

Queens College, CUNY, Department of Computer Science
Numerical Methods
CSCI 361 / 761
Spring 2018
Instructor: Dr. Sateesh Mane

© Sateesh R. Mane 2018

December 15, 2018

11 Lecture 11

11.1 Applied linear algebra Part 3

- We continue the study how to solve a set of coupled linear equations in several variables.
- In this lecture we study (some more) matrix algorithms.

11.2 Matrix equations: review

- We wish to solve a set of n coupled linear equations, which are expressed in matrix form as

$$AX = B. \quad (11.2.1)$$

- Here A is an $n \times n$ square matrix, while X and B are $n \times k$ matrices.
- We employ the **$PA = LU$ algorithm with partial pivoting**.
- The matrix A is decomposed into lower and upper triangular matrices L and U , respectively.
- No extra storage is required: the matrices L and U are stored in the memory occupied by A . The original matrix A is overwritten.
- The partial pivoting permutes the rows of the matrix A , so the algorithm also returns an array of the swap indices and the number of swaps performed.
- The array of swap indices can be formed into a permutation matrix P . The relation between the original matrix A and the LU factorization and the permutation matrix P is

$$PA = LU. \quad (11.2.2)$$

- Any square matrix can be factorized in this way. The algorithm is computationally stable.
- Since $PA = LU$, the equations to be solved using the LU decomposition are

$$LUX = PB. \quad (11.2.3)$$

1. We first solve the following equation for a temporary matrix Y

$$LY = PB. \quad (11.2.4)$$

2. We then solve the following equation to obtain the solution matrix X

$$UX = Y. \quad (11.2.5)$$

3. Both sets of equations are solved using backsubstitution. The LU factorization is performed only once. After that, multiple sets of equations with different right-hand sides can be solved.
- The PA=LU algorithm can be used to calculate the inverse matrix A^{-1} , if A is non-singular, we set $B = I_{n \times n}$ and solve the equation

$$LUX = P. \quad (11.2.6)$$

The solution for X yields the inverse matrix: $A^{-1} = X$.

- The PA=LU algorithm can be used to calculate the determinant of A via

$$\det(A) = (-1)^{\text{number of swaps}} \det(U). \quad (11.2.7)$$

11.3 Special cases

- Although it may seem foolish, let us list some obvious special cases where a lot of theory or formalism is not required
- Suppose the matrix A is **diagonal**, say $A = D$ where

$$D = \begin{pmatrix} d_1 & 0 & \dots & 0 \\ 0 & d_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & d_n \end{pmatrix}. \quad (11.3.1)$$

The solution of the equations is obvious. The inverse matrix is

$$D^{-1} = \begin{pmatrix} d_1^{-1} & 0 & \dots & 0 \\ 0 & d_2^{-1} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & d_n^{-1} \end{pmatrix}. \quad (11.3.2)$$

- Suppose the matrix A is **upper triangular**. Say $A = U$ where U is upper triangular. Then the equations are already in LU form and can be solved immediately

$$UX = B. \quad (11.3.3)$$

There is no need for pivoting and swapping of rows, etc.

- Conversely, suppose the matrix A is **lower triangular**. Say $A = L$ where L is lower triangular. Then the equations are already in LU form and can be solved immediately

$$LX = B. \quad (11.3.4)$$

This can be solved “as is” via backsubstitution.

- If A is already lower triangular, *there is no need to rearrange the rows to make the matrix upper triangular.*
- **Pay attention to the structure of the matrix A .**
- Do not automatically rush to a $PA = LU$ formalism.

11.4 Tridiagonal matrices

- Let us now analyze another special structure of the matrix A , very important in practice.
- The matrix A is **tridiagonal**.
- A tridiagonal matrix has nonzero elements only on the main diagonal and the two neighboring diagonals immediately above and below the main diagonal.
- In other words, $a_{ij} = 0$ if $|i - j| > 1$.
- The structure of a tridiagonal matrix T is shown below.

$$T = \begin{pmatrix} a_1 & c_1 & 0 & & \dots & 0 \\ b_2 & a_2 & c_2 & 0 & & 0 \\ 0 & b_3 & a_3 & c_3 & 0 & 0 \\ & & & \ddots & & \\ & & & & \ddots & \\ 0 & \dots & 0 & b_{n-1} & a_{n-1} & c_{n-1} \\ 0 & \dots & & 0 & b_n & a_n \end{pmatrix}. \quad (11.4.1)$$

- This looks messy. Let us clean it up by replacing all the zeroes with blanks.
- Then the matrix equation $T\mathbf{x} = \mathbf{r}$ looks like this (“ \mathbf{r} ” for right hand side).

$$\begin{pmatrix} a_1 & c_1 & & & & \\ b_2 & a_2 & c_2 & & & \\ & b_3 & a_3 & c_3 & & \\ & & & \ddots & & \\ & & & & \ddots & \\ & & & & & b_{n-1} & a_{n-1} & c_{n-1} \\ & & & & & & b_n & a_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ \vdots \\ r_{n-1} \\ r_n \end{pmatrix}. \quad (11.4.2)$$

- For formal purposes, we set $b_1 = 0$ and $c_n = 0$.
Then we can write general formulas for $i = 1, \dots, n$ without special cases for $i = 1$ and $i = n$.
- For a tridiagonal matrix, we only need to store the elements in the three diagonals, which is totally $3n$ elements (actually $3n - 2$). We do not need storage space for n^2 matrix elements.
- All the calculations can be formulated using only the elements in three diagonals.
- A tridiagonal set of equations can be solved using $O(n)$ computations. By comparison, LU factorization requires $O(n^3)$ computations. This saving in computation time is important in practical applications.

11.5 Tridiagonal algorithm Part 1

- The matrix equation to solve is $T\mathbf{x} = \mathbf{r}$, shown in eq. (11.4.2).
- The problem can be solved in $O(n)$ steps as follows.
 1. We use the first equation to express x_1 in terms of x_2 .
 2. We substitute for x_1 in the second equation, to obtain an equation in x_2 and x_3 .
 3. Then we express x_2 in terms of x_3 .
 4. Then we substitute for x_2 in the third equation, obtain an equation involving x_3 and x_4 , and use that to express x_3 in terms of x_4 .
 5. We repeat the process until we reach the last equation. We substitute for x_{n-1} in the last equation, and we obtain an equation **involving x_n only**.
 6. Hence we solve for x_n .
 7. Then we work backwards (backsubstitution) to compute the values of x_{n-1}, \dots, x_1 in reverse order.
 8. There are totally $n - 1$ elimination steps for x_1, \dots, x_{n-1} , one step to solve for x_n , then $n - 1$ backsubstitution steps for x_{n-1}, \dots, x_1 .
 9. Hence there are totally $2n - 1$ equations to process, i.e. the tridiagonal equations are solved in $O(n)$ steps.
- We can obviously formulate the algorithm in the opposite direction.
We eliminate x_n, \dots, x_2 , solve for x_1 , then perform backsubstitution to solve for x_2, \dots, x_n .
- **There is no pivoting** in the tridiagonal algorithm.
- For this reason the tridiagonal algorithm can fail even if the matrix is non-singular.
- In most practical applications, the lack of pivoting is not a serious problem.
- Unlike LU decomposition, the original matrix T is **not overwritten**.
- The same problems of inconsistent or ill-conditioned equations (or not linearly independent) also exist for tridiagonal matrix equations. Such difficulties are connected with the structure of the equations themselves, not with a computational algorithm.
- The tridiagonal algorithm is a good choice, if it is applicable to a problem. If it fails, one can use the $PA = LU$ algorithm.

11.6 Tridiagonal algorithm Part 2

11.6.1 Elimination of unknowns

- The equation in the first row is

$$a_1x_1 + c_1x_2 = r_1. \quad (11.6.1.1)$$

- We express x_1 in terms of r_1 and x_2 as follows

$$x_1 = \frac{r_1}{a_1} - \frac{c_1}{a_1}x_2 \equiv \beta_1 - \alpha_1x_2. \quad (11.6.1.2)$$

- The equation in the second row is

$$b_2x_1 + a_2x_2 + c_2x_3 = r_2. \quad (11.6.1.3)$$

- We substitute $x_1 = \beta_1 - \alpha_1x_2$ into the above equation, to obtain an equation in two unknowns x_2 and x_3 .

$$\begin{aligned} b_2(\beta_1 - \alpha_1x_2) + a_2x_2 + c_2x_3 &= r_2 \\ (a_2 - b_2\alpha_1)x_2 + c_2x_3 &= r_2 - b_2\beta_1. \end{aligned} \quad (11.6.1.4)$$

- We follow the pattern and express x_2 in terms of r_2 and x_3 as follows

$$x_2 = \frac{r_2 - b_2\beta_1}{a_2 - b_2\alpha_1} - \frac{c_2}{a_2 - b_2\alpha_1}x_3 \equiv \beta_2 - \alpha_2x_3. \quad (11.6.1.5)$$

- This is the general pattern: for row i ($2 \leq i \leq n-1$), we write

$$x_i = \beta_i - \alpha_i x_{i+1}. \quad (11.6.1.6)$$

- The parameters β_i and α_i are given by

$$\alpha_i = \frac{c_i}{a_i - b_i\alpha_{i-1}}, \quad \beta_i = \frac{r_i - b_i\beta_{i-1}}{a_i - b_i\alpha_{i-1}}. \quad (11.6.1.7)$$

- If we define $b_1 = 0$ and $c_n = 0$ then we can extend the above expressions to all $i = 1, \dots, n$.
- **Note that if $a_1 = 0$ or $a_i - b_i\alpha_{i-1} = 0$ for $i = 2, \dots, n$, the algorithm will encounter a division by zero and will fail. The tridiagonal algorithm has no pivoting.**
- Although this is a weak point, it is not a serious problem in most practical applications.

11.6.2 Solution: backsubstitution

- Finally we reach the last equation, which is

$$b_n x_{n-1} + a_n x_n = r_n . \quad (11.6.2.1)$$

- We eliminate x_{n-1} by using $x_{n-1} = \beta_{n-1} - \alpha_{n-1} x_n$. This yields an equation in x_n only.

$$\begin{aligned} b_n(\beta_{n-1} - \alpha_{n-1} x_n) + a_n x_n &= r_n \\ (a_n - b_n \alpha_{n-1}) x_n &= r_n - b_n \beta_{n-1} , . \end{aligned} \quad (11.6.2.2)$$

- The solution for x_n is

$$x_n = \frac{r_n - b_n \beta_{n-1}}{a_n - b_n \alpha_{n-1}} . \quad (11.6.2.3)$$

- We now work backwards by backsubstitution to calculate x_{n-1}, \dots, x_1 in reverse order.

$$\begin{aligned} x_{n-1} &= \beta_{n-1} - \alpha_{n-1} x_n , \\ &\vdots \\ x_i &= \beta_i - \alpha_i x_{i+1} , \\ &\vdots \\ x_1 &= \beta_1 - \alpha_1 x_2 . \end{aligned} \quad (11.6.2.4)$$

11.7 Tridiagonal algorithm Part 3

- A careful examination of the algorithm in Sec. 11.6 shows that an array to hold the values of β_i is unnecessary. We can use the elements of the solution vector x_i to hold β_i .
- The only temporary storage required is to hold the values of α_i , $i = 1, \dots, n - 1$.
- For computational purposes we can therefore restructure the algorithm as follows.
- **Elimination**

1. Row 1 ($i = 1$)

$$x_1 = \frac{r_1}{a_1}, \quad \alpha_1 = \frac{c_1}{a_1}. \quad (11.7.1)$$

2. Rows $i = 2, \dots, n - 1$. It is convenient to define a temporary local variable γ .

$$\gamma = a_i - b_i \alpha_{i-1}, \quad x_i = \frac{r_i - b_i x_{i-1}}{\gamma}, \quad \alpha_i = \frac{c_i}{\gamma}. \quad (11.7.2)$$

- **Backsubstitution**

1. Final row ($i = n$).

$$x_n = \frac{r_n - b_n \beta_{n-1}}{a_n - b_n \alpha_{n-1}}. \quad (11.7.3)$$

2. Rows $i = n - 1, \dots, 1$.

$$\begin{aligned} x_{n-1} &:= x_{n-1} - \alpha_{n-1} x_n, \\ &\vdots \\ x_i &:= x_i - \alpha_i x_{i+1}, \\ &\vdots \\ x_1 &:= x_1 - \alpha_1 x_2. \end{aligned} \quad (11.7.4)$$

11.8 Diagonal dominance

- A square matrix M (not necessarily tridiagonal) is called **strongly diagonally dominant** if for every row the magnitude of the diagonal element exceeds the sum of the amplitudes of all the other elements in that row

$$|m_{ii}| > \sum_{j \neq i} |m_{ij}| \quad (i = 1, \dots, n). \quad (11.8.1)$$

- Then **weak diagonal dominance** means

$$|m_{ii}| \geq \sum_{j \neq i} |m_{ij}| \quad (i = 1, \dots, n). \quad (11.8.2)$$

- Some authors use the term “**diagonal dominance**” without a “strong” or “weak” qualifier. In that case they mean strong diagonal dominance. We shall append the qualifier “strong” or “weak” in these lectures.
- A tridiagonal matrix is (strongly) diagonally dominant if for every row

$$|a_i| > |b_i| + |c_i| \quad (i = 1, \dots, n). \quad (11.8.3)$$

- If a tridiagonal matrix is strongly diagonally dominant, then a zero pivot cannot occur in the elimination process in Sec. 11.7.
- For many practical applications involving tridiagonal matrices, the matrix is strongly diagonally dominant. Hence a zero pivot does not occur.
- There is a related concept called **strong column diagonal dominance** where for every column the magnitude of the diagonal element exceeds the sum of the amplitudes of all the other elements in that column

$$|m_{jj}| > \sum_{i \neq j} |m_{ij}| \quad (j = 1, \dots, n). \quad (11.8.4)$$

11.9 Tridiagonal matrix: inverse

- The inverse of a tridiagonal matrix T is obtained by solving the equation

$$TX = I_{n \times n}. \quad (11.9.1)$$

- Each column of X is calculated using the algorithm in Sec. 11.7.
- The solution for X is the matrix inverse $T^{-1} = X$.
- In general, the inverse of a tridiagonal matrix is a full matrix, which has nonzero elements in every row and every column.

1. Consider the following 5×5 tridiagonal matrix (which is strongly diagonally dominant)

$$T = \begin{pmatrix} 1 & -0.25 & & & \\ 0.25 & 1 & -0.25 & & \\ & 0.25 & 1 & -0.25 & \\ & & 0.25 & 1 & -0.25 \\ & & & 0.25 & 1 \end{pmatrix}. \quad (11.9.2)$$

2. The inverse matrix is

$$T^{-1} = \begin{pmatrix} 0.944272 & 0.22291 & 0.0526316 & 0.0123839 & 0.00309598 \\ -0.22291 & 0.891641 & 0.210526 & 0.0495356 & 0.0123839 \\ 0.0526316 & -0.210526 & 0.894737 & 0.210526 & 0.0526316 \\ -0.0123839 & 0.0495356 & -0.210526 & 0.891641 & 0.22291 \\ 0.00309598 & -0.0123839 & 0.0526316 & -0.22291 & 0.944272 \end{pmatrix}. \quad (11.9.3)$$

- It is therefore a **bad idea** to solve the tridiagonal matrix equation $T\mathbf{x} = \mathbf{r}$ by computing the inverse and solving for \mathbf{x} via

$$\mathbf{x} = T^{-1}\mathbf{r}. \quad (11.9.4)$$

- The calculation of $T^{-1}\mathbf{r}$ requires n^2 computations.
- By contrast, the solving for \mathbf{x} by applying the tridiagonal algorithm to the equation $T\mathbf{x} = \mathbf{r}$ requires $O(n)$ computations.

11.10 Tridiagonal matrix: determinant

- The determinant of a tridiagonal matrix can be computed by solving a recurrence.
- Using eq. (11.4.1), let us define Δ_{n-i} to be the determinant of the bottom right $(n-i) \times (n-i)$ tridiagonal matrix.
- Hence we wish to compute $\det(T) = \Delta_n$.
- The recurrence is

$$\Delta_{n-i} = a_{i+1}\Delta_{n-i-1} - b_{i+2}c_{i+1}\Delta_{n-i-2} \quad (i = 0, \dots, n-3). \quad (11.10.1)$$

- The initial values are

$$\Delta_1 = a_n, \quad \Delta_2 = a_{n-1}a_n - b_nc_{n-1}. \quad (11.10.2)$$

- The recurrence can be computed in $n-3$ steps to obtain $\det(T)$.
- The determinant of the matrix in eq. (11.9.2) is 1.26172 to five decimal places.

11.11 C++ code

- Examples of working C++ functions to implement the tridiagonal algorithm are given below.
- Note that the indexing of all the arrays follows the C/C++ convention.
- **Hence the indices run from 0 through $n - 1$ for an array of length n .**

11.11.1 C++ code: Tridiagonal algorithm

```
int Tridiagonal_solve(const int n,
                     const std::vector<double> & a,
                     const std::vector<double> & b,
                     const std::vector<double> & c,
                     const std::vector<double> & rhs,
                     std::vector<double> & x)
{
    const double tol = 1.0e-14;
    x.clear();
    if ((n < 1) || (a.size() < n) || (b.size() < n) || (c.size() < n)
        || (rhs.size() < n)) return 1;    // fail
    double alpha[n]; // temporary storage

    x.resize(n, 0.0);

    // initial equation i = 0
    int i = 0;
    double gamma = a[i];
    if (std::abs(gamma) <= tol) return 1; // fail
    x[i] = rhs[i]/gamma;
    alpha[i] = c[i]/gamma;

    // forward pass: elimination
    for (i = 1; i < n-1; ++i) {
        gamma = a[i] - b[i]*alpha[i-1];
        if (std::abs(gamma) <= tol) return 1; // fail
        x[i] = (rhs[i] - b[i]*x[i-1])/gamma;
        alpha[i] = c[i]/gamma;
    }

    // solve final equation i = n-1
    i = n-1;
    gamma = a[i] - b[i]*alpha[i-1];
    if (std::abs(gamma) <= tol) return 1; // fail
    x[i] = (rhs[i] - b[i]*x[i-1])/gamma;

    // backward substitution
    for (i = n-2; i >= 0; --i) {
        x[i] -= alpha[i]*x[i+1];
    }
    return 0;
}
```

11.11.2 C++ code: Tridiagonal inverse

```
int Tridiagonal_inverse(const int n,
                        const std::vector<double> & a,
                        const std::vector<double> & b,
                        const std::vector<double> & c,
                        std::vector<std::vector<double>>> & inv_matrix)
{
    if (n < 1) return 1;    // fail

    for (int j = 0; j < n; ++j) {
        std::vector<double> rhs(n, 0.0);
        std::vector<double> x(n, 0.0);
        rhs[j] = 1.0;

        inv_matrix[j].clear();

        int rc = Tridiagonal_solve(n, a, b, c, rhs, x);
        if (rc) {
            for (j = 0; j < n; ++j) {
                inv_matrix[j].clear(); // fail, clear everything
            }
            return rc; // fail
        }

        for (int i = 0; i < n; ++i) {
            inv_matrix[i][j] = x[i];
        }
    }

    return 0;
}
```

11.11.3 C++ code: Tridiagonal determinant

```
int Tridiagonal_determinant(const int n,
                           const std::vector<double> & a,
                           const std::vector<double> & b,
                           const std::vector<double> & c,
                           double & det)
{
    det = 0;
    if (n < 1) return 1;    // fail

    double F[n];    // temporary storage
    F[0] = a[n-1];
    if (n == 1) {
        det = F[n-1];
        return 0;
    }
    F[1] = a[n-2]*a[n-1] - b[n-1]*c[n-2];
    if (n == 2) {
        det = F[n-1];
        return 0;
    }

    for (int j = 2; j < n; ++j) {
        F[j] = a[n-1-j]*F[j-1] - b[n-j]*c[n-1-j]*F[j-2];
    }
    det = F[n-1];
    return 0;
}
```