

"I recommend this *modus operandi*. You will hardly eliminate directly anymore, at least not when you have more than two unknowns. The indirect method can be pursued while half asleep or while thinking about other things." C.F. Gauss (1777-1855).

6

ITERATIVE METHODS FOR LINEAR EQUATIONS

6.1 INTRODUCTION

Gauss (c. 1823) was recommending an iterative *modus operandi* over a direct method (Gaussian elimination) because iterative methods are easier to use (implement) than direct methods.

We must be clear about the distinction between the two types of algorithms : a direct method, using exact arithmetic, calculates an exact solution $x = A^{-1}b$ in a finite number of steps; an iterative method, using exact arithmetic, calculates a sequence $\{x_1, x_2, \dots, x_k, \dots\}$, where $\lim_{k \rightarrow \infty} x_k = x = A^{-1}b$. Thus an iterative method, by definition, cannot calculate an exact solution in a finite number of steps.

It may seem perverse that Gauss was recommending a method that cannot calculate an exact solution over one that can. Let us follow Trefethen & Bau's explanation (pages 243 – 249), where they say that iterative methods are likely to dominate large-scale computing in the future.

6.1.1 Trefethen & Bau's Overview

Direct algorithms, such as Gaussian elimination, require $O(n^3)$ ops. Table 6.1 shows typical problem-sizes solved by direct methods at various times over the last 50 years. In this period the size of problems solved by direct methods has increased by a factor of 10^3 . This may seem impressive, but it is not when compared with processor speeds : these have increased by a factor of 10^9 in the same period.

The explanation for this gap is simple. Let S_p be the speed of a processor in ops per unit time. Let $c_d n^3$ be the number of ops required to solve a problem of size n by a direct method. Then the size of problem solvable in 1 time unit is given by the equation

$$c_d n^3(\text{ops}) = S_p(\text{ops per time unit}) \times 1(\text{time unit}) \quad \text{or} \quad n = O(\sqrt[3]{S_p}).$$

Thus we see why a processor speed-up of 10^9 gives an increase of $\sqrt[3]{10^9} = 10^3$ in problem size.

Table 6.1: Size of Typical Problem

Date	n	Software
1950	20	Wilkinson
1965	200	Forsythe & Moler
1980	2000	LINPACK
1995	20000	LAPACK

6.1.2 Gaussian Elimination and Processor Speed vs Memory Size

Here is a paragraph from Alan Edelman's notes, Math 18.337, Spring 2004, MIT.

It is important to understand that space considerations, not processor speeds, are what bound the ability to tackle such large systems. Memory is the bottleneck in solving these large dense systems. Only a tiny portion of the matrix can be stored inside the computer at any one time. It is also instructive to look at how technological advances change some of these considerations.

For example, in 1996, the record setter of size $n = 128,600$ required $(2/3)n^3 = 1.4 \times 10^{15}$ arithmetic operations (or four times that many if it is a complex matrix) for its solution using Gaussian elimination. On a fast uniprocessor workstation in 1996 running at 140 MFlops/sec, that would take ten million seconds, about $16\frac{1}{2}$ weeks; but on a large parallel machine, running at 1000 times this speed, the time to solve it is only 2.7 hours. The storage requirement was $8n^2 = 1.3 \times 10^{13}$ bytes, however. Can we afford this much main memory? Again, we need to look at it in historical perspective. In 1996, the price was as low as \$10 per megabyte it would cost \$130 million for enough memory for the matrix. Today, however, the price for the memory is much lower. At 5 cents per megabyte, the memory for the same system would be \$650,000. The cost is still prohibitive, but much more realistic. In contrast, the Earth Simulator which can solve a dense linear algebra system with $n = 1,0412,16$ would require $(2/3)n^3 = 7.5 \times 10^{17}$ arithmetic operations (or four times that many if it is a complex matrix) for its solution using Gaussian elimination. For a 2.25 GHz Pentium 4 uniprocessor based workstation available today, at a speed of 3 GFlops/sec this would take 250 million seconds or roughly 414 weeks or about 8 years! On the Earth Simulator running at its maximum of 35.86 TFlops/sec or about 10000 times the speed of a desktop machine, this would only take about 5.8 hrs! The storage requirement for this machine would be $8n^2 = 8.7 \times 10^{14}$ bytes which at 5 cents a megabyte works out to about \$43.5 million. This is still equally prohibitive although the figurative 'bang for the buck' keeps getting better.

Iterative methods have the general form $x^{(k+1)} := Cx^{(k)} + d$. The main calculation is the matrix-vector product $Cx^{(k)}$ which requires $O(n^2)$ ops per iteration, or $k_c O(n^2)$ ops total, where k_c is the number of iterations needed to meet some convergence criterion.

Iterative methods are particularly suited to solving very large **sparse** systems of equations that are arising with increasing frequency. Many of these systems have hundreds of thousands, if not millions, of variables. A sparse matrix has n_z non-zeros per row, and this number is typically very small. Partial differential equations lead to matrices that have $n = 10^5$ while n_z may be as low as 5 or 10. This sparsity can be exploited by storing only the non-zeros, which leads to a

reduction in the number of ops for the matrix-vector operation from $O(n^2)$ to $O(n_2n) = O(n)$ ops per iteration. Careful exploitation of matrix sparsity can lead to huge reductions in both storage and processing time.

6.2 GENERAL ITERATIVE METHODS

We use the ideas developed in Chapter 4 on successive approximations and fixed points, applying them to the linear vector equation $Ax = b$.

6.2.1 Conditions for Convergence

The iterative solution of $Ax = b$ requires the equation to be re-arranged into *fixed point form* as follows :

$$x = Cx + d \equiv T(x). \quad (6.1)$$

If $T(x)$ is a contraction mapping then by Banach's Fixed Point Theorem it has a unique solution which is the limit of the sequence generated by $\{x^{k+1} = T(x^k)\}$. If $T(x)$ is a contraction mapping then for any x and y we have

$$\begin{aligned} \|T(x) - T(y)\| &= \|(Cx + d) - (Cy + d)\| \\ &= \|C(x - y)\| \\ &\leq \|C\| \|x - y\|. \end{aligned}$$

Hence $T(x) = Cx + d$ is a contraction mapping if $\|C\| < 1$. (6.2)

This gives us a sufficient condition for the convergence of $\{x^{k+1} = Cx^k + d\}$. Of course, C must be chosen so that

$$\lim_{k \rightarrow \infty} x^k = x = A^{-1}b.$$

Theorem 6.1. (Convergence) Given an $n \times n$ matrix C , a necessary and sufficient condition for $\{x^{k+1} = Cx^k + d\}$ to converge to the solution of $Ax = b$, for all starting vectors x_0 and all b , is

$$\rho(C) < 1, \quad (6.3)$$

where $\rho(C) = \max\{|\lambda| : \lambda \in \lambda(C)\}$, is the spectral radius of C . □

The computation of $\rho(C)$ is expensive and is rarely used, except for special cases. The following corollary allows us to use a less expensive norm calculation, but is less stringent than the theorem.

Corollary 6.1. A sufficient condition for $\{x^{k+1} = Cx^k + d\}$ to converge to the solution of $Ax = b$, for all starting vectors x_0 and all b , is

$$\|C\| < 1, \quad (6.4)$$

□

Recall that $\rho(C) \leq \|C\|$ for any norm and so $\|C\| < 1$ may not be necessary for convergence. Ultimately, the convergence of the sequence $\{x^{k+1} = Cx^k + d\}$ depends on the eigenvalues of C , and we are free to choose C , to a limited extent.

6.2.2 Order of Convergence

Assume the sequence $\{x^{k+1} = T(x^k)\}$ converges to the fixed point x and define $e^{k+1} = x - x^{k+1}$, the error at the k th iteration. Then we have

$$\begin{aligned} e_{k+1} = x - x^{k+1} &= x - (Cx^k + d) \\ &= Cx + d - (Cx^k + d) \\ &= C(x - x^k) = Ce_k. \end{aligned}$$

Hence $\|e^{k+1}\| \leq \|C\| \|e^k\|$, i.e., Linear or $O(1)$ convergence. Using successive substitution we obtain

$$\|e_{k+1}\| \leq \|C\|^k \|e_0\| \quad (6.5)$$

It is obvious that the smaller $\|C\|$ is, the faster the iterations converge to a solution.

6.2.3 Matrix Splittings

There are many ways to transform $Ax = b$ into fixed point form. The usual methods transform the matrix A by *splitting* it in various ways to obtain the fixed point form. The term *splitting* is used to distinguish it from *factorization*.

A general splitting of $Ax = b$ has the form

$$Mx^{k+1} = Nx^k + b, \text{ where } A = M - N. \quad (6.6)$$

We may view this as : given M, N, x^k, b , solve $Mx^{k+1} = b'$, where $b' = Nx^k + b$.

The equation $Mx^{k+1} = Nx^k + b$ must be repeatedly solved until the sequence $\{x^k\}$ has converged. For this reason M must be chosen so that this equation is easy to solve. As we will see, in the Jacobi and Gauss-Seidel methods, M is diagonal and lower triangular, respectively.

Using Theorem 6.1 and re-writing $Mx^{k+1} = Nx^k + b$ as $x^{k+1} = M^{-1}Nx^k + M^{-1}b = Cx + d$ we see that this iteration scheme converges to a solution of $Ax = b$ if, and only if

$$\rho(M^{-1}N) < 1. \quad (6.7)$$

There are two competing goals in choosing M and N , once they satisfy (6.7) :

- M^{-1} must be easy to calculate or $Mx^{k+1} = Nx^k + b$ easy to solve.
- $\rho(M^{-1}N)$ must be as small as possible.

In what follows we will be using the following matrices, which are splittings of A :

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix} \quad (6.8)$$

6.2.4 Transforming $Ax = B$ to $Mx = Nx + B$

We start by considering the 3×3 case $Ax = b$:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned}$$

Re-write $Ax = b$ as follows :

$$\begin{aligned} a_{11}x_1 &= -a_{12}x_2 - a_{13}x_3 + b_1 \\ a_{22}x_2 &= -a_{21}x_1 - a_{23}x_3 + b_2 \\ a_{33}x_3 &= -a_{31}x_1 - a_{32}x_2 + b_3 \end{aligned}$$

This is equivalent to $Mx = Nx + b$, with $M = D$ and $N = -L - U$, or $Dx = (-L - U)x + b$, whose solution is

$$x = D^{-1}[(-L - U)x + b], \quad \text{or} \quad x^{(k+1)} = D^{-1}[(-L - U)x^{(k)} + b]. \quad (6.9)$$

Thus the fixed point form for our 3×3 system is as follows :

$$\begin{aligned} x_1^{(k+1)} &= (-a_{12}x_2^{(k)} - a_{13}x_3^{(k)} + b_1)/a_{11} \\ x_2^{(k+1)} &= (-a_{21}x_1^{(k)} - a_{23}x_3^{(k)} + b_2)/a_{22} \\ x_3^{(k+1)} &= (-a_{31}x_1^{(k)} - a_{32}x_2^{(k)} + b_3)/a_{33} \end{aligned}$$

As usual, we start with a given vector $x^{(0)} = [x_1^{(0)}, x_2^{(0)}, x_3^{(0)}]^T$.

Example 6.1.

$$A = \begin{bmatrix} 20 & 1 & 4 \\ 5 & 10 & 1 \\ 1 & 2 & 30 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix},$$

which has the fixed point form

$$\begin{aligned} x_1^{(k+1)} &= (-x_2^{(k)} - 4x_3^{(k)} + 1)/20 \\ x_2^{(k+1)} &= (-5x_1^{(k)} - x_3^{(k)} + 1)/10 \\ x_3^{(k+1)} &= (-x_1^{(k)} - 2x_2^{(k)} + 1)/30 \end{aligned}$$

Table 6.2: Jacobi Iterations

k	x_1	x_2	x_3	r^k
0	0.0	0.0	0.0	0.11667
1	0.05	0.1	0.033333	0.031754
2	0.038333	0.071667	0.025	0.0076902
3	0.041417	0.078333	0.027278	0.0020131
4	0.040628	0.076564	0.026731	0.0005116
5	0.040826	0.077013	0.026875	0.00012971
6	0.040774	0.0769	0.026838	0.00003337
\vdots	\vdots	\vdots	\vdots	\vdots
28	0.040785	0.076923	0.026846	3.8079e-017

6.3 THE JACOBI METHOD

The method just illustrated is in fact the Jacobi Method. The general component form is

$$\text{for } i := 1 \text{ to } n \text{ do } x_i^{k+1} := (b_i - \sum_{j=1}^{i-1} a_{ij}x_j^k - \sum_{j=i+1}^n a_{ij}x_j^k) / a_{ii} \text{ endfor} \quad (6.10)$$

In matrix-vector form this is this is

$$x^{k+1} := D^{-1}(b - Lx^k - Ux^k) = -D^{-1}(L + U)x^k + D^{-1}b \quad (6.11)$$

This iteration formula can be written in *correction form* as follows :

$$\text{for } i := 1 \text{ to } n \text{ do } x_i^{k+1} := x_i^k + (b_i - \sum_{j=1}^n a_{ij}x_j^k) / a_{ii} \text{ endfor} \quad (6.12)$$

In *matrix-vector correction form* this is

$$x^{k+1} := x^k + D^{-1}(b - Ax^k) = x^k + D^{-1}r^k, \quad (6.13)$$

where r^k is the residual at the end of the k th iteration.

In the standard form $Mx = Nx + b$ for the Jacobi method we have $M_J = D$ and $N_J = -(L + U)$.

Here is the correction form of the Jacobi algorithm.

```

algorithm Jacobi( $a, b, n, maxits, \epsilon, x_{old}$ )


---


  for  $k := 1$  to  $maxits$  do
    for  $i := 1$  to  $n$  do
       $sum := 0$ 
      for  $j := 1$  to  $n$  do
         $sum := sum + a[i, j] * x_{old}[j]$ 
      endfor
       $x_{new}[i] := x_{old}[i] + (b[i] - sum) / a[i, i]$ 
    endfor
    if  $\|x_{new} - x_{old}\| \leq \epsilon$  then return  $x_{new}$ 
    for  $i := 1$  to  $n$  do
       $x_{old}[i] := x_{new}[i]$ 
    endfor
  endfor
endalg Jacobi

```

It is obvious that this algorithm requires $O(n^2)$ flops/iteration. The single statement in the inner-most loop requires 1 add and 1 mult. Hence the total is, for k_c iterations,

$$\sum_{k=1}^{k_c} \sum_{i=1}^n \sum_{j=1}^n (1 \text{ add} + 1 \text{ mult}) = 2k_c n^2 \text{ ops} = 2n^2 \text{ ops / iter}$$

6.3.1 Convergence of the Jacobi Method.

Using (6.2) and (6.11), a sufficient condition for convergence is

$$\|C\| < 1 \quad \text{or} \quad \|-D^{-1}(L+U)\| < 1.$$

If we use the row-sum matrix norm $\|A\|_1$, a sufficient condition for convergence is

$$\max_i \sum_{i \neq j} |a_{ij} / a_{ii}| < 1, \quad i = 1 \dots n.$$

This condition is called **strict row diagonal dominance**. This condition can be re-written as

$$|a_{ii}| > \sum_{i \neq j} |a_{ij}| \quad i = 1 \dots n.$$

We note that if A is a diagonal matrix then $C = 0$ and only one iteration is needed for the Jacobi method to converge. The 'closer' A is to a diagonal matrix the quicker the method converges.

6.4 THE GAUSS-SEIDEL METHOD

In the Jacobi method new components of x are not used in the right-hand side of the iteration formula until all n new components have been calculated. The Gauss-Seidel method uses a new component as soon as it becomes available.

We alter the Jacobi iteration formula $x^{k+1} := D^{-1}(b - Lx^k - Ux^k)$. This gives the iteration formula

$$(D + L)x^{k+1} = Ux^k + b, \quad \text{or} \quad x^{k+1} = (D + L)^{-1}Ux^k + (D + L)^{-1}b. \quad (6.14)$$

Hence for the Gauss-Seidel method we have

$$M = (D + L) \quad \text{and} \quad N = U, \quad \text{which gives} \quad C = -(D + L)^{-1}U \quad \text{and} \quad d = (D + L)^{-1}b.$$

Each component of the new vector x^{k+1} can be calculated using the original A and b as follows :

$$\text{for } i := 1 \text{ to } n \text{ do} \quad x_i^{k+1} := (b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^n a_{ij}x_j^k) / a_{ii}. \quad \text{endfor}$$

The correction form of this iteration formula is

$$\text{for } i := 1 \text{ to } n \text{ do} \quad x_i^{k+1} := x_i^k + (b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i}^n a_{ij}x_j^k) / a_{ii}. \quad \text{endfor} \quad (6.15)$$

The matrix-vector correction form is

$$x^{k+1} := x^k + D^{-1}(b - Lx^{k+1} - (D + U)x^k) = x^k + D^{-1}r^{k,k+1}, \quad (6.16)$$

where $r^{k,k+1}$ is the 'residual' after the k th iteration.

In the standard form $Mx = Nx + b$ for the Gauss-Seidel method we have $M_G = D + L$ and $N_G = -U$.

Convergence of the Gauss-Seidel Method. The conditions for convergence of the Gauss-Seidel method are more difficult to derive than those for the Jacobi method. However, a sufficient condition is diagonal dominance.

6.4.1 THE ORDER OF COMPUTATIONS

In the Jacobi method the vector x^{k+1} is not updated until all components of the residual r^k have been computed. These components can be computed in any order. Indeed they could be computed simultaneously in parallel.

In the Gauss-Seidel method the i th component of the residual $r_i^{k,k+1}$ depends on old and new components of x and so the components of x cannot be updated simultaneously. This means that parallel versions of the Jacobi method are possible but not possible with the Gauss-Seidel, which is inherently serial.

An equally important implication is that the order in which the new components x_i^{k+1} are computed in the Gauss-Seidel method is important. If we compute the components x_i^{k+1} in the order $i = 1, 2, \dots, n$ then these will be different if we compute them in, say, reverse order, $i = n, n-1, \dots, 1$.

6.5 SUCCESSIVE OVER-RELAXATION

This method modifies the correction term of the Gauss-Seidel method as follows

$$\text{for } i := 1 \text{ to } n \text{ do} \quad x_i^{k+1} := x_i^k + \omega(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i}^n a_{ij}x_j^k) / a_{ii} \quad \text{endfor} \quad (6.17)$$

In matrix-vector form this is

$$x^{k+1} := x^k + \omega D^{-1}(b - Lx^{k+1} - (D + U)x^k) = x^k + \omega D^{-1}r^{k,k+1}, \quad (6.18)$$

Re-writing this in the standard form $Mx = Nx + b$ we have

$$M_S = D + \omega L \quad \text{and} \quad N_S = (1 - \omega)D - \omega U. \quad (6.19)$$

When $\omega = 1$ we have the Gauss-Seidel method; with $\omega > 1$ it is called *successive over-relaxation*; with $\omega < 1$ it is called *successive under-relaxation*. The parameter ω is called the *relaxation parameter* and its purpose is to modify the spectral radius of the resulting $C = M_S^{-1}N_S$ matrix so that convergence is faster – although still linear.

6.6 COMPARISON OF ITERATIVE METHODS

We wish to compare the iterative methods we have discussed so far. We include Iterative Refinement in this comparison. We saw that this method is used to refine solutions obtained by LUP Decomposition. The method has the following steps :

- (1) Compute the residual $r^k = b - Ax^k$ in high precision
- (2) Solve $A\delta x^k = r^k$ in low precision
- (3) Set $x^{k+1} = x^k + \delta x^k$ in high precision

Remember that step 1 uses the original A matrix, while step 2 uses the LUP decomposition of A to solve the equation in $O(n^2)$ time by Forward- and Back-Substitution. Hence each iteration of this method requires $O(n^2)$ work, as do the other methods.

All the methods above have first order convergence, i.e., $\|e^{k+1}\| \leq \|C\| \|e^k\|$, where C depends on the method used. The similarities between the methods can be seen most easily if we write them in matrix-vector correction form :

Table 6.3: Iterative Methods

Method	Iteration Formula	M, N Splitting
Jacobi	$x^{k+1} := x^k + D^{-1}(b - Ax^k) = x^k + D^{-1}r^k$	$M = D, N = -(L + U)$
GS-SOR	$x^{k+1} := x^k + \omega D^{-1}(b - Lx^{k+1} - (D + U)x^k) = x^k + \omega D^{-1}r^{k,k+1}$	$M = \omega(L + D), N = -\omega U$
It. Ref	$x^{k+1} := x^k + A^{-1}(b - Ax^k) = x^k + A^{-1}r^k$	$M = A, N = 0$

It can be seen from these equations that the Jacobi, Gauss-Seidel, and Successive Over-Relaxation methods use different approximations to the A^{-1} matrix of Iterative Refinement.

6.6.1 Complexity Analysis of Iterative Methods

We will analyse the SOR algorithm because Jacobi and Gauss-Seidel have the same complexity.

<pre> algorithm Dense-SOR($a, b, n, \epsilon, \omega, maxits$) for $k := 1$ to $maxits$ do for $i := 1$ to n do $sum := 0$ for $j := 1$ to n do $sum := sum + a[i, j] * x[j]$ endfor $x[i] := x[i] + \omega * (b[i] - sum) / a[i, i]$ endfor if Converged return x endfor endalg Dense-SOR </pre>

We can see that there are three nested loops indexed by k , i , and j . The single statement in the inner-most loop requires 1 add and 1 mult. Hence the total is

$$\sum_{k=1}^{k_c} \sum_{i=1}^n \sum_{j=1}^n (1 \text{ add} + 1 \text{ mult}) = 2k_c n^2 \text{ ops} = 2n^2 \text{ ops /iter}$$

This algorithm requires $O(n^2)$ ops /iteration and $O(n^2)$ storage.

6.7 Modern Iterative Methods

6.7.1 Stationary Methods

The iterative methods for solving $Ax = b$ that we have studied all have the form

$$\begin{aligned} Mx^{(k+1)} &:= Nx^{(k)} + b, \quad \text{where } A = M - N, \\ x^{(k+1)} &:= M^{-1}Nx^{(k)} + M^{-1}b \\ x^{(k+1)} &:= Cx^{(k)} + d, \quad \text{where } C = M^{-1}N. \end{aligned}$$

These are called **stationary methods** because M , N , C , and d do not depend on the iteration counter k . In other words, M , N , C , and d do not vary with time. Hence the adjective 'stationary'.

6.7.2 Non-Stationary Methods

We merely list here the more modern non-stationary methods given in the book by Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd Edition, SIAM, 1994. This book and MATLAB implementations of the various methods are available free at <http://www.netlib.org/templates/>

- | | |
|--|--|
| 1. Conjugate Gradient Method (CG) | 6. Quasi-Minimal Residual (QMR) |
| 2. MINRES and SYMMLQ | 7. Conjugate Gradient Squared Method (CGS) |
| 3. CG on the Normal Equations, CGNE and CGNR | 8. BiConjugate Gradient Stabilized (Bi-CGSTAB) |
| 4. Generalized Minimal Residual (GMRES) | 9. Chebyshev Iteration |
| 5. BiConjugate Gradient (BiCG) | |

6.8 MATLAB NOTES

The two functions below show how easy it is to implement iterative methods in MATLAB.

```
function xnew = Jacobi(A,xold,b,maxits,tol);
M = diag(diag(A)); N = M-A;
for k = 1:maxits
    xnew = M\ (N*xold+b);
    if norm(xnew-xold) < tol + eps(norm(xnew))
        return
    end;
    xold = xnew;
end;
```

```
function x = SOR(A,xold,b,w,maxits,tol);
D = diag(diag(A)); L = tril(A)-D;
U = A-D-L;
M = D + w*L; N = (1-w)*D-w*U;
bp = w*b;
x = xold;
for k = 1:maxits
    x = M\ (N*x+bp);
    if norm(x-xold) < tol + eps(norm(x))
        return
    end;
    xold = x;
end;
```


7

SPARSE MATRICES

7.1 INTRODUCTION

There is no strict definition of a sparse matrix, but generally speaking it is an $n \times n$ matrix which has $O(n)$ non-zero elements. This means that matrices with thousands of rows and columns may have just 1 – 10 non-zeros per row. A full or dense matrix has $O(n^2)$ non-zero elements. Wilkinson has given a concise operational definition:

Definition 7.1. (Wilkinson) A sparse matrix is a matrix with enough zeros that it is worth taking advantage of them.

Sparse matrices occur naturally in the solution of many practical problems, e.g. electrical, gas, and water distribution systems; civil and mechanical engineering (structural analysis); production and financial planning (inventory control and portfolio selection); national and local government operations (income tax analysis and scheduling of fire and ambulance services); economics (Input-Output analysis). At a more theoretical level sparse matrices arise in Graph Theory, Linear Programming, Finite Element Methods, and the solution of ordinary and partial differential equations. Duff, *et al.* [D6] contains a long list of application areas, with references.

Sparse matrices are important because of the need to solve on a computer, linear equations or linear optimization problems that have many thousands of variables (possibly millions) but whose coefficients are mostly zeros. Generally, this sparsity can be exploited, giving large savings in computer time and storage. Indeed, were it not possible to exploit sparsity then many important problems could not be solved on present or future computers.

The sparse matrices that arise in practice are not only sparse but are also highly structured, i.e. the non-zeros form very definite patterns. Figures 1.1 to 1.4 below are taken from NIST's *Matrix Market*¹ site and show some typical sparse matrices. This structure can be exploited also, giving even greater savings in space and time.

¹<http://math.nist.gov/MatrixMarket/> National Institute of Science & Technology, formerly known as the National Bureau of Standards

7.2 HISTORY

The interest in sparse matrices seems to have started in the late 1950's when researchers in electrical power system analysis began to solve realistic problems on computers [?S56]. Electric utilities routinely solve large systems of equations during short circuit, load flow, and stability studies. It was noticed that when real problems were modelled by systems of linear equations, the resulting matrices were sparse with highly structured non-zero patterns. It was also noticed that these matrices were large and that the sparsity would need to be exploited if these problems were to be solved on the rather small and slow computers available at that time. At the same time civil engineers solving structural analysis problems and the oil industry solving large linear programs were writing Fortran code that exploited the sparsity of the very large matrices that arose in these problems.

Sparse matrices became increasingly important in the 1960's and numerical analysts finally realised that this was an important research area but that most of the knowledge was in specialist applications journals which were not widely read. To highlight the importance of sparse matrices, Ralph Willoughby² of IBM organised the first conference on sparse matrices which was held at the IBM Research Center, Yorktown Heights, in 1968 [?W1]. Since then there have been at least six international conferences, with published proceedings, devoted entirely to sparse matrices. The number of research papers on sparse matrices is very large. The 1997 bibliography of Arantes³ contains 2000 items. There are at least three textbooks on sparse matrices, while most textbooks on numerical analysis and data structures contain sections or chapters on the subject.

7.3 SPARSE MATRIX MODELS

Most physical and social structures are sparse in the sense that the elements of these structures are loosely connected. For example, the atoms of very large molecules are directly connected only to a few other atoms; towns are directly connected to 2 or 3 other towns; a person in an organization of 10,000 probably communicates with less than 10-20 people in any week.

A mathematical model of these connections is the *Adjacency Matrix* A , where $a_{ij} = 1$ if element i is connected directly to element j , and $a_{ij} = 0$ or ∞ otherwise. A more general model is: $a_{ij} \neq 0$ if element i is connected directly to element j , and $a_{ij} = 0$ or ∞ otherwise. This allows us to represent a road network by an adjacency matrix, where a_{ij} is the distance or travel time between towns i and j directly connected by a road, and $a_{ij} = \infty$ otherwise. Such adjacency matrices occur in many applications and they all have the characteristic that most of their elements are 0 (or ∞).

In most applications involving sparse matrices the size of the matrix is very large. Typically n is in the range 1,000–1,000,000, with 2–20 non-zeros per row. Storing such large matrices is impossible, even on supercomputers. Besides, most of the storage would be wasted on zeros and, worse still, most of the calculations would be wasted on zeros. We will see in the next two sections that great savings in storage and calculations can be made if the matrix is stored in sparse form.

The World Wide Web has been modelled by various types of matrices which are possibly the largest ever used. *Google's PageRank* requires the calculation of the principal eigenvector of a

²Sometimes called the father of Sparse Matrix Theory. See obituary on page 35

³<http://openlink.br.inter.net/sparse/sparsbib.bib>

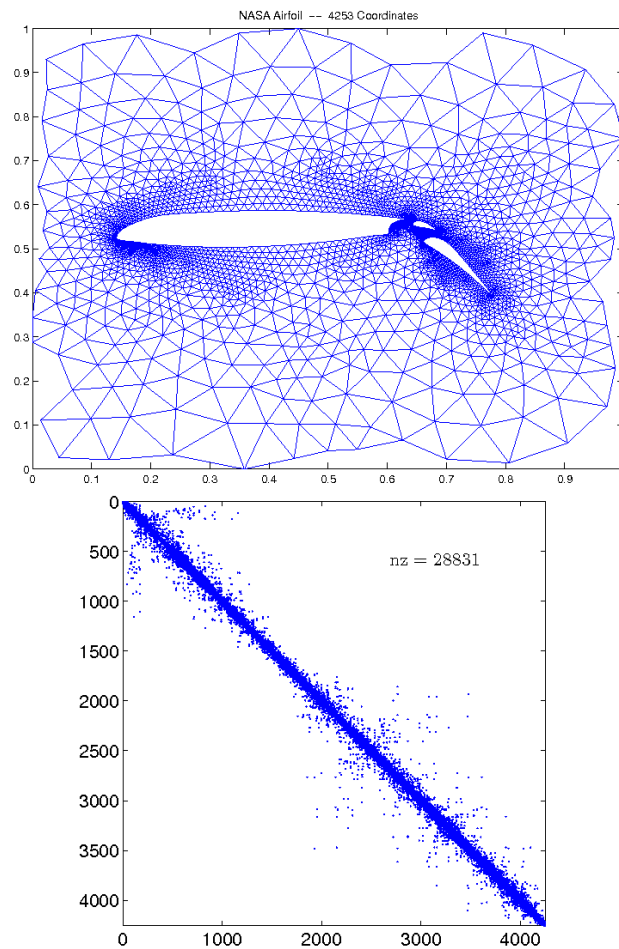


Figure 7.1 : Airfoil Triangulation

matrix of order 2.7 billion. Such calculations are not attempted without great attention being paid to the sparsity, structure, and storage, of such matrices.

7.4 STORAGE OF SPARSE MATRICES

There are many different structures for storing sparse matrices. The choice of storage structure will depend on the problem being solved and the algorithm used to solve it. We will concentrate on a storage structure that is useful when using the *Successive Over-Relaxation* method for linear equations. Detailed discussions of other structures can be found in George & Liu, [G5], and Tewarson [T5].

7.4.1 Dense Matrix Storage

A matrix A is stored in dense form by fixing the location of the first element a_{11} in memory, followed by the remaining elements of the first row, then the second row, etc., as shown in Figure 7.2.

To access an element a_{ij} of a matrix A we must calculate its position relative to the location of the first element a_{11} of A . Thus,

$$\begin{aligned} \text{loc}(a_{ij}) &= \text{loc}(a_{11}) + n(i - 1) + j - 1 && \text{Stored row by row,} \\ \text{loc}(a_{ij}) &= \text{loc}(a_{11}) + n(j - 1) + i - 1 && \text{Stored row by column.} \end{aligned}$$

This calculation assumes that all n^2 elements of A are present and in a predetermined place relative to the first element.

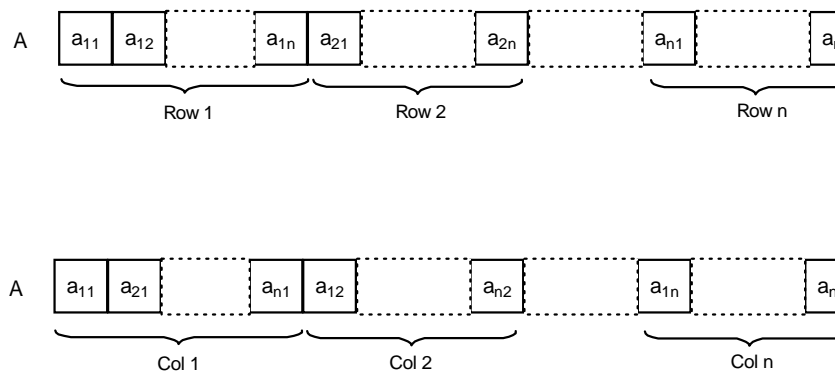


Figure 7.2 : Dense Matrix Storage in Row and Column Major Order

7.4.2 Static Sparse Matrix Storage

We now describe a sparse storage structure that stores the non-zeros only, along with indexing information for each non-zero. The indexing information needed to locate each non-zero must be stored explicitly because the non-zeros can occur at random places in each row and so they are not in pre-determined positions relative to the first element.

The sparse storage structure we use requires 3 arrays, assuming there are n_z non zeros in A : one array *Val* of length n_z holds the non-zeros in row order; one array *Col* of length n_z holds the corresponding column indices; one array *RowStart* of length $n + 1$ indicates where each row starts in the arrays above. An example of this storage structure is shown in Figure 3.

$$A_{6 \times 6} = \begin{bmatrix} 21 & 0 & 0 & 12 & 0 & 0 \\ 0 & 0 & 49 & 0 & 0 & 0 \\ 31 & 16 & 0 & 0 & 0 & 23 \\ 0 & 0 & 0 & 85 & 0 & 0 \\ 55 & 0 & 0 & 0 & 91 & 0 \\ 0 & 0 & 0 & 0 & 0 & 41 \end{bmatrix}$$

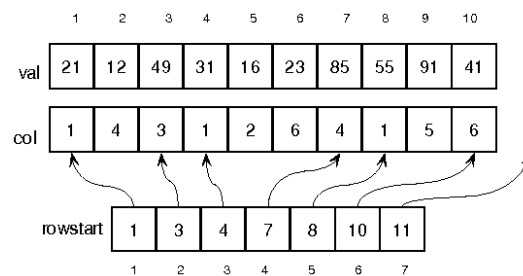


Figure 7.3 : Sparse Matrix Storage in Row Major Order

This storage structure requires $2n_z + n + 1$ boxes. It does not allow easy access to a randomly chosen value a_{ij} , but fortunately random access is rarely needed in matrix-vector computations. We will use this structure to implement the successive over-relaxation method for linear equations.

7.4.3 Dynamic Sparse Matrix Storage

The storage structure above is one of many possible *static* structures. We call it static because once it has been constructed for a given matrix, it is very difficult to alter.

If we tried to implement the *Gaussian Elimination* algorithm on this sparse data structure we would run into serious programming difficulties. Gaussian elimination creates and destroys non-zeros at each of its $n - 1$ stages. These changes could not be elegantly and efficiently handled with the *static* sparse data structure we have used. This structure was chosen because iterative algorithms, in general, do not alter the problem data (see iterative versus transformational algorithms in Chapter 1).

A transformational algorithm, by its nature, dynamically alters the problem data as it proceeds to a solution. If the problem (data) is sparse then the sparsity (non-zeros, patterns of non-zeros, etc.) will change dynamically. This means that we need a dynamic sparse data structure, which, in general, requires *dynamic storage allocation*. This requires knowledge and programming skills that the average ‘computational scientist’ does not have.

7.4.4 Re-Ordering to Reduce Fill-in

Figure 7.4 below shows a sparse 882×882 symmetric matrix A . This is a model of an 882-bus, 345KV electrical transmission network from Quebec Hydro. In dense form this would have 777,924 elements stored in 6.2MB. In sparse form it has 3354 non-zero elements stored in 44KB. (The `MATLAB` statement `whos` gives this information).

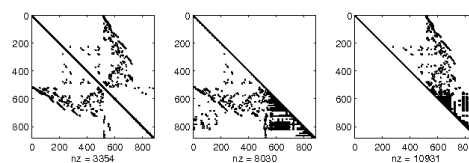


Figure 7.4 : LU decomposition causes fill-in

The second and third matrices above are L and U after LU decomposition is performed on A .

The number of non-zeros has increased by 15607 or 500%.

The effect of LU decomposition on the sparsity pattern of a matrix can be clearly seen in Figure 7.5 below. The matrix A has ones in the 1st row, 1st column, and diagonal. Decomposition causes 100% fill-in, despite the fact there is *no pivoting*. If rows and columns $i = 1, 2, \dots, n$ are re-labelled $i = n, n - 1, \dots, 1$ the the resulting matrix A' suffers little fill-in under LU decomposition. This is equivalent to a symmetric row and column permutation $A' = PAP^T$, with

$$P = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} = P^T$$

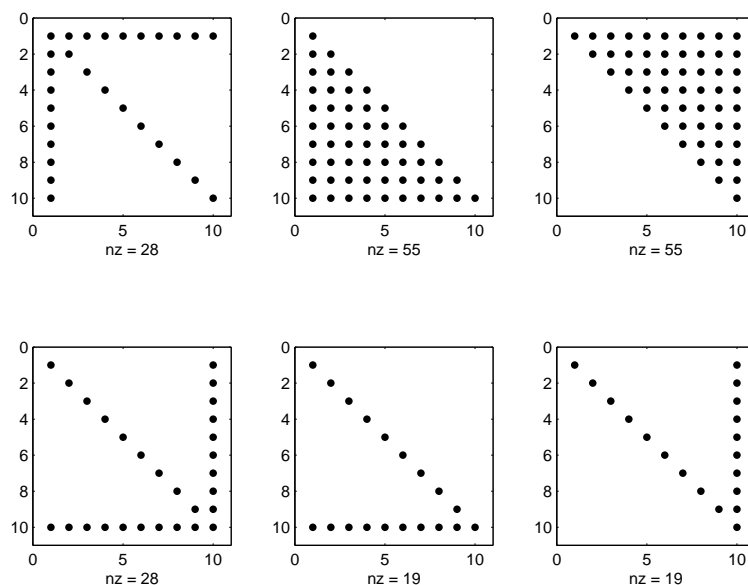


Figure 7.5 : LU decomposition with and without fill-in

We can overcome the fill-in problem by re-ordering the rows or columns or both so that LU decomposition causes little or no fill-in, *before* we do the decomposition. There has been a vast amount of research done on this problem, and it continues today. The A matrix in Figure 7.7 below has been re-ordered using the *Reverse Cuthill-McKee* ordering, provided by the `MATLAB` statements `r = symrcm(A); Ar = A(r,r)`. The beneficial effects of this re-ordering are obvious.

Although we re-order the matrix to reduce fill-in before any computations are done on the matrix, once LU decomposition starts it can still cause fill-in because it re-orders rows with partial pivoting. We cannot tell beforehand what row interchanges LU decomposition will make, unless the matrix has special properties. Hence we need a dynamic storage structure for general matrices undergoing LU decomposition.

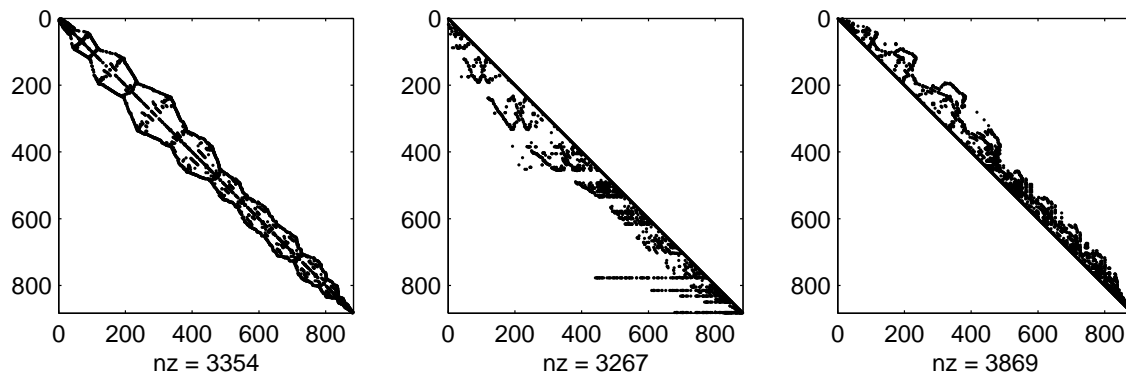


Figure 7.6 : Reordering rows and columns reduces fill-in

7.4.5 Symmetric Positive Definite Matrices

A matrix is *symmetric* if $A = A^T$. A matrix A is *positive definite* if $x^T A x > 0$, for all $x \neq 0$. Matrices that are both symmetric and positive definite have many nice properties, some of which are :

1. Non-singular.
2. Every principal sub-matrix is symmetric positive definite.
3. Each eigenvalue is a positive real number.
4. No pivoting is need for LU decomposition. The diagonals a_{ii} are stable pivots.
5. It has a factorization $A = CC^T$, where C is lower-triangular. This is called the *Cholesky Factorization*.

The MATLAB code below implements the Cholesky factorization.

```

%=====
function C = Cholesky(A);
% Forms the factorization A = CC'
% A must be symmetric positive definite
% Derek O'Connor, UCD, Nov 2005.
%-----
[n,n] = size(A);
C = A;
for k = 1:n
    for j = k+1:n
        C(j,j:n) = C(j,j:n) - C(k,j:n)*C(k,j)/C(k,k);
    end;
    C(k,k:n) = C(k,k:n)/sqrt(C(k,k));
end;
%-----

```

A test on a 1000×1000 spd matrix showed that this code took about 80 sec. on a PIII Xeon, 800MHz machine. MATLAB's built-in function chol did it in 0.85 secs! Very impressive, given that MATLAB's matrix multiplication took 4 secs.

Sparse Cholesky Factorization

We saw in the previous section that re-ordering the matrix reduced the fill-in, but that the pivoting in LU decomposition could cause subsequent fill-in. This is true for all general matrices. If the A matrix is spd then no pivoting is required and we use the Cholesky algorithm. This means we can separate a *Sparse Cholesky* into two independent parts:

- Combinatorial : (1) Find the permutation P such that $A' = PAP^T$ has minimum fill-in.
(2) Perform a symbolic factorization to find where fill-in occurs.
- Numerical : Perform Cholesky on the sparse representation of A' .

We can now use a static storage structure tuned to spd matrices (only an upper or lower triangular part needs to be stored) and the Cholesky algorithm. This can give very fast results.

Ironically, now the hardest part of the problem is finding the P that minimizes fill-in.

7.5 SPARSE ITERATIVE ALGORITHMS

We saw in the Chapter 6 that the correction form of SOR is

$$x_i^{k+1} := x_i^k + \omega(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i}^n a_{ij}x_j^k) / a_{ii}, \quad \text{or} \quad x_i^{k+1} := x_i^k + \delta x_i^k$$

Here is the dense SOR algorithm.

<pre> algorithm Dense-SOR($a, b, n, maxits, \epsilon, \omega, x$) <hr/> $k := 0$ while (not converged) and $k \leq maxits$ do for $i := 1$ to n do $sum := 0$ for $j := 1$ to n to $sum := sum + a[i, j] * x[j]$ endfor $x[i] := x[i] + \omega * (b[i] - sum) / a[i, i]$ endfor $k := k + 1$ endwhile endalg Dense-SOR </pre>

It is obvious that this algorithm requires $O(n^2)$ storage and $O(n^2)$ flops/iteration. The single statement in the inner-most loop requires 1 add and 1 mult. Hence the total is

$$\sum_{k=1}^{k_c} \sum_{i=1}^n \sum_{j=1}^n (1 \text{ add} + 1 \text{ mult}) = 2k_c n^2 \text{ ops} = 2n^2 \text{ ops / iter,}$$

where k_c is the number of iterations to convergence.

computer with 640 KB RAM⁴ using a tri-diagonal matrix with $a_{i,i-1} = 10$, $a_{i,i} = 100$, and $a_{i,i+1} = 15$. The results of this test are shown in Table 7.2 This confirms that the time per

Table 7.2: Sparse SOR on 10 MHz PC/AT

n	ITERS	TIME	TIME/ITER
500	40	10 secs	0.25 secs
1500	40	32 secs	0.75 secs
3000	40	60 secs	1.50 secs

iteration is $O(n)$.

7.5.3 The Convergence Test

To implement the x -convergence test

$$\text{IF } \|x_{old} - x_{new}\|_{\infty} \leq tol + 4.0 \epsilon_m \|x_{new}\|_{\infty} \text{ THEN } \dots$$

we do not need two vectors, nor do we need an $O(n)$ norm(x) function. We must remember that in an algorithm or program statement such as (\star) above, the $x[i]$ on the left of an assignment operator ($:=$) is a 'new' value while that on the right is an 'old' value. So, apart from the convergence test, we need one x -vector only.

The term $\|x_{old} - x_{new}\|_{\infty}$, on the face of it, seems to require two vectors. However this term is really $\|\delta x\|_{\infty}$, where $x_{new} = x_{old} + \delta x$. Looking at the (\star) statement above we see that $\delta x_i = \omega * (b[i] - sum) / d[i]$. If we store this (scalar) value in the variable $delx$ we can use it to incrementally calculate $\|x_{old} - x_{new}\|_{\infty}$. Likewise, we can incrementally calculate $\|x_{new}\|_{\infty}$, and so we do not need a separate norm(x) function. These ideas are implemented in the updated *Sparse-SOR-1* algorithm below.

⁴As usual, we are right at the (b)leading edge.

```

algorithm Sparse-SOR-1 ( $a, b, n, maxits, tol, \omega$ )  $\rightarrow (x, k)$ 

The sparse matrix structure  $a$  contains  $val, col, row, d$ 

for  $k := 1$  to  $maxits$  do
   $normx := 0$ ;  $dnormx := 0$ 
  for  $i := 1$  to  $n$  do
     $sum := 0$ 
    for  $j := rowstart[i]$  to  $rowstart[i + 1] - 1$  do
       $sum := sum + val[j] \times x[col[j]]$ 
    endfor  $j$ 
     $delx := \omega \times (b[i] - sum) / d[i]$  [*]
     $x[i] := x[i] + delx$ 
     $normx := \max(normx, abs(x[i]))$ 
     $dnormx := \max(dnormx, abs(delx))$ 
  endfor  $i$ 
  if  $dnormx \leq tol + 4 \times \epsilon_m \times normx$  then return  $(x, k)$ 
endfor  $k$ 
endalg Sparse-SOR-1

```

Notice that $normx$ and $dnormx$ are incrementally calculated in the i -loop for each iteration of the k -loop. If we want the $\|\cdot\|_1$ norm then we just change 'max' to 'add' in the norm statements above.

7.5.4 Regaining Space used to Store the Diagonal

We saw in the Section 5.5 of the Class Notes that the correction form of SOR is

$$x_i^{k+1} := x_i^k + \omega(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i}^n a_{ij}x_j^k) / a_{ii} \quad (7.1)$$

or

$$x_i^{k+1} := x_i^k + \delta x_i^k$$

Notice that (7.1) uses the diagonal element a_{ii} in two places : in the second summation and in the (only) division. Let us re-write this formula so that a_{ii} is isolated.

$$\begin{aligned}
 x_i^{k+1} &:= x_i^k + \omega(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^n a_{ij}x_j^k) / a_{ii} - \omega a_{ii}x_i^k / a_{ii} \\
 &= (1 - \omega)x_i^k + \omega(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^n a_{ij}x_j^k) / a_{ii} \\
 &= (1 - \omega)x_i^k + \Delta x_i^k, \text{ where } \Delta x_i^k = \delta x_i^k + \omega x_i^k
 \end{aligned} \quad (7.2)$$

Notice that the lower limit on the second summation in (7.2) is now $j = i + 1$ so that a_{ii} is not used here. We see that a_{ii} is used in the single division operation only. We can take advantage of this by a slight modification of the sparse store structure : store the diagonals in a separate array but do *not* store the diagonals with the other non-zeros.

How does this modification affect *SparseSOR-1* algorithm above? Very little. The inner j -loop sums over whatever non-zeros are present and so the second part of the re-written formula above is correctly calculated, because a_{ii} is not included in the *val* and *col* arrays. The scalar variable *delx* now contains the correct value for Δx_i^k and only modification needed is in the statement after the $[\star]$ statement, i.e.,

change $x[i] := x[i] + delx$ to $x[i] := (1 - \omega) \times x[i] + delx$.

These simple modifications to the sparse data structure and algorithm have two benefits : reduced storage and reduced computation because a_{ii} no longer appears in the inner j -loop. We save $O(2n)$ storage and $O(nn_z)$ multiplications per iteration, where n_z is the number of non-zeros. Cost? One multiplication per iteration : $(1 - \omega) \times x[i]$. We call this modified algorithm *SparseSOR-2*.

7.5.5 Preconditioning the A Matrix

Many iterative linear algebra algorithms use *pre-conditioning* to alter the problem data in a way that helps the algorithm. That is, before the algorithm starts solving a problem P , the problem (data) is re-arranged ($P \rightarrow P'$) so that the algorithms runs faster or is easier to implement or both. Also, pre-conditioning may be used to alter the sparsity structure of a problem. Thus we are using a combination of transformational and iterative algorithms.

In our sparse *SOR* algorithm we can eliminate the diagonal elements completely by pre-conditioning the sparse problem $P = (A, b)$ as follows :

1. Divide each row i of A by a_{ii} . ($A \rightarrow A'$)
2. Divide each b_i by a_{ii} . ($b \rightarrow b'$)

We now have a pre-conditioned problem $P' = (A', b')$ in which the diagonal elements of A' have value 1. This means that

1. We do not need to store the diagonal elements. We know their value is 1.
2. We do not need to divide by the diagonal elements. We know that $x/1 = x$.

Solving the pre-conditioned problem P' simplifies both the sparse data structure and the *SparseSOR-2* algorithm. All we need to do is delete the division by $d[i]$ in the $[\star]$ statement of *SparseSOR-1*. Thus we get the modified algorithm

algorithm Sparse-SOR-2-PC($a, b, n, maxits, tol, \omega$) $\rightarrow (x, k)$

The sparse pre-conditioned matrix structure a contains val, col, row
 $a_{ii} = 1$ and is NOT stored

```

for  $k := 1$  to  $maxits$  do
   $norm_x := 0; dnormx := 0$ 
  for  $i := 1$  to  $n$  do
     $sum := 0$ 
    for  $j := row[i]$  to  $row[i+1] - 1$  do
       $sum := sum + val[j] \times x[col[j]]$ 
    endfor  $j$ 
     $\Delta x := \omega \times (b[i] - sum)$  [*]
     $x[i] := (1 - \omega) \times x[i] + \Delta x$ 
     $normx := \max(normx, \text{abs}(x[i]))$ 
     $dnormx := \max(dnormx, \text{abs}(\Delta x))$ 
  endfor  $i$ 
  if  $dnormx \leq tol + 4 \times \epsilon_m \times normx$  then return( $x, k$ )
endfor  $k$ 
endalg Sparse-SOR-2-PC

```

This modification saves a further $O(n)$ storage and $O(n)$ divisions per iteration. Cost? $O(n_z + n)$ divisions before the algorithm starts (each non-zero and b_i need to be divided by the diagonal elements). Because the pre-conditioning is done only once, its cost may be amortized over many runs of the algorithm using the same A' matrix.

Questions

1. Is the solution x altered by pre-conditioning?
2. Why not divide by $|a_{ii}|$ instead of a_{ii} ?
3. Is diagonal dominance preserved?

7.6 IMPLEMENTATION of SpSOR

```

%=====
% SpSOR1(n,val,col,rowstart,d,w,tol,maxits)
% Uses Successive Over-Relaxation to solve the
% sparse system Ax = b, where A is stored in
% [val,col,rowstart,d] form. The diagonal elements
% of A are stored in a separate array d[1..n]
% Returns [x,itors,mflops].
% Written by Derek O'Connor Dec 6 2003
%-----
function [x,itors,mflops] =
    SpSOR1(n,val,col,rowstart,d,b,w,tol,maxits)

nnz = size(val); x=zeros(n,1);
k=0; converged = 0;
while (~converged) & (k <= maxits)
    normx = 0.0; normdx = 0.0;
    for i = 1:n
        sum = 0.0;
        for j = rowstart(i):rowstart(i+1)-1
            sum = sum + val(j)*x(col(j));
        end;
        dx = w*(b(i)-sum)/d(i);
        x(i) = x(i) + dx;
        normdx = max(normdx,abs(dx));
        normx = max(normx,abs(x(i)));
    end; % i-loop
    converged = normdx <= tol + 4.0*eps*normx;
    k=k+1;
end; % k-loop
mflops = (2*nnz + 6*n)*k*1.0e-6;
%----- End of SpSOR1 -----

```

7.6.1 Testing

Example 1 — Tri-Diagonal Matrix. Tri-diagonal matrices of size n were used, with $a_{i,i-1} = 48, a_{ii} = 100,$ and $a_{i,i+1} = 48.$

TABLE III – SPBOR TIMES

n	k_c iters	t secs	mflops	mflops/iters	mflops/sec
1000	23	0.062	0.22991	0.01	3.7082
10000	23	0.500	2.2999	0.1	4.5998
100000	23	4.422	23	1	5.2012
1000000	23	44.578	230	10	5.1595

Example 2 — Finite Difference Matrix. The following non-linear differential equation cannot be solved analytically :

$$\frac{d^2 f(x)}{dx^2} + \cos(x)f(x) = \log(x+4), \quad f(0) = 0, f(1) = 1$$

A *finite difference* approximation to this equation over the *discretized* interval

$$x = [x_0 = 0, x_1, x_2, \dots, x_{n-1}, x_n = 1]$$

gives the set of linear equations $Af = u$

$$\begin{bmatrix} -2 + \frac{\cos(x_1)}{n^2} & 1 & 0 & 0 & \cdots & 0 \\ 1 & -2 + \frac{\cos(x_2)}{n^2} & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -2 + \frac{\cos(x_{n-1})}{n^2} \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} \frac{\log(x_1+4)}{n^2} \\ \frac{\log(x_2+4)}{n^2} \\ \vdots \\ 1 + \frac{\log(x_{n-1}+4)}{n^2} \end{bmatrix}$$

```

%=====
function [A,b] = SpMLCos(n);
% Forms the Matlab sparse matrix A and rhs vector b
% of problem 1(b).
% Derek O'Connor, UCD, Nov 2003.
%-----
x=1/n:1/n:1-1/n;
diagval = -2 + cos(x)./n^2;
i = 1:n-1;
j = 1:n-1;
A = sparse(i,j,diagval,n-1,n-1);
AL = sparse(2:n-1,1:n-2,1,n-1,n-1);
AU = sparse(1:n-2,2:n-1,1,n-1,n-1);
A = A + AL + AU;
b = log(x'+4)/n^2;
b(n-1) = 1 + b(n-1);
%----- End of SpMLCos(n) -----

```

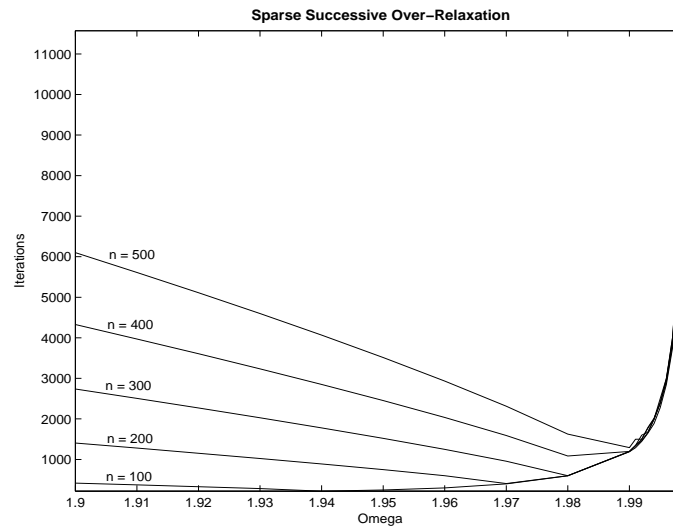


Figure 7.7 : Iterations as a function of ω

7.6.2 Algorithms, Data Structures, and Sparsity

This implementation of the sparse *SOR* algorithm is a good example of the rule :

Design a data structure that suits the algorithm.

In general the design of an algorithm to solve a problem and the design of a data structure to represent the problem go hand-in-hand. In fact this is an iterative design process: alter the data structure to suit the algorithm, alter the algorithm to suit the data structure, etc. We saw this process when we designed the Sparse *SOR* solver.

The data structure used to store the sparse matrix is simple : four arrays `val`, `col`, `rowstart`, and `diag`. The *SOR* algorithm is easy to implement on this data-structure. Indeed, it is more difficult to construct the sparse form of a dense A matrix than to solve $Ax = b$.

If we tried to implement the *Gaussian Elimination* algorithm on this sparse data structure we would run into serious programming difficulties. Gaussian elimination creates and destroys non-zeros at each of its $n - 1$ stages. These changes could not be elegantly and efficiently handled with the *static* sparse data structure we have used. This structure was chosen because iterative algorithms, in general, do not alter the problem data (see iterative versus transformational algorithms in Chapter 1).

A transformational algorithm, by its nature, dynamically alters the problem data as it proceeds to a solution. If the problem (data) is sparse then the sparsity (non-zeros, patterns of non-zeros, etc.) will change dynamically. This means that we need a dynamic sparse data structure, which, in general, requires *dynamic storage allocation*. This requires knowledge and programming skills that the average 'computational scientist' does not have.

7.6.3 Linear Programming and Sparse Matrices

Until recently there has been a curious gap between the development of numerical linear algebra and linear programming. The history of these subjects show that they were developed

independently by different people who were unaware of (or did not care about) each other's work. We have noted a similar gap between electrical and nuclear engineers, and numerical analysts.

Karmarkar's famous linear programming algorithm [[?Karmarkar1984](#)] depended crucially on clever sparse matrix data structures. In fact, after Karmarkar published his famous paper in 1984 there was much controversy about his claims that it was superior to the simplex algorithm. These were based on an implementation he would not reveal because it was proprietary to Bell Labs. He finally revealed in 1989 the details of his sparse matrix data structures in [[?A1](#)].

Karmarkar's algorithm is an interior point linear programming algorithm which moves from an initial point in the feasible region along a path that is always strictly inside the feasible region. Dantzig's simplex algorithm might be called a boundary point algorithm because it starts at a point on the boundary of the feasible region and move along a path that is always on the boundary of the feasible region.

The main computational requirement of Karmarkar's algorithm (and other implementations ?) is the solution of *a sequence of sparse symmetric positive definite systems of linear equations*.

The main implementational advances in linear programming algorithms have been due to the proper use of sparse matrix data structures and their application to crucial parts of algorithms, such as Cholesky factorization.

The paper by Suhl, U. and Suhl, L. [[?S55](#)] has a good description of sparse *LU* factorizations for large linear programs.

7.6.4 Conclusion

. Most real problems are sparse. This is true of linear equations, differential equations, networks, and even functions (e.g., high-degree polynomials with a few non-zero coefficients such as $p_{360}(x) = x^{360} + 20x^{120} + 1$).

Most real problems are huge. Here is Richard Varga ⁵ talking about sparse matrices and iterative methods in 1960:

As an example of the magnitude of problems that have been successfully solved on digital computers by cyclic iterative methods, the Bettis Atomic Power Laboratory of the Westinghouse Electric Corporation had in daily use in 1960 a two-dimensional program that could treat as a special case, Laplacian-type matrix equations of order 20,000

He adds this footnote:

This program, called "PDQ-4", was specifically written for the Philco-2000 computer with 32,000 words of core storage. Even more staggering is Bettis'[s] use of a three-dimensional program, "TNT-1", which treats coupled matrix equations of order 108,000.

Varga's comments show that large problems were being solved routinely on small computers (128 KB core) nearly 50 years ago. Hence, sparse matrix techniques have been around for a long time, even though it took linear programmers⁶, MATLAB⁷, and others until the 1980s and 90s to discover this.

⁵Richard S. Varga, *Matrix Iterative Analysis*, Prentice-Hall, 1962, pages 1 and 2.

⁶CPLEX did not have sparse matrices until Version ??, 199?

⁷MATLAB did not have sparse matrices until Version 4, 1992

7.7 MATLAB and SPARSE MATRICES

MATLAB introduced sparse matrices in Version 4 (1992), which was somewhat tardy, given that their importance was recognized in the late 1950s. The paper by Gilbert, Moler, & Schreiber, 1992 [?G109], gives a fairly good description of MATLAB's implementation of sparse matrices.

The storage structure they use is the same as the static structure used above except that they store matrices in column-major order rather than row-major order. They use dynamic memory allocation, but only on whole matrices. Thus it seems they use a semi-dynamic storage structure as a compromise between the flexible but expensive dynamic structure and the inflexible but efficient static structure.

7.7.1 Design Philosophy

A matrix A may be stored in many different ways : dense integer, sparse floating point, row-major order, etc. Hence we must distinguish between a matrix A and its *storage class*.⁸ This means that two identical matrices A and B can have different storage classes. We consider just two classes : *full* and *sparse*.

1. The time for a sparse matrix operation should be $O(n_z)$, i.e., proportional to the number of operations on non-zero elements.
2. Operation Result Class : $\text{op}(\text{full}, \text{full}) \rightarrow \text{full}$, $\text{op}(\text{full}, \text{sparse}) \rightarrow \text{sparse}$
3. Operation Value Class : The *value* of the result of an operation does not depend on the storage class of the operands. The storage class of the result may depend on the class of the operands. See 2 above.

7.7.2 The Sparse Accumulator

TO BE COMPLETED

⁸MATLAB'S term. Educated computer scientists call this a *type*.

7.8 Matlab's Sparse Matrix Functions

Sparse Matrix Creation

Name	Description	Use
spdiags	Sparse matrix formed from diagonals.	
speye	Sparse identity matrix.	
sprandn	Sparse random matrix.	
sprandsym	Sparse symmetric random	

Visualizing Sparse Matrices

Name	Description	Use
gplot	Plot graph, as in "graph theory"	
spy	Visualize sparsity structure.	

Full to Sparse Conversion

Name	Description	Use
find	Find indices of nonzero entries.	
full	Convert sparse matrix to full matrix.	
sparse	Create sparse matrix from non-zeros and indices.	
spconver	Convert from sparse matrix external format.	

Working with Nonzero Entries of Sparse Matrices

Name	Description	Use
iss	True if matrix is sparse.	
nnz	Number of non-zeros	
nonzeros	Nonzero entries.	
nzmax	Amount of storage allocated for nonzero entries.	
spalloc	Allocate memory for nonzero entries.	
spfun	Apply function to nonzero entries.	
spones	Replace nonzero entries with ones.	

Norm, Condition Number, and Rank

Name	Description	Use
condest	Estimate 1-norm condition.	
normest	Estimate 2-norm.	
sprank	Structural rank.	

Reordering Algorithms

Name	Description	Use
colmmd	Column minimum degree.	
colperm	Order columns based on nonzero count.	
dmperm	Dulmage-Mendelsohn decomposition.	
randperm	Random permutation vector.	
symmmd	Symmetric minimum degree.	
symrcm	Reverse Cuthill-McKee ordering.	

Miscellaneous

Name	Description	Use
spaugment	Form least squares augmented system.	
spparms	Set parameters for sparse matrix routines.	
symbfact	Symbolic factorization analysis.	

© Copyright 1994 by The MathWorks, Inc

7.9 NOTES

Henk A. van der Vorst

According to ISI, the most cited paper in the field of mathematics in the 1990s was

Henk A. van der Vorst, "Bi-CGSTAB — a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear-systems," *SIAM Journal of Scientific and Statistical Computing.*, 13[2]: 631-644, March 1992.

An essay by Dr. Henk A. van der Vorst

In this essay, Dr. Henk A. van der Vorst discusses what influenced his decision to pursue a career in mathematics and what led him to produce his highly cited work, "Bi-CGSTAB — a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear-systems," (SIAM J. Sci. Stat. Comput., Vol 13(2), pp. 631-644, 1992.). This paper has been cited 379 times, making it the most-cited paper of the 1990s in the field of mathematics. Dr. van der Vorst joined the Mathematical Institute of the University of Utrecht in the Netherlands in 1990 as a full professor in applied mathematics.

After his visit to a Shell Research Laboratory, my high school teacher in math told us in class (now more than 40 years ago) that he was so happy with his education, because mathematics had helped him to understand the explanations and demonstrations that had been given by the Shell researchers. He said, "If you master mathematics then you potentially understand everything." That was certainly a slight exaggeration, but it nevertheless sounded like a golden message. Since I definitely wanted to have a better understanding of what was going on around me, mathematics seemed the obvious way to go. Also, if it was not much beyond high school math, then it was pretty easy in addition. What could one wish more? So I enrolled in Utrecht University in 1961. Pretty soon I discovered that mathematics was much more than a set of principles that helped one to solve intellectual riddles. It was not a finished system that one could aim to master after some limited time, but it was really a way of thinking, a means of expressing creativity: endless, an old established science, but still fresh and with undiscovered green meadows, nearby and far away. I also learned that mathematics was more than merely an intellectual activity: it was a necessary tool for getting a grip on all sorts of problems in science and engineering. Without mathematics there is no progress. However, mathematics could also show its nasty face during periods in which problems that seemed so simple at first sight refused to be solved for a long

time. Every researcher will recognize these periods of frustration and helplessness. My first position outside the academic world was with the Dutch Nuclear Research Center and it was an eye-opener for me in that mathematical techniques, in combination with computers, could be used for solving very complicated real-life problems, such as predicting and controlling the behavior of a nuclear reactor. I was deeply impressed by the numerical masterpieces of Jim Wilkinson and Dick Varga. They led the way in showing how one could overcome some serious limitations of computers for solving linear systems of equations. Such systems are immensely important, because most scientific computations lead in one way or another to the necessity to solve linear systems. Although the real world seems to be highly nonlinear, we have to linearize first in order to get insight and to produce meaningful solutions. Many problems in physics, chemistry, engineering, earth sciences, etc., lead to very large systems, and it has always been a challenge for me to help shift our limits in tackling the computational complexity. It is very much the same as the drive of an athlete to try to break barriers. Research by many of us has now led to the ability to solve systems of, say, billions of unknowns. The increase in speed of computers and the human intelligence in discovering faster methods have contributed almost in equal part to the progress made since the early days of Gauss and Jacobi (who solved systems of order 7). By do-

ing this kind of research, one is highly rewarded for useful ideas. Other scientists use them to their advantage and report on their progress. This is what is hidden beneath the cool citation scores. It is a great feeling to realize that my work is used by so many other people. My first success in research was in the mid-1970s, when I proposed, together with Koos Meijerink, the so-called incomplete LU decompositions of matrices, as a way to accelerate the convergence of the Conjugate Gradient method. Our ICCG method became a widely used tool. Then, around 1980, I became heavily inspired by the work of Lanczos, Paige and Saunders, Manteuffel, and others. This led to the completion of my Ph.D. thesis in 1982, at the age of 38. In 1984, I resumed a position in the academic world again: first as a professor at Delft University of Technology, and since 1990 as a professor at the University of Utrecht. Being a professor I felt certainly obliged to something in return for the honor and I devoted much of my (spare) time to research. Together with my advisor Bram van der Sluis, I published a paper that helped to further the understanding of the Conjugate Gradients method ("The rate of convergence of conjugate gradients," *Numer. Math.*, 48[5]: 543-60, 1986). Early ideas by Sonneveld (1984) for improvements in the bi-Conjugate Gradient (Bi-CG) method, for the solution of unsymmetric linear systems, intrigued me for a long time. Sonneveld had a brilliant idea for doubling the speed of convergence of Bi-CG for virtually the same computational costs: CGS. He also published a rather obscure method under the name of IDR. I doubt whether that paper got more than two or three citations altogether. The eventual understanding of that method and the reformulation of it, so that rounding errors had much less bad influence on its speed of convergence, led to the so frequently cited Bi-CGSTAB paper (1992). Since some of the more successful methods had

been published in *SIAM J. Sci.* (for instance, GMRES, 1986), that journal was the obvious choice for me. Also the presentation of Bi-CGSTAB at one of the famous IBM workshops in Oberlech (what a pity that IBM stopped that activity!) was extremely helpful in making other scientists acquainted with the new technique. Bi-CGSTAB is a surprisingly simple algorithm for the combination of two successful techniques: the fast but irregularly converging Bi-CG and the stabilizing effect of GMRES: some 15 lines of computer code. This has helped many people in research and industry solve their complicated computational problems. It has also stimulated further research in my own area. For instance, many new preconditioning techniques are used in combination with Bi-CGSTAB, which explains some of the many citations. Bi-CGSTAB has also been included in the popular mathematical research platform MATLAB. Over the past few years, I have shifted my focus of attention to eigenvalue problems. This is also an exciting area, with many unsolved problems and with many applications in other sciences (plasma physics, astronomy, climate modeling, acoustics, mechanical engineering, etc.). My first success in this area has been the development of the Jacobi-Davidson method (with Gerard Sleijpen, 1996), which was awarded the SIAG-LA prize in 1998 (Sleijpen, G.L.G., and van der Vorst, H.A., "A Jacobi-Davidson iteration method for linear eigenvalue problems," *SIAM J Matrix Anal. A.*, 17[2]: 401-25, April 1996). I trust that there are numerous nuggets to be uncovered in this research area and I hope to be able to find more of these. Meanwhile, I keep an eye on acceleration techniques. One never knows. . .

Henk A. van der Vorst

University of Utrecht

Mathematical Institute Utrecht, The Netherlands

©2006 The Thomson Corporation

7.9.1 Obituary of Ralph Arthur Willoughby (1924 – 2001)

Jane Cullum, Los Alamos National Laboratory, with the help of Iain Duff, Fred Gustavson, Werner Lingner, and Ann and Nona Willoughby. From *SIAM News*, Volume 35, Number 7, September 2002

Ralph Arthur Willoughby, whom some have called the father of sparse matrix theory, died peacefully on July 21, 2001, in Walnut Creek, California, at the age of 77. Ralph spent his childhood and youth

in California, in Nevada City and Yuba City. After serving in World War II, he returned to California as a student at the University of California at Berkeley, where he met and married Nona Chris-

tensen.

On receiving a PhD in mathematics from Berkeley, he accepted a teaching position at Georgia Tech, which he held until 1955, working during the summers at the Oak Ridge National Laboratory. In 1955, he joined Babcock and Wilcox in Lynchburg, Virginia, and became involved in atomic energy research. Two years later he joined the small Mathematics Group at IBM Research.

An initial focus of his work at IBM was circuit simulation. Presented with "stiff" equations, he and Werner Liniger realized that this "multiscale" phenomenon was not unique to circuit models. Together, they devised A-stable implicit differential integration formulae for general time-dependent nonlinear differential equations that included free parameters the user could exploit to mitigate the effects of the different time scales. This concept of "exponential fitting" is recorded in their paper "Efficient integration methods for stiff systems of ordinary differential equations" (SIAM Journal on Numerical Analysis, Vol. 7, 1970, 47-66), which was often quoted in the 1970s.

These new integration methods addressed the issue of stiffness and allowed practical time-step sizes. Use of these methods to solve nonlinear circuit equations, however, required the repeated solution of large systems of equations $Ax = b$. Large, at that point in time, meant several thousand variables, and such systems took hours to solve, occupying entire computers and requiring backup storage.

Ralph and Werner Liniger observed that although these equations were large, they were also sparse, with the sparsity structure fixed throughout the simulation. If a solver code that worked only with nonzero quantities could be produced, they reasoned, the amount of computation and memory required might be reduced to manageable levels. They proceeded to write an internal IBM memo detailing these thoughts.

Initially, special-purpose solver codes were built for each circuit problem. Subsequently seizing the challenge, Fred Gustavson, a colleague of Ralph and Liniger at IBM, devised a clever symbolic triangular factorization program that automatically generated a Fortran code specific for any given problem. Their frequently referenced joint work (Gustavson, Liniger, Willoughby) appeared as "Symbolic generation of an optimal Crout algorithm for sparse systems of linear equations" (*Journal of the Association for Computing Machinery*, Vol. 17, 1970, 87-109). Prior to this research, large

sparse matrices were typically handled by iterative techniques; direct methods were considered appropriate only for small full matrices.

While Ralph and his IBM colleagues were engaged in solving circuit simulation problems, they became aware of a body of sparse matrix knowledge and techniques being developed in certain other application areas, e.g., power systems analysis, linear programming, and structural analysis. This knowledge was buried in application-specific codes or was being published only in applications journals that were not known to the numerical analysis community.

The IBM researchers were convinced from their own work of the emerging importance of sparse matrix problems. Ralph, following a path of the sort he pursued throughout his professional career, decided to organize a conference that would encourage people from these application areas to talk with numerical analysts and thereby stimulate research on this important new topic. He enlisted the financial and scientific support of IBM Research and organized the first *Symposium on Sparse Matrices and Their Applications*. The meeting was held at the IBM T.J. Watson Research Center in Yorktown Heights, New York, September 9-10, 1968.

The proceedings of this conference appeared only as an IBM Research Report, *Sparse Matrix Proceedings* (RA1 3-12-69). By 1970 several thousand requests for copies had come in from all over the world.

Ralph was a keynote speaker at a subsequent sparse matrix conference, organized by the Institute of Mathematics and Its Applications and held at Oxford, April 5-8, 1970. The proceedings of that conference were published as *Large Sparse Sets of Linear Equations* by Academic Press in 1970, with John Reid as editor.

A third meeting, organized jointly by Ralph and Donald Rose, was again held at the IBM T.J. Watson Research Center. At this meeting, held September 9-10, 1971, many of the talks were oriented toward algorithms for working with sparse matrices and toward direct methods for solving large sparse systems of linear equations. The meeting was part of the IBM Research Symposia Series, initiated in 1970, and the proceedings, *Sparse Matrices and Their Applications*, were published by Plenum Press in 1972, with Ralph and Donald Rose as co-editors.

During these years, Ralph was something of a pi-

oneer in promoting the association of graphs and matrices. His work in this area focused on matrix reducibility and orderings, as in the IBM Research Technical Report A Characterization of Matrix Irreducibility via Triangular Factorization (RC 3931, 1972).

These sparse matrix conferences were remarkable in that many of the issues discussed are still considered important today. Examples include the effects of accessing various levels within a memory hierarchy on the efficiency of the computations, and the design of algorithms that are hierarchy-aware; the importance of matching data storage patterns to the pattern of use of that data within the algorithm, now known as "data locality," and blocking and reuse of data; the use of multiple functional units to speed computation; the need for computer architects to work with numerical analysts; the key role that preservation of structure can play in the repeated use of linear solvers with the nonlinear systems of equations that arise in most applications; reorderings to preserve sparsity; and the desirability of fast sparse vector operations, as in the sparse blas. I think it is fair to say that these meetings had a tremendous impact on the numerical linear algebra community and on its research agenda.

In 1973 Ralph organized another in the series of IBM Research Symposia, this time on stiff differential equations. The meeting was held in Wildbad, Germany, in October 1973, under the auspices of IBM Europe. The proceedings were published by Plenum Press.

In the latter part of the 1970s, Ralph's interests shifted to large-scale eigenvalue problems. We began working together and jointly developed the idea of using variants of Lanczos recursions as the basis for a series of algorithms for eigenvalue problems of various types. This work differed from other work in the literature by its refusal to conform to the accepted way of designing eigenvalue algorithms in terms of strictly orthogonal projections. The consequences of this approach, the benefits to be gained from its use, and discussions of practical implementations can be found in the two-volume book *Lanczos Algorithms for Large Symmetric Eigenvalue Computations* (Cullum and Willoughby, Birkhäuser Boston, 1985). These algorithms have been used in a wide variety of problems in computational physics and chemistry. Volume 1 will be republished this fall as part of the SIAM Classics Series. In 1985, Ralph and I

organized an IBM European Institute Workshop titled Large Scale Eigenvalue Problems, which was held July 8-12, 1985, in Oberlech, Austria. Our paper, "A practical procedure for computing eigenvalues of large sparse nonsymmetric matrices," which demonstrated that a great deal of information about the spectrum of any nonsymmetric problem can be obtained by a simple generalization of the ideas used in our symmetric algorithms, was published in the proceedings of this workshop, *Large Scale Eigenvalue Problems*, which we co-edited (Mathematical Studies, Vol. 127, 1986, North-Holland, 193-240).

In 1989, working with Wolfgang Kerner, we extended this work to generalized eigenvalue problems for computing the Alfvén spectrum of magneto-hydrodynamic models (see "A generalized nonsymmetric Lanczos procedure," *Computer Physics Communications*, Vol. 53, 1989, 19-48). In our paper "Computing eigenvalues of large matrices, some Lanczos algorithms and a shift and invert strategy" (in *Advances in Numerical Partial Differential Equations and Optimization*, S. Gómez, J.P. Hennart, R.A. Tapia, eds., SIAM, 1991, 198-246), we discussed the selection of suitable shifts for use in shift-and-invert eigenvalue algorithms.

In 1991, Ralph retired from IBM, and he and Nona moved to Walnut Creek to be near their two daughters, Carol and Ann.

Ralph will be remembered as a gentle person whose primary concern was never himself, but the welfare of his colleagues and family. He showed a particular interest in students, treating them as equal colleagues, asking questions to make them think about what they were doing, and showing interest in their progress by introducing them to leaders in the field and mentioning their work to others in the field.

A history buff, he seemed to know all there was to know about the history of numerical linear algebra, recent wars, and college football.

Ralph is survived by Nona, his wife of 54 years, his daughters Carol and Ann Willoughby, grandsons James and Paul Abers, two nephews (the sons of his late sister Eleanor McFadden), and by his brother, Peter Putnam, and his two children.

Contributions in his memory can be sent to the Scholarship Fund at the University of California, Berkeley.