计算物理期末大作业

梁旭民*

Cuiying Hornors College, Lanzhou University
liangxm15@lzu.edu.cn

Abstract

本次作业求解的是矩形导体边界接地情况下内部有一个点电荷的*Poisson*方程和圆形导体边界接地情况下内部有一个点电荷的*Poisson*方程，通过在*Cartesian*坐标系和极坐标系下分别生成差分格式矩阵，并处理对应的点电荷，得到了线性方程组。分别利用*Gauss*消元法、*Jacobi*迭代法、*Gauss Seidel*迭代法和*SOR*超松弛迭代法，可以解出线性方程组，即通过有限差分法求得了*Poisson*方程的解，不断增加了分成的格点我们可以比较不同方法求解线性方程的速度快慢。

I. Introduction

Poisson方程是数学中一个常见于静电学、机械工程和理论物理的偏微分方程，因法国数学家、几何学家及物理学家Poisson而得名的。

A. 方程的叙述

Poison方程为

$$\Delta\varphi = f$$

在这里Δ代表的是Laplace算子，而f和φ可以是在流形上的实数或复数值的方程。当流形属于Euclid空间，而Laplace算子通常表示为∇^2，因此Poisson方程通常写成

$$\nabla^2\varphi = f$$

在三维直角坐标系，可以写成

$$\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}+\frac{\partial^2}{\partial z^2}\right)\varphi(x,y,z) = f(x,y,z)$$

如果有$f(x,y,z)$恒等于0，这个方程就会变成一个齐次方程，这个方程称作“Laplace方程”

$$\Delta\varphi = 0$$

Poisson方程可以用Green函数来求解，现在也发展出很多种数值解，如松弛法迭代法。

*指导老师：齐新老师

B. 数学表达

通常泊松方程表示为

$$-\Delta\varphi = f$$

这里Δ代表Laplace算子，f为已知函数，而φ为未知函数。当$f = 0$时，这个方程被称为Laplace方程。

为了解Poisson方程我们需要更多的信息，比如Dirichlet边界条件:

$$\begin{cases} -\Delta\varphi = f & \text{in } \Omega \\ \varphi = g & \text{auf } \partial\Omega \end{cases}$$

其中$\Omega \subset \mathbb{R}^n$为有界开集。

这种情况下利用基础函数构建Poisson方程的解，Laplace方程的基础函数为:

$$\Phi(x) = \begin{cases} -\dfrac{1}{2\pi}\ln|x| & n = 2 \\ \dfrac{1}{n(n-2)\omega_n}\dfrac{1}{|x|^{n-2}} & n \geq 3 \end{cases}$$

其中ω_n为n维欧几里得空间中单位球面的体积，此时可通过卷积$(\Phi * f)$得到$-\Delta\varphi = f$的解。

为了使方程满足上述边界条件，我们使用Green函数

$$G(x,y) = \Phi(y-x) - \phi^x(y)$$

ϕ^x为一个校正函数，它满足

$$\begin{cases} \Delta\phi^x = 0 & \text{in } \Omega \\ \phi^x = \Phi(y-x) & \text{auf } \partial\Omega \end{cases}$$

通常情况下ϕ^x是依赖于Ω。

通过$G(x,y)$可以给出上述边界条件的解

$$u(x) = -\int_{\partial\Omega} g(y)\frac{\partial G}{\partial \nu}(x,y)\mathrm{d}\sigma(y) + \int_{\Omega} f(y)G(x,y)\mathrm{d}y$$

其中σ表示$\partial\Omega$上的曲面测度。

此方程的解也可通过变分法得到。

II. The Method

A. 问题描述

先通过有限差分法生成矩阵，利用消元法(或者LU法)、Jacobi迭代、Gauss Seidel迭代、超松弛(SOR)迭代分别求解矩形和圆形接地导体边界情况下,点电荷在空间中的电势分布。并不断增加矩阵尺寸,比较各种方法计算速度。其中矩形边界用Cartesian坐标系，圆形边界用极坐标系。

B. 有限差分法简介

在数学中，有限差分法（finite-difference methods，简称FDM），是一种微分方程数值方法，是通过有限差分来近似导数，从而寻求微分方程的近似解。

1. 由Taylor展开式的推导

首先假设要近似函数的各级导数都有良好的性质，依照Taylor定理，可以得到以下的Taylor展开式：

$$f(x_0+h) = f(x_0) + \frac{f'(x_0)}{1!}h + \frac{f^{(2)}(x_0)}{2!}h^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}h^n + R_n(x)$$

其中$n!$表示是n的阶乘，$R_n(x)$为余数，表示泰勒多项式和原函数之间的差。可以推导函数f一阶导数的近似值：

$$f(x_0+h) = f(x_0) + f'(x_0)h + R_1(x)$$

设$x_0 = a$，可得：

$$f(a+h) = f(a) + f'(a)h + R_1(x)$$

除以h可得：

$$\frac{f(a+h)}{h} = \frac{f(a)}{h} + f'(a) + \frac{R_1(x)}{h}$$

求解$f'(a)$：

$$f'(a) = \frac{f(a+h)-f(a)}{h} - \frac{R_1(x)}{h}$$

假设$R_1(x)$相当小，因此可以将函数$f(x)$的一阶导数近似为：

$$f'(a) \approx \frac{f(a+h)-f(a)}{h}$$

2. 准确度及误差

近似解的误差定义为近似解及解析解之间的差值。有限差分法的两个误差来源分别是舍入误差及截尾误差（或称为离散化误差），前者是因为电脑计算小数时四舍五入造成的误差，后者则是计算机内数字位数限制造成的误差。有限差分法是以在格点上函数的值为准

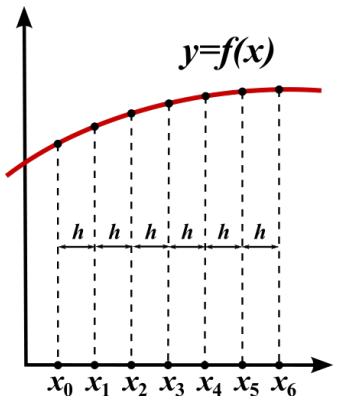

Figure 1: 有限差分法是以在格点上函数的值为准

在运用有限差分法求解一问题（或是说找到问题的近似解）时，第一步需要将问题的定义域离散化。一般会将问题的定义域用均匀的网格分割（见Figure 1）。因此有限差分法会制造一组导数的离散数值近似值。

一般会关注近似解的局部截尾误差，会用大O符号表示，局部截尾误差是指应用有限差分法一次后产生的误差，因此为$f'(x_i) - f'_i$，此时$f'(x_i)$是实际值，而f'_i为近似值。Taylor多项式的余数项有助于分析局部截尾误差。利用$f(x_0+h)$Taylor多项式的余数项，也就是

$$R_n(x_0+h) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(h)^{n+1}$$

其中$x_0 < \xi < x_0 + h$

可以找到局部截尾误差的主控项，例如用前项差分法计算一阶导数，已知$f(x_i)=f(x_0+ih)$,则有

$$f(x_0+ih)=f(x_0)+f'(x_0)ih+\frac{f''(\xi)}{2!}(ih)^2$$

利用一些代数的处理，可得

$$\frac{f(x_0+ih)-f(x_0)}{ih}=f'(x_0)+\frac{f''(\xi)}{2!}ih$$

注意到左边的量是有限差分法的近似，右边的量是待求解的量再加上一个余数，因此余数就是局部截尾误差。上述范例可以用下式表示：

$$\frac{f(x_0+ih)-f(x_0)}{ih}=f'(x_0)+O(h)$$

在此例中，局部截尾误差和时间格点的大小成正比。

C. Cartesian坐标下有限差分格式

Cartesian坐标下静电场的二维Poisson方程可以表示为

$$\nabla^2\varphi(x,y)=\frac{\partial^2\varphi}{\partial x^2}+\frac{\partial^2\varphi}{\partial y^2}=-\frac{-\rho(x,y)}{\varepsilon_0}$$

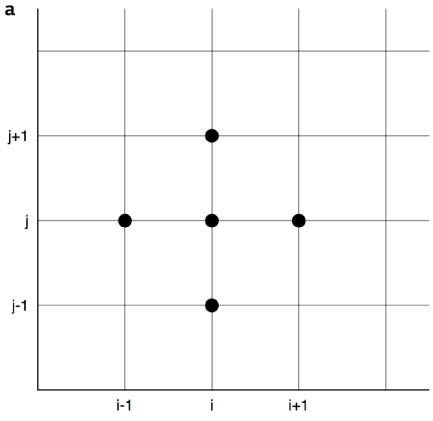

Figure 2: 二维*Cartesian*坐标网络五点差分格式

在二维网格中的差分格式可以写作

$$\nabla^2\varphi(x_i,y_j)=-\frac{\rho_{i,j}}{\varepsilon_0}$$
$$=\frac{1}{h^2}(\varphi_{i-1,j}+\varphi_{i+1,j}+\varphi_{i,j-1}+\varphi_{i,j+1}-4\phi_{i,j})$$

矩阵法是将未知量和已知量之间的线性关系表示为矩阵的形式,通过对矩阵进行消元和变换直接求出未知量的值。矩阵的边长等于总格点个数。可将矩阵表示为如下形式

$$\left[\begin{array}{ccc|ccc|ccc}
 & & & e & & & & & \\
 & \mathbf{B} & & & \ddots & & & & \\
 & & & & & e & & & \\
\hline
d & & & & & & e & & \\
 & \ddots & & & \ddots & & & \ddots & \\
 & & d & & & & & & e \\
\hline
 & & & d & & & & & \\
 & & & & \ddots & & & \mathbf{B} & \\
 & & & & & d & & &
\end{array}\right]$$

其中，矩阵B与参数分别为

$$\mathbf{B}=\begin{bmatrix} a & c & & \\ b & \ddots & \ddots & \\ & \ddots & \ddots & c \\ & & b & a \end{bmatrix} \qquad \begin{cases} a=4 \\ b=c=-1 \\ d=e=-1 \end{cases}$$

D. 极坐标下的有限差分格式

在极坐标系下，我们选择厚度非常小的圆柱(实际上是为了保证物理把厚度直接当1算)，则有：

$$\int_V\frac{\rho}{\varepsilon_0}dV=\frac{1}{\varepsilon_0}(\rho_{i,j}r_i\Delta\theta\Delta r)$$
$$\oint\vec{E}\cdot d\vec{S}=E^r_{i+\frac{1}{2},j}\cdot r_{i+\frac{1}{2}}\Delta\theta-E^r_{i-\frac{1}{2},j}\cdot r_{i-\frac{1}{2}}\Delta\theta$$
$$+E^\theta_{i,j+\frac{1}{2}}\cdot\Delta r-E^\theta_{i,j-\frac{1}{2}}\cdot r_{i+\frac{1}{2}}\Delta r$$

根据Gauss定理，即：

$$\int_V\frac{\rho}{\varepsilon_0}dV=\oint\vec{E}\cdot d\vec{S}$$

又：

$$\begin{cases} E^r_{i+\frac{1}{2},j}=-\dfrac{\varphi_{i+1,j}-\varphi_{i,j}}{\Delta r} \\ E^r_{i-\frac{1}{2},j}=-\dfrac{\varphi_{i,j}-\varphi_{i-1,j}}{\Delta r} \\ E^\theta_{i,j+\frac{1}{2}}=-\dfrac{\varphi_{i,j+1}-\varphi_{i,j}}{r_i\Delta\theta} \\ E^\theta_{i,j-\frac{1}{2}}=-\dfrac{\varphi_{i,j}-\varphi_{i,j-1}}{r_i\Delta\theta} \end{cases}$$

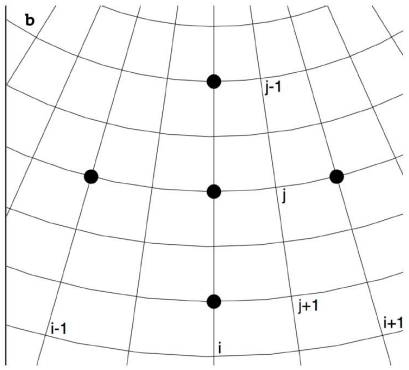

Figure 3: 二维极坐标网络五点差分格式

可以得到非原点处的五点差分公式：

$$
\begin{aligned}
&-\frac{\rho_{i,j}}{\varepsilon_0}=\frac{1}{r_i^2\Delta\theta^2}(\varphi_{i,j+1}-2\varphi_{i,j}+\varphi_{i,j-1})\\
&+\frac{1}{r_i\Delta r^2}\left(\varphi_{i+1,j}r_{i+\frac{1}{2}}-2\varphi_{i,j}r_i+\varphi_{i-1,j}r_{i-\frac{1}{2}}\right)\\
&=-\left(\frac{2}{r_i^2\Delta\theta^2}+\frac{2}{r_i\Delta r^2}\right)\varphi_{i,j}+\frac{r_{i+\frac{1}{2}}}{r_i\Delta r^2}\varphi_{i+1,j}\\
&+\frac{r_{i-\frac{1}{2}}}{r_i\Delta r^2}\varphi_{i-1,j}+\frac{1}{r_i^2\Delta\theta^2}\varphi_{i,j+1}+\frac{1}{r_i^2\Delta\theta^2}\varphi_{i,j-1}
\end{aligned}
$$

对于坐标原点，同样可以解得其差分公式为

$$
\frac{\rho_0}{\varepsilon_0}=\frac{2\Delta\theta}{\pi\Delta r^2}\sum_{j=1}^{N_\theta}(\varphi_{1,j}-\varphi_0)
$$

矩阵A可以表示为

$$
\left[\begin{array}{c|c|c|c}
m & n\dots n & & \\
\hline
d_1 & & e_1 & \\
\vdots & \mathbf{B} & \ddots & \\
d_1 & & & e_1 \\
\hline
 & d_2 & & e_2 \\
 & \ddots & \ddots & \ddots \\
 & & d_i & & e_i \\
\hline
 & & d & \\
 & & \ddots & \mathbf{B} \\
 & & & d
\end{array}\right]
$$

其中矩阵B和各项系数为

$$
\mathbf{B}=\begin{bmatrix}
a_i & c_i & & b_i \\
b_i & \ddots & \ddots & \\
 & \ddots & \ddots & c_i \\
b_i & & b_i & a_i
\end{bmatrix}
$$

$$
\begin{cases}
a_i=\dfrac{2}{\Delta r^2}+\dfrac{2}{r_i^2\Delta\theta^2} \\
b_i=c_i=-\dfrac{1}{r_i^2\Delta\theta^2} \\
d_i=-\dfrac{r_{i-\frac{1}{2}}}{r_i\Delta r^2} \qquad e_i=-\dfrac{r_{i+\frac{1}{2}}}{r_i\Delta r^2} \\
m=\dfrac{2N_\theta\Delta\theta}{\pi\Delta r^2} \qquad n=-\dfrac{2\Delta}{\pi\Delta r^2}
\end{cases}
$$

E. Gauss消元法

数学上，Gauss消元法是线性代数中的一个算法，可用来为线性方程组求解，求出矩阵的秩，以及求出可逆方阵的逆矩阵。该方法以数学家Karl Gauss命名，但最早出现于中国古籍《九章算术》，成书于约公元前150年。由于Gauss消元法在线性代数课程中为最基本的算法，这里只做简单分析:

Gauss消元法的算法复杂度是$O(n3)$；这就是说，如果系数矩阵的是$n\times n$，那么Gauss消元法所需要的计算量大约与n^3成比例。

Gauss消元法可以用在电脑中来解决数千条等式及未知数。不过，如果有过百万条等式时，这个算法会十分费时。一些极大的方程组通常会用迭代法来解决。亦有一些方法特地用来解决一些有特别排列的系数的方程组。

Gauss消元法可用在任何域中。

Gauss消元法对于一些矩阵来说是稳定的。对于普遍的矩阵来说，Gauss消元法在应用上通常也是稳定的，不过亦有例外。

F. Jacobi迭代法

在数值线性代数中，Jacobi迭代法是一种解对角元素几乎都是各行和各列的绝对值最大的值的线性方程组的算法。求解出每个对角元素并插入近似值。不断迭代直至收敛。这个算法是雅可比矩阵的精简版。方法的名字来源于德国数学家Kerl Jacobi。

1. 数学描述

给定一个$n\times n$的线性方程组

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

其中:

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

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

A可以分解成对角部分D和剩余部分R:

$$A = D + R$$

其中

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

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

线性方程组可以重写为:

$$D\mathbf{x} = \mathbf{b} - R\mathbf{x}$$

Jacobi法是一种迭代方法。在每一次迭代中，上一次算出的x被用在右侧，用来算出左侧的新的x。这个过程可以如下表示：

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

对每个元素可以用以下公式：

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i \in \mathbb{N}_+$$

注意计算$x_i^{(k+1)}$需要$x^{(k)}$中除了自己之外的每个元素。不像Gauss Seidel迭代，我们不能用$x_i^{(k+1)}$覆盖$x^{(k)}$，因为在接下来的计算中还要用到这些值。这是雅可比和高斯-塞德尔方法最显著的差别，也是为什么前者可以用并行算法而后者不能的原因。最小需要的存储空间是两个长度为n的向量。

2. 算法描述

选择一个初始猜想值x^0

while 未收敛

$for\ i := 1\ step\ until\ n\ do$

$\sigma = 0$

$for\ j := 1\ step\ until\ n\ do$

$if\ j! = i\ then$

$\sigma = \sigma + a_{ij} x_j^{(k-1)}$

$end\ if$

$end(j - loop)$

$x_i^{(k)} = \frac{(b_i - \sigma)}{a_{ii}}$

$end(i - loop)$

检查是否收敛

end (未收敛时继续循环)

3. 收敛

标准的收敛情况是当迭代矩阵的谱半径

$$\rho(D^{-1}R) < 1$$

保证收敛的条件是矩阵A为严格或不可约地对角占优矩阵。严格的行对角占优矩阵指对于每一行，对角线上的元素之绝对值大于其余元素绝对值的和，即

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

有时即使不满足此条件，Jacobi法仍可收敛。

G. Gauss Seidel迭代法

Gauss Seidel法是数值线性代数中的一种迭代法，可用来求出线性方程组解的近似值。该方法以Kerl Gauss和Philipp Ludwig von Seidel命名。同Jacobi法一样，Gauss Seidel迭代是基于矩阵分解原理。

1. 数学描述

对于一个含有n个未知量及n个等式的如下线性方程组

$$\begin{aligned} a_{11} \cdot x_1 + a_{12} \cdot x_2 + \ldots + a_{1n} \cdot x_n &= b_1, \\ a_{21} \cdot x_1 + a_{22} \cdot x_2 + \ldots + a_{2n} \cdot x_n &= b_2, \\ \vdots \qquad\qquad\qquad\qquad &= \vdots \\ a_{n1} \cdot x_1 + a_{n2} \cdot x_2 + \ldots + a_{nn} \cdot x_n &= b_n \end{aligned}$$

为了求这个方程组的解$\vec{x}$，我们使用迭代法。k用来计量迭代步数。给定该方程组解的一个近似值$\vec{x}^k \in \mathbb{R}^n$。在求$k+1$步近似值时，我们利用第m个方程求解第m个未知量。在求解过程中，所有已解出的$k+1$步元素都被直接使用。这一点与Jacobi法不同。对于每个元素可以使用如下公式(其中，$1 \le m \le n$)

$$x_m^{k+1} = \frac{1}{a_{mm}} \left(b_m - \sum_{j=1}^{m-1} a_{mj} \cdot x_j^{k+1} - \sum_{j=m+1}^{n} a_{mj} \cdot x_j^k \right)$$

重复上述的求解过程，可以得到一个线性方程组解的近似值数列：$\vec{x}^0, \vec{x}^1, \vec{x}^2, \ldots$。在该方法收敛的前提下，此数列收敛于$\vec{x}$。为了保证该方法可以进行，主对角线元素$a_{mm}$需非零。

2. 矩阵分解

线性方程组的系数$a_{ij}\,(i,j = 1,2,\ldots,n)$可以被写成矩阵形式$A \in \mathbb{R}^{n\times n}$, 该矩阵的第i行第j列元素满足$(A)_{i,j} = a_{ij}$。方程组的右边项可以被写成向量形式$\vec{b} \in \mathbb{R}^n$：$(\vec{b})_i = b_i$。线性方程组因此可以被写成矩阵运算形式

$$A\vec{x} = \vec{b}$$

矩阵A可以分解成如下形式

$$A = D + L + U$$

其中$D \in \mathbb{R}^{n\times n}$为一个对角矩阵满足$(D)_{i,i} = (A)_{i,i}, L, U$均为严格三角矩阵：$L$为严格下三角矩阵，$U$为严格上三角矩阵。

Gauss Seidel迭代的每一步可以写成如下形式

$$D\vec{x}^{k+1} = \vec{b} - L\vec{x}^{k+1} - U\vec{x}^k$$

该公式推导如下：

如上形式来自于Gauss Seidel迭代的元素公式: 对于第m个未知量$(\vec{x}^{k+1})_m = x_m^{k+1}$, 我们可以得出

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

$$\Rightarrow$$

$$a_{mm}x_m^{k+1} = b_m - \sum_{j=1}^{n}(L)_{m,j}x_j^{k+1} - \sum_{j=1}^{n}(U)_{m,j}x_j^k$$

已知$a_{mm} \neq 0, (L)_{m,j} = 0\,(\forall j \ge m)$以及$(U)_{m,j} = 0\,(\forall j \le m)$，因此可以得出

$$x_m^{k+1} = \frac{1}{a_{mm}} \left(b_m - \sum_{j=1}^{m-1}(L)_{m,j}x_j^{k+1} - \sum_{j=m+1}^{n}(U)_{m,j}x_j^k \right) = \frac{1}{a_{mm}} \left(b_m - \sum_{j=1}^{m-1} a_{mj}x_j^{k+1} - \sum_{j=m+1}^{n} a_{mj}x_j^k \right)$$

3. 算法描述

因为元素可以被重新赋值为在这个算法中计算得到的新值，所以只需要保存一个向量，而向量索引被省略。该算法如下：

输入:A, b
输出:ϕ

初始化一个的猜测结果ϕ

repeat until convergence（收敛）
  for i from 1 *until n do*
    $\sigma \leftarrow 0$
    for j from 1 *until n do*
      if $j \neq i$ *then*
        $\sigma \leftarrow \sigma + a_{ij}\phi_j$
      end if
    end$(j - loop)$
    $\phi_i \leftarrow \frac{1}{a_{ii}}(b_i - \sigma)$
  end$(i - loop)$
  check if convergence is reached
(检查是否已收敛)
end$(repeat)$

H. 超松弛(SOR)迭代法

在数值线性代数中，超松弛(SOR)的方法是用于求解线性方程组的Gauss-Seidel方法的变形，可以产生更快地收敛。类似的方法可以用于任何缓慢收敛的迭代过程。

它由David M. Young，Jr和Stanley P. Frankel于1950年同时设计，目的是自动解决数字计算机上的线性系统。在Young和Frankel的工作之前已经使用了过度放松的方法。一个例子是Lewis Fry Richardson的方法，以及RV

Southwell开发的方法。然而，这些方法是为人类计算器计算而设计的，它们需要一些专业知识来确保解决方案的融合，这使得它们不适用于数字计算机上的编程。这些方面在David M. Young，Jr的论文中讨论。

1. 数学描述

给定具有未知$\mathbf{x}$的n个线性方程的平方系统：

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

其中

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

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

然后A可以分解为对角分量D，严格分为下三角分量L和上三角分量D：

$$A = D + L + U$$

其中

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

$$L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}$$

$$U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

线性方程组可以改写为：

$$(D + \omega L)\mathbf{x} = \omega\mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x}$$

对于常数$\omega > 1$，称为弛豫因子。

连续过度松弛的方法是一种迭代技术，它使用$\mathbf{x}$的右侧的前一个值来解决$\mathbf{x}$的该表达式的左侧。从分析上看，这可以写成：

$$\begin{aligned} \mathbf{x}^{(k+1)} &= (D + \omega L)^{-1}\left(\omega\mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x}^{(k)}\right) \\ &= L_w\mathbf{x}^{(k)} + \mathbf{c} \end{aligned}$$

其中，$\mathbf{x}^{(k)}$是k阶的近似或迭代$\mathbf{x}$和$\mathbf{x}^{(k+1)}$是下一个或$k+1$次迭代$\mathbf{x}$。但是，通过利用$(D + \omega L)$的三角形形式，可以使用正向替换顺序计算$\mathbf{x}^{(k+1)}$的元素：

$$\begin{aligned} x_i^{(k+1)} &= (1 - \omega)x_i^{(k)} \\ &+ \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{(k+1)} - \sum_{j>i} a_{ij}x_j^{(k)}\right) \end{aligned}$$

其中，$i \in \mathbb{N}_+$

选择合适的弛豫因子ω并不容易，并且取决于系数矩阵的性质。1947年，Ostrowski证明了如果A是一个正定对角的矩阵，则对于$0 < \omega < 2$有$\rho(L_\omega) < 1$。因此，迭代过程的收敛如下，但我们通常对更快的收敛而不仅仅是收敛感兴趣。

可以分析地导出SOR方法的收敛速度。假设

- 合适的松弛因子$\omega \in (0, 2)$
- Jacobi迭代矩阵$C_{Jac} := I - D^{-1}A$有实数本征值
- Jacobi迭代法是收敛的:$\mu := \rho(C_{Jac}) < 1$
- 存在唯一解:$detA \neq 0$

则收敛速度$\rho(C_\omega)$可以表示为

$$\begin{cases} \frac{1}{4}\left(\omega\mu + \sqrt{\omega^2\mu^2 - 4(\omega - 1)}\right)^2, & 0 < \omega \leq \omega_{\text{opt}} \\ \omega - 1, & \omega_{\text{opt}} < \omega < 2 \end{cases}$$

其中最合适的松弛因子由下式给出

$$\omega_{\text{opt}} := 1 + \left(\frac{\mu}{1 + \sqrt{1 - \mu^2}}\right)^2$$

2. 算法描述

由于元素可以在此算法中计算时被覆盖，因此只需要一个存储向量，并省略向量索引。算法如下：

输入:A, b, ω

输出:ϕ

选择一个初始猜测ϕ重复直到收敛的解决方案

$$
\begin{aligned}
&for\ i\ from\ 1\ until\ n\ do\\
&\quad \sigma \leftarrow 0\\
&\quad for\ j\ from\ 1\ until\ n\ do\\
&\quad\quad if\ j \neq i\ then\\
&\quad\quad\quad \sigma \leftarrow \sigma + a_{ij}\phi_j\\
&\quad\quad end\ if\\
&\quad end(j-loop)\\
&\quad 检查是否达到收敛\\
&end(repeat)
\end{aligned}
$$

需要注意的是$(1-\omega)\phi_i + \frac{\omega}{a_{ii}}(b_i - \sigma)$也可以写为$\phi_i + \omega\left(\frac{b_i-\sigma}{a_{ii}} - \phi_i\right)$从而在每次for-loop的迭代的外部保存一个乘法，减少计算量。

III. Results and Discussion

在(-3.5,-2)位置处(可以在生成矩阵的程序修改x0,y0参数从而改变位置，见附录A)放置一个点电荷，这里点电荷我们直接用1代替，求解矩阵大小为169，则可以求解出电势随坐标分布见Figure 4。

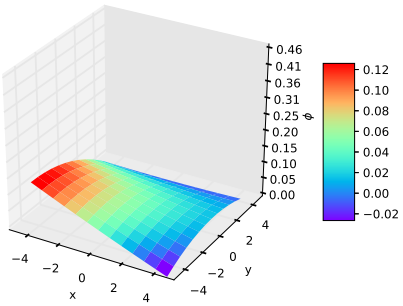

Figure 4: 矩形边界接地内部点电荷电势分布

由于使用2次幂级数拟合使得电势分布相对平滑，并不特别符合物理事实，所以使用Matlab中的Curve fitting工具进行拟合，矩形边界条件电势分布见Figure 5。观察图像可以发现电势最高点为(-3.5,-2.0)位置附近，其余位置的电势近似成负幂次衰减，因此得出结论：有限差分法求解矩形边界条件Poisson方程得到的解较为符合物理认识，并且整个图像较为合理。

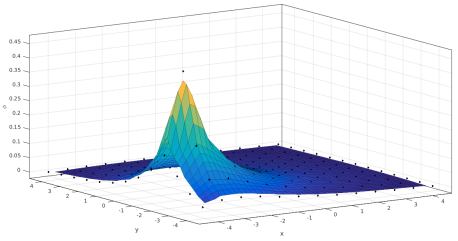

Figure 5: 矩形边界接地内部点电荷电势分布

在(2,0)位置处(可以在生成矩阵的程序修改x0,y0参数从而改变位置，见附录B)放置一个点电荷，这里点电荷我们直接用1代替，求解矩阵大小为170，则可以求解出电势随坐标分布见Figure 6。

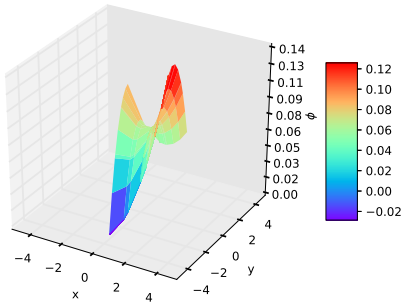

Figure 6: 圆形边界接地内部点电荷电势分布

同样使用Matlab中的Curve fitting工具进行拟合，极坐标边界条件电势分布见Figure 7。

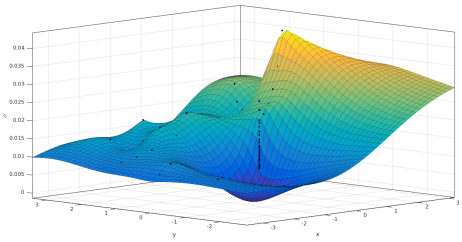

Figure 7: 圆形边界接地内部点电荷电势分布

为了对各种求解线性方程组的方法进行计算速度分析，我们统计了不同算法处理相同矩阵的运行时间，见Figure 8和Figure 9。

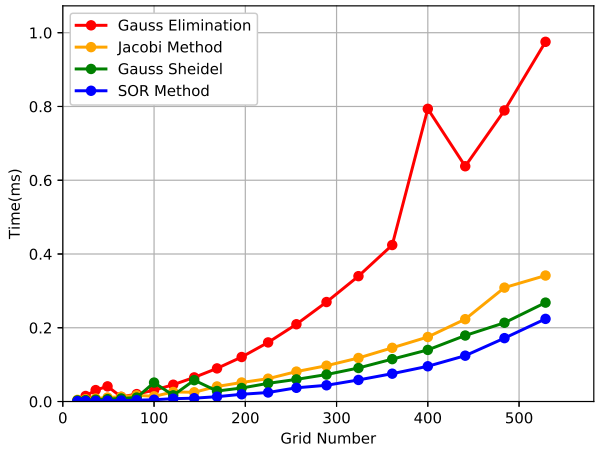

Figure 8: 矩形边界下不同算法计算速度分析

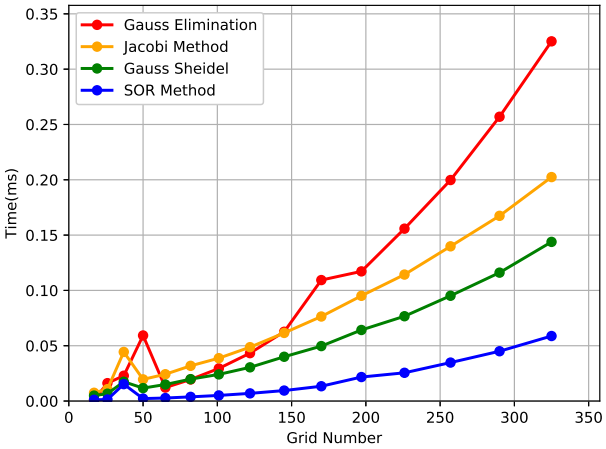

Figure 9: 圆形边界下不同算法计算速度分析

由于采用了Python time宏包内部的time.clock()函数，可能对于记录时间一问题处理并不是特别完美，可以看到曲线可能会存在波动，但是总体可以看出，不同求解线性方程组的方法中计算速度从快到慢依次为:SOR超松弛迭代法、Gauss Seidel迭代法、Jacobi迭代法、Gauss消元法。

Appendices

A. 矩形边界生成Cartesian坐标下有限差分格式矩阵

矩形边界生成Cartesian坐标下有限差分格式矩阵,为了可以不断扩大矩阵尺寸，我们将不同大小的线性方程组分别存到Rmatrix.xlsx工作簿的不同Sheet表格中。

Input Python source:

```
import numpy as np
import openpyxl
from openpyxl import Workbook
wb=Workbook()
wb.remove_sheet(wb.get_sheet_by_name("Sheet"))
#参数设置
ba=-5.0;bb=5.0;bc=5.0;bd=-5.0
nmin=5;nmax=18;x0=2.0;y0=0.0;rho=1
for index in range(nmin,nmax):
    h=(bc-ba)/index;N=(index-1)*(index-1);m=index-1
#生成数组矩阵
    A=np.zeros(shape=(N,N))
    for i in range(N):
        if(i>=0):
            A[i-1][i]=-1.0
            A[i][i]=4.0
            A[i][i-1]=-1.0
        if(i+m-N<0):
            A[i][(i+m)-N]=-1.0
            A[(i+m)-N][i]=-1.0
    for j in range(0,N,m):
        A[j-1][j]=0;A[j][j-1]=0
    b=np.zeros(shape=(N))
    b[int((x0-ba)/h)+int((y0-bd)/h)]=rho
#储存数组矩阵'''
    sheet=wb.create_sheet("Sheet%d"%(index-nmin+1))
    for i in range(N):
        sheet.cell(row=i+1,column=N+1,value=b[i])
        for j in range(N):
            sheet.cell(row=i+1,column=j+1,value=A[i][j])
wb.save('Rmatrix.xlsx')
```

B. 圆形边界生成极坐标下有限差分格式矩阵

圆形边界生成极坐标下有限差分格式矩阵,为了可以不断扩大矩阵尺寸，我们将不同大小的线性方程组分别存到Cmatrix.xlsx工作簿的不同Sheet表格中。

Input Python source:

```
import numpy as np
import openpyxl
from openpyxl import Workbook
wb=Workbook()
wb.remove_sheet(wb.get_sheet_by_name("Sheet"))
#常数设置
pi=3.14159265358979323846264433
#参数设置
ba=-5.0;bb=5.0;bc=5.0;bd=-5.0
nmin=5;nmax=18;x0=2.0;y0=0.0;rho=1
for index in range(nmin,nmax):
    M=index-1;N=1+(index-1)*(index-1);Deltar=bb/index;Deltatheta=2*pi/index
    #生成系数初始化
    a=np.zeros(shape=M+1)
    b=np.zeros(shape=M+1)
    c=np.zeros(shape=M+1)
    d=np.zeros(shape=M+1)
    e=np.zeros(shape=M+1)
    for i in range(1,M+1):
        a[i]=2.0/(Deltar*Deltar)+2.0/(i*Deltar*i*Deltar*Deltatheta*Deltatheta)
        b[i]=-1.0/(i*Deltar*i*Deltar*Deltatheta*Deltatheta)
        c[i]=-1.0/(i*Deltar*i*Deltar*Deltatheta*Deltatheta)
        d[i]=-1.0*(i-0.5)*Deltar/(i*Deltar*Deltar*Deltar)
        e[i]=-1.0*(i+0.5)*Deltar/(i*Deltar*Deltar*Deltar)
    m=(2.0*M*Deltatheta)/(pi*Deltar*Deltar)
    n=-1.0*(2.0*Deltatheta)/(pi*Deltar*Deltar)
    #数组矩阵初始化
    A=np.zeros(shape=(N,N))
    A[0][0]=m
    for i in range(1,M+1):
        A[0][i]=n
    for i in range(1,M+1):
        A[i][0]=d[1]
    for i in range(N):
        if((i!=0)and(i+M-N)<0):
            A[i][(i+M)-N]=e[int((i-1)/M)-M]
    for i in range(N):
        if((i!=0)and(i+M-N)<0):
            A[(i+M)-N][i]=d[int((i-1)/M)+1-M]
    for k in range(M):
        B=np.zeros(shape=(M,M))
        for l in range(M):
            B[l-1][l]=c[k+1]
            B[l][l]=a[k+1]
            B[l][l-1]=b[k+1]
        for i in range(1+k*M,M+1+k*M):
            for j in range(1+k*M,M+1+k*M):
                A[i][j]=B[i-k*M-1][j-k*M-1]
    b=np.zeros(shape=(N))
    b[int(np.sqrt(x0*x0+y0*y0)/Deltar)+int(np.arctan(y0/x0)/Deltatheta)]=rho
#生成并储存矩阵
    sheet=wb.create_sheet("Sheet%d"%(index-nmin+1))
    for i in range(N):
        sheet.cell(row=i+1,column=N+1,value=b[i])
        for j in range(N):
            sheet.cell(row=i+1,column=j+1,value=A[i][j])
wb.save('Cmatrix.xlsx')
```

C. 不同求解线性方程组的方法

将求解线性方程组的方法分别写成子函数合在一起写成一个叫做method.py的Python程序,方便调用。

Input Python source:

```
import numpy as np
def gaussian_elimination(A,b):
    n=len(A)
    x=np.ones([n,1])
    b=b.reshape(n,1)
    A_b=np.hstack((A,b))
    for i in range(n):
        for j in range(i+1,n):
            A_b[j]=A_b[j]-A_b[i]*(A_b[j][i]/A_b[i][i])
    for i in range(n-1,-1,-1):
        sum=A_b[i][n]
        for j in range(i+1,n):
            sum=sum-A_b[i][j]*x[j]
        x[i]=sum/A_b[i][i]
    return x.T[0]
def jacobi(A,b):
    n=len(A)
    x=np.ones([n,1])
    b=b.reshape(n,1)
    B=np.zeros([n,n])
    D=np.zeros([n,n])
    d=np.ones([n,1])
    B=A.copy()
    for i in range(n):
        B[i,i]=0.0
        D[i,i]=A[i,i].copy()
    B=np.dot(-np.linalg.inv(D),B)
    d=np.dot(np.linalg.inv(D),b)
    x1=x
    x2=np.dot(B,x)+d
    while(np.linalg.norm(x1-x2,np.inf)>10e-10):
        x1=x2
        x2=np.dot(B,x1)+d
    return x2.T[0]
def gauss_seidel(A,b):
    n=len(A)
    x=np.ones([n,1])
    b=b.reshape(n,1)
    B=np.tril(A)
    U=np.triu(A,k=1)
    G=-np.dot(np.linalg.inv(B),U)
    d=np.dot(np.linalg.inv(B),b)
    x1=x
    x2=np.dot(G,x)+d
    while(np.linalg.norm(x1-x2,np.inf)>10e-10):
        x1=x2
        x2=np.dot(G,x1)+d
    return x1.T[0]
def sor(A,b):
    w=1.15
    n=len(A)
    x=np.ones([n,1])
    b=b.reshape(n,1)
    L=np.tril(A,k=-1)
    U=np.triu(A,k=1)
    D=np.zeros([n,n])
    for i in range(n):
```

```
58          D[i,i]=A[i,i].copy()
59      x1=x
60      x2=np.dot(np.dot(np.linalg.inv(D+w*L),((1-w)*D-w*U)),x)+w*np.dot(np.linalg.
61  inv(D+w*L),b)
62      while(np.linalg.norm(x1-x2,np.inf)>10e-10):
63          x1=x2
64          x2=np.dot(np.dot(np.linalg.inv(D+w*L),((1-w)*D-w*U)),x1)+w*np.dot(np.linalg.
65  inv(D+w*L),b)
66      return x1.T[0]
```

D. 计算分析绘图

通过读取Rmatrix.xlsx和Cmatrix.xlsx文件中的矩阵数据，并调用上述子函数分别进行运算，我们可以绘制出不同大小矩阵的Cartesian坐标下矩形边界内部有一个点电荷的导体接地的Poisson方程及极坐标下圆形边界内部有一个点电荷的导体接地的Poisson方程的解，这里为了展示方便，只绘制选定最大限度的矩阵的情况。

Input Python source:

```
import numpy as np
import scipy.linalg
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.interpolate import griddata
from matplotlib import cm
from matplotlib.ticker import LinearLocator,FormatStrFormatter
import method as med
import openpyxl
from openpyxl import Workbook
#直角坐标线性方程组输入
#wb=openpyxl.load_workbook('Rmatrix.xlsx')
#极坐标线性方程组输入
wb=openpyxl.load_workbook('Cmatrix.xlsx')
sheets=wb.get_sheet_names()
GridNumber=np.zeros(shape=(len(sheets)))
#选择不同的index从而输出分格个数不同的解的图像
index=10#选择Sheet10进行计算
sheet=wb.get_sheet_by_name('Sheet'+str(index))
A=np.zeros(shape=(sheet.max_row,sheet.max_row))
b=np.zeros(shape=(sheet.max_row))
GridNumber[index-1]=(sheet.max_row)
for k in range(1,sheet.max_column):
    b[k-1]=sheet.cell(row=k,column=sheet.max_column).value
for i in range(1,sheet.max_column):
    for j in range(1,sheet.max_column):
        A[i-1][j-1]=sheet.cell(row=i,column=j).value
'''
#直角坐标设置
ba=-5.0;bb=5.0;bc=5.0;bd=-5.0;nmin=5;h=(bc-ba)/(index+nmin-1)
#位置坐标设置
xx=np.arange(ba+h,bc,h)
x=np.repeat(xx,index+nmin-2)
yy=np.arange(bd+h,bb,h)
y=np.tile(yy,index+nmin-2)
'''
#极坐标设置
ba=-5.0;bb=5.0;bc=5.0;bd=-5.0
#常数设置
pi=3.14159265358979323846264433
#参数设置
nmin=5;x0=1.0;y0=1.0;rho=1
M=index-1;N=1+(index+nmin-2)*(index+nmin-2)
Deltar=bb/(index+nmin-1);Deltatheta=2*pi/(index+nmin-1)
#位置坐标设置
x=np.zeros(shape=N);x[0]=0
y=np.zeros(shape=N);y[0]=0
for i in range(1,M+1):
    for j in range(1,M+1):
        x[i*j]=(i*Deltar)*np.cos(j*Deltatheta)
        y[i*j]=(i*Deltar)*np.sin(j*Deltatheta)

#绘制解的图像部分
phi1=med.gaussian_elimination(A,b)
phi2=med.jacobi(A,b)
```

```
phi3=med.gauss_seidel(A,b)
phi4=med.sor(A,b)
print(x,y,phi1)
#这部分代码可以实现将不同方法生成的图像分别绘制并储存
for fn in range(1,5):
    varName='phi%d'%fn
    data=np.c_[x,y,locals()[varName]]
    fig=plt.figure()
    #X,Y=np.meshgrid(np.arange(ba+h,bc,h),np.arange(bd+h,bb,h))
    X,Y=np.meshgrid(np.sort(x),np.sort(y))
    XX=X.flatten()
    YY=Y.flatten()
    A=np.c_[np.ones(data.shape[0]),data[:,:2],np.prod(data[:,:2],axis=1),data[:,:2]**2]
    C,_,_,_=scipy.linalg.lstsq(A,data[:,2])
    Z = np.dot(np.c_[np.ones(XX.shape),XX,YY,XX*YY,XX**2,YY**2],C).reshape(X.shape)
    fig=plt.figure()
    ax=fig.gca(projection='3d')
    surf=ax.plot_surface(X,Y,Z,rstride=1,cstride=1,cmap='rainbow')
    ax.set_xlim(ba,bc)
    ax.set_ylim(bd,bb)
    ax.set_zlim(0,np.max(locals()[varName]))
    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('$\phi$')
    fig.colorbar(surf,shrink=0.5,aspect=5)
    fname='figR%d.eps'%fn
    #fname='figC%d.eps'%fn
    plt.savefig(fname)
    plt.clf()
    del data
```

E. 计算分析绘图

通过读取Rmatrix.xlsx和Cmatrix.xlsx文件中的矩阵数据，并调用上述子函数分别进行运算，通过统计不同算法的运行时间，我们可以绘制出不同算法运行时间与矩阵大小的关系。

Input Python source:

```
import numpy as np
import time
import method as med
import openpyxl
from openpyxl import Workbook
#直角坐标线性方程组输入
wb=openpyxl.load_workbook('Rmatrix.xlsx')
#极坐标线性方程组输入
#wb=openpyxl.load_workbook('Cmatrix.xlsx')
sheets=wb.get_sheet_names()
t1=np.zeros(shape=(len(sheets)))
t2=np.zeros(shape=(len(sheets)))
t3=np.zeros(shape=(len(sheets)))
t4=np.zeros(shape=(len(sheets)))
GridNumber=np.zeros(shape=(len(sheets)))
for index in range(1,len(sheets)+1):
    sheet=wb.get_sheet_by_name('Sheet'+str(index))
    A=np.zeros(shape=(sheet.max_row,sheet.max_row))
    b=np.zeros(shape=(sheet.max_row))
    GridNumber[index-1]=(sheet.max_row)
    for k in range(1,sheet.max_column):
        b[k-1]=sheet.cell(row=k,column=sheet.max_column).value
    for i in range(1,sheet.max_column):
        for j in range(1,sheet.max_column):
            A[i-1][j-1]=sheet.cell(row=i,column=j).value
    time1=time.clock()
    med.gaussian_elimination(A,b)
    time2=time.clock()
    med.jacobi(A,b)
    time3=time.clock()
    med.gauss_seidel(A,b)
    time4=time.clock()
    med.sor(A,b)
    time5=time.clock()
    t1[index-1]=time2-time1
    t2[index-1]=time3-time2
    t3[index-1]=time4-time3
    t4[index-1]=time5-time4
    #为了避免覆盖节约内存我们在每次循环的时候将重复变量删除
    del sheet,A,b,time1,time2,time3,time4,time5
#绘制判断不同方法收敛速度快慢部分
import matplotlib.pyplot as plt
plt.figure()
plt.grid()
plt.plot(GridNumber,t1,'ro-',label="Gauss Elimination",color="red",linewidth=2)
plt.plot(GridNumber,t2,'ro-',label="Jacobi Method",color="orange",linewidth=2)
plt.plot(GridNumber,t3,'ro-',label="Gauss Sheidel",color="green",linewidth=2)
plt.plot(GridNumber,t4,'ro-',label="SOR Method",color="blue",linewidth=2)
plt.xlabel("Grid Number")
plt.ylabel("Time(ms)")
plt.xlim(0,GridNumber[-1]*1.1)
plt.ylim(0,t1[-1]*1.1)
plt.legend()
plt.savefig("timeR.eps")
#plt.savefig("timeC.eps")
```

参考文献

[1] Jackson, Julia A.; Mehl, James P.; Neuendorf, Klaus K. E. (编), Glossary of Geology, American Geological Institute, Springer: 503, 2005, ISBN 9780922152766.

[2] Poisson Equation at EqWorld: The World of Mathematical Equations.

[3] L.C. Evans, Partial Differential Equations, American Mathematical Society, Providence, 1998. ISBN 0-8218-0772-2.

[4] A. D. Polyanin, Handbook of Linear Partial Differential Equations for Engineers and Scientists, Chapman & Hall/CRC Press, Boca Raton, 2002. ISBN 1-58488-299-9.

[5] Crank, J. The Mathematics of Diffusion. 2nd Edition, Oxford, 1975, p. 143.

[6] K.W. Morton and D.F. Mayers, Numerical Solution of Partial Differential Equations, An Introduction. Cambridge University Press, 2005.

[7] Oliver Rübenkönig, The Finite Difference Method (FDM) - An introduction, (2006) Albert Ludwigs University of Freiburg.

[8] Autar Kaw and E. Eric Kalu, Numerical Methods with Applications, (2008).

[9] 第八章方程,《九章算术》.

[10] Atkinson, Kendall A. An Introduction to Numerical Analysis, 第二版, John Wiley & Sons, New York, 1989年ISBN 978-0-471-50023-0.

[11] Golub, Gene H., and Van Loan, Charles F. Matrix computations, 第三版, Johns Hopkins, Baltimore, 1996年ISBN 978-0-8018-5414-9.

[12] Black, Noel; Moore, Shirley; and Weisstein, Eric W. Jacobi method. MathWorld.

[13] A. Hadjidimos, Successive overrelaxation (SOR) and related methods, Journal of Computational and Applied Mathematics 123 (2000), 177-199.

[14] Yousef Saad, Iterative Methods for Sparse Linear Systems, 1st edition, PWS, 1996.

[15] Netlib's copy of "Templates for the Solution of Linear Systems", by Barrett et al.

[16] . Richard S. Varga 2002 Matrix Iterative Analysis, Second ed. (of 1962 Prentice Hall edition), Springer-Verlag.

[17] David M. Young, Jr. Iterative Solution of Large Linear Systems, Academic Press, 1971. (reprinted by Dover, 2003)