opgenomen van subroutines die gebaseerd zijn op algoritmen uit de eerste zes hoofdstukken van dit proefschrift. Met name betreft het routines op het gebied van het oplossen van stelsels lineaire vergelijkingen, voor het inverteren van matrices en voor het oplossen van kleinste-kwadraten problemen.

Eines schickt sich nicht für alle!
Sehe jeder, wie er's treibe,
sehe jeder, wo er bleibe,
und wer steht, dass er nicht falle.

Johann Wolfgang Goethe

Stellingen

behorende bij het proefschrift

Basic Transformations in Linear Algebra for Vector Computing

Walter Hoffmann

19 mei 1989

- I -

Voor het berekenen van de exponentiële functie kan men gebruik maken van een geschikte transformatie naar het interval [-0.125,0) waar een polynoom van de graad 10 een benadering levert met een relatieve fout die kleiner is dan 10^{-24}.

P.W. Hemker, W. Hoffmann, S.P.N. van Kampen, H.L. Oudshoorn and D.T. Winter; Single and double-length computation of elementary functions; NW 7/73, Mathematisch Centrum Amsterdam, 1973.

- II -

Bij gebruik van de QL algoritme voor het bepalen van de eigenwaarden van een symmetrische tridiagonale matrix convergeert het eerste element van de subdiagonaal naar nul met een orde van convergentie die minstens gelijk aan twee is doch in het algemeen groter dan drie.

W. Hoffmann and B. N. Parlett; A new proof of global convergence for the tridiagonal QL algorithm; SIAM J. Numer. Anal. 15, 929-937, 1978.

- III -

Voor een test om de berekening van een LU-ontbinding van een tridiagonale matrix wegens een te kleine pivotwaarde voortijdig te beeindigen, is het aan te bevelen om een afbreekcriterium te hanteren dat gerelateerd is aan een norm van de bijbehorende rij en niet aan een norm van de matrix zoals voor volle matrices gangbaar is.

Procedures DECTRI, DECTRIPIV en DECSYMTRI, daterend uit 1974 en deel uit makend van: NUMAL, Numerical Procedures in ALGOL 60; ed.: P.W. Hemker; MC syllabus 47.1 t/m MC syllabus 47.7; Mathematisch Centrum Amsterdam, 1981.

- IV -

In de lineaire algebra leert men dat de 'nieuwe' coördinaten van een vector bij overgang van de heersende (oude) basis naar een andere (nieuwe) basis berekend kunnen worden door vermenigvuldiging toe te passen met de matrix die de identieke afbeelding beschrijft van de ruimte met de oude basis naar dezelfde ruimte met de nieuwe basis.
Uit didactisch oogpunt is het ongewenst om de *inverse* van deze matrix 'de coördinatentransformatiematrix bij overgang van de oude naar de nieuwe basis' te noemen, zoals in veel leerboeken gebeurt.

G.W. Decnop, H. van Iperen, R. Martini; Dictaat Lineaire Algebra; Delftse Uitgeversmij., Delft, 1976.
Drs. J.F. Deckers; Lineaire Algebra 1; Wolters-Noordhoff, Groningen, 1979.
I.N. Herstein, D.J. Winter; Matrix theory and Linear Algebra; Macmillan Publishing Company, New York, 1988.
S. Lipschutz; Linear Algebra; Schaum's Outline series, McGraw-Hill book Cy, New York etc., 1968.

- V -

De Gauss-Jordan algoritme met kolomverwisselingen levert op een vector-computer een snelle en numeriek stabiele routine voor het inverteren van matrices; het inverteren kan bovendien 'in situ' geschieden.

T.J. Dekker and W. Hoffmann; Numerical Improvement of the Gauss-Jordan algorithm; Proceedings ICIAM 87, Paris-La Villette, june 29 - july 3 1987; Contributions from the Netherlands; ed.: A.H.P. van der Burgh, R.M.M. Mattheij.

- VI -

Het oplossen van (grote) lineaire stelsels met behulp van een routine uit de LINPACK-library op de Alliant FX/4 computer vergt aanmerkelijk minder tijd indien men de berekening in een grotere precisie laat uitvoeren.

- VII -

Als op de vectoren

$$f = \begin{pmatrix} 0.4999 \\ 0.5 \\ 0.5001 \\ 0.5 \end{pmatrix}, \quad g = \begin{pmatrix} 0.5 \\ 0.5001 \\ 0.5 \\ 0.4999 \end{pmatrix}, \quad h = \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{pmatrix},$$

de (klassieke) Gram-Schmidt algoritme wordt toegepast om een orthonormale basis van span$\{f, g, h\}$ te bepalen en de Euclidische norm van de eerste vector wordt afgerond op 1, dan zal een verder *exact* uitgevoerde berekening een stelsel vectoren opleveren waarvan de hoek tussen de tweede en de derde vector gelijk is aan $\pi/4$.

- VIII -

Wil men niet het risico lopen dat veel computervoorzieningen uiteindelijk dubbel betaald worden, dan zal men de faculteiten der Universiteit van Amsterdam voor het gebruik van SARA voorzieningen een bonus moeten verstrekken in plaats van een als malus ervaren doorbelasting in rekening te brengen.

- IX -

Bij T.V. opnames van cultuuruitingen die niet speciaal voor televisie zijn gecreëerd, waaronder opera- en concertuitvoeringen, dient de regisseur zich te onthouden van een eigen 'artistieke' inbreng zoals bijvoorbeeld het registreren van dubbelbeelden en extreme close-up's.

- X -

Dat het ontwerp van de mozaiekvloer in de vleugels en gangen van het Centraal Station te Amsterdam met opzet enige onregelmatigheden vertoonde is zeer onaannemelijk vanwege het feit dat werkzaamheden zijn uitgevoerd voor het herstel van onregelmatigheden. Dat voor het opsporen van deze onregelmatigheden geen 'full proof' algoritme beschikbaar was, wordt gedemonstreerd door de nog altijd aanwezige afwijkingen in het patroon.

- XI -

Het is in Amsterdam voor een fietser gevaarlijker om door groen licht te rijden dan door rood.