数值分析第五、六章补充阅读

朱宇嘉

1 解线性方程组

对于 n 元线性方程组（假定只有 n 个方程）：

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ \quad ... \quad\quad ... \quad\quad ... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n \end{cases}$$

我们主要有以下求解方法：

1.1 克莱姆法则

克莱姆法则是求解线性方程组最为直接的方法。其计算步骤是 $x_i = \frac{D_i}{D}$，其中 D 是系数矩阵的行列式，D_i 是将系数矩阵的第 i 列变为 $(b_1, b_2, ...b_n)^T$ 后的行列式。通过计算 $n+1$ 个行列式的值，我们就可以求解线性方程组的解。

克莱姆法则理论上可以求出方程的精确解（不记舍入误差的情况下），但克莱姆法则也有一个明显的缺点：**运算量过大**。对于 n 阶方程组，我们需要计算 $n+1$ 个 n 阶行列式的值。我们知道一个 n 阶行列式是由 $n!$ 项相加得来的，而每项又是 n 个数的乘积。因此，计算一个 n 阶行列式需要做 $(n-1)\cdot n!$ 次乘法和 $n!-1$ 次加法或减法，共需做 $(n-1)\cdot n! + n! - 1 = n\cdot n! - 1$ 次浮点数运算。所以，在克莱姆法则中计算 $n+1$ 个行列式共需要做 $(n+1)[n\cdot n! - 1] = n\cdot(n+1)! - n - 1$ 次浮点数运算。此外，在计算 x_i 时还要做 n 次除法。因此使用克莱姆法则总共的需要的浮点数运算次数是 $n\cdot(n+1)! - 1$ 次。

这样的运算量是什么概念呢？我们来做一道数学题。

> **问题：**假设我们要求解一个 15 阶的线性方程组。已知个人 PC 机每秒可以做大约 200 亿次浮点数运算，请问使用克莱姆法则求解该线性方程组需要多少时间？
>
> **答：**需要 $(15\times 16! - 1)/2e8(s) = 1569209(s) \approx 436(h) \approx 18(day)$。

所以计算一个 15 阶的线性方程组，克莱姆法则就需要花掉 18 天。此外 $n \cdot (n+1)! - 1$ 随 n 增长的速度非常快，当 n 增长到 20 时，计算时间会变成 162008 年!!! 你即使是愚公也不会有什么办法。

所以克莱姆法则往往只能用在低阶线性方程组的求解上。对于高阶方程组，克莱姆法则往往是不能使用的。

1.2 直接方法

解线性方程组的直接方法包括**高斯消去法**和**矩阵三角分解法**。事实上，这两种方法是同质的，两者之间并没有什么区别。

高斯消去法和矩阵三角分解法在理论上也可以求出线性方程组的精确解。但是由于计算机的浮点数长度限制会产生一定的输入误差，在一些特定的方程下，舍入误差会在计算过程中扩散和传播，使得计算结果造成极大的偏离。因此，**高斯主元消去法**和**主元三角分解法**被提出以避免舍入误差的影响。

使用高斯消去法仅需做 $O(n^3)$ 次浮点数运算，大大提升了运算和求解的速度。

此外，对于部分特殊的方程，我们可以使用**平方根法**和**追赶法**继续提高运算速度。

1.3 迭代方法

迭代方法主要包括**雅克比迭代法**，**高斯-赛德尔迭代法**和**超松弛迭代法**。

与直接方法主要研究**时间复杂度不同**，迭代方法主要研究的是**是否收敛**。

本章，我们主要用程序语言来实现各种线性方程组的求解方法。我们将使用 Python 语言实现解线性方程组的直接方法；使用 R 语言实现解线性方程组的迭代方法。

2 解线性方程组的直接方法

我们用 Python 语言来实现各类解线性方程组的直接方法。需要实现的方法包括高斯消去法，列主元消去法，矩阵三角分解和主元三角分解。平方根法和追赶法则留给大家作为联系。

2.1 高斯消去法

高斯消去法包括消元计算和回代计算两个步骤。当 $a_{kk}^{(k)} \neq 0$ 时，我们可以重复两个步骤来实现高斯消去法。我们可以用两个 for 循环写出以下代码：

```
import numpy as np

def Guass(A,b):
    n = A.shape[0]
    assert n == A.shape[1], 'A should be a square matrix'
    assert b.shape[0] == n, 'b should have same length as A'
    ## 连接 A 和 b
    Ab = np.concatenate((A,b.reshape(-1,1)),axis=1).astype('float32')
    ## 消元计算
    for i in range(n-1):
        ## a_kk 是否为 0
        assert Ab[i,i] != 0, 'Unsolveable for Guassian Elimination Method'
        ## 与下方相减
        for j in range(i+1,n):
            Ab[j,:] -= Ab[i,:] * Ab[j,i] / Ab[i,i]
    ## 回代计算求 x
    x = np.zeros(n)
    A = Ab[:,:-1]
    b = Ab[:,-1]
    for i in range(n):
        x[n-1-i] = (b[n-1-i]-np.sum(A[n-1-i,:]*x))/A[n-1-i,n-1-i]
    return x
```

这些代码的思路应该是比较清晰的在这里就不做解释了，我们用以下方程来做一下验证：

$$\begin{cases} x_1 + x_2 + x_3 = 3 \\ 4x_2 - x_3 = 3 \\ 2x_1 - 2x_2 + x_3 = 1 \end{cases}$$

这个方程的根是 $(x_1, x_2, x_3) = (1, 1, 1)$，我们可以看到输出结果没有异常。

```
A = np.array([[1,1,1],[0,4,-1],[2,-2,1]])
b = np.array([3,3,1])
Guass(A,b)

## array([1., 1., 1.])
```

2.2 列主元消去法

列主元消去法和高斯消去法的区别是：在每次消元计算前，我们要将该列中绝对值最大的元素调至该行，以减小误差。因此，我们只需在高斯消去法的代码中加入几行实现这一过程即可。

```
def ColGuass(A,b):
    n = A.shape[0]
    assert n == A.shape[1], 'A should be a square matrix'
    assert b.shape[0] == n, 'b should have same length as A'
    ## 连接 A 和 b
    Ab = np.concatenate((A,b.reshape(-1,1)),axis=1).astype('float32')
    ## 消元计算
    for i in range(n):
        ## 选取列主元
        pivot = np.argmax(abs(Ab[i:,i]))
        if pivot!=0:
            Ab[[i,i+pivot],:] = Ab[[i+pivot,i],:]
            print("pivot changed")
        assert Ab[i,i] != 0, 'Unsolveable for Guassian Elimination Method'
        ## 与下方相减
        for j in range(i+1,n):
            Ab[j,:] -= Ab[i,:] * Ab[j,i] / Ab[i,i]
    ## 回代计算求 x
    x = np.zeros(n)
    A = Ab[:,:-1]
    b = Ab[:,-1]
    for i in range(n):
        x[n-1-i] = (b[n-1-i] - np.sum(A[n-1-i,:]*x))/A[n-1-i,n-1-i]
    return x
```

我们仍然可以运行代码求解之前的方程，但事实上，这样的运行意义并不大，因为这个方程并不存在 $a_{kk}^{(k)} \approx 0$ 的情况。但我们仍然运行一下以显示程序的正确性。可以看到，在求解过程中，程序进行了一次换主元操作。

```
A = np.array([[1,1,1],[0,4,-1],[2,-2,1]])
b = np.array([3,3,1])
ColGuass(A,b)
```

```
## pivot changed
## array([1., 1., 1.])
```

2.3 矩阵三角分解

矩阵三角分解就是将一个方阵 A 分解为两个矩阵的乘积，即 $A = LU$，其中 L 是对角线为 1 的下三角矩阵，U 是一个上三角矩阵。这个算法实现起来并不复杂，但如果要节省计算机的内存，可以使用一些技巧。我们可以用计算出的 L, U 直接覆盖原矩阵 A，同时我们只输出一个矩阵，其下三角部分存储 L，上三角和对角线部分存储 U。

```
def LUDecomp(A):
    n = A.shape[0]
    assert A.shape[1] == n, 'A should be a square matrix'
    ## A 矩阵下半部分存储 L，上半部分存储 U
    A = A.astype('float32')
    ## 迭代
    for i in range(n):
        ## 计算 U
        A[i,i:] = A[i,i:] - np.sum(A[:i,i:] * A[i,:i].reshape(-1,1),axis=0)
        ## 计算 L
        assert A[i,i] != 0, 'can not do LU decomposition'
        A[(i+1):,i] -= np.sum(A[(i+1):,:i] * A[:i,i].reshape(1,-1) ,axis=1)
        A[(i+1):,i] = A[(i+1):,i]/A[i,i]
    return A
```

我们用书上的例 5 来验证一下：

```
A = np.array([[1,2,3],[2,5,2],[3,1,5]])
LUDecomp(A)
```

```
## array([[  1.,   2.,   3.],
##        [  2.,   1.,  -4.],
##        [  3.,  -5., -24.]], dtype=float32)
```

输出结果为：

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & -4 \\ 3 & -5 & -24 \end{pmatrix}$$

因此，我们知道 L 和 U 矩阵分别是：

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & -5 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & -4 \\ 0 & 0 & -24 \end{pmatrix}$$

2.4 主元三角分解

主元三角分解与三角分解的区别和列主元高斯消去法和高斯消去法的区别类似。我们也只需对代码做小部分修改即可。但是，我们还需要一个 list 来存储主元位置，以便可以在日后求解 $PA = LU$。

```
def PivotLUDecomp(A):
    n = A.shape[0]
    assert A.shape[1] == n, 'A should be a square matrix'
    A = A.astype('float32')
    Ip = []
    ## LU 矩阵下半部分存储 L，上半部分存储 U
    ## 迭代
    for i in range(n):
        ## 选主元，并交换两行元素
        A[i:,i] = (A[i:,i] - np.sum(A[i:,:i] * A[:i,i].reshape(1,-1) ,axis=1))
        pivot = i + np.argmax(abs(A[i:,i]))
        Ip.append(pivot)
        if pivot != i: A[[i,pivot],:] = A[[pivot,i],:]
        ## 计算 L
        assert A[i,i] != 0, 'can not do Pivot LU decomposition'
        A[(i+1):,i] = A[(i+1):,i]/A[i,i]
        ## 计算 U
        A[i,(i+1):] = A[i,(i+1):] - np.sum(A[:i,(i+1):]*A[i,:i].reshape(-1,1),axis=0)
    return A,Ip
```

我们输出结果, 可以发现如今 L 矩阵中的元素绝对值均已小于 1。

```
A = np.array([[1,2,3],[2,5,2],[3,1,5]])
LU, Ip = PivotLUDecomp(A)
print("Decomposition:\n",LU,"\nPivot:",Ip)
```

```
## Decomposition:
##  [[ 3.          1.          5.        ]
##  [ 0.6666667   4.3333335  -1.3333335 ]
##  [ 0.33333334  0.38461536  1.8461537 ]]
## Pivot: [2, 1, 2]
```

3 解线性方程组的迭代法

我们使用 R 语言实现各种线性方程组的迭代法。需要实现的方法有雅克比迭代法，高斯-赛德尔迭代法和超松弛迭代法。

3.1 雅克比迭代法

雅克比迭代法将矩阵 A 分解成 $A=M-N=D-(L+U)$，并使用迭代公式 $x^{(k+1)}=D^{-1}(L+U)x^{(k)}+D^{-1}b:=B^Jx^{(k)}+f$ 进行迭代。按照这个思路，我们可以写出以下代码：

```
Jacobi<-function(A,b,esp=1e-3){
  n = ncol(A)
  if(n!=nrow(A)) stop("A should be a square matrix")
  if(n!=length(as.vector(b))) stop("b should have same length as A")
  ## 分解 A
  M<-diag(diag(A));N<-M-A
  B<-solve(M)%*%N;f<-solve(M)%*%b
  ## 迭代
  iter<-1;x<-rep(0,n);temp<-x+1
  while(mean(abs(temp-x))>esp){
    temp<-x;x<-B%*%x+f
    if(iter>100|max(abs(x))==Inf) stop("not converge")
    iter<-iter+1
  }
  return(x)
}
```

我们用以下方程来检验雅克比迭代法的编写：

$$\begin{cases}8x_1-3x_2+2x_3=20\\4x_1+11x_2-x_3=33\\6x_1+3x_2+12x_3=36\end{cases}$$

```
A<-matrix(c(8,4,6,-3,11,3,2,-1,12),ncol=3)
b<-c(20,33,36)
Jacobi(A,b,esp=1e-3)
```

```
##           [,1]
## [1,] 3.0002816
## [2,] 1.9999118
```

```
## [3,] 0.9997405
```

可以看出，在 1e-3 的误差范围内，使用雅克比迭代法求得的根为 $(3.0002, 1.9999, 0.9997)^T$，与真实解 $(3, 2, 1)^T$ 还存在一些差距。

3.2 高斯-赛德尔迭代法

高斯-赛德尔迭代法将矩阵 A 分解成 $A = M - N = (D - L) - U$，并使用迭代公式 $x^{(k+1)} = (D - L)^{-1}Ux^{(k)} + (D - L)^{-1}b := B^G x^{(k)} + f$ 进行迭代。按照这个思路，我们可以写出以下代码：

```
GuassSeidel<-function(A,b,esp=1e-3){
  n = ncol(A)
  if(n!=nrow(A)) stop("A should be a square matrix")
  if(n!=length(as.vector(b))) stop("b should have same length as A")
  ## 分解 A
  N<--A;N[lower.tri(N,diag=T)]<-0;M<-A+N
  B<-solve(M)%*%N;f<-solve(M)%*%b
  ## 迭代
  iter<-1;x<-rep(0,n);temp<-x+1
  while(mean(abs(temp-x))>esp){
    temp<-x;x<-B%*%x+f
    if(iter>100|max(abs(x))==Inf) stop("not converge")
    iter<-iter+1
  }
  return(x)
}
```

我们只需要将代码中有关矩阵 A 的分解部分进行修正即可。我们用同样的例子来验证：

```
A<-matrix(c(8,4,6,-3,11,3,2,-1,12),ncol=3)
b<-c(20,33,36)
GuassSeidel(A,b,esp=1e-3)
```

```
##          [,1]
## [1,] 2.999842
## [2,] 2.000072
## [3,] 1.000061
```

我们看到，求出的根是 $(2.9998, 2.0001, 1.0001)^T$，与真实解 $(3, 2, 1)^T$ 也存在一些差距。

3.3 超松弛迭代法

超松弛迭代法引进了松弛因子 ω，以试图加速收敛。在编写程序时，我们要注意，因为只有 $0 < \omega < 2$ 时，超松弛迭代法才可能收敛。因此对于不在这个范围内的 ω，我们可直接暂停程序的运行。

```
SOR<-function(A,b,omega=1,esp=1e-3){
  n = ncol(A)
  if(n!=nrow(A)) stop("A should be a square matrix")
  if(n!=length(as.vector(b))) stop("b should have same length as A")
  if(omega<=0|omega>=2) stop("not converge")
  ## 分解 A
  D<-diag(diag(A));L<--A;U<--A
  L[upper.tri(L,diag=T)]<-0;U[lower.tri(U,diag=T)]<-0
  B<-solve(D-omega*L)%*%((1-omega)*D+omega*U)
  f<-omega*solve(D-omega*L)%*%b
  ## 迭代
  iter<-1;x<-rep(0,n);temp<-x+1
  while(mean(abs(temp-x))>esp){
    temp<-x;x<-B%*%x+f
    if(iter>300|max(abs(x))==Inf) stop("not converge")
    iter<-iter+1
  }
  return(list(root=x,iter=iter))
}
```

我们来尝试复现一下书上的表 6-1。我们考虑以下方程：

$$\begin{cases} -4x_1 + x_2 + x_3 + x_4 = 1 \\ x_1 - 4x_2 + x_3 + x_4 = 1 \\ x_1 + x_2 - 4x_3 + x_4 = 1 \\ x_1 + x_2 + x_3 - 4x_4 = 1 \end{cases}$$

我们遍历松弛因子 ω 从 1.0 至 1.9，分别求其迭代步数。

```
A<-matrix(1,ncol=4,nrow=4);diag(A)<--4
b<-rep(1,4)
for(omega in 10:19){
  iter<-SOR(A,b,omega=omega/10,esp=1e-5)$iter
  print(paste("omega is:",omega/10,",iteration time is:",iter))
}
```

```
## [1] "omega is: 1 ,iteration time is: 22"
```

```
## [1] "omega is: 1.1 ,iteration time is: 17"
## [1] "omega is: 1.2 ,iteration time is: 13"
## [1] "omega is: 1.3 ,iteration time is: 12"
## [1] "omega is: 1.4 ,iteration time is: 15"
## [1] "omega is: 1.5 ,iteration time is: 19"
## [1] "omega is: 1.6 ,iteration time is: 24"
## [1] "omega is: 1.7 ,iteration time is: 34"
## [1] "omega is: 1.8 ,iteration time is: 53"
## [1] "omega is: 1.9 ,iteration time is: 110"
```

因为误差计算方式的不同（书上使用了二范数，而我们使用了一范数），我们的结果与书上有着较小的差别，不过迭代次数的变化趋势仍然是十分正确的。在 $\omega = 1.3$ 时，程序有着最快的收敛速度。