Linear Systems

Solving for $\mathbf{x}$ in

$$A\mathbf{x} = \mathbf{b},$$

where A is n-by-n and nonsingular, $\mathbf{b}$ is n-by-1.

Some textbook methods

Method 1: Cramer's rule.
We need to compute $n+1$ determinants of order n, each of which is a sum of $n!$ products and each product requires $n-1$ multiplications. Total of $n!\,(n+1)(n-1)$ floating-point multiplications and additions. It is prohibitively expensive. How many floating-point operations (multiplication or addition) are required for solving a system of order 20?

Method 2: Compute A^{-1}, then $\mathbf{x} = A^{-1}\mathbf{b}$ (invert and multiply).
Usually, it is unnecessary to compute A^{-1}. This method is inefficient and inaccurate. Even for a scalar system $7\mathbf{x} = 21$, using $\beta = 10$ and $p = 4$, we have $1/7 = 0.1429$ and $0.1429 \times 21 = 3.001$ (one division and one multiplication). Whereas, $\mathbf{x} = 21/7 = 3.000$ (one division).

Practical methods for dense and sparse systems.
We first decompose A into a product of simple matrices, then we solve a sequence of simple systems. For example, we can decompose A into a product of two triangular matrices, then we solve two triangular systems.

1 Triangular Decomposition

Gaussian elimination (method for general matrices)

Example 1 *$\beta = 10$, $p = 4$.*

$$\begin{bmatrix} 10 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 4 \\ 6 \end{bmatrix}$$

Stage 1. Forward elimination

Step 1. Eliminate x_1 in equations (2) and (3).

$-(-3/10) \times (1) + (2) \to (2)$

$-(5/10) \times (1) + (3) \rightarrow (3)$
pivot: $a_{11} = 10$
multipliers: $m_{21} = -(-3/10)$, $m_{31} = -(5/10)$

$$\begin{bmatrix} 10 & -7 & 0 \\ 0 & -0.1 & 6 \\ 0 & 2.5 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 6.1 \\ 2.5 \end{bmatrix}$$

Step 2. Eliminate x_2 in equation (3).
$-(2.5/-0.1) \times (2) + (3) \rightarrow (3)$
pivot: $a_{22} = -0.1$
multiplier: $m_{32} = 2.5/0.1$.

$$\begin{bmatrix} 10 & -7 & 0 \\ 0 & -0.1 & 6 \\ 0 & 0 & 155 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 6.1 \\ 155 \end{bmatrix}$$

Stage 2. Backward substitution.

$$\begin{bmatrix} 10 & -7 & 0 \\ 0 & -0.1 & 6 \\ 0 & 0 & 155 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 6.1 \\ 155 \end{bmatrix}$$

$$\begin{aligned} x_3 &= 155/155 = 1.000 \\ x_2 &= (6.1 - 6x_3)/(-0.1) = -1.000 \\ x_1 &= (7 - (-7)x_2 - 0x_3)/10 = 0 \end{aligned}$$

Matrix notations

Elementary matrix

$$M_1 = \begin{bmatrix} 1 & 0 & 0 \\ m_{21} & 1 & 0 \\ m_{31} & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0.3 & 1 & 0 \\ -0.5 & 0 & 1 \end{bmatrix}$$

Easy to invert,

$$M_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -m_{21} & 1 & 0 \\ -m_{31} & 0 & 1 \end{bmatrix}$$

$$M_1 A = \begin{bmatrix} 10 & -7 & 0 \\ 0 & -0.1 & 6 \\ 0 & 2.5 & 5 \end{bmatrix} \qquad M_1 b = \begin{bmatrix} 7 \\ 6.1 \\ 2.5 \end{bmatrix}$$

$$M_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & m_{32} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 25 & 1 \end{bmatrix}$$

$$\begin{aligned} M_2 M_1 A &= \begin{bmatrix} 10 & -7 & 0 \\ 0 & -0.1 & 6 \\ 0 & 0 & 155 \end{bmatrix} = U \\ M_2 M_1 b &= \begin{bmatrix} 7 \\ 6.1 \\ 155 \end{bmatrix} \end{aligned}$$

Triangularization of A.

$$A = M_1^{-1} M_2^{-1} U$$

$$\begin{aligned} M_1^{-1} M_2^{-1} &= \begin{bmatrix} 1 & 0 & 0 \\ -m_{21} & 1 & 0 \\ -m_{31} & -m_{32} & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ -0.3 & 1 & 0 \\ 0.5 & -25 & 1 \end{bmatrix} = L \end{aligned}$$

In general, $A = LU$, the LU decomposition (L: lower triangular; U: upper triangular).

Algorithm. Gaussian elimination without pivoting.
Given an n-by-n matrix A, this algorithm computes lower triangular L and upper triangular U so that $A = LU$. On return, A is overwritten by L and U.

```
for k=1 to n-1
   A(k+1:n,k) = A(k+1:n,k)/A(k,k);
   A(k+1:n,k+1:n) = A(k+1:n,k+1:n)
                - A(k+1:n,k)*A(k,k+1:n);
end
```

On return, L and U can be obtained by

$$L = \text{eye}(n, n) + \text{tril}(A, -1); \quad U = \text{triu}(A);$$

Two basic operations in the above algorithm:

1. Scalar-vector multiplication $A(k+1:n,k)/A(k,k)$;
2. Matrix update $A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k) * A(k,k+1:n)$.

Basic Linear Algebra Subprograms (BLAS)
Computer manufacturers provide libraries of these subprograms optimized for their machine architectures, such as memory hierarchies and pipeline structures. For example, there is a subprogram `scal.f` for scalar-vector multiplication in BLAS.

```
      subroutine  dscal(n,da,dx,incx)
c
c     scales a vector by a constant.
c     uses unrolled loops for increment equal to one.
c     jack dongarra, linpack, 3/11/78.
c     modified 3/93 to return if incx .le. 0.
c     modified 12/3/93, array(1) declarations changed to array(*)
c
      double precision da,dx(*)
      integer i,incx,m,mp1,n,nincx
c
      if( n.le.0 .or. incx.le.0 )return
      if(incx.eq.1)go to 20
c
c        code for increment not equal to 1
c
      nincx = n*incx
      do 10 i = 1,nincx,incx
        dx(i) = da*dx(i)
   10 continue
      return
c
```

```
c        code for increment equal to 1
c
c
c        clean-up loop
c
   20 m = mod(n,5)
      if( m .eq. 0 ) go to 40
      do 30 i = 1,m
        dx(i) = da*dx(i)
   30 continue
      if( n .lt. 5 ) return
   40 mp1 = m + 1
      do 50 i = mp1,n,5
        dx(i) = da*dx(i)
        dx(i + 1) = da*dx(i + 1)
        dx(i + 2) = da*dx(i + 2)
        dx(i + 3) = da*dx(i + 3)
        dx(i + 4) = da*dx(i + 4)
   50 continue
      return
      end
```

Note that the loop unrolling technique is used when the increment `incx` equals one. Manufacturers can optimize the above code for their machines. Now, we can rewrite this part of our program in the form:

```
call dscal(n, 1.0/A(k,k), A(k+1,k), incx)
```

What is the value of `incx`? In Fortran, matrix A is stored $\cdots, A(k+1,k), A(k+2,k), \cdots$. Thus `incx` is one. In C, however, A is stored $\cdots, A[k+1][k], A[k+1][k+1], \cdots, A[k+1][n-1], A[k+2][0], \cdots, A[k+2][k], \cdots$. Thus, `incx` should be n. As shown in the above code, in this case, loop unrolling technique cannot be applied.

The matrix update $A(k+1:n, k+1:n) = A(k+1:n, k+1:n) - A(k+1:n, k) * A(k, k+1:n)$ can be written in several different ways. For example, it can be written in the form of column vector updates:

```
for j=k+1:n
   A(k+1:n,j) = A(k+1:n,j) - A(k,j)*A(k+1:n,k);
end
```

The column vector update in the loop has the form $a\mathbf{x} + \mathbf{y}$. There is a subprogram `daxpy.f` ($a\mathbf{x}$ plus $\mathbf{y}$) in BLAS.

```
      subroutine daxpy(n,da,dx,incx,dy,incy)
c
c     constant times a vector plus a vector.
c     uses unrolled loops for increments equal to one.
c     jack dongarra, linpack, 3/11/78.
c     modified 12/3/93, array(1) declarations changed to array(*)
c
      double precision dx(*),dy(*),da
      integer i,incx,incy,ix,iy,m,mp1,n
c
      if(n.le.0)return
      if (da .eq. 0.0d0) return
      if(incx.eq.1.and.incy.eq.1)go to 20
c
c        code for unequal increments or equal increments
c          not equal to 1
c
      ix = 1
      iy = 1
      if(incx.lt.0)ix = (-n+1)*incx + 1
      if(incy.lt.0)iy = (-n+1)*incy + 1
      do 10 i = 1,n
        dy(iy) = dy(iy) + da*dx(ix)
        ix = ix + incx
        iy = iy + incy
   10 continue
      return
c
c        code for both increments equal to 1
c
c
c        clean-up loop
c
   20 m = mod(n,4)
      if( m .eq. 0 ) go to 40
```

```
      do 30 i = 1,m
        dy(i) = dy(i) + da*dx(i)
   30 continue
      if( n .lt. 4 ) return
   40 mp1 = m + 1
      do 50 i = mp1,n,4
        dy(i) = dy(i) + da*dx(i)
        dy(i + 1) = dy(i + 1) + da*dx(i + 1)
        dy(i + 2) = dy(i + 2) + da*dx(i + 2)
        dy(i + 3) = dy(i + 3) + da*dx(i + 3)
   50 continue
      return
      end
```

Only when the increments in both vectors are one, loop unrolling can be applied. In Fortran, the above column vector version has increments one for both vectors. We can write the matrix update as

```
        do 10 j = k+1,n
            call daxpy(n-k,-A(k,j),A(k+1,k),1,A(k+1,j),1)
    10  continue
```

In C, we can write the matrix update in the form of row vector updates:

```
    for i=k:n-1
       A(i,k:n-1) = A(i,k:n-1) - A(i,k-1)*A(k-1,k:n-1);
```

Then, both increments are one. The above matrix update can be written in the form:

```
    for (i=k; i<n; i++)
       daxpy(n-k,-A[i][k-1],&A[k-1][k],1,&A[i][k],1);
```

Memory considerations

Increment one not only allows loop unrolling, which reduces the number of branches, but also takes advantage of modern memory organization.

- Computer storage consists of hierarchy of memories, slow and fast. When a variable is referenced but it is not in fast memory, a block of data containing this variable is moved from slow memory into fast memory. Thus, increment one improves hit ratio.
- Memory is organized in banks. Variables stored in consecutive locations are distributed in different banks. Increment one can reduce memory bank conflicts.

Level two BLAS
The BLAS subprograms `scal` and `axpy` operate on vectors. To reduce the dependency of an algorithm on array orientations (Fortran v.s. C), level two BLAS provides matrix operations. For example, subprogram `ger` performs a rank-one matrix update $A := \alpha\,\mathbf{x}\mathbf{y}^{\mathrm{T}} + A$. This is exactly what we need in matrix update, where the matrix to be updated is $A(k+1:n, k+1:n)$, $\alpha = -1$, $\mathbf{x} = A(k+1:n,k)$, and $\mathbf{y}^{\mathrm{T}} = A(k, k+1:n)$.

A block algorithm using level three BLAS
Suppose that we have already computed the first $i-1$ columns of L and rows of U:

$$A := \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & I & 0 \\ L_{31} & 0 & I \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & \tilde{A}_{22} & \tilde{A}_{23} \\ 0 & \tilde{A}_{32} & \tilde{A}_{33} \end{bmatrix}$$

where
A_{11}: $(i-1)$-by-$(i-1)$
A_{22}: b-by-b
and b is the block size. Suppose that we perform the LU decomposition using Gaussian elimination

$$\begin{bmatrix} \tilde{A}_{22} \\ \tilde{A}_{32} \end{bmatrix} = \begin{bmatrix} L_{22} \\ L_{32} \end{bmatrix} U_{22},$$

then we have

$$\begin{aligned} \begin{bmatrix} \tilde{A}_{22} & \tilde{A}_{23} \\ \tilde{A}_{32} & \tilde{A}_{33} \end{bmatrix} &= \begin{bmatrix} L_{22}U_{22} & \tilde{A}_{23} \\ L_{32}U_{22} & \tilde{A}_{33} \end{bmatrix} \\ &= \begin{bmatrix} L_{22} & 0 \\ L_{32} & I \end{bmatrix} \begin{bmatrix} U_{22} & L_{22}^{-1}\tilde{A}_{23} \\ 0 & \tilde{A}_{33} - L_{32}(L_{22}^{-1}\tilde{A}_{23}) \end{bmatrix} \\ &=: \begin{bmatrix} L_{22} & 0 \\ L_{32} & I \end{bmatrix} \begin{bmatrix} U_{22} & U_{23} \\ 0 & \tilde{A}_{33} \end{bmatrix}. \end{aligned}$$

This is similar to one step of Gaussian elimination except that we eliminate a block of columns before (delayed) updating the right-bottom part of A. We have the following block algorithm rich in level 3 BLAS matrix-matrix operations.

```
for i=1 to n-b step b
   perform LU decomposition A(i:n, i:(i+b-1)) ...
                      to compute  L_22, L_32, and U_22;
   form U_23 = L_22^(-1) A(i:(i+b-1), (i+b):n) ...
                           (solving triangular system);
   form new A_33: A((i+b):n,(i+b):n) = ...
                                A((i+b):n,(i+b):n) ...
          - A((i+b):n,i:(i+b-1))*A(i:(i+b-1),(i+b):n);
end;
perform LU decomposition of the last remaining block.
```

2 Solving Triangular Systems

Given the decomposition $LU = PA$,
solve $A\mathbf{x} = \mathbf{b}$

Solve two triangular systems:
stage 1, solve $L\mathbf{y} = P\mathbf{b}$ (forward elimination)
stage 2, solve $U\mathbf{x} = \mathbf{y}$ (back substitution)

$$LU\mathbf{x} = P\mathbf{b}$$

Consider $U\mathbf{x} = \mathbf{y}$, given U and $\mathbf{y}$, the solution $\mathbf{x}$ overwrites $\mathbf{y}$

Row version (C):

```
y[n-1] = y[n-1] / U[n-1][n-1]
for (i=n-2; i>=0; i--) {
    y[i] = y[i] - U[i][i+1:n-1]y[i+1:n-1];
    y[i] = y[i]/U[i][i]
}
```

Column version (FORTRAN):

```
    y(n) = y(n)/U(n,n)
```

```
      do 10 i=n-1,1,-1
         y(1:i) = y(1:i) - y(i+1)U(1:i,i+1)
         y(i) = y(i)/U(i,i)
 10   continue
```

3 Pivoting

Change the system slightly.

$$\begin{bmatrix} 10 & -7 & 0 \\ -3 & 2.099 & 6 \\ 5 & -1 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 3.901 \\ 6 \end{bmatrix}$$

Exact solution: $(0, -1, 1)^{\mathrm{T}}$.

Forward elimination ($\beta = 10$, $p = 4$)
Step 1

$$\begin{bmatrix} 10 & -7 & 0 \\ 0 & -0.001 & 6 \\ 0 & 2.5 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 6.001 \\ 2.5 \end{bmatrix}$$

pivot: 10, multipliers: 0.3, -0.5
Step 2

$$\begin{bmatrix} 10 & -7 & 0 \\ 0 & -0.001 & 6 \\ 0 & 0 & 1.501 \times 10^4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 6.001 \\ 1.500 \times 10^4 \end{bmatrix}$$

pivot: -0.001; multiplier: 2500.

Backward substitution

$$\begin{aligned} x_3 &= 1.501 \times 10^4 / 1.500 \times 10^4 = 1.001 \\ x_2 &= (6.001 - 6x_3)/(-0.001) = 5.0 \end{aligned}$$

What went wrong?
The pivot -0.001 is small. Consequently, the entry 1.501×10^4 is large. The rounding error in 1.501×10^4 is expected as $0.0005 \times 10^4 = 5.0$, same size as the solution.

- A large multiplier causes growth in the entries in the lower-right corner.

Solution: keep multipliers small.
How? Interchange equations (rows).
Forward elimination step 2:

$$\begin{bmatrix} 10 & -7 & 0 \\ 0 & -0.001 & 6 \\ 0 & 2.5 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 6.001 \\ 2.5 \end{bmatrix}$$

Interchange equations (2) and (3),

$$\begin{bmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & -0.001 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 2.5 \\ 6.001 \end{bmatrix}$$

pivot: 2.5, multiplier: 4×10^{-4}

$$\begin{bmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & 0 & 6.002 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 2.5 \\ 6.002 \end{bmatrix}$$

Backward substitution:

$$\begin{aligned} x_3 &= 6.002/6.002 = 1.000 \\ x_2 &= (2.5 - 5x_3)/2.5 = -1.000 \\ x_1 &= (7 - (-7)x_2 - 0x_3)/10 = 0 \end{aligned}$$

Matrix notations:

$$M_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -0.3 & 1 & 0 \\ 0.5 & 0 & 1 \end{bmatrix} \quad P_2 = P_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$M_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -4 \times 10^{-4} & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & 0 & 6.002 \end{bmatrix}$$

$(M_1^{-1}P_2M_2^{-1})U = A$
But

$$M_1^{-1}P_2M_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -0.3 & -4\times 10^{-4} & 1 \\ 0.5 & 1 & 0 \end{bmatrix}$$

is not lower triangular.
Interchanging m_{21} and m_{31}:

$$\hat{M}_1^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ -0.3 & 0 & 1 \end{bmatrix}$$

In programming, A is overwritten by L and U and L is stored in the lower part of A. When we interchange rows of A, we also interchange corresponding rows of L. Then

$$\hat{M}_1^{-1}M_2^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ -0.3 & -4\times 10^{-4} & 1 \end{bmatrix} = L$$

is lower triangular and

$$LU = \begin{bmatrix} 10 & -7 & 0 \\ 5 & -1 & 5 \\ -3 & 2.009 & 6 \end{bmatrix} = P_2A$$

In general, $LU = P_{n-1}\cdots P_2P_1A$, LU decomposition of a permuted A.

Algorithm. Gaussian elimination with partial pivoting.

```
for k=1 to n-1
   find the pivot:  max(|A(k:n,k)|);
   record the pivoting row index in p(k);
   interchange rows A(k,:) and A(p(k),:);
   A(k+1:n,k) = A(k+1:n,k)/A(k,k);
   A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k)*A(k,k+1:n);
end
```

4 Error Analysis

We assume the coefficient matrix A is nonsingular. In principle, the singularity of A can be detected during the computation. In practice, it may be difficult to decide.

Let $\widehat{\mathbf{x}}$ be the computed solution, can we use the size of the residual vector $\mathbf{r} = \mathbf{b} - A\widehat{\mathbf{x}}$ to measure the accuracy of the solution?

A contrived example.
$\beta = 10$, $p = 3$

$$\begin{bmatrix} 1.15 & 1.00 \\ 1.41 & 1.22 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2.15 \\ 2.63 \end{bmatrix}$$

Apply Gaussian elimination with partial pivoting.
Interchange two rows; multiplier: $1.15/1.41 = 0.816$;

$$\begin{bmatrix} 1.41 & 1.22 \\ 0 & 0.004 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2.63 \\ 0.00 \end{bmatrix}$$

$\hat{x}_2 = 0$, $\hat{x}_1 = 2.63/1.41 = 1.87$.
Residual: $r_1 = 0.01$, $r_2 = 0$.
Exact solution: $x_1 = x_2 = 1$.

- Small residual does not imply small error.

Check the pivots?

- If the pivots are small, A is nearly singular; however, A might be nearly singular but none of the pivots are small.

Backward error analysis
Let $\widehat{L}$ and $\widehat{U}$ be the lower and upper triangular matrices computed by Gaussian elimination, then the backward error analysis gives

$$\widehat{L}\widehat{U} = A + \Delta A,$$

where

$$|\Delta A| \le \gamma_n |\widehat{L}|\,|\widehat{U}|, \quad \gamma_n = \frac{nu}{1 - nu},$$

where $|A|$ denotes the matrix $[|A_{ij}|]$. This says that the stability of Gaussian elimination is determined not by the size of the multipliers but by the size of the matrix $|\widehat{L}|\,|\widehat{U}|$.

For solving the system $Ax = b$, the computed solution $\hat{x}$ satisfies

$$(A + \Delta A)\hat{x} = b$$

where

$$|\Delta A| \leq 2\gamma_n |\widehat{L}|\, |\widehat{U}|.$$

In the normwise form,

$$\|\Delta A\|_\infty \leq 2n^2 \gamma_n \rho_n \|A\|_\infty,$$

where

$$\rho_n = \frac{\max_{i,j,k} |A_{ij}^{(k)}|}{\max_{i,j} |A_{ij}|}$$

is the growth factor and the matrix norm $\|A\|_\infty$ is defined by

$$\|A\|_\infty = \max_i \sum_j |A_{ij}|, \qquad \text{max row sum.}$$

Condition number

The condition number for solving a linear system is the sensitivity of the solution $\mathbf{x}$ to the change in A and/or $\mathbf{b}$. Let $\check{\mathbf{x}}$ be the solution of the perturbed system:

$$(A + \Delta A)\check{\mathbf{x}} = \mathbf{b}.$$

First, we consider the relative residual

$$\frac{\|\mathbf{b} - A\check{\mathbf{x}}\|}{\|A\|\, \|\check{\mathbf{x}}\|} = \frac{\|\Delta A \check{\mathbf{x}}\|}{\|A\|\, \|\check{\mathbf{x}}\|} \leq \frac{\|\Delta A\|}{\|A\|}.$$

This says that the relative residual is insensitive to the change in data. Then, we consider the solution. The change in $\mathbf{x}$ (relative error) is $\|\check{\mathbf{x}} - \mathbf{x}\|/\|\check{\mathbf{x}}\|$ and the change in A is $\|\Delta A\|/\|A\|$. We use the ratio of the two errors as the measurement of the sensitivity, called condition number:

$$\text{cond} = \frac{\|\check{\mathbf{x}} - \mathbf{x}\|/\|\check{\mathbf{x}}\|}{\|\Delta A\|/\|A\|} = \frac{\|A^{-1}\mathbf{b} - \check{\mathbf{x}}\|}{\|\check{\mathbf{x}}\|} \cdot \frac{\|A\|}{\|\Delta A\|} \leq \|A^{-1}\|\, \|A\|.$$

So $\|A^{-1}\|\, \|A\|$ is the condition number for the problem of solving a linear system, a relative error magnification factor.

Putting the above backward error analysis and perturbation analysis together, we get the following two inequalities (forward errors) for relative residual and solution:

$$\frac{\|\mathbf{b} - A\hat{\mathbf{x}}\|}{\|A\|\,\|\hat{\mathbf{x}}\|} \le 2n^2\gamma_n\rho_n$$

and

$$\frac{\|\mathbf{x} - \hat{\mathbf{x}}\|}{\|\hat{\mathbf{x}}\|} \le \|A^{-1}\|\,\|A\| 2n^2\gamma_n\rho_n.$$

5 PageRank

PageRank$^{\mathrm{TM}}$ is an algorithm used in the successful search engine Google. The basic idea of the algorithm is as follows. Let n be the number of pages that can be reached by following a chain of hyperlinks starting at some root page. For Google, by the end of 2002, n was over 3 billion. Let G be the n-by-n connectivity matrix defined by $g_{ij} = 1$ if there is a hyperlink to page i from page j, and $g_{ij} = 0$ otherwise. The matrix G is very large and very sparse. Let the row and column sums:

$$r_i = \sum_j g_{ij}, \qquad c_j = \sum_i g_{ij}.$$

The r_j and c_j are the in-degree and out-degree of the jth page.

When surfing the Web, we go from a page to another following a link. Sometimes, we end up with a page with no out going links, then we choose a random page. Let p be the probability that the random walk follows a link, then $1 - p$ is the probability that a random page is chosen. A typical value of p is 0.85. An n-by-n matrix A is constructed:

$$a_{ij} = p\, g_{ij}/c_j + \delta, \text{ where } \delta = (1-p)/n.$$

Most of the elements of A are δ. When $n = 3 \times 10^9$, $p = 0.85$, then $\delta = 5 \times 10^{-11}$.

The matrix A is the transition probability matrix of the Markov chain, characterized by its elements are all strictly between zero and one and its column sums are all equal to one. An important result in matrix theory known as the *Perron-Frobenius Theorem* concludes that for such matrices a nonzero solution of the equation

$$x = Ax$$

exists and is unique to within a scaling factor. If the scaling factor is chosen so that

$$\sum_i x_i = 1,$$

then x is the *state vector* of the Markov chain and is Google's PageRank.

The matrix A can be written in the form:

$$A = pGD + \delta\, ee^T,$$

where D is a diagonal matrix whose diagonal elements are c_j^{-1} and e is a vector of ones. Thus $e^T x = 1$ and the equation $Ax = x$ can be written

$$pGDx + \delta\, e = x.$$

The state vector x is the solution of

$$(I - pGD)x = \delta\, e.$$

When $p < 1$, the coefficient matrix $I - pGD$ is nonsingular. This approach can take advantage of the sparsity of G, but it breaks down as $p \to 1$ and $\delta \to 0$.

Once G is generated, we need its column sums

```
c = sum(G)
```

In Matlab, we can construct the diagonal matrix D as a sparse matrix:

```
D = spdiags(1./c',0,n,n)
```

The product GD is computed by sparse matrix multiplication. Then

```
p = 0.85;
delta = (1-p)/n;
e = ones(n,1);
x = (speye(n,n) - p*G*D)\(delta*e)
```

efficiently computes PageRank by solving the sparse linear system with Gaussian elimination.

6 A special case

Real symmetric: $A^{\mathrm{T}} = A$, or $a_{ij} = a_{ji}$
Positive definite: $\mathbf{x}^{\mathrm{T}} A \mathbf{x} > 0$, for any $\mathbf{x} \neq 0$
Properties:

- nonsingular
- all principal submatrices are symmetric and positive definite

The method of choice: Cholesky factorization

$$A = LL^{\mathrm{T}}$$

L: lower triangular

Derivation

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{12} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & & & 0 \\ l_{21} & l_{22} & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & \cdots & l_{1n} \\ & l_{22} & \cdots & l_{2n} \\ & & \ddots & \vdots \\ & & & l_{nn} \end{bmatrix}$$

Consider the first row of L: $l_{11} = \sqrt{a_{11}}$. If we have computed the rows 1 through $i-1$ of L, then

$$\begin{aligned} l_{ij} &= (a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk})/l_{jj} \quad j = 1 : i-1 \\ l_{ii} &= \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2} \end{aligned}$$

Algorithm. Cholesky factorization (scalar version)
This algorithm computes a lower triangular L from an n-by-n symmetric and positive definite A so that $A = LL^{\mathrm{T}}$.

```
for i=1 to n
   if i>1
      % compute L(i,1:i)
      for j=1 to i-1
```

```
         L(i,j) = (A(i,j) - L(i,1:j-1)*L(j,1:j-1)')/L(j,j);
      end
   end
   L(i,i) = sqrt(A(i,i) - L(i,1:i-1)*L(i,1:i-1)');
end
```

Cholesky factorization

- storage $n(n+1)/2$
- flops $n^3/3 + O(n^2)$
- stable
- the cheapest way to check if a symmetric matrix is positive definite

Performance
Exploiting memory hierarchy. BLAS1 and BLAS2.

7 Some Classical Iterative Methods

We split A into three parts: D, the diagonal part; L, the strictly lower triangular part; U, the strictly upper triangular part;

$$A = D - L - U.$$

We assume that D is nonsingular.

Jacobi's Iteration

If $\mathbf{x}$ is the solution of the system $A\mathbf{x} = \mathbf{b}$, then

$$D\mathbf{x} = \mathbf{b} + (L + U)\mathbf{x},$$

which suggests an iteration

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} + (L + U)\mathbf{x}^{(k)}).$$

Write in scalar form:

$$x_i^{(k+1)} = a_{ii}^{-1}(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}).$$

This is called Jacobi's iteration.

Gauss-Seidel Iteration

Rearranging D, L, and U, we have

$$(D - L)\mathbf{x} = \mathbf{b} + U\mathbf{x},$$

which leads to the iteration

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} + L\mathbf{x}^{(k+1)} + U\mathbf{x}^{(k)}).$$

Write in scalar form:

$$x_i^{(k+1)} = a_{ii}^{-1}(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}).$$

Successive Overrelaxation

This method is called SOR(ω). Its basic idea is to take a weighted average $(1-\omega)\mathbf{x}^{(k)} + \omega\mathbf{x}^{(k+1)}$ in Gauss-Seidel iteration to improve the convergence. Thus we have the iteration

$$\mathbf{x}^{(k+1)} = (1-\omega)\mathbf{x}^{(k)} + \omega\mathbf{x}^{(k+1)},$$

where ω is the relaxation parameter. In matrix form:

$$\mathbf{x}^{(k+1)} = (1-\omega)\mathbf{x}^{(k)} + \omega D^{-1}(\mathbf{b} + L\mathbf{x}^{(k+1)} + U\mathbf{x}^{(k)}).$$

Rearranging the above, we get

$$(I - \omega D^{-1}L)\mathbf{x}^{(k+1)} = ((1-\omega)I + \omega D^{-1}U)\mathbf{x}^{(k)} + \omega D^{-1}\mathbf{b}.$$

When $\omega = 1$, SOR(ω) is Gauss-Seidel method; when $\omega < 1$, called underrelaxation; when $\omega > 1$, called overrelaxation. Intuitively, overrelaxation is a better choice, since it moves in the direction of $\mathbf{x}^{(k+1)}$.

A General Splitting Method

Let

$$A = M - N,$$

where M is nonsingular, then we have an iteration scheme:

$$x^{(k+1)} = M^{-1}(\mathbf{b} + N\mathbf{x}^{(k)}).$$

Obviously, Jacobi's iteration and Gauss-Seidel iteration are special cases of this method.

Now, we discuss the convergence of this method. Let us examine the error $\mathbf{x}^{(k)} - \mathbf{x}$. Since the true solution $\mathbf{x}$ satisfies

$$\mathbf{x} = M^{-1}(\mathbf{b} + N\mathbf{x}),$$

we have

$$\mathbf{x}^{(k+1)} - \mathbf{x} = (M^{-1}N)(\mathbf{x}^{(k)} - \mathbf{x}) = \cdots = (M^{-1}N)^k(\mathbf{x}^{(0)} - \mathbf{x}).$$

A necessary and sufficient condition for the convergence of this method is

$$\lim_{k\to\infty}(M^{-1}N)^k = 0$$

or $\|M^{-1}N\| < 1$ for some matrix norm. It can be shown that

$$\rho\,(X) = \inf_{\|\cdot\|}\|X\|,$$

where $\rho\,(X)$ is the spectral radius of X. Thus, a necessary and sufficient condition for the convergence is

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

Convergence of Jacobi's, Gauss-Seidel, and SOR methods

- If the magnitude of each diagonal element of A is larger than the sum of the magnitudes of the other elements in the same row, called strictly row diagonally dominant, then both Jacobi's and Gauss-Seidel methods converge, and Gauss-Seidel method is faster.
- The above is a sufficient condition. There is a weaker condition, called irreducibility.
- For SOR(ω), $0 < \omega < 2$ is necessary for convergence. If A is positive definite, then $0 < \omega < 2$ is also sufficient for convergence.
- In SOR(ω), we try to find ω to make the spectral radius of the matrix $(I - \omega D^{-1}L)^{-1}((1-\omega)I + \omega D^{-1}U)$ as small as possible.

8 Krylov Subspace Methods

In this part, we study a family of methods, called Krylov subspace methods. These methods are well suited for large sparse or structured problems, because the major operation in these methods is matrix-vector multiplication.

- In sparse problems, by taking advantage of zeros, we can efficiently multiply a sparse matrix with a vector.
- A sparse matrix is often stored in a special data structure unavailable to the user, and a function is usually provided for matrix-vector multiplication.
- There exist fast matrix-vector multiplication algorithms for structured matrices, such as circulant.

Krylov subspace methods can be applied to

- eigenvalue problems
- linear systems

Although there are programs available, these programs cannot be used as black boxes. The user must understand them in order to use them intelligently.

Krylov Matrix

Given a matrix A of order n and an initial vector $\mathbf{u}$, we form a sequence of vectors

$$\mathbf{u}, A\mathbf{u}, A^2\mathbf{u}, ..., A^{n-1}\mathbf{u} \tag{1}$$

by multiplying A. This sequence is called Krylov sequence and the matrix

$$K = [\mathbf{u}\ A\mathbf{u}\ \cdots\ A^{n-1}\mathbf{u}] \tag{2}$$

is called Krylov matrix. Again, we multiply matrix A and establish a relation between AK and K:

$$AK = KC, \tag{3}$$

where

$$C := [\mathbf{e}_2\ \mathbf{e}_3\ \cdots\ \mathbf{e}_n\ \ -\mathbf{c}] \equiv \begin{bmatrix} 0 & 0 & \cdots & 0 & -c_1 \\ 1 & 0 & \cdots & 0 & -c_2 \\ 0 & 1 & \cdots & 0 & -c_3 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -c_n \end{bmatrix}$$

and the vector $\mathbf{c}$ satisfies the linear equation

$$K\mathbf{c} = -A^n\mathbf{u}.$$

As we know, C is called companion matrix, whose eigenvalues are the roots of the polynomial

$$\lambda^n + c_n\lambda^{n-1} + \cdots + c_1.$$

Suppose that the vectors in the Krylov sequence (1) are linearly independent or the Krylov matrix (2) is nonsingular, then the eigenvalues of C are those of A, since C and A are similar. Thus we have the following algorithm for computing the eigenvalues of A.

Algorithm. Compute the eigenvalues of A.

1. Select an initial vector $\mathbf{u}$;
2. Form $K = [\mathbf{u}\ A\mathbf{u}\ \cdots\ A^{n-1}\mathbf{u}]$;
3. Solve for $\mathbf{c}$ in $K\mathbf{c} = -A^n\mathbf{u}$;
4. Find the roots of $\lambda^n + c_n\lambda^{n-1} + \cdots + c_1$.

There is a major drawback in the above algorithm. The Krylov matrix K is often ill-conditioned even for a moderate size n.

Example 2 *Let* $n = 20$*,* $\mathbf{u} = [1, 1, ..., 1]^{\mathrm{T}}$*, and* $A = \mathrm{diag}$*(0.9501, 0.9355, 0.9218, 0.9169, 0.8936, 0.8913, 0.8214, 0.7919, 0.7621, 0.7382, 0.6154, 0.6068, 0.4860, 0.4565, 0.4447, 0.4103, 0.4057, 0.2311, 0.1763, 0.0185), a random diagonal matrix. The vectors* $A^{18}\mathbf{u}$ *and* $A^{19}\mathbf{u}$ *are almost linearly dependent in that*

$$\frac{(A^{18}\mathbf{u})^{\mathrm{T}}(A^{19}\mathbf{u})}{\|A^{18}\mathbf{u}\|_2\|A^{19}\mathbf{u}\|_2} = 0.9998.$$

To solve the linear dependency problem, we orthogonalize K by performing a QR decomposition:

$$K = QR.$$

Substituting K in (3) with $K = QR$, we obtain

$$AQ = Q(RCR^{-1}).$$

Since R and R^{-1} are upper triangular and C is upper Hessenberg, RCR^{-1} is upper Hessenberg. Denoting RCR^{-1} by H, we have

$$AQ = QH. \tag{4}$$

The matrices A and H are similar. Now, we show how Q and H can be computed using matrix-vector multiplication. Partitioning $Q = [\mathbf{q}_1, \mathbf{q}_2, ..., \mathbf{q}_n]$ and comparing the jth $(1 \le j < n)$ columns of the both sides of (4), we get

$$A\mathbf{q}_j = h_{1,j}\mathbf{q}_1 + \cdots + h_{j,j}\mathbf{q}_j + h_{j+1,j}\mathbf{q}_{j+1}.$$

Since the columns of Q are orthogonal, we get

$$h_{i,j} = \mathbf{q}_i^{\mathrm{T}} A\mathbf{q}_j, \quad i = 1, ..., j.$$

Let $\mathbf{r} = A\mathbf{q}_j - h_{1,j}\mathbf{q}_1 - \cdots - h_{j,j}\mathbf{q}_j$, then

$$h_{j+1,j} = \|\mathbf{r}\|_2 \quad \text{and} \quad \mathbf{q}_{j+1} = \mathbf{r}/h_{j+1,j},$$

since $\mathbf{q}_{j+1}$ is a unit norm vector.

Algorithm. The Arnoldi algorithm for reduction to Hessenberg.

```
Initialize a unit norm vector Q(:,1);
for j = 1 to n
   r = A*Q(:,j);
   for i = 1 to j
      H(i,j) = Q(:,i)'*r;
      r = r - H(i,j)*Q(:,i);
   end
   H(j+1,j) = norm2(r);
   if H(j+1,j) = 0, quit;
   Q(:,j+1) = r/H(j+1,j);
end
```

As pointed out, A is large, it is impractical to compute the n-by-n orthogonal matrix Q. In practice, we only carry out the first k iterations. Partitioning $Q = [Q_k,\ Q_u]$, where Q_k consists of the first k columns of Q and Q_u consists of the last $n-k$ columns, we have

$$H = Q^{\mathrm{T}} A Q = [Q_k,\ Q_u]^{\mathrm{T}} A \, [Q_k,\ Q_u] \equiv \begin{bmatrix} H_k & H_{uk} \\ H_{ku} & H_u \end{bmatrix}.$$

If we carry out only k iterations, we know Q_k and H_k.

- H_k is upper Hessenberg;
- H_{ku} has at most one nonzero entry, $h_{j+1,j}$, in its upper right corner;
- Q_u, H_u, and H_{uk} are unknown.
- The eigenvalues of H_k are approximations of the extreme ones of A.
- Suppose $\mathbf{u}$ is an eigenvector of H_k, then $Q_k\mathbf{u}$ is an approximation of an eigenvector of A.

When A is symmetric, so is H, therefore H is tridiagonal, denoted by

$$T = \begin{bmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \ddots & \ddots & \\ & \ddots & \ddots & \beta_{n-1} \\ & & \beta_{n-1} & \alpha_n \end{bmatrix}.$$

The Arnoldi algorithm is significantly simplified.

Algorithm. Lanczos algorithm for reducing a symmetric A to a symmetric tridiagonal matrix.

```
b(0) = 0; Q(:,0) = 0;
for j = 1 to k
   r = A*Q(:,j);
   a(j) = Q(:,j)'*r;
   r = r - a(j)*Q(:,j) - b(j-1)*Q(:,j-1);
   b(j) = norm2(r);
   if b(j) = 0, quit;
   Q(:,j+1) = r/b(j);
end
```

What do we know about Q_k?

- The columns of Q_k form an orthonormal basis for the Krylov subspace $\mathcal{K}_k(A, \mathbf{q}_1) = \text{span}(\mathbf{q}_1, A\mathbf{q}_1, ..., A^{k-1}\mathbf{q}_1)$.
- In the presence of roundoff error, the vectors $\mathbf{q}_j$ computed by the Lanczos algorithm can quickly lose orthogonality.

Solving $A\mathbf{x} = \mathbf{b}$

How do we solve $A\mathbf{x} = \mathbf{b}$ using the partial reduction algorithms? Since we only have Q_k, whose columns form an orthonormal basis for the Krylov subspace, we search for the "best" approximate solution in the Krylov subspace.

- When A is symmetric and positive definite, we search for $\mathbf{x}_k$ minimizing $\|\mathbf{r}_k\|_{A^{-1}}$, where $\mathbf{r}_k = \mathbf{b} - A\mathbf{x}_k$ is the residual vector. This norm $\|\mathbf{r}_k\|_{A^{-1}}$ is the same as $\|\mathbf{x}_k - \mathbf{x}\|_A$. That is we find $\mathbf{x}_k$ over the Krylov subspace which is closest, in terms of A-norm, to the true solution. This method is called the conjugate gradient method.
- When A is symmetric and positive definite, if we initialize $\mathbf{q}_1 = \mathbf{b}/\|\mathbf{b}\|_2$, then $\mathbf{x}_k = Q_k T_k^{-1}\mathbf{e}_1\|\mathbf{b}\|_2$ minimizes $\|\mathbf{r}_k\|_{A^{-1}}$ over the Krylov subspace $\mathcal{K}_k(A, \mathbf{b})$. Also, $\mathbf{r}_k = \pm\|\mathbf{r}_k\|_2\mathbf{q}_{k+1}$, which means that $\mathbf{r}_k \perp \mathcal{K}_k$.

Conjugate Gradient Method

The conjugate gradient method is the best choice for symmetric and positive definite systems. The Lanczos method computes Q_k and T_k. The solution $\mathbf{x}_k$ is given by $Q_k T_k^{-1}\mathbf{e}_1\|\mathbf{b}\|_2$. The conjugate gradient method efficiently computes the approximate solutions $\mathbf{x}_k$, the residual vectors $\mathbf{r}_k \equiv \mathbf{b} - A\mathbf{x}_k$, which are parallel to $\mathbf{q}_{k+1}$, and the conjugate gradients $\mathbf{p}_k$.

- Since A is symmetric and positive definite, so is $T_k = Q_k^{\mathrm{T}} A Q_k$. Let $T_k = L_k D_k L_k^{\mathrm{T}}$ be the Cholesky factorization of T_k, where

$$L_k = \begin{bmatrix} 1 & & & 0 \\ l_1 & \ddots & & \\ & \ddots & \ddots & \\ 0 & & l_{k-1} & 1 \end{bmatrix}$$

is unit lower bidiagonal and

$$D_k = \begin{bmatrix} d_1 & & & 0 \\ & \ddots & & \\ & & \ddots & \\ 0 & & & d_k \end{bmatrix}$$

is diagonal. Then

$$\begin{aligned} \mathbf{x}_k &= Q_k T_k^{-1} \mathbf{e}_1 \|\mathbf{b}\|_2 \\ &= (Q_k L_k^{-\mathrm{T}})(D_k^{-1} L_k^{-1} \mathbf{e}_1 \|\mathbf{b}\|_2) \\ &\equiv (\tilde{P}_k)(\mathbf{y}_k). \end{aligned}$$

- The columns of $\tilde{P}_k$ are A-conjugate in that $\tilde{P}_k^{\mathrm{T}} A \tilde{P}_k = D_k$.
- Recurrence for $\tilde{P}_k$: $\tilde{P}_k = [\tilde{P}_{k-1}, \tilde{\mathbf{p}}_k]$. In other words, $\tilde{P}_{k-1}$ equals the first $k-1$ columns of $\tilde{P}_k$.
- Recurrence for $\mathbf{y}_k$: $\mathbf{y}_k = \begin{bmatrix} \mathbf{y}_{k-1} \\ \eta_k \end{bmatrix}$. In other words, $\mathbf{y}_{k-1}$ equals the first $k-1$ entries of $\mathbf{y}_k$.
- Recurrence for $\mathbf{x}_k$: $\mathbf{x}_k = \mathbf{x}_{k-1} + \eta_k \tilde{\mathbf{p}}_k$.
- Recurrence for $\mathbf{r}_k$: $\mathbf{r}_k = \mathbf{r}_{k-1} - \eta_k A \tilde{\mathbf{p}}_k$.
- Recurrence for $\tilde{\mathbf{p}}_k$: $\tilde{\mathbf{p}}_k = \mathbf{q}_k - l_{k-1} \tilde{\mathbf{p}}_{k-1}$.
- To eliminate $\mathbf{q}_k$, we use $\mathbf{q}_k = \mathbf{r}_{k-1}/\|\mathbf{r}_{k-1}\|_2$ and define $\mathbf{p}_k \equiv \|\mathbf{r}_{k-1}\|_2 \tilde{\mathbf{p}}_k$ to get

$$\begin{aligned} \mathbf{x}_k &= \mathbf{x}_{k-1} + \nu_k \mathbf{p}_k, \quad \nu_k = \eta_k / \|\mathbf{r}_{k-1}\|_2 \\ \mathbf{r}_k &= \mathbf{r}_{k-1} - \nu_k A \mathbf{p}_k, \\ \mathbf{p}_k &= \mathbf{r}_{k-1} + \mu_k \mathbf{p}_{k-1}, \quad \mu_k = -\|\mathbf{r}_{k-1}\|_2 l_{k-1} / \|\mathbf{r}_{k-2}\|_2 \end{aligned}$$

- Formulas for ν_k and μ_k:

$$\begin{aligned} \nu_k &= \frac{\mathbf{r}_{k-1}^{\mathrm{T}} \mathbf{r}_{k-1}}{\mathbf{p}_k^{\mathrm{T}} A \mathbf{p}_k} \\ \mu_k &= -\frac{\mathbf{p}_{k-1}^{\mathrm{T}} A \mathbf{r}_{k-1}}{\mathbf{p}_{k-1}^{\mathrm{T}} A \mathbf{p}_{k-1}} = \frac{\mathbf{r}_{k-1}^{\mathrm{T}} \mathbf{r}_{k-1}}{\mathbf{r}_{k-2}^{\mathrm{T}} \mathbf{r}_{k-2}} \end{aligned}$$

Algorithm. Conjugate gradient method.

$k = 0$; $\mathbf{x}_0 = 0$; $\mathbf{r}_0 = \mathbf{b}$; $\mathbf{p}_1 = \mathbf{b}$;
repeat
  $k = k + 1$;
  $\mathbf{z} = A\mathbf{p}_k$;
  $\nu_k = (\mathbf{r}_{k-1}^{\mathrm{T}}\mathbf{r}_{k-1})/(\mathbf{p}_k^{\mathrm{T}}\mathbf{z})$;
  $\mathbf{x}_k = \mathbf{x}_{k-1} + \nu_k\mathbf{p}_k$;
  $\mathbf{r}_k = \mathbf{r}_{k-1} - \nu_k\mathbf{z}$;
  $\mu_{k+1} = (\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k)/(\mathbf{r}_{k-1}^{\mathrm{T}}\mathbf{r}_{k-1})$;
  $\mathbf{p}_{k+1} = \mathrm{r}_k + \mu_{k+1}\mathbf{p}_k$;
until $\|\mathbf{r}_k\|_2$ is small enough.

- Each iteration requires only one matrix-vector multiplication ($A\mathbf{p}_k$), two inner products ($\mathbf{p}_k^{\mathrm{T}}\mathbf{z}$ and $\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k$, $\mathbf{r}_{k-1}^{\mathrm{T}}\mathbf{r}_{k-1}$ can be saved in the previous iteration), three axpy operations ($\mathbf{x}_k = \mathbf{x}_{k-1} + \nu_k\mathbf{p}_k$, $\mathbf{r}_k = \mathbf{r}_{k-1} - \nu_k\mathbf{z}$, $\mathbf{p}_{k+1} = \mathbf{r}_k + \mu_{k+1}\mathbf{p}_k$), and a few scalar operations. The only vectors need to be stored are the current values of $\mathbf{x}$, $\mathbf{r}$, $\mathbf{p}$, and $\mathbf{z}$.

- In exact arithmetic, CG is a direct method because $\mathbf{r}_n$ must be zero. In practice, however, in the presence of roundoff error, it is an iterative method. Often, it provides accurate answers after $k \ll n$ steps.

- The convergence rate depends on the condition number. More precisely, it also depends on the eigenvalue distribution.

Preconditioning

To accelerate the convergence rate, the system $A\mathbf{x} = \mathbf{b}$ is replaced by $M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}$, where M is chosen so that

- $M \approx A$ is symmetric and positive definite.

- M^{-1} is well conditioned or has few extreme eigenvalues.

- $M\mathbf{u} = \mathbf{b}$ is easy to solve.

The preconditioner M is often problem dependent. Since $M^{-1}A$ is generally not symmetric, we apply CG implicitly to $\hat{A}\hat{\mathbf{x}} = \hat{\mathbf{b}}$, where $\hat{A} = M^{-1/2}AM^{-1/2}$, $\hat{\mathbf{x}} = M^{1/2}\mathbf{x}$, and $\hat{B} = M^{-1/2}\mathbf{b}$.

Other methods in this family

- Find $\mathbf{x}_k$ minimizing $\|\mathbf{r}_k\|_2$. The algorithms are called MINRES (*minimum residual*) for symmetric A and GMRES (*generalized minimum residual*) for nonsymmetric A.

- Find $\mathbf{x}_k$ making $\mathbf{r}_k$ orthogonal to the Krylov subspace spanned by the first k columns in the Krylov matrix K. In other words, $Q_k^{\mathrm{T}}\mathbf{r}_k = 0$. The algorithm is called SYMMLQ for symmetric A. When A is symmetric and positive definite, this method is equivalent to the conjugate gradient method.

Summary

- Gaussian elimination without pivoting: Working on one matrix, matrix update (BLAS, portability and efficiency), cause of unstability (small pivots)

- Gaussian elimination with partial pivoting: Working on one matrix (efficiency)

- Error estimates: Condition number of a matrix, backward error, relative residual

- Special systems: Trianglar, symmetric and positive definite (Cholesky factorization).

- Classical iterative methods: Jacobi's iteration, Gauss-Seidel iteration, SOR(ω) method, splitting method.

- Krylov subspace methods: Lanczos tridiagonalization, conjugate gradient method.

Software packages

- Subroutines of the direct methods for dense systems can be found at `http://www.netlib.org/lapack`

- Implementations of the direct methods for sparse systems can also be found in netlib by searching SuperLU or SPARSE.

- An on-line help for selecting iterative methods is available at `http://www.netlib.org/templates`
- For large systems, `http://www.netlib.org/scalapack`

References

R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, 1994.
Available http://www.netlib.org/templates/Templates.html

James W. Demmel, *Applied Numerical Linear Algebra*, Society for Industrial and Applied Mathematics, Philadelphia, 1997, §6.1–6.6.

G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd Ed., The Johns Hopkins University Press, Baltimore, MD, 1996. Chapter 3.