一类椭圆型偏微分方程离散格式的求解

信息计算科学专业：康国胜　　指导老师：谭敏

摘要：本文针对经典椭圆型偏微分方程的有限元离散系统 Ax=b，讨论了该离散系统在不同存储格式下直接法和迭代法的求解问题。采用的存储格式包括：满矩阵存储格式、带宽存储格式和按行压缩稀疏存储格式。并分别用直接法（以高斯循序消去法为例）和迭代法（以 Gauss-seidel 迭代法为例）实现了几种存储格式下的线性代数方程组的算法求解，然后用该问题的求解来进行说明和比较，并利用C++语言实现了算法。最后对各种存储格式算法求解的优劣进行了分析。数值实验表明：在多种存储格式中，带宽存储格式的存储量大量少于满矩阵存储格式，且带宽存储格式耗时比满矩阵存储格式要少很多。而一维变带宽存储虽然比等带宽少占了一些内存，但计算过程中寻找元素较二维等带宽复杂，占用机时多，因此，两种方法的利弊要通盘考虑。通常当带宽变化不大，计算机内存又允许，采用等带宽存储还是合适的。特别地，在迭代法中，按行压缩稀疏存储格式在存储量和计算时间上比满矩阵和带宽存储均是最少的。故数值算例证明在迭代法中，按行压缩稀疏存储格式在时间和存储上都较为占优，可靠高效，能够较好地应用于有限元线性方程组的迭代求解。

关键词：满矩阵；带宽；按行压缩稀疏存储；高斯消元；Gauss-seidel 迭代

To Solve the Discrete Format of a Type of Elliptic Differential Equations

Calculation of Science and Technology Information: Kang Guosheng　　Instructor: Tan Min

Abstract: This paper aims at the finite element discrete systems of classic elliptic differential equations :Ax = b, discussed the discrete systems in different storage formats to solve the problem through the direct method and iterative method. The storage formats include full matrix storage format, bandwidth storage format and the compressed sparse storage format according to row. And by means of the direct method (Gaussian gradual elimination method as an example) and iterative method (Gauss-seidel iterative method for example), the algorithm-solvings of several storage formats of linear algebraic equations have achieveed.And then use the problem of solving to explain and compare the pros and cons of these storage formats.At last use C + + language to achieve these algorithms. Finally, analysis the pros and cons of he various storage format algorithm of solving . Numerical experiments show that in a variety of storage formats, memory spaces and time-consuming of bandwidth storage format are less than the full matrix storage format's greatly,and the memory spaces of one-dimensional variable bandwidth storages ormat are less than the equiband storage format's,but the process of calculating to find elements is more complex than the two-dimensional equiband, occupying for more time.So the pros and cons of the two methods should take into consideration. When the bandwidths is usually little changed and the computer's memory capacity allowed, using equiband storage format is appropriate. In particular, in the iterative method, the compressed sparse storage format according to row is the best than the full matrix storage and bandwidth in storage and computing time. The numerical experiments show that the compressed sparse storage format according to row is dominant in time and storage.It is efficient and reliable, so it can be used to solve the linear equations of finite element in the iterative method preferably.

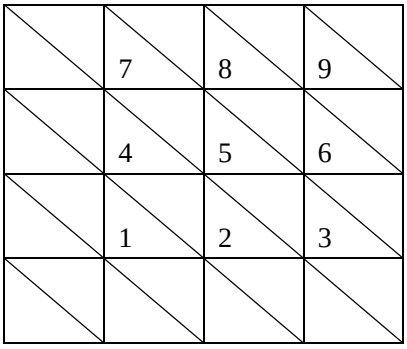
Key words: full matrix; bandwidth; the compressed sparse storage according to row; Gauss elimination; Gauss-seidel iterative

一　椭圆型偏微分方程及其离散

考虑经典的椭圆型偏微分方程边值问题：

$$\begin{cases} -\Delta u = f, (\mathrm{x},\ \mathrm{y}) \in \Omega \\ u\big|_{\Gamma} = 0,\ (x,\ y) \in \Gamma \end{cases} \qquad (1)$$

其中Ω=[0,1;0,1]，Γ是Ω的边界。右端 $f = 2\pi^2 \sin \pi x \sin \pi y$，其真解为：$u = \sin \pi x \sin \pi y$。本文中网格剖分均采用三角形剖分（如图（1），剖分为 4 × 4 时）。其中 1～9 为求解域内部节点的局部编号。



图（1）

对（1）采用有限元离散格式，得到一个线性代数系统 Ax=b，其系数矩阵通常具有对称、带状分布且稀疏的特点。这种对称分为两种：元素对称（即$a_{ij} = a_{ji}$）和位置对称（即若$a_{ij} \neq 0$，则有$a_{ji} \neq 0$，但不一定有$a_{ij} = a_{ji}$）两种。本文主要研究了元素对称的情形，对位置对称的情形稍加修改即可。以下若无特别说明均指元素对称的情形。对问题（1），当剖分为 4 × 4 时，其系数矩阵为：

$$A=\begin{bmatrix} 8 & -2 & 0 & -2 & & & & & \\ -2 & 8 & -2 & 0 & -2 & & & & \\ 0 & -2 & 8 & 0 & 0 & -2 & & & \\ -2 & 0 & 0 & 8 & -2 & 0 & -2 & & \\ & -2 & 0 & -2 & 8 & -2 & 0 & -2 & \\ & & -2 & 0 & -2 & 8 & 0 & 0 & -2 \\ & & & -2 & 0 & 0 & 8 & -2 & 0 \\ & & & & -2 & 0 & -2 & 8 & -2 \\ & & & & & -2 & 0 & -2 & 8 \end{bmatrix}$$

其中 b= $(1.052076,1.572131,1.171253,1.572131,2.223330,1.572131,1.171253,1.572131,1.052076)^{T}$，是采用做了边界处理以后得到的。

二　对不同存储格式的线性代数方程组的求解

本文主要针对具有上述特点的稀疏矩阵进行求解。对线性方程组来说，不同的存储格式和不同的算法有不同的效率。本文分别以直接法中的高斯循序消去法和迭代法中的 Gauss-seidel 迭代法对上述一类椭圆型偏微分方程的边值问题离散格式的线性代数方程组的多种存储格式进行算法求解。

线性代数方程组的解法可以分为两大类，即直接解法和迭代解法。直接解法的特点是，选定某种形式的直接解法以后，对于给定的线性代数方程组，事先可以按规定的算法步骤计算出它需要的算术运算操作数，直接给出最后的结果；迭代解法的特点是，对于一个给定的线性代数方程组，首先假设一个初始解，然后按一定的算法公式进行迭代。在每次迭代过程中对解的误差进行检查，并通过增加迭代次数不断降低解的误差，直至满足解的精度要求，并输出最后的解答。

2.1 矩阵的存储格式及举例

2.1.1 满矩阵存储格式

在一般的线性方程组的计算方法中，采用满矩阵存储的方式。即采用一个二维数组依次存储矩阵的每一个元素。若矩阵的阶数为 n，则可用二维数组 a 来存储，其中数组的界为 n×n。每个 a(ij)存储一个元素。若每个元素占一个存储单元，则在计算机中需要 n×n 个单元来存储。矩阵元素与数组元素的一一对应关系如图 2.1.1.1 所示。

$$\begin{bmatrix} a_{11}\,a_{12}\cdots a_{1n} \\ a_{21}\,a_{22}\cdots a_{2n} \\ \cdots\ \cdots\ \cdots\ \cdots \\ a_{n1}\,a_{n2}\cdots a_{nn} \end{bmatrix} \Longrightarrow \begin{bmatrix} a(11) & a(12) & \cdots & a(1,\ n-1) \\ a(21) & a(22) & \cdots & a(1,\ n-1) \\ \cdots & \cdots & \cdots & \cdots \\ a(n-1,1) & a(n-1,1) & \cdots & a(n-1,\ n-1) \end{bmatrix}$$

图 2.1.1.1

2.1.2 半阵的带宽存储格式

半阵即矩阵的一半。有限元法中，线性方程组的系数矩阵是对称的，因此可以只存储一个上三角（或下三角）矩阵，但是由于矩阵的稀疏性，仍然会发生零元占绝大多数的情况。考虑到非零元素的分布呈带状的特点，在计算机中系数矩阵的存储一般采用二维等带宽存储或一维变带宽存储。

2.1.2.1 二维等带宽存储

对于 n 阶的系数矩阵，若取最大的半带宽（半带宽即为从主对角元素开始直至最右的非零元素为止的所有元素个数，即该列号最大的非零元素为止的宽度。）D 为带宽，则上三角阵中的全部非零元素都将包括在这条以主对角元素为一边的一条等宽带中，如图 2.1.2.1 所示。二维带宽存储就是将这样一条带中的元素，以二维数组，如图 2.1.2.2 的形式存储在计算机中，二维数组的界是 n×D，其中右下三角形内的元素全部是零（见如图 2.1.2.2），有 D×(D-1)/2 个。

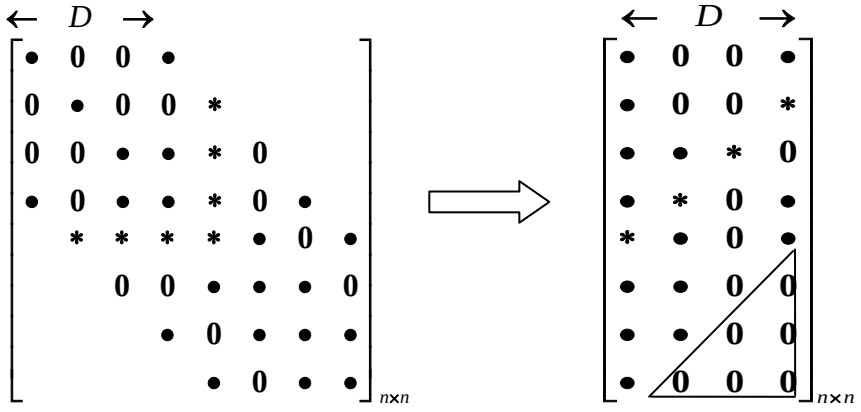


图 2.1.2.1　　图 2.1.2.2

以上两图是方阵存贮和等带宽存贮位置对应关系关系。我们以具体的例子来说明种存储是如何进行的。图 2.1.2.3 为一个 8×8 的系数矩阵，它的最大带宽 D=4。将每行在带宽的元素按行置于 8×4 的二维数组中，图 2.1.2.4 表示的系数矩阵在二维数组中的实际位置。图 2.1.2.5 表示的是元素在二维

数组中的编号。可以看到，由于对角元素都排在二维数组的第一列，因此二维数组中元素的列数都较原来的列数有一错动，而行则保持不变。若将元素原来的行、列码记为 i、j，它在二维数组中新的行、列码记为i^*、j^*，则有：

$$i^* = i$$

$$j^* = j - i + 1 \qquad (2.1.2.1)$$

即有表 2.1.2.1。

存储方式	行号	列号
方阵存储	i	j
等带宽存储	i	j-i+1

表 2.1.2.1 方阵存贮和等带宽存储地址关系

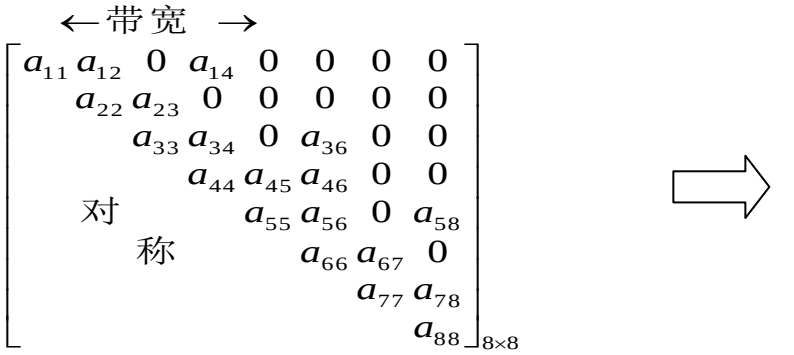


图 2.1.2.3 系数矩阵

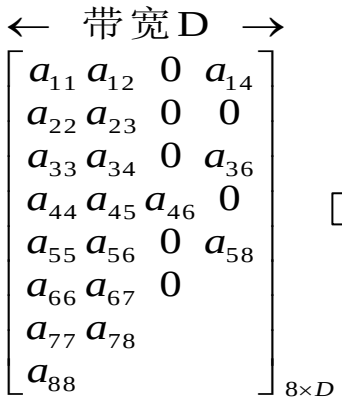


图 2.1.2.4

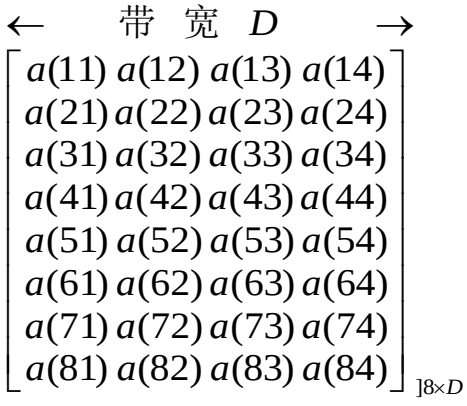


图 2.1.2.5 二维等带宽存储

如矩阵元素a_{67}在二维等带宽存储中应是 a(62)。

2.1.2.2 一维变带宽存储

一维变带宽存储就是将变化的带宽内的元素按一定的顺序存储在一维数组 a 中。由于它不按最大带宽存储，因此较二维等带宽存储更能节省内存，按照解法可分为按行的一维变带宽存储及按列的一维变带宽存储。现在仍旧利用图 2.1.2.3，按照一般习惯，进行按行的一维变带宽存储。

按行一维变带宽存储是按行依次存储元素，每行应从主对角元素直至最右的非零元素，即该列号最大的非零元素为止，这是迭代法的一维变带宽存储格式。高斯循序消去法一维变带宽存储格式稍有

不同，它的一维变带宽存储可能要稍多存储一些元素，虽然也是按行依次存储，但每行中从主对角元素直至前面行（包括当前行）最右的非零元素中列号最大的非零元素为止。按行存储对消元法是方便的，而按列存储适用于平方根法或其他解法。在一维变带宽存储中，由各行存储的元素中列号最大的元素组成的折线称为高度轮廓线（见图 2.1.2.6、图 2.1.2.7）。显然这种存储较二维等带宽存储少存了一些元素，但是对于高度轮廓线下的零元素（在迭代法的一维变带宽存储格式中亦即夹在非零元素内的零元素），如 a_{13}、a_{35} 等必须存储。图 2.1.2.6 中表示的是这些元素按行在迭代法中一维数组中的排列。图 2.1.2.7 是这些元素按行在消去法的一维数组中的排列。由图 2.1.2.3、图 2.1.2.4、图 3.1.2.5 及图 2.1.2.6 和图 2.1.2.7，可知二维等带宽与一维变带宽存储元素的对应关系。

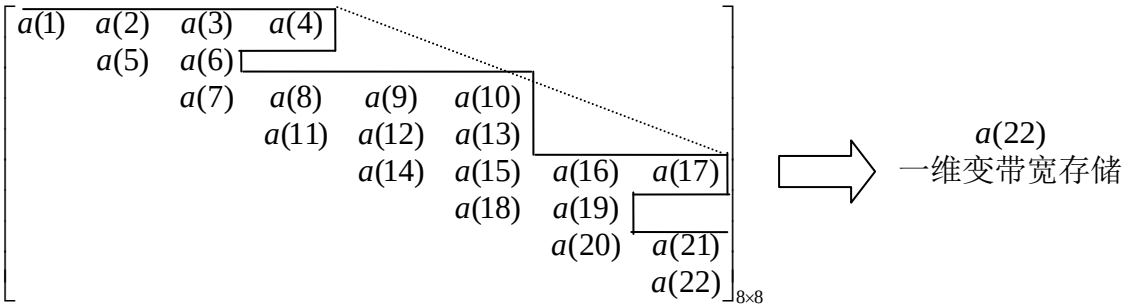


图 2.1.2.6 迭代法的按行一维变带宽存储

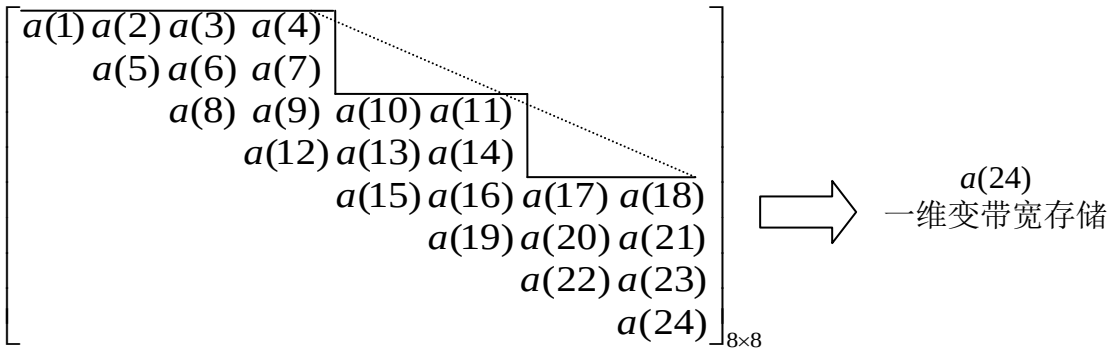


图 2.1.2.7 消去法的按行一维变带宽存储

将系数矩阵中的元素紧凑存储在一维数组中，必须有辅助的数组帮助记录原系数矩阵的性状。以迭代法的按行一维变带宽存储为例，例如对角元素的位置、每列元素的个数等。我们将辅助数组 M(n+1)，用以记录主对角元素在一维数组中的位置。对于图 2.1.2.6 的一维数组，它的 M(8+1) 数组是：

[1 5 7 11 14 18 20 22 23]

其中前 n 个数记录的是主对角元素的位置，最后一个数是一维数组长度加 1（在消去法的按行变带宽存储中最后一个不用存储，即元素 23 不用存储）。利用辅助数组 M，除了知道各主元在一维数组中的位置以外，还可以用以计算每行元素的行宽 Ni，即每行元素的个数，以及每行非零元素存储的末尾列号 mi（在迭代法中亦即每行非零元素的末尾列号）。

$$\begin{aligned} Ni &= M(i+1) - M(i) \\ mi &= i + Ni - 1 \end{aligned} \qquad (2.1.2.2)$$

设 A 中任一元素 a_{ij} 在 a 中的位置（序号）为 IJ，则由图 2.1.2.6 或图 2.1.2.7 可知

$$IJ=M(i)+(j-i) \qquad (2.1.2.3)$$

例如求第 4 行元素个数 N4，则有

$$N4=M(5)-M(4)=14-11=3$$

求第 6 行元素的个数及非零元的末尾列号 N6 及 m6，应有

N6=M(7)-M(6)=20-18=2

m6=6+2-1=7

有了辅助数组 M 后，可以找到一维数组中相应的元素，然后进行方程组的求解。

2.1.3 按行压缩稀疏存储格式

按行压缩稀疏存储格式就是将变化的带宽内的元素按行的顺序存储在一维数组 a 中。由于它不按其本身的带宽存储，只存储对称部分的非零元素。因此较二维等带宽存储及一维变带宽存储更能节省内存，它适合于迭代法的求解。现在仍旧利用图 2.1.2.3，看看按行压缩稀疏存储格式的形式，如图 2.1.3.1。

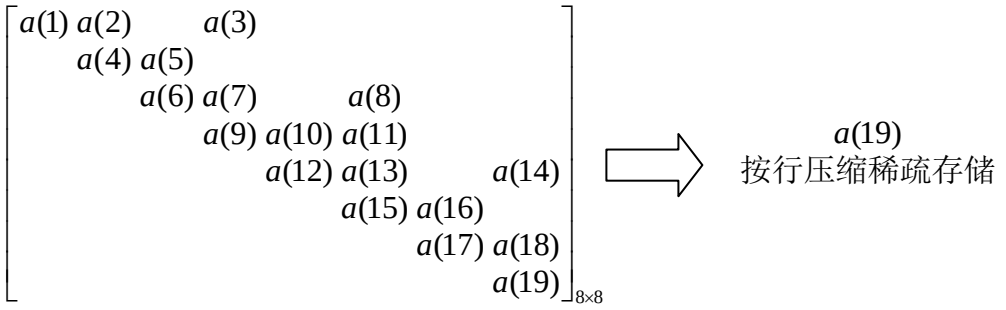


图 2.1.3.1 按行压缩稀疏存储

按行压缩稀疏存储是按行依次存储元素，每行应从主对角元素开始的第一个非零元直至最右的非零元素，即该列号最大的非零元素为止。而该行中夹在非零元素内的零元素不给予存储。如 a13、a35 等不给予存储。显然这种存储较二维等带宽存储及一维变带宽存储少存了一些元素，图 2.1.3.1 中表示的是这些元素按行在一维数组中的排列。由图 2.1.2.3、图 2.1.2.4、图 2.1.2.5 图 2.1.2.6 及图 2.1.3.1，可知其与二维等带宽和一维变带宽存储的元素对应关系。

同一维变带宽存储一样，将系数矩阵中的元素紧凑存储在一维数组中，必须有辅助的数组帮助记录原系数矩阵的性状。这里用辅助数组 ia(n+1)，ja。ia 用来记录每行中从对角线元素开始的第一个非零元在一维数组 a 的位置；数组 ja 的长度同数组 a 的长度一致，ja(i)用来记录 a(i)在原来矩阵中的列号。对于图 2.1.3.1 的一维数组，它的 ia(8+1)数组是：

[1 4 6 9 12 15 17 19 20]

其中前 n 个数记录的是每行中从对角线元素开始的第一个非零元在一维数组 a 的位置，最后一个是一维数组长度加 1。它的 ja 数组是：

[1 2 4 2 3 3 4 6 4 5 6 5 6 8 6 7 7 8 8]

利用辅助数组 ia、ja 可以计算每一行中第一个非零元的列号 Fi 和最后一个非零元的列号 Li。

$$\begin{cases} Fi = ja(ia(i)) \\ Li = ja(ia(i+1)+1) \end{cases} \qquad (2.1.3.1)$$

例如求第 4 行第一个非零元的列号 Fi，则有

Fi=ja(ia(4))=ja(9)=4

最后一个非零元的列号 Li，则有

Li=ja(ia(4+1)-1)=ja(11)=6

实际上，在迭代法种主对角线元素都不能为零，故每行第一个非零元就是对角元，即有 Fi=i。

有了辅助数组 ia、ja 后，可以找到一维数组中相应的元素，然后就可以进行方程组的迭代求解。

2.1.4 几种存储格式的举例

下面以问题（1）离散得到的一个 9×9 矩阵的线性代数方程组的求解为例来进行说明和比较。线性代数方程组如（2.1.4.1）式：

$$\begin{bmatrix} 8&-2&0&-2&0&0&0&0&0\\ -2&8&-2&0&-2&0&0&0&0\\ 0&-2&8&0&0&-2&0&0&0\\ -2&0&0&8&-2&0&-2&0&0\\ 0&-2&0&-2&8&-2&0&-2&0\\ 0&0&-2&0&-2&8&0&0&-2\\ 0&0&0&-2&0&0&8&-2&0\\ 0&0&0&0&-2&0&-2&8&-2\\ 0&0&0&0&0&-2&0&-2&8 \end{bmatrix}_{9\times9} \begin{pmatrix} x_1\\x_2\\x_3\\x_4\\x_5\\x_6\\x_7\\x_8\\x_9 \end{pmatrix} = \begin{pmatrix} 1.052076\\1.572131\\1.171253\\1.572131\\2.223330\\1.572131\\1.171253\\1.572131\\1.052076 \end{pmatrix} \quad (2.1.4.1)$$

几种储格式的图示及存储量如下：

$$\begin{bmatrix} 8&-2&0&-2&0&0&0&0&0\\ -2&8&-2&0&-2&0&0&0&0\\ 0&-2&8&0&0&-2&0&0&0\\ -2&0&0&8&-2&0&-2&0&0\\ 0&-2&0&-2&8&-2&0&-2&0\\ 0&0&-2&0&-2&8&0&0&-2\\ 0&0&0&-2&0&0&8&-2&0\\ 0&0&0&0&-2&0&-2&8&-2\\ 0&0&0&0&0&-2&0&-2&8 \end{bmatrix}_{9\times9}$$

图 2.1.4.1 满矩阵存储需 9×9=81 个单元

$$\begin{bmatrix} 8&-2&0&-2\\ 8&-2&0&-2\\ 8&0&0&-2\\ 8&-2&0&-2\\ 8&-2&0&-2\\ 8&0&0&-2\\ 8&-2&0&0\\ 8&-2&0&0\\ 8&0&0&0 \end{bmatrix}_{9\times4} \Rightarrow \begin{bmatrix} a(11)&a(12)&a(13)&a(14)\\ a(21)&a(22)&a(23)&a(24)\\ a(31)&a(32)&a(33)&a(34)\\ a(41)&a(42)&a(43)&a(44)\\ a(51)&a(52)&a(53)&a(54)\\ a(61)&a(62)&a(63)&a(64)\\ a(71)&a(72)&a(73)&a(74)\\ a(81)&a(82)&a(83)&a(84)\\ a(91)&a(92)&a(93)&a(94) \end{bmatrix}_{9\times4}$$

图 2.1.4.2 二维等带宽存储需 9×4=36 个单元

$$\begin{bmatrix} 8&-2&0&-2&&&&&\\ &8&-2&0&-2&&&&\\ &&8&0&0&-2&&&\\ &&&8&-2&0&-2&&\\ &&&&8&-2&0&-2&\\ &&&&&8&0&0&-2\\ &&&&&&8&-2&\\ &&&&&&&8&-2\\ &&&&&&&&8 \end{bmatrix}_{9\times9} \Rightarrow \begin{bmatrix} a(1)&a(2)&a(3)&a(4)&&&&&\\ &a(5)&a(6)&a(7)&a(8)&&&&\\ &&a(9)&a(10)&a(11)&a(12)&&&\\ &&&a(13)&a(14)&a(15)&a(16)&&\\ &&&&a(17)&a(18)&a(19)&a(20)&\\ &&&&&a(21)&a(22)&a(23)&a(24)\\ &&&&&&a(25)&a(26)&\\ &&&&&&&a(27)&a(28)\\ &&&&&&&&a(29) \end{bmatrix}_{9\times9}$$

图 2.1.4.3 一维变带宽存储（以迭代法的存储为例，消元法的存储需多存储一个元素 a_{79}）需 29 个单元

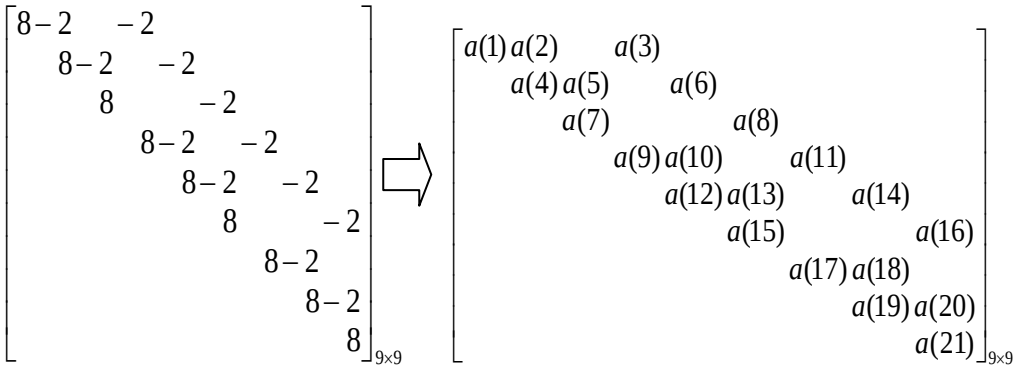


图 2.1.4.4 按行压缩稀疏存储需 21 个单元

对上述的例子，下面看三种存储格式的存储量，如下表（2.1.4.1）：

存储格式	满矩阵存储	带宽存储		按行压缩稀疏存储
		二维等带宽存储	一维变带宽存储	
存储需求单元数	81	36	29	21

表（2.1.4.1）

注：存储单元数中只包括系数矩阵数组的存储，因为在大型或超大型矩阵中元素的个数远远大于辅助数组的存储元素的个数，辅助数组的存储可以忽略不计，故不把它包括在存储单元数计算中。以下均同。

2.2 直接法对离散格式的求解

首先介绍高斯循序消去法，循序消去法是高斯消去法的基本形式。它的基本思想是通过初等变换消去方程组系数矩阵主对角线以下的元素，而使方程组化为等价的上三角方程组，再通过回代求出方程组的解。

下面讨论，一般的 n 阶线性方程组的高斯循序消去法。

记 Ax=b 为 $A^{(1)}$x=$b^{(1)}$，$A^{(1)}$ 和 $b^{(1)}$ 的元素分别记为 $a_{ij}^{(1)}$ 和 $b_i^{(1)}$，i，j=1，2，…，n，系数上标（1）代表第 1 次消元之前的状态。

第 k 次消元（2≤k≤n−1）时，设 k-1 次消元已完成，即有

$$A^{(k)}\text{x}=b^{(k)}$$

其中

$$A^{(k)}=\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & & \cdots & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & \cdots & a_{2n}^{(2)} \\ & & \ddots & & & \vdots \\ & & & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} \\ & & & \vdots & & \vdots \\ & & & a_{nk}^{(k)} & & a_{nn}^{(k)} \end{bmatrix} \quad b^{(k)}=\begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_k^{(k)} \\ \vdots \\ b_n^{(k)} \end{bmatrix} \qquad (2.2.1)$$

设 $a_{kk}^{(k)}\neq 0$，计算乘数

$$m_{ik}=\frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} \quad ，\ \text{i=k+1，…，n}$$

用-m_{ik}乘以第 k 个方程，加到第 i 个方程到第 n 个方程的未知数x_k，得到

$$A^{(k+1)}\text{x}=b^{(k+1)}$$

其中

$$\begin{cases} a_{ij}^{(k+1)}=a_{ij}^{(k)}-m_{ik}a_{kj}^{(k)} \\ b_i^{(k+1)}=b_i^{(k)}-m_{ik}b_k^{(k)} \end{cases} \qquad \text{i，j=k+1，…，n}$$

只要$a_{kk}^{(k)}\neq 0$，消元过程就可以进行下去，直到经过 n-1 次消元之后，消元过程结束，得

$$A^{(n)}\text{x}=b^{(n)}$$

或者写成

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ & & \ddots & \vdots \\ & & & a_{nn}^{n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(n)} \end{bmatrix}$$

这是一个与原方程组等价的上三角形方程组。把经过 n-1 次消元将线性方程组化为上三角形方程组的计算过程叫做消元过程。

当$a_{nn}^{(n)}\neq 0$ 时，对上三角形方程组自下而上逐步回代解方程组计算x_n，x_{n-1}，…，x_1，即

$$\begin{cases} x_n=\dfrac{b_n^{(n)}}{a_{nn}^{(n)}} \\ x_i=(b_i^{(i)}-\sum_{j=i+1}^{n}a_{ij}^{(i)}x_j)/a_{ii}^{(i)},\text{i}=\text{n-1},\cdots,1 \end{cases}$$

$a_{kk}^{(k)}$（k=1，2，…，n）称为各次消元的主元素；m_{ik}（k=1，2，…，n-1，i=k+1，…，n-1，n）称为各次消元的乘数，主元素所在的行称为主行。

当用 k 表示消元过程的次序时，高斯消去的计算步骤：

1）消元过程：

设$a_{kk}^{(k)}\neq 0$，对 k=1，2，…，n-1 计算

$$\begin{cases} m_{ik}=a_{ik}^{(k)}/a_{kk}^{(k)} \\ a_{ij}^{(k+1)}=a_{ij}^{(k)}-m_{ik}a_{kj}^{(k)} \\ b_i^{(k+1)}=b_i^{(k)}-m_{ik}b_k^{(k)} \end{cases} \qquad (2.2.2)$$

i，j=k+1，k+2，…，n

2）回代过程：

$$\begin{cases} x_n=b_n^{(n)}/a_{nn}^{(n)} \\ x_i=(b_i^{(i)}-\sum_{j=i+1}^{n}a_{ij}^{(i)}x_j)/a_{ii}^{(i)} \end{cases} \qquad \text{i=n-1，…，2，1}$$

为了使公式看起来简洁，把（n-1）轮消元后所得到的最后方程式去掉上标（k）（k=1～n），有如下形式：

$$\begin{cases} x_n = b_n / a_{nn} \\ x_i = (b_i - \sum_{j=i+1}^{n} a_{ij} x_j) / a_{ii} \end{cases} \qquad (2.2.3)$$

i=n-1，…，2，1；j=i+1，…，n

综上所述画出了高斯循序消去法的算法框图如图 2.2.1 所示。

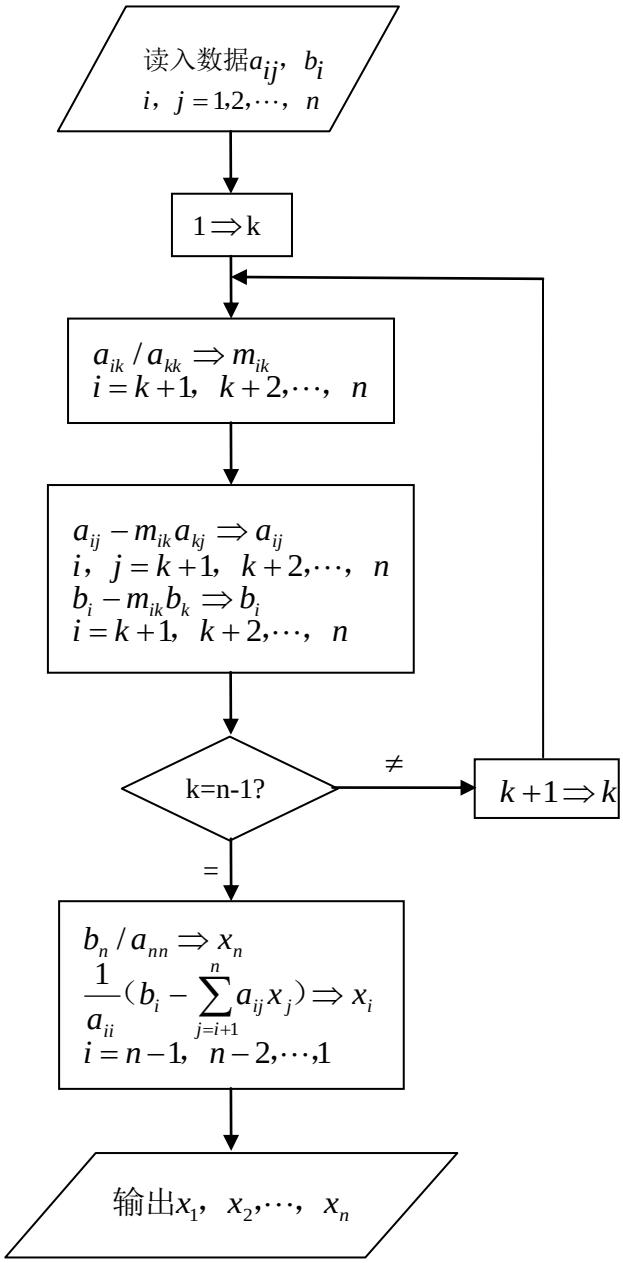


图 2.2.1 高斯循序消去法算法框图

因为对上述离散格式的线性代数方程组的系数矩阵 A 为严格对角占优矩阵，即

$$|a_{ii}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} \{a_{ij}\}，\ i=1，2，\cdots，n \qquad （2.2.4）$$

故可适用于高斯循序消去法的求解。

若 A 具有对称性，可以采用半阵存储，即只存主对角线以上的元素。因为$a_{ij}=a_{ji}$，可将（2.2.2）式的消元过程修改如下：

$$\begin{cases} a_{ij}^{(k)} = a_{ij} - \dfrac{a_{ki}}{a_{kk}} a_{kj} \\ b_i^{(k)} = b_i - \dfrac{a_{ki}}{a_{kk}} b_k \end{cases} \qquad （2.2.5）$$

k=1，2，…，n-1；i，j=k+1，k+2，…，n

（2.2.5）式等号右边项去掉了上标（k-1），（2.2.3）式不修改。

2.2.1 满矩阵存储格式的高斯循序消去解法

高斯循序消去法的分量和前面的介绍一致。计算步骤：消元过程（2.2.2），回代过程（2.2.3）。元素的查找按数组元素的下标及图 2.1.1.1 的对应关系，即可直接获得。即若将元素原来的行、列码记为 i、j，它在二维数组中新的行、列码记为i^*、j^*，则有：

$$\begin{cases} \mathrm{i}^* = \mathrm{i} \\ \mathrm{j}^* = \mathrm{j} \end{cases} \qquad (2.2.1.1)$$

故a_{ij}=a(i^*，j^*)，可直接运用上述的高斯循序消去法求解，算法框图同图 2.2.1。

2.2.2 二维等带宽存储格式的高斯循序消去解法

设矩阵 A 的二维等带宽存储的等带宽矩阵为A^*，只要找元素在 A 和A^*中的对应关系，等带宽高斯循序消去法公式的推导就迎刃而解了。

1．元素在 A 和A^*中的对应关系

由(2.1.2.1)式和图 2.2.2.1 可见，两个矩阵的元素及列码对应关系如下（行码不变）：

$a_{ij} \rightarrow a_{iJ}^*$　　J=j-i+1　（j=J+i-1）

$a_{kk} \rightarrow a_{k1}^*$

$a_{ki} \rightarrow a_{kL}^*$　　L=i-k+1

$a_{kj} \rightarrow a_{kM}^*$　　M=j-k+1=J+i-k　（J 为循环变量）

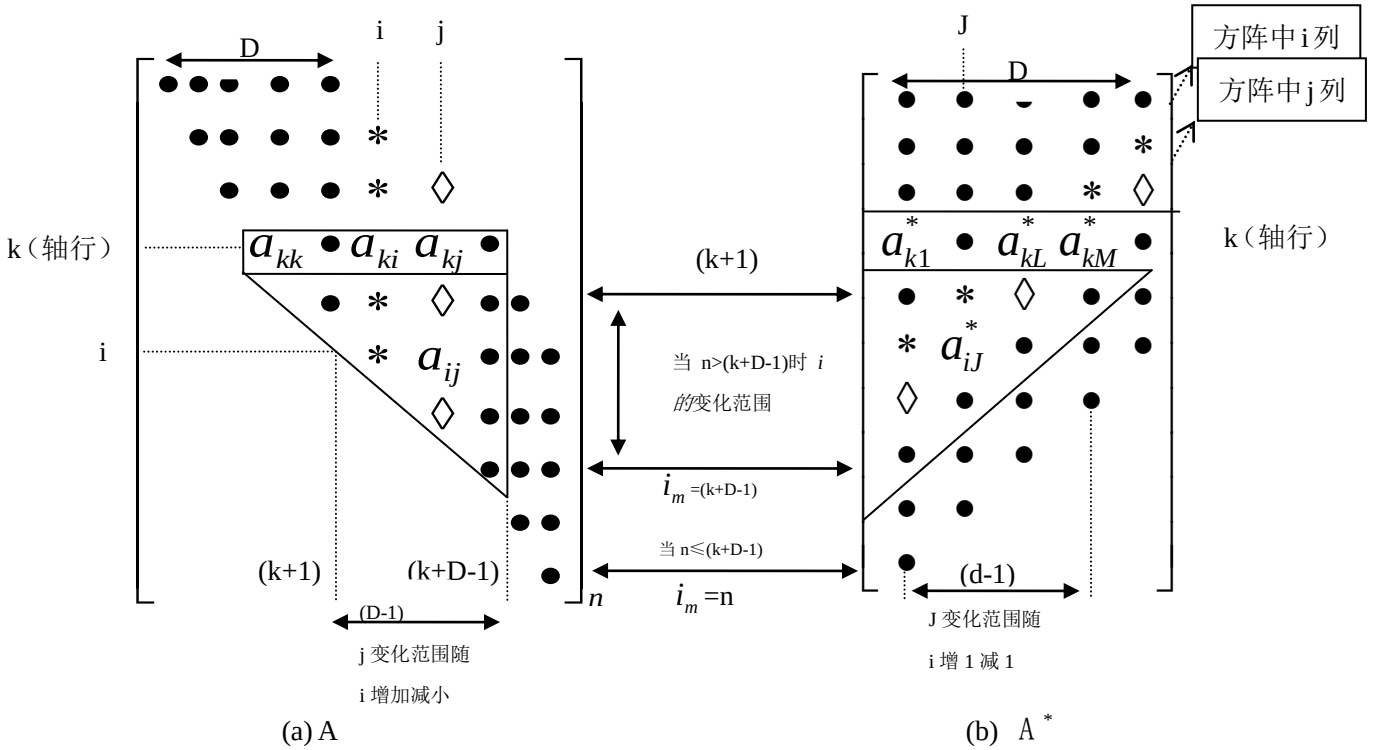


图 2.2.2.1 A 与 A^* 中元素和行列玛对应关系

这样对应后，消元过程的循环码做相应的修改，如下：

（1）消元轮次码 k 不变，其取值范围仍为（1～（n-1））。

（2）消元码 i 不变，但其取值范围发生了变化。这是因为在消元过程中变化的系数只局限在三角形（见图 2.2.2.1）所框的范围内，i 的取值范围是（k+1）～i_m，而 i_m 的取值又分两种情况。当（k+D-1）＜n 时（即图 2.2.2.1 中的 n＞（k+D-1）），i_m=(k+D-1)，而当（k+D-1）≥n 时（即图 2.2.2.1 中的 n≤（k+D-1）），i_m=n。

（3）消元列码 j 改变为 J。J 的取值范围是 1～J_m，而 J_m 是随行码 i 增加而减小，由下式计算：J_m=D-(i-k)。在程序中可用 J_m=D-L+1(因为 L=i-k+1，i-k=L-1)。

2．消元公式的修改

对（2.2.5）式修改如下：

$$\begin{cases} a_{iJ}^{*(k)} = a_{iJ}^{*} - \dfrac{a_{kL}^{*}}{a_{k1}^{*}} a_{kM}^{*} \\ b_i^{(k)} = b_i - \dfrac{a_{kL}^{*}}{a_{k1}^{*}} b_k \end{cases} \qquad (2.2.2.1)$$

k=1，2，…，n-1；

i=k+1，k+2，…，$\begin{cases} i_m = k+d-1 \\ i_m = n \end{cases}$；

J=1，2，…，J_m=D-L+1

在程序中，将（2.2.2.1）式中的上标*和(k)都去掉。

3．回代公式的修改：

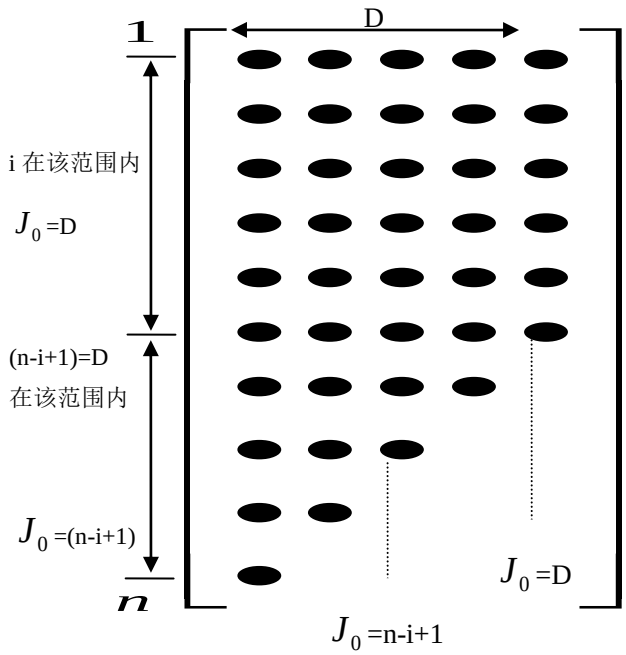
对（2.2.3）式修改如下：

$$\begin{cases} x_n = b_n / a_{n1} \\ x_i = (b_i - \sum_J a_{iJ} x_H) / a_{k1} \end{cases} \qquad (2.2.2.2)$$

i=n-1，n-2，…，1；

J=2，3，…，$\begin{cases} J_0 = n-i+1 \\ J_0 = D \end{cases}$

式中 H=J+i-1



图（2.2.2.2） 回代公式A^*的最大列码

说明：

（1）x_H在（2.2.2.2）式中为x_j，由 j 与 J 的对应关系 j=J+i-1，所以，此处 H=J+i-1 是以 H 取代 j，避免重复。

（2）有图（2.2.2.2）可见，列号 J 由 2 开始，但其截止码为J_0。当(n-i+1)≥D 时，J_0=D；当(n-i+1)＜D，J_0=（n-i+1）

综上所述画出了等带宽存储的高斯循序消去法算法框图如图 2.2.2.3 所示。

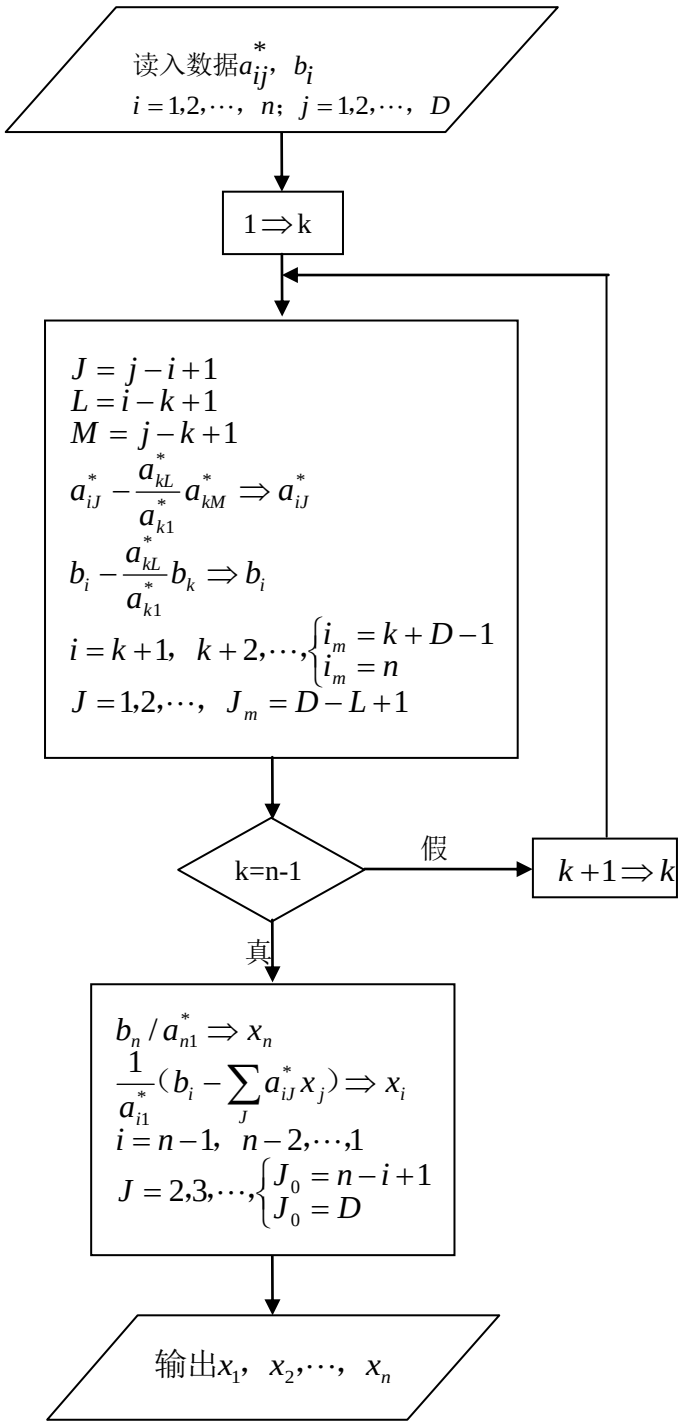


图 2.2.2.3 等带宽存储的高斯循序消去法算法框图

2.2.3 一维变带宽存储格式的高斯循序消去解法

一维变带宽存储的高斯循序消去法计算公式，可以由半阵存储的高斯消元公式（2.2.5）修改而得（如图 2.2.3.1）。

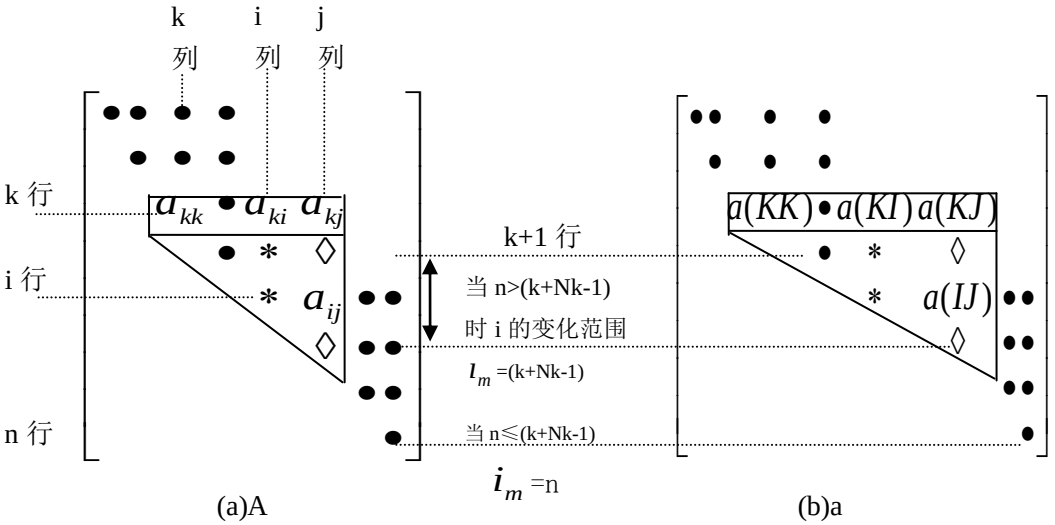


图 2.2.3.1 A 与 a 中元素的对应关系

1．元素的对应关系

由图 2.2.3.1 可见，公式(2.1.2.3)，（2.2.5）中的个元素与 a 中元素的对应关系如下：

a_{ij} ＝a(IJ)　　IJ=M(i)+j-i

a_{kk} =a(KK)　　KK=M(k)

a_{ki} =a(KI)　　KI=M(k)+i-k

a_{kj} =a(KJ)　　KJ=M(k)j-k

2．修改后的公式

（1）消元公式

$$\begin{cases} a(IJ)=a(IJ)-\dfrac{a(KI)}{a(KK)}\bullet a(KJ) \\ b_i=b_i-\dfrac{a(KI)}{a(KK)}\bullet b_k \end{cases} \qquad (2.2.3.1)$$

（k=1，2，…，（n-1）；i=(k+1)，…，i_m；j=i，（i+1），…，j_m）

（2）回代公式

$$x_n=b_n\ /\ a(NN) \qquad NN=M(n) \qquad (2.2.3.2)$$

$$x_i=(b_i-\sum_{j=i+1}^{j_m}a(IJ)x_j)\ /\ a(II) \qquad (2.2.3.3)$$

(i=(n-1)，(n-2)，…，1；j=(i+1)，…，j_0；II=M(i))

综上所述画出了半带宽存储的高斯循序消去法算法框图如图 2.2.3.2 所示。

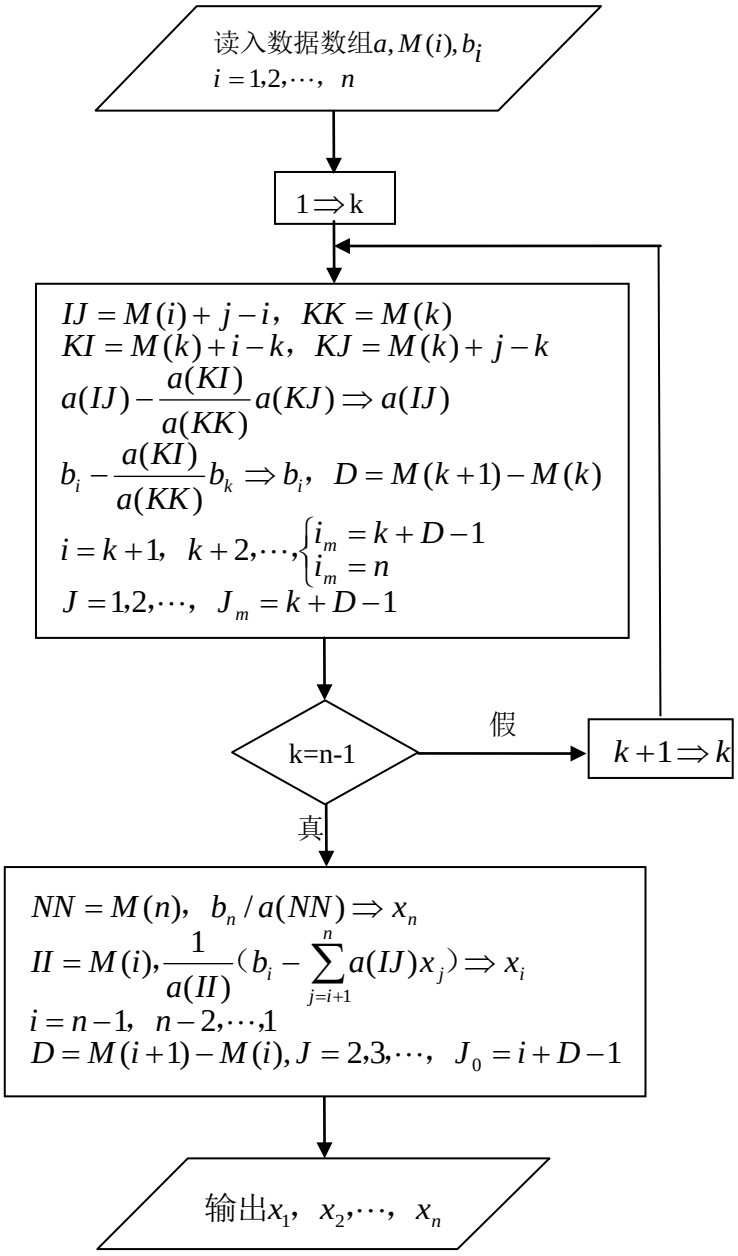


图 2.2.3.2 半带宽存储的高斯循序消去法算法框图

(3)循环码的说明

因为 a 中元素的位置可以由(2.1.2.3)式算出，所以编程时仍用 k，i，j 做循环码，只是 k，i，j 的循环范围要加以修改。修改的方法可参考等带宽高斯循序消去法的公式（2.2.2.1）和（2.2.2.2)。

“等带宽”和“变带宽”的主要区别在于 A 中每一行存储的元素个数是否相等，只要将图 2.2.2.1 与图 2.1.2.7 加以比较便可知。由图 2.2.2.1 可见，等带宽存储消元工作在三角行围内进行，变化的元素有(D-1)行，但每一行要变化的元素的列数都不同；最上行有(D-1)列，下面各行是每增加一行减少一列。因为是等带宽存储，消元过程中三角行的大小（即三角形内元素的行、列数）是不变的，即半带宽 D 为常数。而在图 2.1.2.7 中，每一行的带宽（D=Nk，Nk 的值由式(2.1.2.2)得到，表示第 k 行的带宽）是不同的，因此，不同的行其消元范围三角形的大小是不一样的。此时，三角形有（Nk－1）行；最上一行有（Nk－1）列，下面各行，每增加一行前边减少一列。所以得到(2.2.3.1)式的循环码。

行 k 变化范围与（2.2.2.1）式一样，不再多讲。消元行 i 的变化范围从（k+1）行到i_m行。i_m的值也可以按两种情况去取，当（k+Nk-1）＜n 时，i_m=（k+Nk-1），当（k+Nk-1）≥n 时，i_m=n(这里只是将（2.2.2.1）式中的 D 改为 Nk)。消元列号 j 的变化较复杂，由图 2.2.3.1 可见，（k+1）行由（k+1）列至j_m列，（k+2）行由（k+2）列至j_m，…，恰 i 的变化规律为（k+1），（k+2），所以起始列取 i，而j_m= k+Nk-1。

因为消元完了各元素仍存在原来的位置上，回代仍按图 2.2.3.1(a)来进行，所以(2.2.3.3)式中的j_0= k+Nk-1。

2.2.4 小结

下面对几种存储格式的存储，按高斯循序消去法对问题（1）中离散的线性代数方程组的求解来进行说明和比较，线性代数方程组如（2.2.4.1）式：

$$\begin{bmatrix} 8 & -2 & 0 & -2 & 0 & 0 & 0 & 0 & 0 \\ -2 & 8 & -2 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & -2 & 8 & 0 & 0 & -2 & 0 & 0 & 0 \\ -2 & 0 & 0 & 8 & -2 & 0 & -2 & 0 & 0 \\ 0 & -2 & 0 & -2 & 8 & -2 & 0 & -2 & 0 \\ 0 & 0 & -2 & 0 & -2 & 8 & 0 & 0 & -2 \\ 0 & 0 & 0 & -2 & 0 & 0 & 8 & -2 & 0 \\ 0 & 0 & 0 & 0 & -2 & 0 & -2 & 8 & -2 \\ 0 & 0 & 0 & 0 & 0 & -2 & 0 & -2 & 8 \end{bmatrix}_{9\times 9} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} 1.052076 \\ 1.572131 \\ 1.171253 \\ 1.572131 \\ 2.223330 \\ 1.572131 \\ 1.171253 \\ 1.572131 \\ 1.052076 \end{pmatrix} \qquad (2.2.4.1)$$

下面来看三种存储格式的高斯循序消去法的计算时间（程序分别见附录）。

通过运行程序，得到近似解和计算时间，其中近似解为:

$x=(0.466984,0.670949,0.481881,0.670949,0.948865,0.670949,0.481881,0.670949,0.466984)^T$

再结合前面的表（2.1.4.1），我们由此例可以得出如下表 2.2.4.1：

存储格式	满矩阵存储	带宽存储	
		二维等带宽存储	一维变带宽存储
存储需求单元数	81	36	30
cpu 时间（秒）	0.010445	0.00756	0.009914

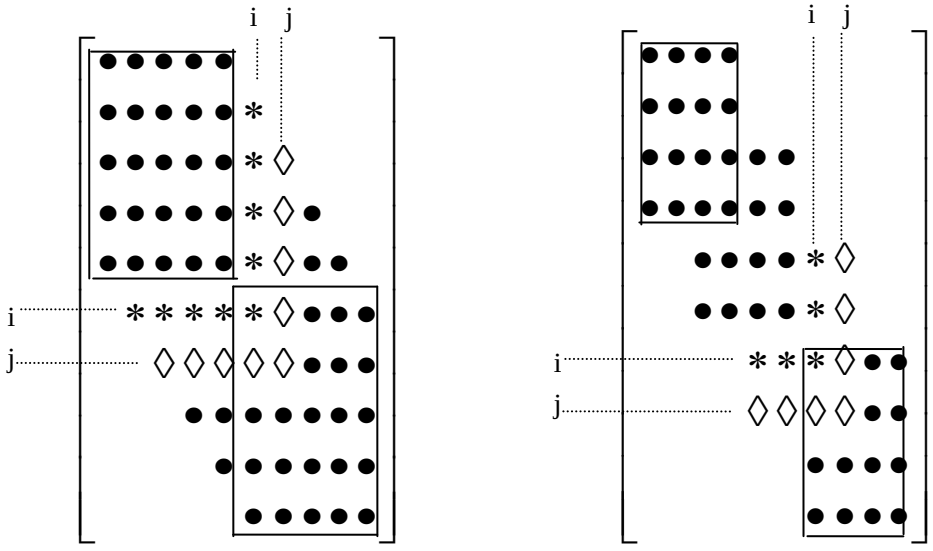
表 2.2.4.1

注：以上计算时间均为同一操作系统下循环调用 1000 次算法的平均时间。

数值实验表明：在高斯消元多种存储格式中，半阵存储格式的存储量和计算时间比满阵存储要少很多，而一维变带宽存储虽然比等带宽少占了一些内存，但消元过程中寻找元素较二维等带宽复杂，占用机时多，因此，两种方法的利弊要通盘考虑。通常当带宽变化不大，计算机内存又允许，采用等带宽存储还是合适的。

在半阵的带宽存储中，若对于只是位置对称的矩阵，可增加相应下三角的的存储（亦包括对角线元素），为使与原来矩阵元素对应关系和元素对称的矩阵保持一致。可将下三角先转置，然后再和元素对称的矩阵存储一致。例如：半带宽存储可再将下三角先转置后加存储于另一个半带宽矩阵 B 中；变带宽存储也可再下三角先转置后按行存储于另一个一维数组 b 中，使矩阵元素存储完整。同样用于

高斯循序消元法。但此时消元的特点不再是工作三角形，而是工作矩形。



(a)等带宽存储　　（b）变带宽存储

图（2.2.4.1）位置对称矩阵的带宽存储的高斯消元的特点——工作矩形

图（2.2.4.1）中分别画出了等带宽和变带宽的第 1 次和第 i 次消元的工作矩形。其消元过程可参考半阵带宽存储的公式，稍做变化。而回代过程是一样的，不必变化。显然容易实现，故不再多讲。其算法的优劣同上述分析一致的。

2.3 迭代法对离散格式的求解

首先，为介绍 Gauss-seidel 迭代法，我们先介绍一下 Jacobi 迭代法。

设有方程组：

$$Ax=b \qquad (2.3.1)$$

其展开形式为：

$$\begin{bmatrix} a_{11}\ a_{12}\cdots a_{1n} \\ a_{21}\ a_{22}\cdots a_{2n} \\ \cdots\ \cdots\cdots\cdots \\ a_{n1}\ a_{n2}\cdots a_{nn} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \qquad (2.3.2)$$

改写线性方程组（2.3.2）式，将第 i 个方程（i=1～n）表示为 x_i 的表达式：

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij} x_j \right) \quad （i=1，2，3，…，n） \qquad （2.3.3）$$

Jacobi 迭代法是由一组 x_i 的初值 x_i^0（i=1，2，…，n），然后按下式进行迭代：

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij} x_j^k \right) \quad \begin{matrix} (i=1,2,3,\cdots,n) \\ (k=0,1,2,3\cdots) \end{matrix} \qquad （2.3.4）$$

上标 k 代表迭代次数。上式还可以改写成更便于编程的如下形式，即

$$x_i^{k+1}=x_i^k+\frac{1}{a_{ii}}\left(b_i-\sum_{j=1}^{n}a_{ij}x_j^k\right)\qquad\begin{matrix}(i=1,2,3,\cdots n)\\(k=0,1,2,3\cdots)\end{matrix}\qquad(2.3.5)$$

迭代一直进行到满足精度要求为止。迭代的精度可采用以下准则进行检查和控制，即

$$\max_{1\le i\le n}\left|x_i^{k+1}-x_i^k\right|\le\varepsilon\qquad(k=0,1,2,3\cdots)\qquad(2.3.6)$$

式中 ε 是允许的误差。

由 Jacobi 迭代法可知，在计算 x^{k+1} 的过程中，采用的都是上一迭代的结果 x^k 。考察其计算过程，显然在计算新值的分量 x_i^{k+1} 时，已经计算得到了新的分量，x_1^{k+1}，x_2^{k+1}，…，x_{i-1}^{k+1}。有理由认为新计算出来的分量可能比上次迭代得到的分量有所改善。希望充分利用新计算出来的分量以提高迭代解法的效率，这就是 Gauss-seidel 迭代法。

对（2.3.4）式进行改变可以得到 Gauss-seidel 迭代法的分量形式：

$$x_i^{k+1}=\frac{1}{a_{ii}}\left(b_i-\sum_{j=1}^{i-1}a_{ij}x_j^{k+1}-\sum_{j=i+1}^{n}a_{ij}x_j^k\right)\qquad\begin{matrix}(i=1,2,3,\cdots,n)\\(k=0,1,2,3\cdots)\end{matrix}\qquad(2.3.7)$$

Gauss-seidel 迭代法的分量形式亦可表示为：

$$x_i^{k+1}=x_i^k+\frac{1}{a_{ii}}\left(b_i-\sum_{j=1}^{i-1}a_{ij}x_j^{k+1}-\sum_{j=i}^{n}a_{ij}x_j^k\right)\qquad\begin{matrix}(i=1,2,3,\cdots,n)\\(k=0,1,2,3\cdots)\end{matrix}\qquad(2.3.8)$$

综上所述画出了 Gauss-seidel 迭代法算法框图如图 2. 3.1 所示。

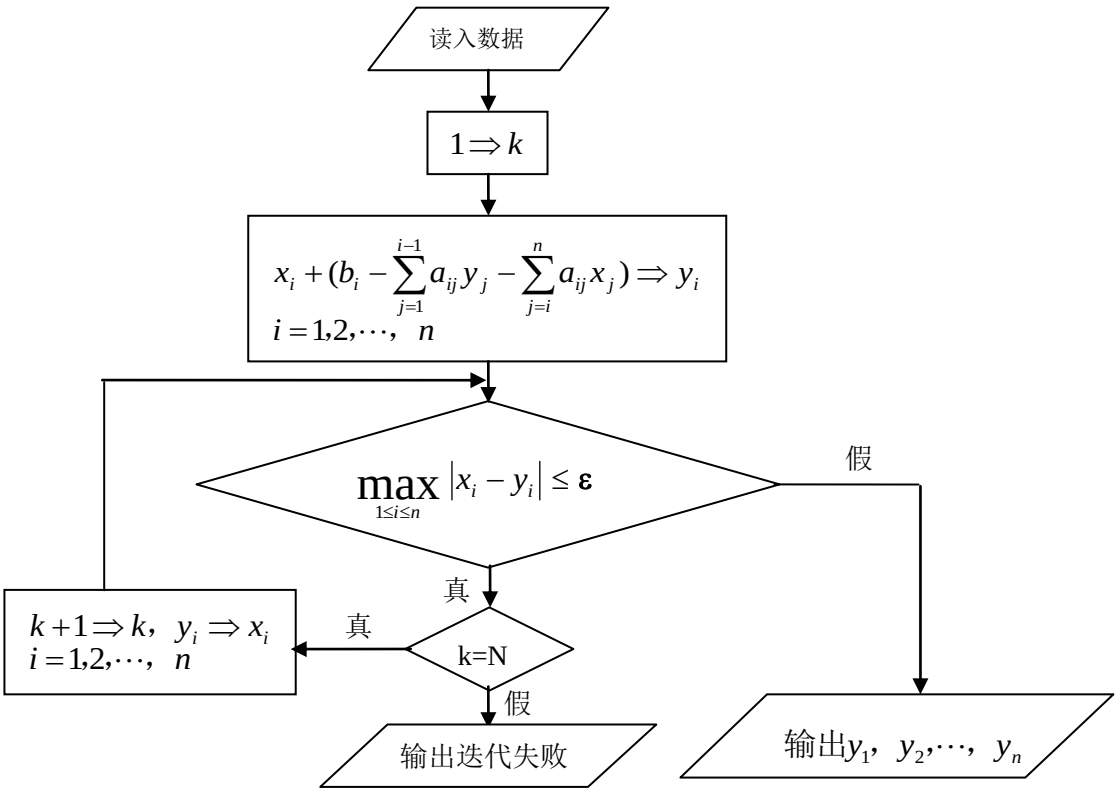


图 2. 3.1Gauss-seidel 迭代法算法框图

2.3.1 满矩阵存储格式的 Gauss-seidel 迭代解法

Gauss-seidel 迭代法的分量形式同（2.3. 8 ）式。元素的查找按数组元素的下标及图 3.1. 1. 1 的对应关系，即可直接获得。即若将元素原来的行、列码记为 i、j，它在二维数组中新的行、列码记为i^*、j^*，则有：

$$i^*=i$$

$$j^*=j \qquad (2.3.1.1)$$

故a_{ij}=a(i^*，j^*)，可直接运用（2.3. 8 ）式的迭代公式求解，算法框图同图 2. 3.1。

2.3.2 二维等带宽存储的 Gauss-seidel 迭代解法

在（2.3. 8 ）式的迭代格式中，用到了所有的元素，而事实上在带宽外的元素均为零，迭代时与x_j^k相乘仍为零，而计算机程序计算的时间主要花在计算乘除运算中，故可不必参与运算和存储，以减少计算时间及内存的花费。所以我们在二维等带宽存储的迭代中只需要用到带宽内的元素，较满矩阵存储节省了很多时间和内存。具体迭代算法如下：

在第 k+1 次迭代中，当 1=<i<=D 时，若 j>i+D-1，则a_{ij}在带宽外部，即a_{ij}=0。故 Gauss-seidel 迭代法的分量形式为：

$$x_i^{k+1}=x_i^k+\frac{1}{a_{ii}}\left(b_i-\sum_{j=1}^{i-1}a_{ij}x_j^{k+1}-\sum_{j=i}^{i+D-1}a_{ij}x_j^k\right)\quad\begin{matrix}(i=1,2,3,\cdots,\ D)\\(k=0,1,2,3\cdots)\end{matrix}\qquad(2.3.2.1)$$

当 D<i<n-D 时，若 j>i+D-1 或 j<i-D+1， 则a_{ij}在带宽外部，即a_{ij}=0。故 Gauss-seidel 迭代法的分量形式为：

$$x_i^{k+1}=x_i^k+\frac{1}{a_{ii}}\left(b_i-\sum_{j=i-D+1}^{i-1}a_{ij}x_j^{k+1}-\sum_{j=i}^{i+D-1}a_{ij}x_j^k\right)\quad\begin{matrix}(i=i-D+1,\cdots,\ i+D-1)\\(k=0,1,2,3\cdots)\end{matrix}\qquad(2.3.2.2)$$

当 n-D<=i<=n 时，若 j<i-D+1，则a_{ij}在带宽外部，即a_{ij}=0。故 Gauss-seidel 迭代法的分量形式为：

$$x_i^{k+1}=x_i^k+\frac{1}{a_{ii}}\left(b_i-\sum_{j=i-D+1}^{i-1}a_{ij}x_j^{k+1}-\sum_{j=i}^{n}a_{ij}x_j^k\right)\quad\begin{matrix}(i=i-D+1,\cdots,\ i+D-1)\\(k=0,1,2,3\cdots)\end{matrix}\qquad(2.3.2.3)$$

其中元素的查找可按数组元素的下标及(2. 1. 2. 1)式，即可获得，即a_{ij}=a(i，j-i+1)。当 i>j 时，查找其对称的元素即可。通过对不同的x_i分为不同迭代的分量形式，对带宽外的零元忽略掉，从而也不用查找，节省了部分时间。在带宽内若所夹零元较非零明显多时，元素查找到后可先判断是否为零，是零的话就可不必参与运算，从而节省计算时间。算法的框图可参照图 2. 3.1 类似。

采用二维等带宽存储，消除了带宽以外的全部零元素，较之全部上三角阵都大大节省了内存。但是由于取最大带宽为存储范围，因此它不能排除在带宽范围内的零元素。当系数矩阵的带宽特别大的情况时，采用二维等带宽是合适的，求解也是便的。但当出现局部带宽特别大的情况时，采用二维

等带宽时将由于局部带宽过大而使整体系数矩阵的存储大大增加，此时可采用一维变带宽存储。

2.3.3 一维变带宽存储的 Gauss-seidel 迭代解法

同二维等带宽存储的 Gauss-seidel 迭代解法一样，每一行高度轮廓线外的元素均为零，迭代时与 x_j^k 相乘仍为零，故可不必运算和存储。

在第 k+1 次迭代中，计算 x_i^{k+1} 时，先计算本行的带宽 Ni=M(i+1)-M(i)，若 j>i+Ni-1，则 a_{ij} 在带宽外部，即 a_{ij}=0，不必查找和运算。故 Gauss-seidel 迭代法的分量形式为：

$$x_i^{k+1}=x_i^k+\frac{1}{a_{ii}}\left(b_i-\sum_{j=1}^{i-1}a_{ij}x_j^{k+1}-\sum_{j=i}^{i+Ni-1}a_{ij}x_j^k\right)\ \begin{matrix}(i=1,2,3,\cdots,\ n)\\(k=0,1,2,3\cdots)\end{matrix} \qquad (2.3.3.1)$$

其中元素的查找可按数组元素的下标及辅助数组 M(n+1)，可知各元素与数组 a 中元素的对应关系为：

a_{ij}=a(M(i)+j-i)

当 i>j 时，查找其对称的元素即可。其他的未查找到的即为 0，故也不必参与运算。同等带宽存储一样，在带宽内若所夹零元较非零元明显多时，元素查找到后可先判断是否为零，是零的话就可不必参与运算，从而节省计算时间。算法的框图可参照图 2. 3.1 类似。

一维变带宽存储是比二维等带宽存储更节省内存的一种存储方法，但由于寻找元素较二维等带宽存储稍复杂，因而编写程序亦较麻烦，并且计算机耗时可能也比二维等带宽存储时较多。因此在选用二维存储方式上要权衡两者的利弊，统盘考虑。通常，当带宽变化不大而计算机内寸允许时，采用二维等带宽存储是合适的。当出现局部带宽过大和带宽范围内的零元素过多时，不管用二维等带宽存储还是用一维变带宽存储都会使整体系数矩阵的存储大大增加，此时可采用按行压缩稀疏存储格式。

2. 3. 4 按行压缩稀疏存储的 Gauss-seidel 迭代解法

按行压缩稀疏存储中，可如一维变带宽存储一样计算其带宽（带宽也指为从主对角元素直至最右的非零元素，即该列号最大的非零元素为止），使其需要查找的元素的范围缩小。

在第 k+1 次迭代中，计算 x_i^{k+1} 时，先计算本行的带宽 D= Li-Fi+1，若 j>i+D-1，则 a_{ij} 在带宽外部，即 a_{ij}=0。故 Gauss-seidel 迭代法的分量形式为：

$$x_i^{k+1}=x_i^k+\frac{1}{a_{ii}}\left(b_i-\sum_{j=1}^{i-1}a_{ij}x_j^{k+1}-\sum_{j=i}^{i+D-1}a_{ij}x_j^k\right)\ \begin{matrix}(i=1,2,3,\cdots,\ n)\\(k=0,1,2,3\cdots)\end{matrix} \qquad (2.3.4.1)$$

即同一维变带宽存储的 Gauss-seidel 迭代法分量形式一样。其中元素的查找按数组元素的下标 i、j，在 ja(Fi)至 ja(Li)查找是否有等于 j 的元素，若有 ja(k)=j，则 a_{ij}=a(k)。当 i>j 时，查找其对称的元素即可。其他的未查找到的即为 0，也不必运算。算法的框图可参照图 2. 3.1 类似。

2. 3. 5 小结

下面对几种存储格式的存储，按 Gauss-seidel 迭代法对问题（1）中离散的线性代数方程组的求解来进行说明和比较，线性代数方程组如（2.3.5.1）式：

$$\begin{bmatrix} 8 & -2 & 0 & -2 & 0 & 0 & 0 & 0 & 0 \\ -2 & 8 & -2 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & -2 & 8 & 0 & 0 & -2 & 0 & 0 & 0 \\ -2 & 0 & 0 & 8 & -2 & 0 & -2 & 0 & 0 \\ 0 & -2 & 0 & -2 & 8 & -2 & 0 & -2 & 0 \\ 0 & 0 & -2 & 0 & -2 & 8 & 0 & 0 & -2 \\ 0 & 0 & 0 & -2 & 0 & 0 & 8 & -2 & 0 \\ 0 & 0 & 0 & 0 & -2 & 0 & -2 & 8 & -2 \\ 0 & 0 & 0 & 0 & 0 & -2 & 0 & -2 & 8 \end{bmatrix}_{9\times 9} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} 1.052076 \\ 1.572131 \\ 1.171253 \\ 1.572131 \\ 2.223330 \\ 1.572131 \\ 1.171253 \\ 1.572131 \\ 1.052076 \end{pmatrix} \quad (2.3.5.1)$$

下面看三种存储格式的 Gauss-seidel 迭代法的计算时间（程序分别见附录）。

通过运行程序，得到迭代次数、近似解和计算时间，其中近似解为:

$x=(0.466984,0.670949,0.481881,0.670949,0.948865,0.670949,0.481881,0.670949,0.466984)^T$

再结合前面的分析和表（2.1.4.1），我们由此例可以得出如下表格：

存储格式	满矩阵存储	带宽存储		按行压缩稀疏存储
		二维等带宽存储	一维变带宽存储	
迭代次数	24	24	24	24
存储需求单元数	81	36	29	21
cpu 时间（秒）	0. 011456	0. 011416	0. 011426	0. 011406

注：以上计算时间均为同一操作系统下循环调用 1000 次算法的平均时间。

数值实验表明：在迭代多种存储格式中，按行压缩稀疏存储格式存储量和计算时间最少，带宽存储格式的存储量和少于满矩阵存储格式，且半带宽存储格式耗时比满矩阵存储格式要少很多。因为三种存储格式的迭代都是在 Gauss-seidel 迭代法的基础上进行的，故迭代次数总是一致的。可看出按行压缩稀疏存储格式是最优的。在大型的矩阵计算中，按行压缩稀疏存储格式更加显示出它的优越性。故数值算例可证明按行压缩稀疏存储格式在时间和存储上都较为占优，可靠高效，能够应用于有限元线性方程组的迭代求解。

在半阵的带宽及按行压缩稀疏存储中，若对于只是位置对称的矩阵，可增加相应下三角的的存储（包括对角线元素），为使与原来矩阵元素对应关系及查找方法和元素对称的矩阵保持一致。可将下三角先转置，然后再和元素对称的矩阵存储一致。例如：半带宽存储可再将下三角先转置后加存储于另一个半带宽矩阵 B 中；变带宽及按行压缩稀疏存储也可再下三角先转置后按行存储于另一个一维数组 b 中，使矩阵元素存储完整。则查找元素 a_{ij} （i>j）时，先转化为查找 a_{ji}，然后再按 a_{ji} 的下标，按同样的方法在 B 或 b 中查找元素。同样适用于迭代法。

其迭代过程和前述的迭代公式一致，只是元素的查找稍做变化（即查找元素 a_{ij} （i>j）时，要注意是在 B 或 b 中查找）。显然容易实现，故不再多讲。其算法的优劣同上述分析是一致的。

三　小结

本文虽然是针对一类椭圆型偏微分方程离散格式的求解而引入，但许多工程问题，力学问题，动力系统问题最终都归结为有限元线性方程组的求解问题。因此，构造高效率的求解离散方程的方法是计算数学的重要内容。目前主要有2 种方法:直接法和迭代法。 (1) 直接法: 经过有限次数的运算即可得方程组的精确解的方法。其中Gauss 消去法是最有效最直接的方法。算法简单方便，特别是适用于求解多组载荷的情况，因为此时系数矩阵的消元仍只需进行一次。随着计算机性能的不断提高，如有

效字长位数的增加，内存的扩大及计算速度的提高，一般情况下，直接法仍是首选的解法。它计算量小，精确度高。但它的不足之处，一是要求计算机的存储比较大，无论是半带宽还是变带宽存储，都要存储大量的零元素，一般零元素的个数与网格节点编号密切相关(而松弛迭代法只需存储非零元素，且不受节点编号的影响) ，而且程序比较复杂。直接法的另一不足是，它在计算过程中不能对解的误差进行检查和控制。而解的误差只要是由计算机的有效字长，方程的阶数，特别是系数矩阵的性态决定的，在一定的字长条件下，方程组的阶数越高，计算过程中累积的舍入误差越大。基于以上两点，对大型、超大型方程组，迭代解法常更合适的选择。(2) 迭代法:也称辗转法，是一种不断用变量的旧值递推新值的过程。迭代解法的优点之一是，它按行压缩稀疏存储，它不要求保存系数矩阵中高度轮廓线以下的零元素，并且不对他们进行运算，即它们保持为零不变。这样一来计算机只存储系数矩阵的非零元素以及记录它们位置的辅助数组。这不仅可以最大限度地节约存储空间，而且提高了计算效率。这对于求解大型、超大型方程组是很有意义的。另一方面，迭代解法计算过程中可以对解的误差进行检查，并通过迭代次数来降低误差，直至满足的精度要求。这种方法可节省计算机储存量，而且程序简单，但要获得精度较高的解，需要逐次迭代，计算量比较大，有时也可能会因为舍入误差的积累，导致计算的精度很差，还有待于继续研究。

本文稀疏矩阵的存储格式的线性方程组求解还可推广到直解法的列主元素法和LU分解或其他分解法和其他的迭代解法，为有限元线性方程组的求解提供更广阔的天地。

最后感谢我的指导老师谭敏老师的悉心指导，是她为我提出了许多宝贵的意见，才使我的论文逐渐生成和丰盈。也感谢给论文完成给予批评、指正和帮助的老师和同学。谢谢你们！

参考文献：

[1]马东升　雷勇军　《数值计算方法》[M] 第二版　机械工业出版社　2006
[2]李景涌　《有限元法》[M] 北京邮电大学出版社 1999
[3]王勖成　《有限单元法》[M] 清华大学出版社 2002
[4]李荣华　《偏微分方程数值解法》[M] 高等教育出版社 2004
[5]林群　　《微分方程数值解法基础教程》[M] 科学出版社 2001

附录：

直接法的 C++程序：

满矩阵存储的高斯消元法

```
//满矩阵存储的高斯消元法
#include<math.h>
#include<stdio.h>
#define N 9
gauss(double a[N][N],double b[N],int n);
int main()
{
    double a[N][N]={
    {8,-2,0,-2,0,0,0,0,0},
    {-2,8,-2,0,-2,0,0,0,0},
    {0,-2,8,0,0,-2,0,0,0},
    {-2,0,0,8,-2,0,-2,0,0},
    {0,-2,0,-2,8,-2,0,-2,0},
    {0,0,-2,0,-2,8,0,0,-2},
    {0,0,0,-2,0,0,8,-2,0},
```

```
      {0,0,0,0,-2,0,-2,8,-2},
      {0,0,0,0,0,-2,0,-2,8}
      },
      b[N]={1.052076,1.572131,1.171253,1.572131,2.223330,1.572131,1.171253,1.572131,1.052076};
      gauss( a,b,N);
}
gauss(double a[N][N],double b[N],int n)
{
   int i,j,k;
   double tmp;
   double x[N];
   for(i=0;i<n-1;i++)
      for(j=i+1;j<n;j++)//高斯消元
       {
          tmp=-a[j][i]/a[i][i];
          b[j]+=b[i]*tmp;
          for(k=i;k<n;k++)
          a[j][k]+=a[i][k]*tmp;
       }
   x[n-1]=b[n-1]/a[n-1][n-1];//求解方程
   for(i=n-2;i>=0;i--)
   {
       x[i]=b[i];
       for(j=i+1;j<n;j++)
           x[i]-=a[i][j]*x[j];
       x[i]/=a[i][i];
   }
   printf("原方程组的解为:\n");//输出
   for(i=0;i<N;i++) printf("x%d=%f\n",i+1,x[i]);
  return 0;
}
```

等带宽存储的高斯消元算法

```
//等带宽存储的高斯消元法
#include<math.h>
#include<stdio.h>
#define    N 9
#define    NHBW 4
void gauss(int n,int d,double band[N+1][NHBW+1],double x[N+1]);
void main()
{
   double band[N+1][NHBW+1]={
         {0},
         {0,8,-2,0,-2},
```

```
        {0,8,-2,0,-2},
        {0,8,0,0,-2},
        {0,8,-2,0,-2},
        {0,8,-2,0,-2},
        {0,8,0,0,-2},
        {0,8,-2,0,0},
        {0,8,-2,0,0},
        {0,8,0,0,0}},

b[N+1]={0,1.052076,1.572131,1.171253,1.572131,2.223330,1.572131,1.171253,1.572131,1.052076};
    gauss( N, NHBW, band, b);
}
//二维等带宽存储的高斯消元法求解方程组 x[N+1]
void gauss(int n,int d,double band[N+1][NHBW+1],double b[N+1])
{
  int k,i,im,L,J,Jm,M,H;
  double factor,x[N+1];
  for(k=1;k<=n-1;k++)
  {
   im=k+d-1;
   if(im>n) im=n;
   for(i=k+1;i<=im;i++)
   {
       L=i-k+1;
       factor=band[k][L]/band[k][1];
       Jm=d-L+1;
       for(J=1;J<=Jm;J++)
       {
          M=J+i-k;
          band[i][J]=band[i][J]-factor*band[k][M];
       }
       b[i]=b[i]-factor*b[k];
   }
  }

  x[n]=b[n]/band[n][1];
  for(i=n-1;i>=1;i--)
  {
     factor=0;
     Jm=n-i+1;
     if(Jm>d) Jm=d;
     for(J=2;J<=Jm;J++)
     {
```

```
        H=J+i-1;
        factor+=band[i][J]*x[H];
       }
      x[i]=(b[i]-factor)/band[i][1];
   }
   //printf("原方程组的解为:\n");//输出
   for(i=1;i<=N;i++)
       printf("x%d=%lf\n",i,x[i]);
}
```

变带宽存储的高斯消元算法（按行存储）

```
//变带宽存储的高斯消元法（按行存储）
#include<math.h>
#include<stdio.h>
#define N 9
void Gauss(int n,double A[31],double x[N+1],int N1[N+1]);
void main()
{ int       N1[N+1]={0,1,5,9,13,17,21,25,28,30};
   double A[31]={0,8,-2,0,-2,8,-2,0,-2,8,0,0,-2,8,0,0,-2,8,-2,0,-2,8,0,0,-2,8,-2,0,8,-2,8},

x[N+1]={0,1.052076,1.572131,1.171253,1.572131,2.223330,1.572131,1.171253,1.572131,1.052076};
   Gauss(N,A,x,N1);
}

void Gauss(int n,double A[31],double x[N+1],int N1[N+1])
{
   int i,j,k,NUS,im,jm,IJ,KK,KI,KJ,NN,II;
   double factor;
   for(k=1;k<=n-1;k++)
   {
    NUS=N1[k+1]-N1[k];
    im=k+NUS-1;
    jm=k+NUS-1;
    if(im>=n) im=n;
    for(i=k+1;i<=im;i++)
    {
        KK=N1[k];
        KI=N1[k]+i-k;
        factor=0;
        factor=A[KI]/A[KK];
        x[i]=x[i]-factor*x[k];
        for(j=i;j<=jm;j++)
        {
         IJ=N1[i]+j-i;
```

```
            KJ=N1[k]+j-k;
            A[IJ]=A[IJ]-factor*A[KJ];
           }
     }
   }
   NN=N1[n];
   x[n]=x[n]/A[NN];
   for(i=n-1;i>=1;i--)
   {
      factor=0;
      II=N1[i];
      NUS=N1[i+1]-N1[i];
      jm=NUS-1;
      for(j=i+1;j<=i+jm;j++)
      {
        IJ=N1[i]+j-i;
        factor+=A[IJ]*x[j];
      }
      x[i]=(x[i]-factor)/A[II];
   }
   printf("原方程的解为：\n");
   for(i=1;i<=n;i++)
     printf("x%d=%lf\n",i,x[i]);
}
```

迭代法的 C++程序：

满矩阵存储格式的 Gauss-seidel 迭代算法：

```
#include<math.h>
#include<stdio.h>
#define MAXREPT 100 //最大迭代次数
#define N 9
#define epsilon 0.0000001
Gauss_seidel(double a[N][N],double b[N],int n);
void main()
{
      double a[N][N]={
      {8,-2,0,-2,0,0,0,0,0},
      {-2,8,-2,0,-2,0,0,0,0},
      {0,-2,8,0,0,-2,0,0,0},
      {-2,0,0,8,-2,0,-2,0,0},
      {0,-2,0,-2,8,-2,0,-2,0},
      {0,0,-2,0,-2,8,0,0,-2},
      {0,0,0,-2,0,0,8,-2,0},
```

```
    {0,0,0,0,-2,0,-2,8,-2},
    {0,0,0,0,0,-2,0,-2,8}
    },
    b[N]={1.052076,1.572131,1.171253,1.572131,2.223330,1.572131,1.171253,1.572131,1.052076};
    Gauss_seidel( a,b,N);
}
Gauss_seidel(double a[N][N],double b[N],int n)
{
  int i,j,k;
  double err,factor;
  double x[N],nx[N];
  for(i=0;i<N;i++) x[i]=0;
  for(k=1;k<=MAXREPT;k++)
  {
    for(i=0;i<N;i++)
    {
      factor=0;
      for(j=0;j<=i-1;j++)
      {
          factor=factor+a[i][j]*nx[j];
      }
      for(j=i;j<n;j++)
      {

        factor=factor+a[i][j]*x[j];
      }
      nx[i]=x[i]+(b[i]-factor)/a[i][i];
    }
    err=0;
    for(j=0;j<n;j++)
    if(err<fabs(nx[j]-x[j])) err=fabs(nx[j]-x[j]); //误差
    for(j=0;j<n;j++)
    x[j]=nx[j];
    if(err<epsilon)
    {
    printf("迭代次数为%d:\n",k);
    printf("原方程组的解为:\n");//输出
    for(i=0;i<n;i++) printf("x%d=%f\n",i+1,x[i]);
    return 0;
    }
  }
  printf("经过%d 次迭代,没有结果,迭代失败.\n",MAXREPT); //输出
    return 1;
```

```
}
```

二维等带宽存储格式的 Gauss-seidel 迭代算法：

```
#include<math.h>
#include<stdio.h>
#define MAXREPT 100 //最大迭代次数
#define N 9
#define D 4
#define epsilon 0.0000001
Gauss_seidel(double a[N+1][D+1],double b[N+1],int d,int n);
void main()
{
   double band[N+1][D+1]={
         {0},
         {0,8,-2,0,-2},
         {0,8,-2,0,-2},
         {0,8,0,0,-2},
         {0,8,-2,0,-2},
         {0,8,-2,0,-2},
         {0,8,0,0,-2},
         {0,8,-2,0,0},
         {0,8,-2,0,0},
         {0,8,0,0,0}},

b[N+1]={0,1.052076,1.572131,1.171253,1.572131,2.223330,1.572131,1.171253,1.572131,1.052076};
    Gauss_seidel(band,b,D,N);
}
Gauss_seidel(double a[N+1][D+1],double b[N+1],int d,int n)
{
   int i,j,k,lb,ub;
   double err,factor,aij,aii;
   double x[N+1],nx[N+1];
   for(i=1;i<=n;i++) x[i]=0;
   for(k=1;k<=MAXREPT;k++)
   {
      for(i=1;i<=n;i++)
      {
          if(i<=d)    {lb=1;ub=i+d-1;}
            else if(i<n-d) {lb=i-d+1;ub=i+d-1;}
            else {lb=i-d+1;ub=n;}
          factor=0;
          for(j=lb;j<=i-1;j++)
          {
             aij=a[j][i-j+1];
```

```
            factor=factor+aij*nx[j];
        }
        for(j=i;j<=ub;j++)
        {
           aij=a[i][j-i+1];
           factor=factor+aij*x[j];
        }
        aii=a[i][1];
        nx[i]=x[i]+(b[i]-factor)/aii;
      }
      err=0;
      for(j=1;j<=n;j++)
      if(err<fabs(nx[j]-x[j])) err=fabs(nx[j]-x[j]); //误差
      for(j=1;j<=n;j++)
      x[j]=nx[j];
      if(err<epsilon)
      {
      printf("迭代次数为:%d\n",k);
      printf("原方程组的解为:\n");//输出
      for(i=1;i<=n;i++) printf("x%d=%f\n",i,x[i]);
      return 0;
      }
   }
   printf("经过%d 次迭代,没有结果,迭代失败.\n",MAXREPT); //输出
   return 1;
}
```

一维变带宽存储格式的 Gauss-seidel 迭代算法：

```
#include<math.h>
#include<stdio.h>
#define MAXREPT 100 //最大迭代次数
#define N 9
#define epsilon 0.0000001
Gauss_seidel(double a[30],double b[N+1],int M[N+1],int n);
void main()
{ int     M[N+2]={0,1,5,9,13,17,21,25,27,29,30};
   double a[31]={0,8,-2,0,-2,8,-2,0,-2,8,0,0,-2,8,-2,0,-2,8,-2,0,-2,8,0,0,-2,8,-2,8,-2,8},

b[N+1]={0,1.052076,1.572131,1.171253,1.572131,2.223330,1.572131,1.171253,1.572131,1.052076};
     Gauss_seidel(a,b,M,N);
}
Gauss_seidel(double a[30],double b[N+1],int M[N+1],int n)
{
   int i,j,k,II,Ni,mi;
```

```
double err,factor,aij;
double x[N+1],nx[N+1];
for(i=1;i<=N;i++) x[i]=0;
for(k=1;k<=MAXREPT;k++)
{
   for(i=1;i<=N;i++)
   {
      factor=0;

      for(j=1;j<=i-1;j++)
      {
        Ni=M[j+1]-M[j];
        if(i>j+Ni-1) continue;//寻找元素
        else mi=M[j]+i-j; //寻找元素
        aij=a[mi];
        factor=factor+aij*nx[j];
      }
      Ni=M[i+1]-M[i];//计算带宽
      for(j=i;j<=i+Ni-1;j++)
      {
         if(j>i+Ni-1) continue;//寻找元素
         else mi=M[i]+j-i; //寻找元素
         aij=a[mi];
         factor=factor+aij*x[j];
      }
      II=M[i];
      nx[i]=x[i]+(b[i]-factor)/a[II];
   }
   err=0;
   for(j=1;j<=n;j++)
   if(err<fabs(nx[j]-x[j])) err=fabs(nx[j]-x[j]); //误差
   for(j=1;j<=n;j++)
   x[j]=nx[j];
   if(err<epsilon)
   {
   printf("迭代次数为:%d\n",k);
   printf("原方程组的解为:\n");//输出
   for(i=1;i<=n;i++) printf("x%d=%f\n",i,x[i]);
   return 0;
   }
}
printf("经过%d 次迭代,没有结果,迭代失败.\n",MAXREPT); //输出
return 1;
```

```
}
```

按行压缩稀疏存储格式的 Gauss-seidel 迭代算法：

```
#include<math.h>
#include<stdio.h>
#define MAXREPT 100 //最大迭代次数
#define N 9
#define epsilon 0.0000001
Gauss_seidel(double a[23],double b[10],int ia[11],int ja[23],int n);
void main()
{ int       ia[N+2]={0,1,4,7,9,12,15,17,19,21,22},
              ja[22]={0,1,2,4,2,3,5,3,6,4,5,7,5,6,8,6,9,7,8,8,9,9};
   double a[22]={0,8,-2,-2,8,-2,-2,8,-2,8,-2,-2,8,-2,-2,8,-2,-2,8,-2,8,-2,8},

b[N+1]={0,1.052076,1.572131,1.171253,1.572131,2.223330,1.572131,1.171253,1.572131,1.052076};

   Gauss_seidel(a,b,ia,ja,N);

}
Gauss_seidel(double a[23],double b[N+1],int ia[N+2],int ja[23],int n)
{
   int i,j,k,k1,mi,II,Fi,Li,D;
   double err,factor,aij;
   double x[N+1],nx[N+1];
   for(i=1;i<=N;i++) x[i]=0;
   for(k=1;k<MAXREPT;k++)
   {
      for(i=1;i<=N;i++)
      {
         factor=0;
         for(j=1;j<=i-1;j++)
         {
             for(k1=ia[j];k1<ia[j+1];k1++) //寻找元素
                   if(i==ja[k1]) {mi=k1; break;}//寻找到
             if(k1==ia[j+1]) continue;//未寻找到
             aij=a[mi];
             factor=factor+aij*nx[j];
         }
         Fi=ja[ia[i]];
         Li=ja[ia[i+1]-1];
         D=Li-Fi+1;
         for(j=i;j<=i+D-1;j++)
         {
            for(k1=ia[i];k1<ia[i+1];k1++)//寻找元素
```

```
            if(j==ja[k1]) {mi=k1; break;}//寻找到
          if(k1==ia[i+1]) continue;//未寻找到
          aij=a[mi];
          factor=factor+aij*x[j];
        }
        II=ia[i];
        nx[i]=x[i]+(b[i]-factor)/a[II];
      }
      err=0;
      for(j=1;j<=n;j++)
      if(err<fabs(nx[j]-x[j])) err=fabs(nx[j]-x[j]); //误差
      for(j=1;j<=n;j++)
      x[j]=nx[j];
      if(err<epsilon)
      {
      printf("迭代次数为:%d\n",k);
      printf("原方程组的解为:\n");//输出
      for(i=1;i<=n;i++) printf("x%d=%f\n",i,x[i]);
      return 0;
      }
   }
   printf("经过%d 次迭代,没有结果,迭代失败.\n",MAXREPT); //输出
   return 1;
}
```