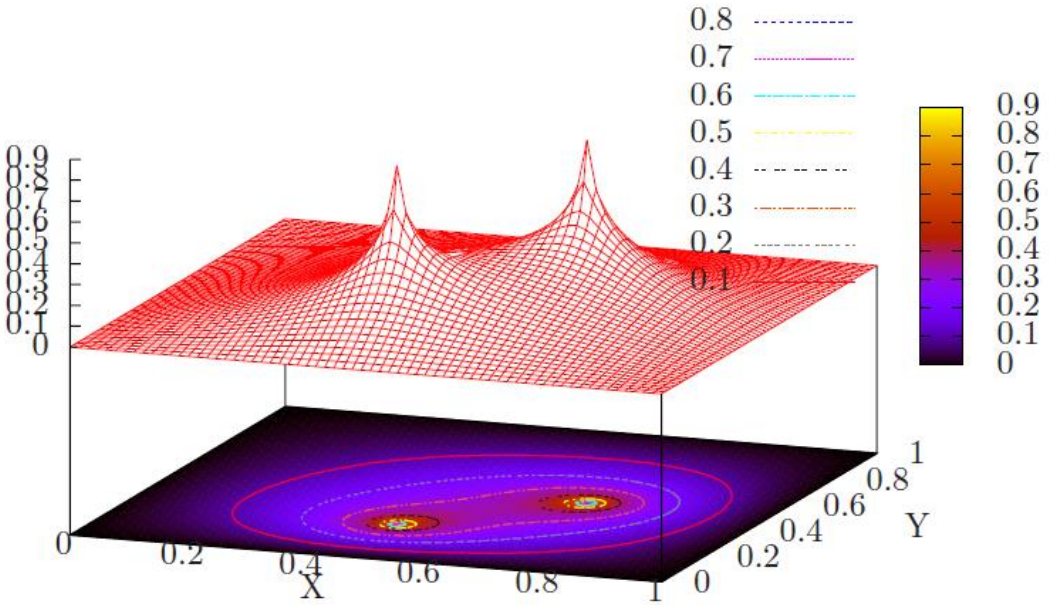
计算物理 第二部分
第4讲



李强　北京大学物理学院中楼411

qliphy0@pku.edu.cn， 15210033542

http://www.phy.pku.edu.cn/~qiangli/CP2017.html

4. 偏微分方程B

2阶线性PDE: Elliptic, Parabolic, Hyperbolic

椭圆方程(泊松方程):

2D/3D

spectral method 傅立叶变换

抛物形方程(2D扩散方程):

二阶线形偏微分方程分类

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Du_x + Eu_y + Fu + G = 0,$$

其中A, B, C, D, E, F, G为x, y的函数.

椭圆(Elliptic)： $B^2 - AC < 0$,

Laplace方程 $\nabla^2 u = u_{xx} + u_{yy} = 0$

Poisson方程 $\nabla^2 u = f(x, y)$

抛物(Parabolic)： $B^2 - AC = 0$,

Diffusion方程 $u_t = au_{xx}$

双曲(Hyperbolic): $B^2 - AC > 0$,

Wave方程 $u_{tt} = cu_{xx}$

$$Lu = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{i,j}\frac{\partial^2 u}{\partial x_i \partial x_j} \quad \text{plus lower-order terms} = 0.$$

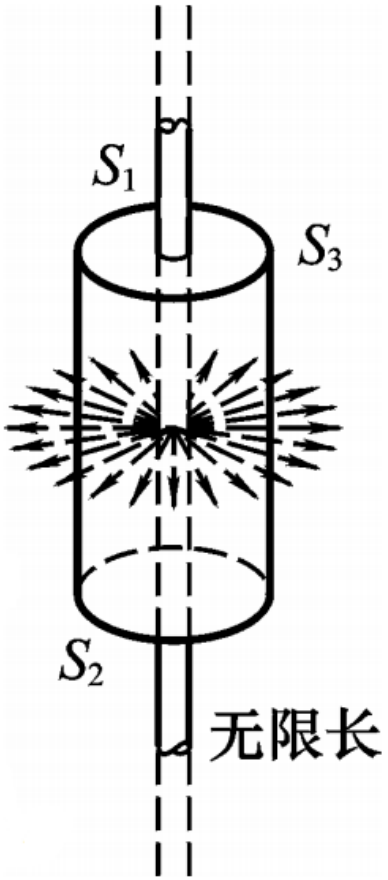
The classification depends upon the signature of the eigenvalues of the coefficient matrix $a_{i,j}$.

1. Elliptic: The eigenvalues are all positive or all negative.
2. Parabolic : The eigenvalues are all positive or all negative, save one that is zero.
3. Hyperbolic: There is only one negative eigenvalue and all the rest are positive, or there is only one positive eigenvalue and all the rest are negative.
4. Ultrahyperbolic: There is more than one positive eigenvalue and more than one negative eigenvalue There is only a limited theory for ultra-hyperbolic equations (Courant and Hilbert, 1962).

2D泊松方程 （椭圆型方程）

$$\frac{\partial^2 u(x,y)}{\partial x^2}+\frac{\partial^2 u(x,y)}{\partial y^2}=v(x,y)$$

$$x_l \le x \le x_h \qquad 0 \le y \le L$$



$$E=\frac{1}{2\pi\varepsilon_0}\frac{\lambda}{r}$$

参照之前介绍过的1D泊松情形，对x和y分布采用二阶中心差分方法

$$u_{xx}h^2=u(x_{i+1})-2u(x_i)+u(x_{i-1})$$
$$u_{yy}h^2=u(y_{i+1})-2u(y_i)+u(y_{i-1})$$

然而，这样的方法得到的一系列方程并不能约化为简单的三对角矩阵。采用迭代算法，需要比较复杂的技术，而且收敛性不是特别好，比如Jacobi方法，Gauss-Seidel方法，Multi-grid方法等等。参见Numerical recipes in C: the art of scientific computing, W.H.Press, S. A. Teukolsky et.al.

下面，我们着重介绍Spectral方法，即傅里叶变换。

大 连 理 工 大 学 学 报
Journal of Dalian University of Technology
Vol. 37, No. 4
Jul. 1 9 9 7

块三对角矩阵方程的追赶法及其应用*

宫 野[a] 龙永兴[b] 王友年[a] 邓新绿[a]
a) (大连理工大学物理系 116024) b) (西南物理研究院 610041)

摘要 导出了块三对角矩阵方程追赶法的一套递推关系式, 并编制出相应的计算机 Code. 该 Code 具有良好的实用价值, 可供在实际问题中使用.

关键词: 三对角矩阵; 矩阵追赶法/递推关系式
分类号: O241.6; O151.21

设线性代数方程组 $\boldsymbol{AX}=\boldsymbol{F}$, 即

$$\begin{pmatrix} B_1 & C_1 & & 0 \\ A_1 & B_2 & C_2 & \\ & \ddots & \ddots & \ddots \\ & A_{n-2} & B_{n-1} & C_{n-1} \\ 0 & & A_{n-1} & B_n \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_{n-1} \\ X_n \end{pmatrix} \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_{n-1} \\ F_n \end{pmatrix}$$

其中: A_i、B_i、C_i 均为 $M\times M$ 阶子矩阵(分块矩阵), F_i 为 M 个列向量. $i=1, 2, \cdots, n$.

2D泊松方程 Dirichlet边界条件

$$u(x,0) = u(x,L) = 0$$

在y方向做傅里叶展开

$$u(x,y) = \sum_{j=1}^{\infty} U_j(x)\sin(j\pi\, y/L)$$

$$v(x,y) = \sum_{j=1}^{\infty} V_j(x)\sin(j\pi\, y/L)$$

y方向的边界条件自动满足。且在区间[0-L]，$\sin(j\pi\, y/L)$构成完备正交基：

$$\frac{2}{L}\int_0^L \sin(j\pi\, y/L)\sin(k\pi\, y/L)dy = \delta_{jk} \qquad j,k > 0$$

$$\frac{2}{L}\sin(j\pi\, y/L)\sin(k\pi\, y/L) = \frac{1}{L}\cos\frac{\pi y}{L}(j-k) - \frac{1}{L}\cos\frac{\pi y}{L}(j+k)$$

$$U_j(x) = \frac{2}{L}\int_0^L u(x,y)\sin(j\pi\, y/L)dy\,,$$

$$V_j(x) = \frac{2}{L}\int_0^L v(x,y)\sin(j\pi\, y/L)dy$$

2D泊松方程 Dirichlet边界条件

$$u(x,0) = u(x,L) = 0$$

$$u(x,y) = \sum_{j=1}^{\infty} U_j(x)\sin(j\pi\, y/L)$$

$$v(x,y) = \sum_{j=1}^{\infty} V_j(x)\sin(j\pi\, y/L)$$

代入

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = v(x,y)$$

$$\frac{d^2 U_j(x)}{dx^2} - \frac{j^2\pi^2}{L^2} U_j(x) = V_j(x), \qquad j=1,\ldots\infty$$

在数值计算中，我们要截断傅里叶展开到第J项，即*j=1, …J*。
这相当于在y方向上做了如下差分：$y_j = j\,L/J.$

$$j + J \leftrightarrow J - j$$

对每一个j，我们可以在x方向做2阶中心差分

$$U_{i-1,j} - (2 + j^2k^2)U_{i,j} + U_{i+1,j} = V_{i,j}(\delta x)^2$$

2D泊松方程 Dirichlet边界条件

$$u(x,0) = u(x,L) = 0$$

$$U_{i-1,j} - (2 + j^2k^2)U_{i,j} + U_{i+1,j} = V_{i,j}(\delta x)^2$$

$$i = 1, N, \text{ and } j = 1, J$$

$$U_{i,j} = U_j(x_i),\ V_{i,j} = V_j(x_i),\ \text{and } k = \pi\delta x/L$$

X方向的边界条件为

$$\alpha_l U_j(x_l) + \beta_L \frac{dU_j(x_l)}{dx} = \Gamma_{lj},$$

$$\alpha_h U_j(x_h) + \beta_h \frac{dU_j(x_h)}{dx} = \Gamma_{hj},$$

$$\Gamma_{lj} = \frac{2}{L}\int_0^L \gamma_l(y)\sin(j\pi y/L)dy$$

$$\Gamma_{hj} = \frac{2}{L}\int_0^L \gamma_h(y)\sin(j\pi y/L)dy$$

$$U_{0,j} = \frac{\Gamma_{lj}\delta x - \beta_l U_{1,j}}{\alpha_l \delta x - \beta_l},$$

$$U_{N+1,j} = \frac{\Gamma_{hj}\delta x - \beta_h U_{N,j}}{\alpha_h \delta x - \beta_h}$$

我们得到了J个Uncoupled三对角矩阵方程！

2D泊松方程 Neumann边界条件

$$\frac{\partial u(x,y=0)}{\partial y}=\frac{\partial u(x,y=L)}{\partial y}=0$$

在y方向做傅里叶展开

$$u(x,y)=\sum_{j=0}^{\infty}U_j(x)cos(j\pi\, y/L)$$

$$v(x,y)=\sum_{j=0}^{\infty}V_j(x)\cos(j\pi\, y/L)$$

y方向的边界条件自动满足。且在区间[0-L]，$\cos(j\pi\, y/L)$构成完备正交基：

$$\frac{2}{L}\int_0^L\cos(j\pi\, y/L)\cos(k\pi\, y/L)dy=\delta_{jk}\qquad j,k\geq 0$$

$$\frac{2}{L}\cos(j\pi\, y/L)\cos(k\pi\, y/L)=\frac{1}{L}\cos\frac{\pi y}{L}(j-k)+\frac{1}{L}\cos\frac{\pi y}{L}(j+k)$$

$$U_j(x)=\frac{2}{L}\int_0^L u(x,y)\cos(j\pi\, y/L)dy\,,$$

$$V_j(x)=\frac{2}{L}\int_0^L v(x,y)\cos(j\pi\, y/L)dy$$

注意，对j=0，

$$\frac{2}{L}\text{ -> }\frac{1}{L}$$

2D泊松方程 Neumann边界条件

$$\frac{\partial u(x, y=0)}{\partial y} = \frac{\partial u(x, y=L)}{\partial y} = 0$$

$$U_{i-1,j} - (2 + j^2 k^2) U_{i,j} + U_{i+1,j} = V_{i,j} (\delta x)^2$$

$$i = 1, N, \text{ and } \boldsymbol{j = 0, J}$$

$$U_{i,j} = U_j(x_i),\ V_{i,j} = V_j(x_i),\ \text{and } k = \pi \delta x / L$$

X方向的边界条件为

$$\alpha_l U_j(x_l) + \beta_L \frac{dU_j(x_l)}{dx} = \Gamma_{lj},$$

$$\alpha_h U_j(x_h) + \beta_h \frac{dU_j(x_h)}{dx} = \Gamma_{hj},$$

$$\Gamma_{lj} = \frac{2}{L} \int_0^L \gamma_l(y) \cos(j\pi y / L) dy$$

$$\Gamma_{hj} = \frac{2}{L} \int_0^L \gamma_h(y) \cos(j\pi y / L) dy$$

$$U_{0,j} = \frac{\Gamma_{lj} \delta x - \beta_l U_{1,j}}{\alpha_l \delta x - \beta_l},$$

$$U_{N+1,j} = \frac{\Gamma_{hj} \delta x - \beta_h U_{N,j}}{\alpha_h \delta x - \beta_h}$$

我们得到了J+1个Uncoupled三对角矩阵方程！

注意，对j=0，$\frac{2}{L}$ -> $\frac{1}{L}$

离散傅里叶变换

```
#include "TMath.h"
void fourier()
{
      gRandom = new TRandom3(0);
      int n=400;
      double f[400], c[400];
      double pi=TMath::ACos(-1.0);
      double xl=0.0, xh=2*pi;
      double x;
      double dx=(xh-xl)/float(n);
      for (int i = 0; i < n; ++i) {
        c[i]=0.0;
      }
      for (int i = 0; i < n; ++i) {
        double rr=gRandom->Uniform(-1,1);
        x=xl+i*dx;
```

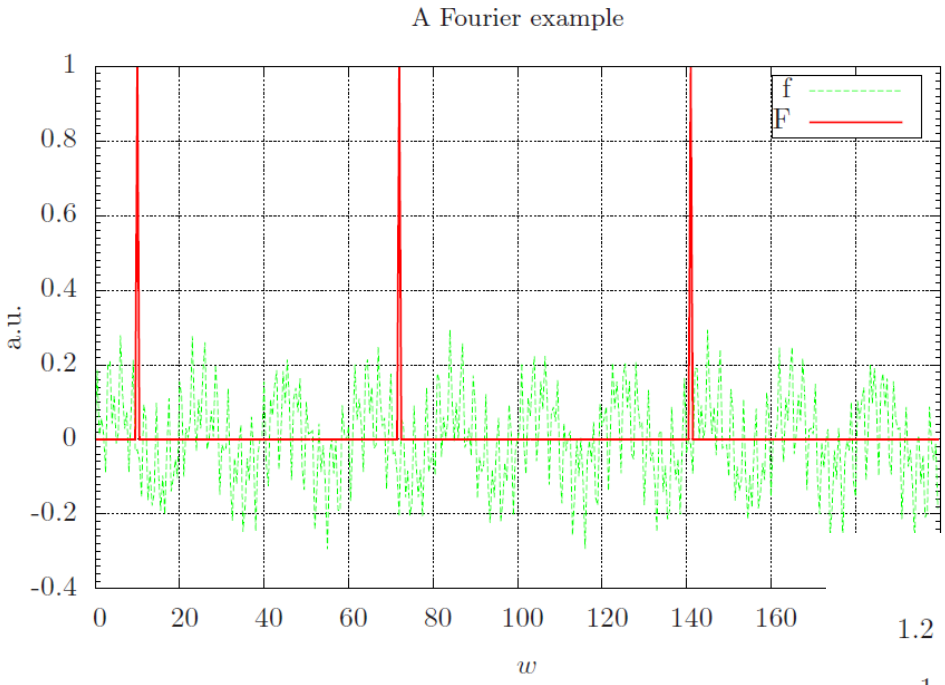
```
f[i]=(TMath::Sin(10*x)+TMath::Sin(72*x)+TMat
h::Sin(141*x));
      }
      for (int i = 0; i < n; ++i) {
        x=xl+i*dx;
        for (int j = 0; j < n; ++j) {

c[i]+=TMath::Sin(i*j*pi/float(n))*f[j]*2.0/float
(n);
        }
      }
      char *out= "plot.gnu";
      FILE *fp = fopen(out,"w");
      for (int i = 0; i < n; ++i) {
      fprintf(fp,"%15.7f %15.7f %15.7f \n",
float(i)/2.0, 0.1*f[i], c[i]);
      }
}
```

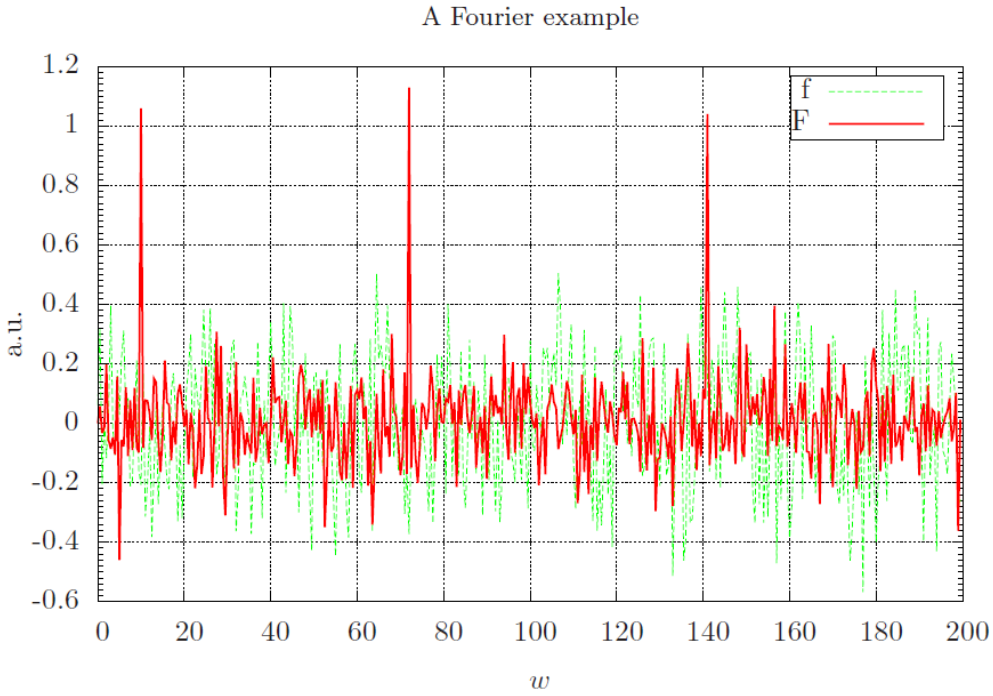
$$\frac{2}{L}\int_0^L \sin(j\pi\, y/L)\sin(k\pi\, y/L)dy = \delta_{jk}$$

$$\Rightarrow \frac{1}{\pi}\int_0^{2\pi} \sin(j\, y/2)\sin(k\, y/2)dy = \delta_{jk}$$

离散傅里叶变换

$$U_j = \frac{2}{L}\int_0^L f(y)\sin(j\pi\, y/L)$$
$$= \frac{1}{\pi}\sum_{i=0}^{n} f(y_i)\sin(j\, y_i/2)\delta y$$
$$= \frac{2}{n}\sum_{i=0}^{n} f(y_i)\sin(\boldsymbol{j} * \boldsymbol{i} * \pi/n)$$





With random shift

离散傅里叶变换

$$\int_0^L e^{imt}e^{-int}dt = L\delta_{mn}$$

$$f(x) = \sum_{m=0}^{N-1} F(m)e^{imx}$$

$$F(n) = \frac{1}{L}\int_0^L f(x)e^{-inx}dx$$

$$\rightarrow \frac{1}{L}\sum_{m=0}^{N-1} f(x_m)e^{-inx_m}\frac{L}{N} = \frac{1}{N}\sum_{m=0}^{N-1} f_m e^{-imnL/N}$$

$x: 0-2\pi$, $t_n = 2\pi n/N$, n=0,1,..N-1

$$\omega \equiv e^{-i2\pi/N}$$

$$a(t_n) = \frac{1}{N}\sum_{k=0}^{N-1} \mathrm{A}(k)e^{ikt_n}$$

$$A_k = \sum_{n=0}^{N-1} a(t_n)\, e^{-ikt_n} = \sum_{n=0}^{N-1} a_n\, \omega^{nk}$$

以n=3，即N=n+1=4=2^2为例， $\boldsymbol{\omega^1}$=-i

$$A_0 = a_0\omega^0 + a_1\omega^0 + a_2\omega^0 + a_3\omega^0,$$
$$A_1 = a_0\omega^0 + a_1\omega^1 + a_2\omega^2 + a_3\omega^3,$$
$$A_2 = a_0\omega^0 + a_1\omega^2 + a_2\omega^4 + a_3\omega^6,$$
$$A_3 = a_0\omega^0 + a_1\omega^3 + a_2\omega^6 + a_3\omega^9$$

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

需要进行$N^2 = 16$次复数乘法和加法运算

快速傅里叶变换

N=n+1=4=2^2

$$A_k=\sum_{n=0}^{3}(-i)^{kn}a_n=a_0+(-i)^k a_1+(-i)^{2k}a_2+(-i)^{3k}a_3=a_0+(-i)^k a_1+(-1)^k a_2+i^k a_3$$

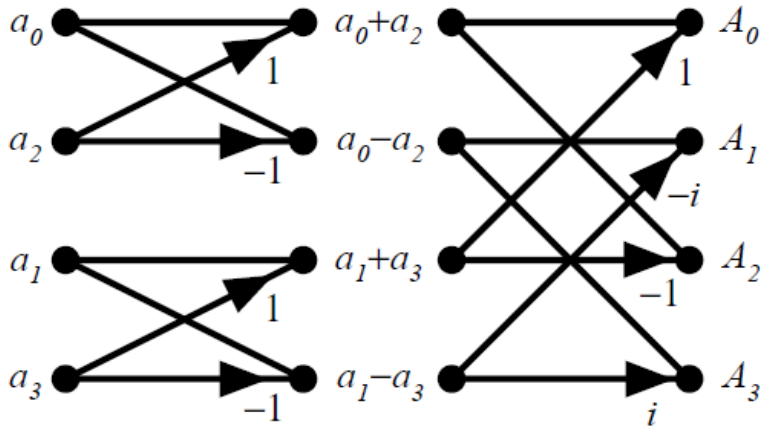
$$\begin{aligned}A_0&=a_0+a_1+a_2+a_3\\A_1&=a_0-ia_1-a_2+ia_3\\A_2&=a_0-a_1+a_2-a_3\\A_3&=a_0+ia_1-a_2-ia_3\end{aligned}$$

$$\begin{pmatrix}A_0\\A_1\\A_2\\A_3\end{pmatrix}=\begin{pmatrix}1&1&1&1\\1&-i&-1&i\\1&-1&1&-1\\1&i&-1&-i\end{pmatrix}\begin{pmatrix}a_0\\a_1\\a_2\\a_3\end{pmatrix}$$

Bit-reversion

为了加快计算，可以如下进行：

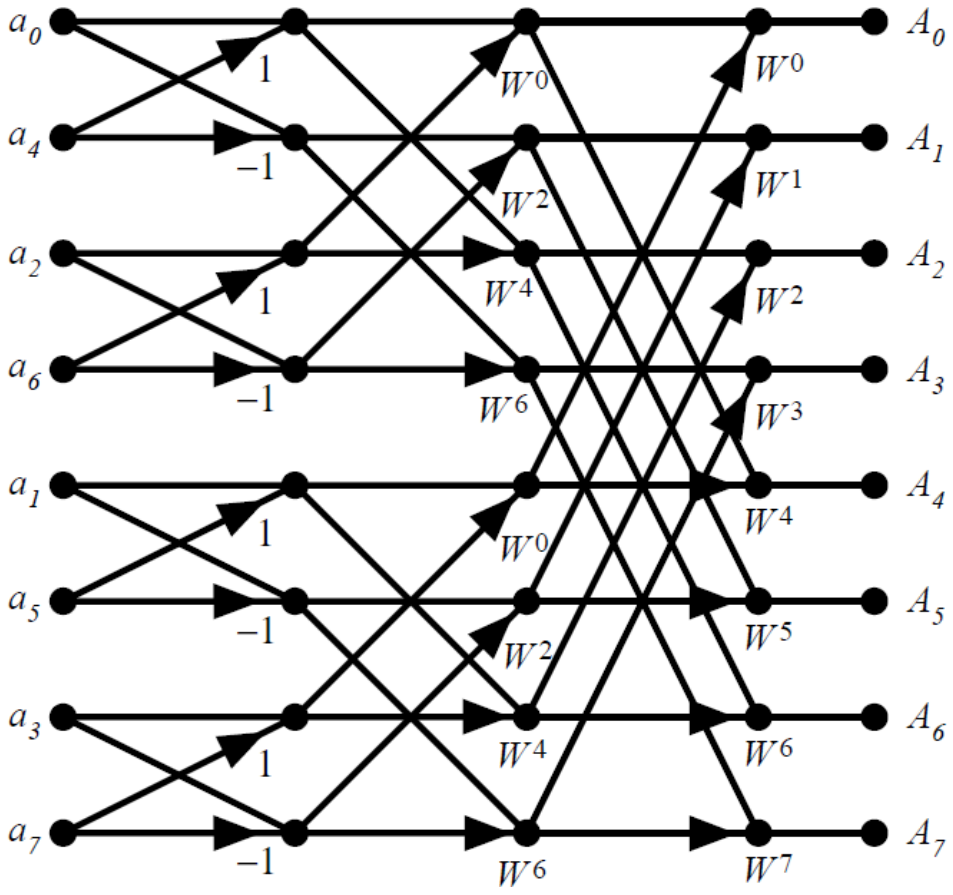
$$\begin{aligned}A_0&=(a_0+a_2)+(a_1+a_3)\\A_1&=(a_0-a_2)-i(a_1-a_3)\\A_2&=(a_0+a_2)-(a_1+a_3)\\A_3&=(a_0-a_2)+i(a_1-a_3)\end{aligned}$$



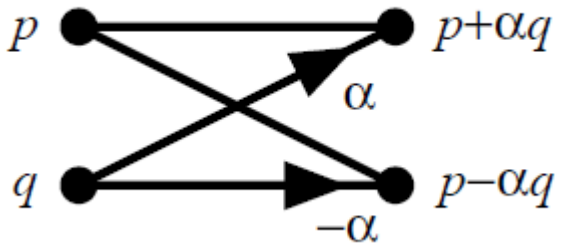
只需要进行8次加法运算

快速傅里叶变换

$N=2^3$



One Butterfly Operation：



Bit-reversion

j	0	1	2	3	4	5	6	7
n_j	0	4	2	6	1	5	3	7
j base 2	000	001	010	011	100	101	110	111
n_j base 2	000	100	010	110	001	101	011	111

The FFT algorithm decomposes the DFT into $\log_2 N$ stages, each of which consists of $N/2$ *butterfly* computations.

快速傅里叶变换

$$W = W_8 = e^{-i\pi/4}$$

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \\ A_7 \end{pmatrix} = \begin{pmatrix} W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 \\ W^0 & W^2 & W^4 & W^6 & W^0 & W^2 & W^4 & W^6 \\ W^0 & W^3 & W^6 & W^1 & W^4 & W^7 & W^2 & W^5 \\ W^0 & W^4 & W^0 & W^4 & W^0 & W^4 & W^0 & W^4 \\ W^0 & W^5 & W^2 & W^7 & W^4 & W^1 & W^6 & W^3 \\ W^0 & W^6 & W^4 & W^2 & W^0 & W^6 & W^4 & W^2 \\ W^0 & W^7 & W^6 & W^5 & W^4 & W^3 & W^2 & W^1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix}$$

multiplying by the dense (no zeros) matrix is more expensive than multiplying by the three sparse matrices on the bottom

For $N = 2^r$, the factorization would involve r matrices of size $N \times N$, each with 2 non- zero entries in each row and column

$$= \begin{pmatrix} 1 & \cdot & \cdot & \cdot & W^0 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & W^1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & W^2 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & W^3 \\ 1 & \cdot & \cdot & \cdot & W^4 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot & \cdot & W^5 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot & \cdot & \cdot & W^6 & \cdot \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & W^7 \end{pmatrix} \begin{pmatrix} 1 & \cdot & W^0 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & W^2 & \cdot & \cdot & \cdot & \cdot \\ 1 & \cdot & W^4 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & W^6 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & W^0 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & W^2 \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & W^4 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & W^6 \end{pmatrix} \begin{pmatrix} 1 & W^0 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & W^4 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & W^0 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & W^4 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & W^0 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & W^4 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & W^0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & W^4 \end{pmatrix} \begin{pmatrix} a_0 \\ a_4 \\ a_2 \\ a_6 \\ a_1 \\ a_5 \\ a_3 \\ a_7 \end{pmatrix}$$

快速傅里叶变换2

对N=2^r,可将N个离散的傅里叶变换写成N/2个偶数项和N/2个奇数项之和：

$$
\begin{aligned}
&r=1, \quad A_j = a_0 + (-1)^j a_1, \\
&r=2, \quad A_j = a_0 + (-1)^j a_2 + \omega^j (a_1 + (-1)^j a_3), \\
&r=3, \quad A_j = a_0 + (-1)^j a_4 + \omega^{2j} (a_2 + (-1)^j a_6) \\
&\qquad\qquad + \omega^j \left[a_1 + (-1)^j a_5 + \omega^{2j} (a_3 + (-1)^j a_7) \right] \\
&\cdots\cdots
\end{aligned}
$$

可以看出，从a_0开始，后一项是前面所有项升指标$2^{r-i}(i=1,\ldots,M)$后乘$\omega^{2^{r-i}j}$

这样，可以把一个N项求和变成两个N/2项和；由于N=2^r，这个过程可以重复r-1次，直到每个求和只有2项。

Algorithms for the Ages 二十世纪十大算法

1946: The Metropolis Algorithm for Monte Carlo. Through the use of random processes, this algorithm offers an efficient way to stumble toward answers to problems that are too complicated to solve exactly.
1947: Simplex Method for Linear Programming. An elegant solution to a common problem in planning and decision-making.
1950: Krylov Subspace Iteration Method. A technique for rapidly solving the linear equations that abound in scientific computation.
1951: The Decompositional Approach to Matrix Computations. A suite of techniques for numerical linear algebra.
1957: The Fortran Optimizing Compiler. Turns high-level code into efficient computer-readable code.
1959: QR Algorithm for Computing Eigenvalues. Another crucial matrix operation made swift and practical.
1962: Quicksort Algorithms for Sorting. For the efficient handling of large databases.
1965: Fast Fourier Transform. Perhaps the most ubiquitous algorithm in use today, it breaks down waveforms (like sound) into periodic components.
1977: Integer Relation Detection. A fast method for spotting simple equations satisfied by collections of seemingly unrelated numbers.
1987: Fast Multipole Method. A breakthrough in dealing with the complexity of n-body calculations, applied in problems ranging from celestial mechanics to protein folding.

好的数值算法的特点

- 面向计算机，提供切实可行的算法。只能包括计算机能够直接处理的加减乘除运算、逻辑运算、以及内置的简单函数计算等。
- 能任意逼近，并有可靠的理论分析(收敛性、稳定性、误差分析)，能达到精度要求，对近似算法要保证收敛性和稳定性。
- 省时间、省资源，并有良好的计算复杂性。计算复杂性是指算法所包含的运算次数和所需的储存量。好的算法计算效率高、计算复杂度低(耗时少，存储小)。
- 多次广泛的上机试验证明算法稳定、高效。

简单来说，好的数值算法本质上有三个要求：**好，快，省**。好，指算的准确，误差小，计算效果好；快，指计算步骤少，效率高；省，指计算的时间和存储代价小。【参考书】N.J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, 1996.

快速傅里叶变换

The **Cooley–Tukey** algorithm, named after J.W. Cooley and John Tukey, is the most common fast Fourier transform (FFT) algorithm. It re-expresses the discrete Fourier transform (DFT) of an arbitrary composite size **N = N1N2** in terms of N1 smaller DFTs of sizes N2, recursively, to reduce the computation time to O(N log N) for highly composite N (smooth numbers). Because of the algorithm's importance, specific variants and implementation styles have become known by their own names, as described below.

An Algorithm for the Machine Calculation of Complex Fourier Series

By James W. Cooley and John W. Tukey

An efficient method for the calculation of the interactions of a 2^m factorial experiment was introduced by Yates and is widely known by his name. The generalization to 3^m was given by Box et al. [1]. Good [2] generalized these methods and gave elegant algorithms for which one class of applications is the calculation of Fourier series. In their full generality, Good's methods are applicable to certain problems in which one must multiply an N-vector by an $N \times N$ matrix which can be factored into m sparse matrices, where m is proportional to $\log N$. This results in a procedure requiring a number of operations proportional to $N \log N$ rather than N^2. These methods are applied here to the calculation of complex Fourier series. They are useful in situations where the number of data points is, or can be chosen to be, a highly composite number. The algorithm is here derived and presented in a rather different form. Attention is given to the choice of N. It is also shown how special advantage can be obtained in the use of a binary computer with $N = 2^m$ and how the entire calculation can be performed within the array of N data storage locations used for the given Fourier coefficients.

An Algorithm for the Machine Calculation of Complex Fourier Series
Author(s): James W. Cooley and John W. Tukey
Source: *Mathematics of Computation*, Vol. 19, No. 90 (Apr., 1965), pp. 297-301
Published by: American Mathematical Society
Stable URL: http://www.jstor.org/stable/2003354
Accessed: 11/11/2008 10:32

快速傅里叶变换

http://www.fftw.org/

FFTW is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data (as well as of even/odd data, i.e. the discrete cosine/sine transforms).

FFTW

研究熟悉

Steven G. Johnson joined the faculty of Applied Mathematics at MIT in 2004. He received his Ph. D. in 2001 from the Dept. of Physics at MIT, where he also received undergraduate degrees in computer science and mathematics. His recent work, besides FFTW, has focused on the theory of photonic crystals: electromagnetism in nano-structured media. This has ranged from general research in semi-analytical and numerical methods for electromagnetism, to the design of integrated optical devices, to the developement of optical fibers that guide light within an air core to circumvent limits of solid materials. His Ph. D. thesis was published as a book, *Photonic Crystals: The Road from Theory to Practice*, in 2002.

Matteo Frigo received his Ph. D. in 1999 from the Dept. of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology (MIT). Besides FFTW, his research interests include the theory and implementation of Cilk (a multi-threaded system for parallel programming), cache-oblivious algorithms, and software radios. In addition, he has implemented a gas analyzer that is used for clinical tests on lungs.

Joint recipient, with Steven G. Johnson, of the 1999 J. H. Wilkinson Prize for Numerical Software, in recognition of their work on FFTW.

http://www.fftw.org/fftw-paper-ieee.pdf

复傅里叶变换 vs Sin-/Cos-傅里叶变换

$$F_j = \frac{1}{2J}\sum_{k=0}^{2J-1} f_k \exp(-ijk\pi/J)$$

$$f_j = \sum_{k=0}^{2J-1} F_k \exp(ijk\pi/J) \qquad \text{for } j = 0, 2J-1,$$

注意，这里指标周期为2J。

$$\exp(-i\frac{k}{J}\pi) = Cos(\frac{k}{J}\pi) - iSin(\frac{k}{J}\pi)$$

$$\exp(-i\frac{2J-k}{J}\pi) = Cos(\frac{k}{J}\pi) + iSin(\frac{k}{J}\pi)$$

$$Sin(\frac{2J-k}{J}\pi) = -Sin(\frac{k}{J}\pi)$$

$$Cos(\frac{2J-k}{J}\pi) = Cos(\frac{k}{J}\pi)$$

Thus, for a sine transform we write:

$$f_{2J-j} = -f_j,$$
$$F_{2J-j} = -F_j,$$

for $j = 1, J-1$, in which case

$$F_j^S = 2iF_j = -2\mathrm{Im}(F_j)$$

for a cosine transform we write:

$$f_{2J-j} = f_j,$$
$$F_{2J-j} = F_j,$$

for $j = 1, J-1$, in which case

$$F_j^C = 2\mathrm{Re}(F_j)$$

复傅里叶变换 vs Sin-/Cos-傅里叶变换

```
// Calculates Fourier-cosine transform of array f in array F
void fft_forward_cos (Array<double,1> f, Array<double,1>& F)
{
  // Find J. Declare local arrays.
  int J = f.extent(0) - 1;
  int N = 2 * J;
  fftw_complex ff[N], FF[N];

  // Load and extend data
  c_re (ff[0]) = f(0); c_im (ff[0]) = 0.;
  c_re (ff[J]) = f(J); c_im (ff[J]) = 0.;
  for (int j = 1; j < J; j++)
    {
      c_re (ff[j]) = f(j); c_im (ff[j]) = 0.;
      c_re (ff[2*J-j]) = f(j); c_im (ff[2*J-j]) = 0.;
    }


  // Call fftw routine
  fftw_plan p = fftw_create_plan (N, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_one (p, ff, FF);
  fftw_destroy_plan (p);

  // Unload data
  F(0) = c_re (FF[0]); F(J) = c_re (FF[J]);
  for (int j = 1; j < J; j++)
    {
      F(j) = 2. * c_re (FF[j]);
    }

  // Normalize data
  F /= 2. * double (J);
}
```

Cos-傅里叶变换

```
// Calculates inverse Fourier-cosine transform of array F in array f
void fft_backward_cos (Array<double,1> F, Array<double,1>& f)
{
  // Find J. Declare local arrays.
  int J = f.extent(0) - 1;
  int N = 2 * J;
  fftw_complex ff[N], FF[N];

  // Load and extend data
  c_re (FF[0]) = F(0); c_im (FF[0]) = 0.;
  c_re (FF[J]) = F(J); c_im (FF[J]) = 0.;
  for (int j = 1; j < J; j++)
    {
      c_re (FF[j]) = F(j) / 2.; c_im (FF[j]) = 0.;
      FF[2*J-j] = FF[j];
    }

  // Call fftw routine
  fftw_plan p = fftw_create_plan (N, FFTW_BACKWARD, FFTW_ESTIMATE);
  fftw_one (p, FF, ff);
  fftw_destroy_plan (p);

  // Unload data
  f(0) = c_re (ff[0]); f(J) = c_re (ff[J]);
  for (int j = 1; j < J; j++)
    {
      f(j) = c_re (ff[j]);
    }
}
```

Cos-傅里叶反变换

复傅里叶变换 vs Sin-/Cos-傅里叶变换

```
// Calculates Fourier-sine transform of array f in array F
void fft_forward_sin (Array<double,1> f, Array<double,1>& F)
{
  // Find J. Declare local arrays.
  int J = f.extent(0) - 1;
  int N = 2 * J;
  fftw_complex ff[N], FF[N];

  // Load and extend data
  c_re (ff[0]) = 0.; c_im (ff[0]) = 0.;
  c_re (ff[J]) = 0.; c_im (ff[J]) = 0.;
  for (int j = 1; j < J; j++)
    {
      c_re (ff[j]) = f(j); c_im (ff[j]) = 0.;
      c_re (ff[2*J-j]) = - f(j); c_im (ff[2*J-j]) = 0.;
    }

  // Call fftw routine
  fftw_plan p = fftw_create_plan (N, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_one (p, ff, FF);
  fftw_destroy_plan (p);

  // Unload data
  F(0) = 0.; F(J) = 0.;
  for (int j = 1; j < J; j++)
    {
      F(j) = - 2. * c_im (FF[j]);
    }

  // Normalize data
  F /= 2. * double (J);
}
```

Sin-傅里叶变换

```
// Calculates inverse Fourier-sine transform of array F in array f
void fft_backward_sin (Array<double,1> F, Array<double,1>& f)
{
  // Find J. Declare local arrays.
  int J = f.extent(0) - 1;
  int N = 2 * J;
  fftw_complex ff[N], FF[N];

  // Load and extend data
  c_re (FF[0]) = 0.; c_im (FF[0]) = 0.;
  c_re (FF[J]) = 0.; c_im (FF[J]) = 0.;


 for (int j = 1; j < J; j++)
   {
     c_re (FF[j]) = 0.; c_im (FF[j]) = - F(j) / 2.;
     c_re (FF[2*J-j]) = 0.; c_im (FF[2*J-j]) = F(j) / 2.;
   }

 // Call fftw routine
 fftw_plan p = fftw_create_plan (N, FFTW_BACKWARD, FFTW_ESTIMATE);
 fftw_one (p, FF, ff);
 fftw_destroy_plan (p);

 // Unload data
 f(0) = 0.; f(J) = 0.;
 for (int j = 1; j < J; j++)
   {
     f(j) = c_re (ff[j]);
   }
}
```

Sin-傅里叶反变换

2D泊松方程

Let us now use the techniques discussed above to solve Poisson's equation in two dimensions. Suppose that the source term is

$$v(x,y) = 6\,x\,y\,(1-y) - 2\,x^3$$

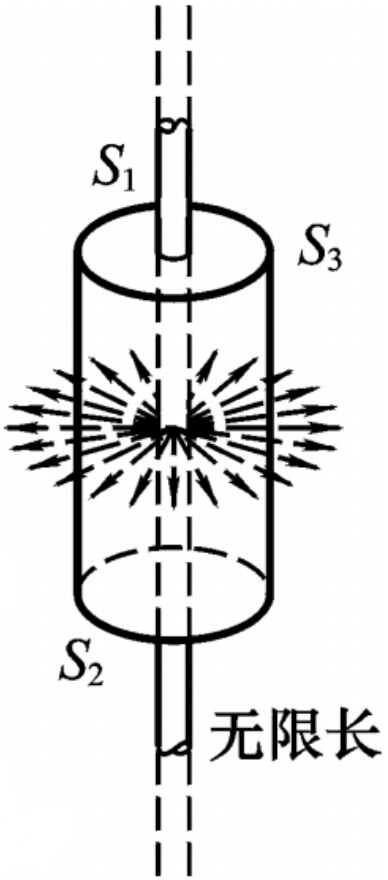
for $0 \leq x \leq 1$ and $0 \leq y \leq 1$. The boundary conditions at $x = 0$ are $\alpha_l = 1$, $\beta_l = 0$, and $\gamma_l = 0$, whereas the boundary conditions at $x = 1$ are $\alpha_h = 1$, $\beta_h = 0$, and $\gamma_h = y\,(1-y)$. The simple Dirichlet boundary conditions $u(x,0) = u(x,1) = 0$ are applied at $y = 0$ and $y = 1$. Of course, this problem can be solved analytically

$$u(x,y) = y\,(1-y)\,x^3.$$

```
// d^2 u / dx^2 + d^2 u / dy^2 = v for xl <= x <= xh and 0 <= y <= L
// alphaL u + betaL du/dx = gammaL(y) at x=xl
// alphaH u + betaH du/dx = gammaH(y) at x=xh
// In y-direction, either simple Dirichlet boundary conditions:
// u(x,0) = u(x,L) = 0
// Matrices u and v assumed to be of extent N+2, J+1.
// Arrays gammaL, gammaH assumed to be of extent J+1.
// Now, (i,j)th elements of matrices correspond to
// x_i = xl + i * dx i=0,N+1
// y_j = j * L / J j=0,J
// Here, dx = (xh - xl) / (N+1) is grid spacing in x-direction.
```

```
#include <fftw/fftw.h>
#include <blitz/array.h>
```

2D泊松方程 静电学示例



$$\frac{\partial^2\phi(x,y)}{\partial x^2}+\frac{\partial^2\phi(x,y)}{\partial y^2}=\nu(x,y)=-\delta(x-x_0)\,\delta(y-y_0),$$

where (x_0, y_0) are the coordinates of the wire. Here, we have conveniently normalized our units such that the factor ϵ_0 is absorbed into the normalization. Assuming that the box is grounded, the potential is subject to the Dirichlet boundary conditions $\phi = 0$ at $x = 0$, $x = 1$, $y = 0$, and $y = 1$. We require the solution in the region $0 \le x \le 1$ and $0 \le y \le 1$.

Note that when discretizing the right-hand side becomes

$$\nu(x_i, y_j) = -\frac{1}{\delta x\,\delta y}$$

on the grid-point closest to the wire, with $\nu(x_i, y_j) = 0$ on the remaining grid-points. Here, δx and δy are the grid spacings in the x- and y- directions, respectively.

2D泊松方程 静电学示例

```
void Tridiagonal (Array<double,1> a,
Array<double,1> b, Array<double,1> c,
Array<double,1> w, Array<double,1>& u)
{…}
```

```
int main()
{
  double alphaL, betaL;
  double alphaH, betaH;
  double dx; double kappa;
  double xl, xh;
  double L;
  int Neumann;
  int N=64;
  int J=64;
  xl=0.0; xh=1.0; L=1.0;
  dx=(xh-xl)/(N+1);
  kappa=Pi*dx/L;
  alphaL=1.0; betaL=0.0;
  alphaH=1.0; betaH=0.0;
  Neumann=0;
```

```
void fft_forward_cos (Array<double,1> f,
Array<double,1>& F)
{…}
void fft_backward_cos (Array<double,1> F,
Array<double,1>& f)
{…}
void fft_forward_sin (Array<double,1> f,
Array<double,1>& F)
{…}
void fft_backward_sin (Array<double,1> F,
Array<double,1>& f)
{…}
```

```
Array<double,2> u(N+2, J+1), v(N+2, J+1);
 Array<double,1> gammaL(J+1), gammaH(J+1);

 for (int j = 0; j <= J; j++)   {
   double y_j = j * L / J ;
   gammaL(j)=0.0;
   gammaH(j)=0.0;
 }
```

2D泊松方程 静电学示例

```
for (int i = 0; i <= N+1; i++) {
  for (int j = 0; j <= J; j++)  {
    double x_i = xl + i * dx;
    double y_j = j * L / J ;
    v(i, j)=0.0;
     if(fabs(x_i-0.4615)<0.001 && fabs(y_j-
0.25)<0.001)
     {v(i,j)=-1.0/dx/(L/J);}
     if(fabs(x_i-0.6923)<0.001 && fabs(y_j-
0.5)<0.001)
     {v(i,j)=-1.0/dx/(L/J);}
  }
 }                    //Souce
```

```
// Find N and J. Declare local arrays.
 Array<double,2> V(N+2, J+1), U(N+2, J+1);
 Array<double,1> GammaL(J+1), GammaH(J+1);
 // Fourier transform boundary conditions
 if (Neumann)
 {
  fft_forward_cos (gammaL, GammaL);
  fft_forward_cos (gammaH, GammaH);
 }
 else
 {
  fft_forward_sin (gammaL, GammaL);
  fft_forward_sin (gammaH, GammaH);
 }
 // Fourier transform source term
 for (int i = 1; i <= N; i++)
 {
  Array<double,1> In(J+1), Out(J+1);
  for (int j = 0; j <= J; j++) In(j) = v(i, j);
  if (Neumann)
  fft_forward_cos (In, Out);
  else
  fft_forward_sin (In, Out);
  for (int j = 0; j <= J; j++) V(i, j) = Out(j);
 }
```

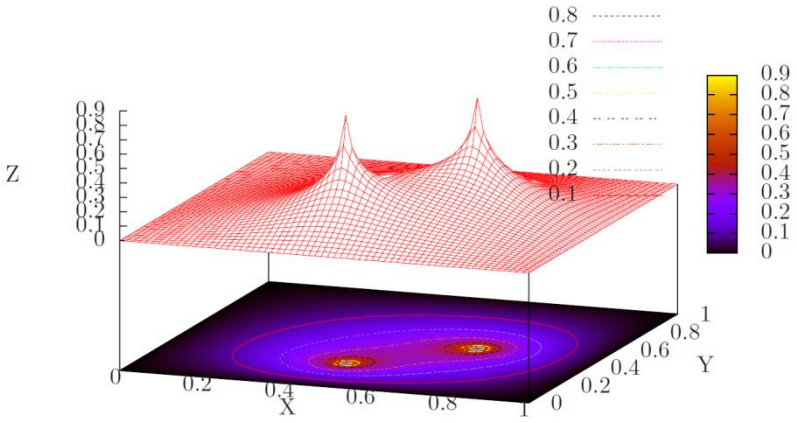
2D泊松方程 静电学示例

```
// Solve tridiagonal matrix equations
 if (Neumann)
 {
  for (int j = 0; j <= J; j++)
  {
  Array<double,1> a(N+2), b(N+2), c(N+2), w(N+2),
uu(N+2);
   // Initialize tridiagonal matrix
   for (int i = 2; i <= N; i++) a(i) = 1.;
   for (int i = 1; i <= N; i++)
   b(i) = -2. - double (j * j) * kappa * kappa;
   b(1) -= betaL / (alphaL * dx - betaL);
   b(N) += betaH / (alphaH * dx + betaH);
   for (int i = 1; i <= N-1; i++) c(i) = 1.;
   for (int i = 1; i <= N; i++)
     w(i) = V(i, j) * dx * dx;
   w(1) -= GammaL(j) * dx / (alphaL * dx - betaL);
   w(N) -= GammaH(j) * dx / (alphaH * dx + betaH);
   // Invert tridiagonal matrix equation
   Tridiagonal (a, b, c, w, uu);
   for (int i = 1; i <= N; i++) U(i, j) = uu(i);
  }
 }
```

```
 else
 {
  for (int j = 1; j < J; j++)
  {
   Array<double,1> a(N+2), b(N+2), c(N+2), w(N+2),
uu(N+2);
   // Initialize tridiagonal matrix
   for (int i = 2; i <= N; i++) a(i) = 1.;
   for (int i = 1; i <= N; i++)
   b(i) = -2. - double (j * j) * kappa * kappa;
   b(1) -= betaL / (alphaL * dx - betaL);
   b(N) += betaH / (alphaH * dx + betaH);
   for (int i = 1; i <= N-1; i++) c(i) = 1.;
   for (int i = 1; i <= N; i++)
    w(i) = V(i, j) * dx * dx;
   w(1) -= GammaL(j) * dx / (alphaL * dx - betaL);
   w(N) -= GammaH(j) * dx / (alphaH * dx + betaH);
   // Invert tridiagonal matrix equation
   Tridiagonal (a, b, c, w, uu);
   for (int i = 1; i <= N; i++) U(i, j) = uu(i);
  }
  for (int i = 1; i <= N ; i++)
  {
   U(i, 0) = 0.; U(i, J) = 0.;
  }
 }
```

2D泊松方程 静电学示例

```
// Reconstruct solution via inverse Fourier transform
for (int i = 1; i <= N; i++)
{
 Array<double,1> In(J+1), Out(J+1);
 for (int j = 0; j <= J; j++) In(j) = U(i, j);
 if (Neumann)
  fft_backward_cos (In, Out);
 else
  fft_backward_sin (In, Out);
 for (int j = 0; j <= J; j++) u(i, j) = Out(j);
}
// Calculate i=0 and i=N+1 values
for (int j = 0; j <= J; j++)
{
 u(0, j) = (gammaL(j) * dx - betaL * u(1, j)) /
  (alphaL * dx - betaL);
 u(N+1, j) = (gammaH(j) * dx + betaH * u(N, j)) /
  (alphaH * dx + betaH);
}
```




```
set hidden3d
set pm3d at b
#set pm3d
set isosample 30,30
set xyplane 1.0
splot "plot.gnu" with lines title ""
```

不同电荷情形测试

2D 扩散问题 Dirichlet边界条件

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} + D\frac{\partial^2 T}{\partial y^2}$$

$$x_l \le x \le x_h \qquad 0 \le y \le L$$

差分： $t_n = t_0 + n\delta t, \quad x_i = x_0 + i\delta x, \quad y_j = y_0 + j\delta y$

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\delta t} = \frac{D}{2}\left[\frac{T_{i-1,j}^{n+1} - 2T_{i,j}^{n+1} + T_{i+1,j}^{n+1}}{(\delta x)^2} + \frac{T_{i-1,j}^{n} - 2T_{i,j}^{n} + T_{i+1,j}^{n}}{(\delta x)^2}\right]$$

Crank-Nicholson 方法

$$+\frac{D}{2}\left[\left(\frac{\partial^2 T}{\partial y^2}\right)_{i,j}^{n+1} + \left(\frac{\partial^2 T}{\partial y^2}\right)_{i,j}^{n}\right]$$

y方向Dirichlet边界条件 $\mathrm{T}(x, 0, \mathrm{t}) = \mathrm{T}(x, L, t) = 0$ $T_{i,0}^{n} = T_{i,J}^{n} = 0$

X方向混合边界条件

$$\alpha_l(t)T(x_l, y, t) + \beta_l(t)\frac{\partial T(x_l, y, t)}{\partial x} = \gamma_l(y, t), \quad \Rightarrow \quad T_{0,\mathrm{j}}^{n} = \frac{\gamma_{lj}^{n}\delta x - \beta_l^{n} T_{1,\mathrm{j}}^{n}}{\alpha_l^{n}\delta x - \beta_l^{n}},$$

$$\alpha_h(t)T(x_h, \mathrm{y}, t) + \beta_h(t)\frac{\partial T(x_h, \mathrm{y}, t)}{\partial x} = \gamma_h(y, t), \quad T_{N+1,\mathrm{j}}^{n} = \frac{\gamma_{hj}^{n}\delta x + \beta_h^{n} T_{N,j}^{n}}{\alpha_h^{n}\delta x + \beta_h^{n}}$$

$$\gamma_{lj}^{n} = \gamma_l(y_j, t_n), \ldots$$

2D 扩散问题 Dirichlet边界条件

$$\mathrm{T}(x,0,\mathrm{t}) = \mathrm{T}(x,L,t) = 0$$

傅里叶展开及逆变换

$$T_{i,j}^{n} = \sum_{k=0}^{J} \hat{T}_{i,k}^{n} \sin(jk\,\pi/J), \qquad \hat{T}_{i,j}^{n} = \frac{2}{J} \sum_{k=0}^{J} T_{i,k}^{n} \sin(jk\,\pi/J)$$

$$-\frac{C}{2}\hat{\mathrm{T}}_{i-1,j}^{n+1} + \{1 + C\,(1 + j^2\,\kappa^2/2)\}\,\hat{\mathrm{T}}_{i,j}^{n+1} - \frac{C}{2}\hat{\mathrm{T}}_{i+1,j}^{n+1} = \frac{C}{2}\hat{\mathrm{T}}_{i-1,j}^{n} + \{1 - C\,(1 + j^2\,\kappa^2/2)\}\,\hat{\mathrm{T}}_{i,j}^{n} + \frac{C}{2}\hat{\mathrm{T}}_{i+1,j}^{n},$$

$$i = 1, N,$$
$$j = 0, J,$$
$$C = D\frac{\delta t}{(\delta x)^2}$$
$$\mathrm{k} = \frac{\pi \delta x}{L}$$

$$\hat{\mathrm{T}}_{0,j}^{n} = \frac{\Gamma_{lj}^{n}\,\delta x - \beta_{l}^{n}\,\hat{\mathrm{T}}_{1,j}^{n}}{\alpha_{l}^{n}\,\delta x - \beta_{l}^{n}},$$

$$\hat{\mathrm{T}}_{N+1,j}^{n} = \frac{\Gamma_{hj}^{n}\,\delta x + \beta_{h}^{n}\,\hat{\mathrm{T}}_{N,j}^{n}}{\alpha_{h}^{n}\,\delta x + \beta_{h}^{n}},$$

$$\hat{\Gamma}_{lj}^{n} = \frac{2}{J}\sum_{k=0}^{J} \gamma_{l,k}^{n}\,\sin(j\,k\,\pi/J),$$

a set of J + 1 uncoupled tridiagonal matrix equations for the $\hat{\mathrm{T}}_{i,j}^{n+1}$, with one equation for each separate value of j.

2D 扩散问题 Neumann边界条件

$$\frac{\partial T(x,0,t)}{\partial y} = \frac{\partial T(x,L,t)}{\partial y} = 0$$

$$T_{i,j}^{n} = \sum_{k=0}^{J} \hat{T}_{i,k}^{n} \cos\left(jk\,\pi/J\right),$$

$$\hat{T}_{i,j}^{n} = \frac{T_{i,0}^{n}}{J} + \frac{2}{J}\sum_{k=1}^{J-1} T_{i,k}^{n} \cos\left(jk\,\pi/J\right) + \frac{(-1)^{j} T_{i,J}^{n}}{J}$$

2D 扩散问题 例子

```
// Function to evolve diffusion equation in 2-d:
// dT / dt = D d^2 T / dx^2 +D d^2 T / dy^2
// alphaL T + betaL dT/dx = gammaL(y) at x=xl
// alphaH T + betaH dT/dx = gammaH(y) at x=xh
//    for xl <= x <= xh  and 0 <= y <= L
// In y-direction, either simple Dirichlet boundary conditions:
//   T(x,0) = T(x,L) = 0
// or simple Neumann boundary conditions:
// dT/dy(x,0) = dT/dy(x,L) = 0
// Matrix T assumed to be of extent N+2, J+1.
// Arrays gammaL, gammaH assumed to be of extent J+1.
// Now, (i,j)th elements of matrices correspond to
// x_i = xl + i * dx i=0,N+1
// y_j = j * L / J j=0,J
// Here, dx = (xh - xl) / (N+1) is grid spacing in x-direction.
// Now, C =D dt / dx^2, and kappa = pi * dx / L
// Finally, Neumann=0/1 selects Dirichlet/Neumann bcs in y-direction.
// Uses Crank-Nicholson scheme.
```

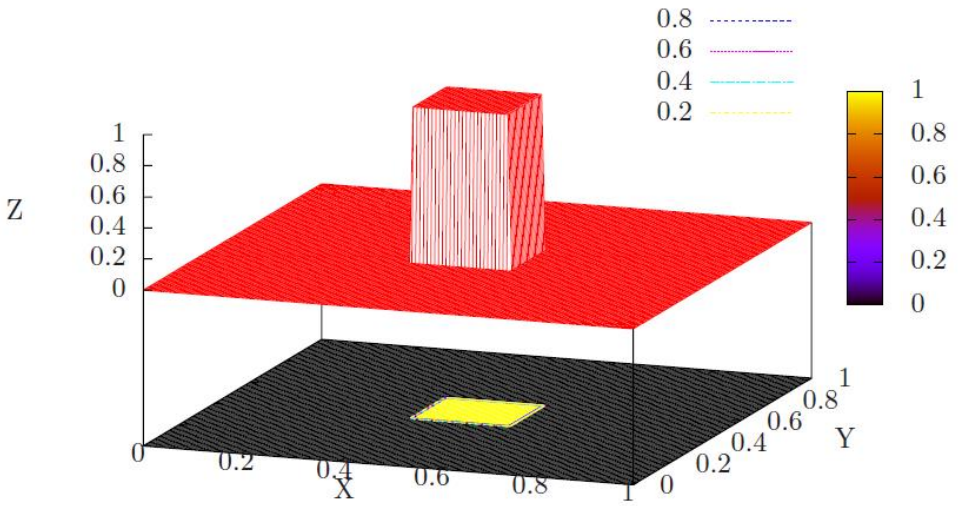
2D 扩散问题 例子

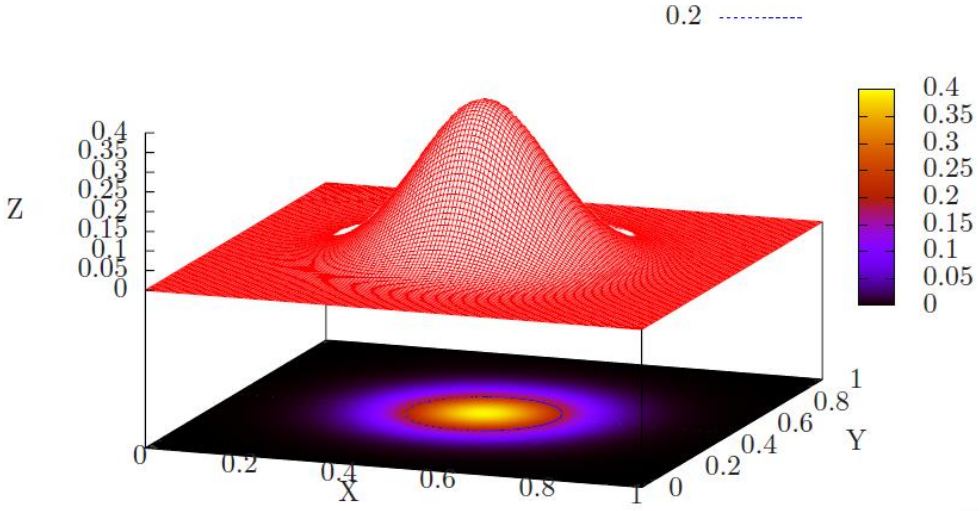
```
double alphaL, betaL;
 double alphaH, betaH;
 double dx; double kappa;
 double xl, xh;
 double L;
 int Neumann;
 int N=128;
 int J=128;
 xl=0.0; xh=1.0; L=1.0;
 dx=(xh-xl)/(N+1);
 double dt;
 double C;
 dt=0.0001;
 C=1.0*dt/dx/dx;
 kappa=Pi*dx/L;
 alphaL=1.0; betaL=0.0;
 alphaH=1.0; betaH=0.0;
 Neumann=0;
 Array<double,2> T(N+2, J+1);
 Array<double,1> gammaL(J+1), gammaH(J+1);
```

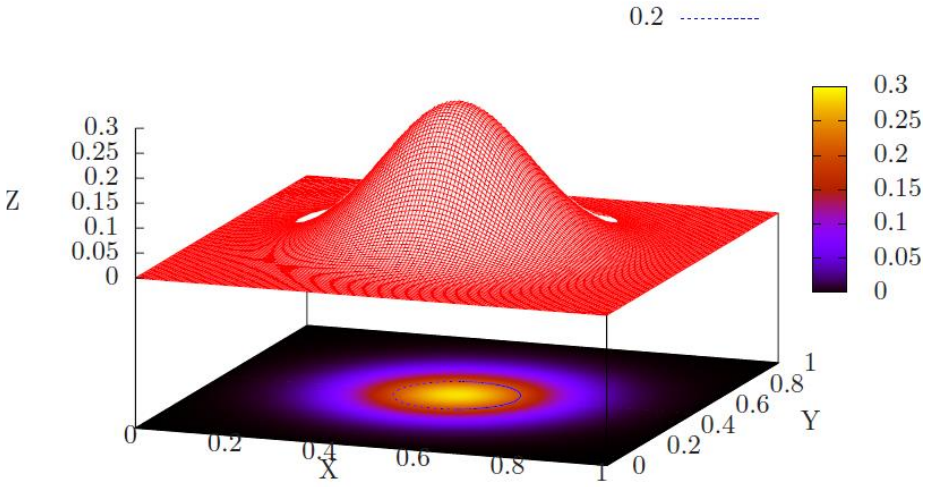
```
for (int j = 0; j <= J; j++)   {
   double y_j = j * L / J ;
   gammaL(j)=0.0;
   gammaH(j)=0.0;
 }
```

```
for (int i = 0; i <= N+1; i++) {
  for (int j = 0; j <= J; j++)   {
   double x_i = xl + i * dx;
   double y_j = j * L / J ;
   T(i, j)=0.0;
   if(fabs(x_i-0.5)<0.1 && fabs(y_j-0.5)<0.1)
    {T(i,j)=1.0;}
    }
   }
```

2D 扩散问题 例子





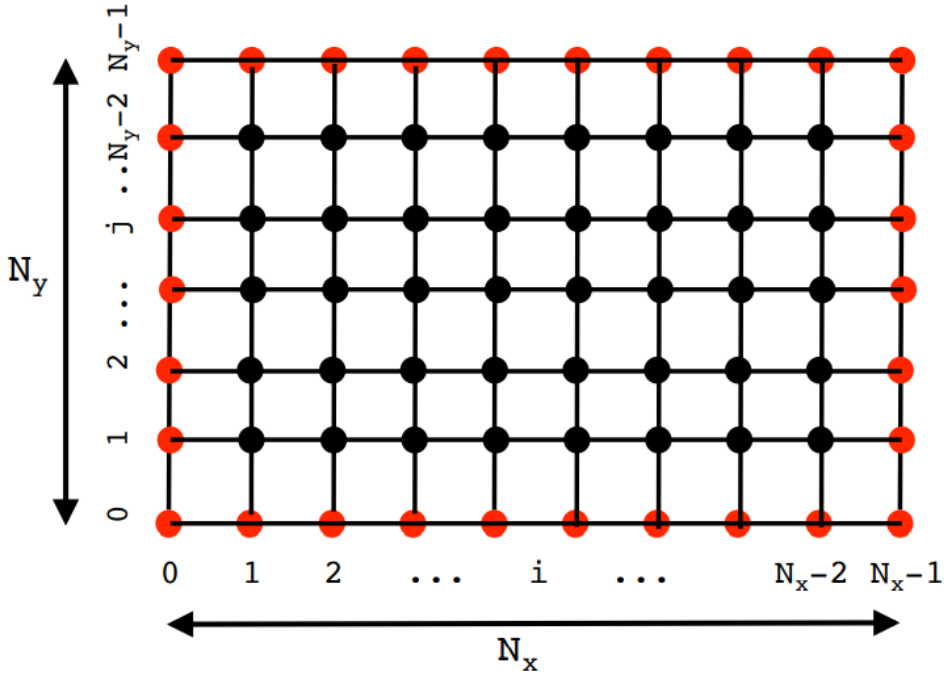


2D泊松方程 （椭圆型方程）

$$\frac{\partial^2 \varphi(x,y)}{\partial x^2} + \frac{\partial^2 \varphi(x,y)}{\partial y^2} = S(x,y)$$

$$x_l \le x \le x_h \qquad 0 \le y \le L$$

$$u_{xx}\Delta x^2 = u(x_{i+1}) - 2u(x_i) + u(x_{i-1})$$
$$u_{yy}\Delta y^2 = u(y_{i+1}) - 2u(y_i) + u(y_{i-1})$$



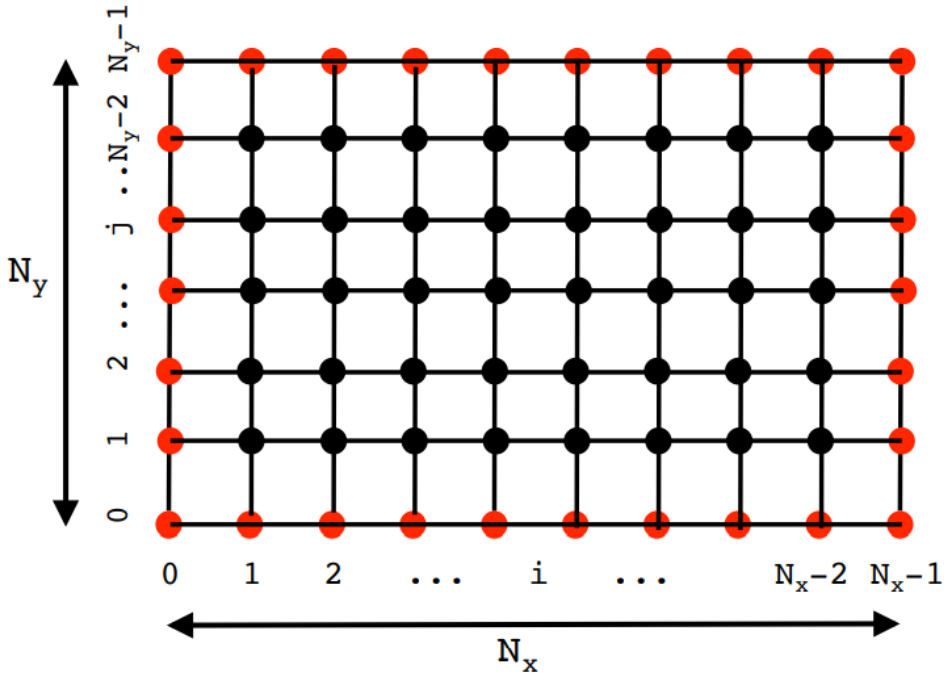
Uniform and equal spacing in both direction is assumed: h=Δx=Δy.

Red points should be specified as boundary conditions while black points are the solution values (unknowns).

2D泊松方程 （椭圆型方程）

$$\frac{\partial^2 \varphi(x,y)}{\partial x^2} + \frac{\partial^2 \varphi(x,y)}{\partial y^2} = S(x,y)$$

$$\frac{\varphi_{i+1,j} - 2\varphi_{i,j} + \varphi_{i-1,j}}{\Delta x^2} + \frac{\varphi_{i,j+1} - 2\varphi_{i,j} + \varphi_{i,j-1}}{\Delta y^2} = S_{i,j}$$



Interior points:

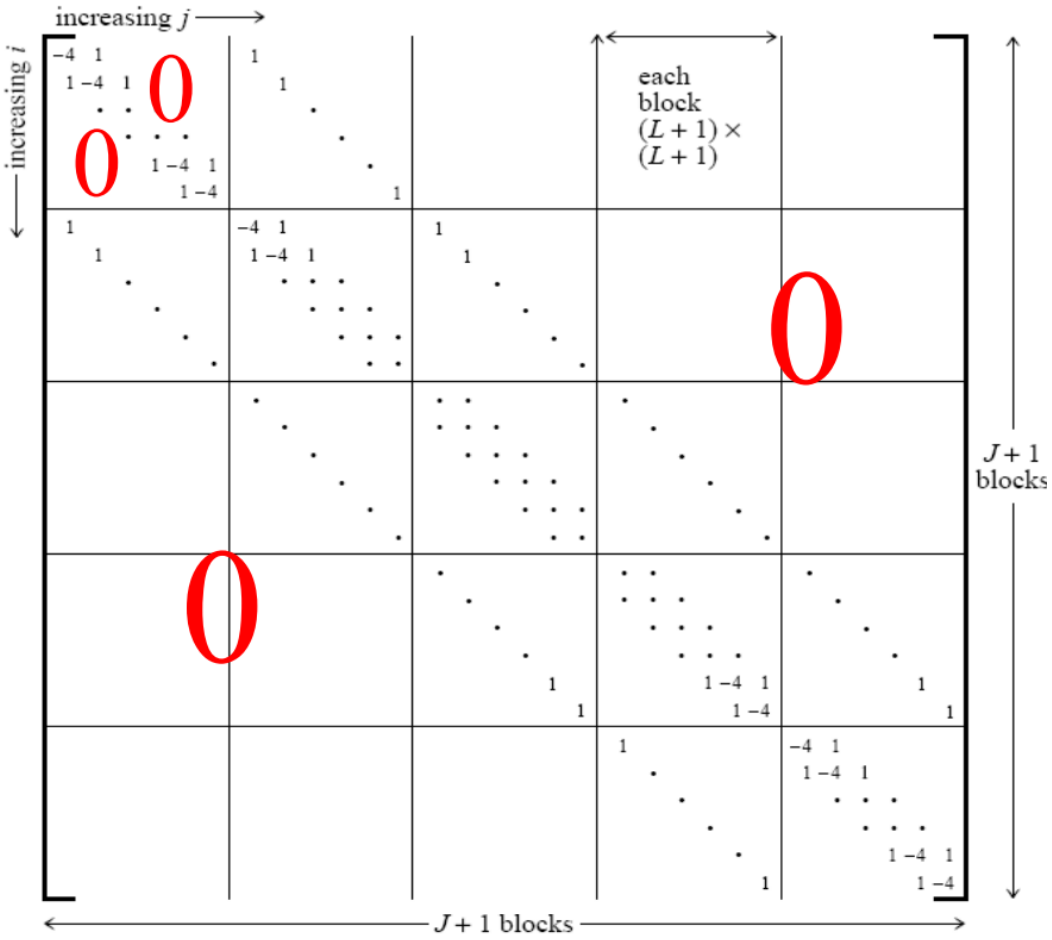
- $i=1\ldots N_x-2$, $j=1\ldots N_y-2$. This is where the solution must be found.

Boundary points:

- *Bottom:* $i=0\ldots N_x-1$ $j=0$
- *Top:* $i=0\ldots N_x-1$ $j=N_y-1$
- *Left:* $i=0$ $j=0\ldots N_y-1$
- *Right:* $i=N_x-1$ $j=0\ldots N_y-1$

$$A\varphi = b$$

Here A is a large sparse matrix of $(N_x-2)^2 x (N_y-2)^2$ points.



For 3D-grids typically used in science application of about 300 x 300 x 300

φ has 300^3=2.7*10^7 grid points,

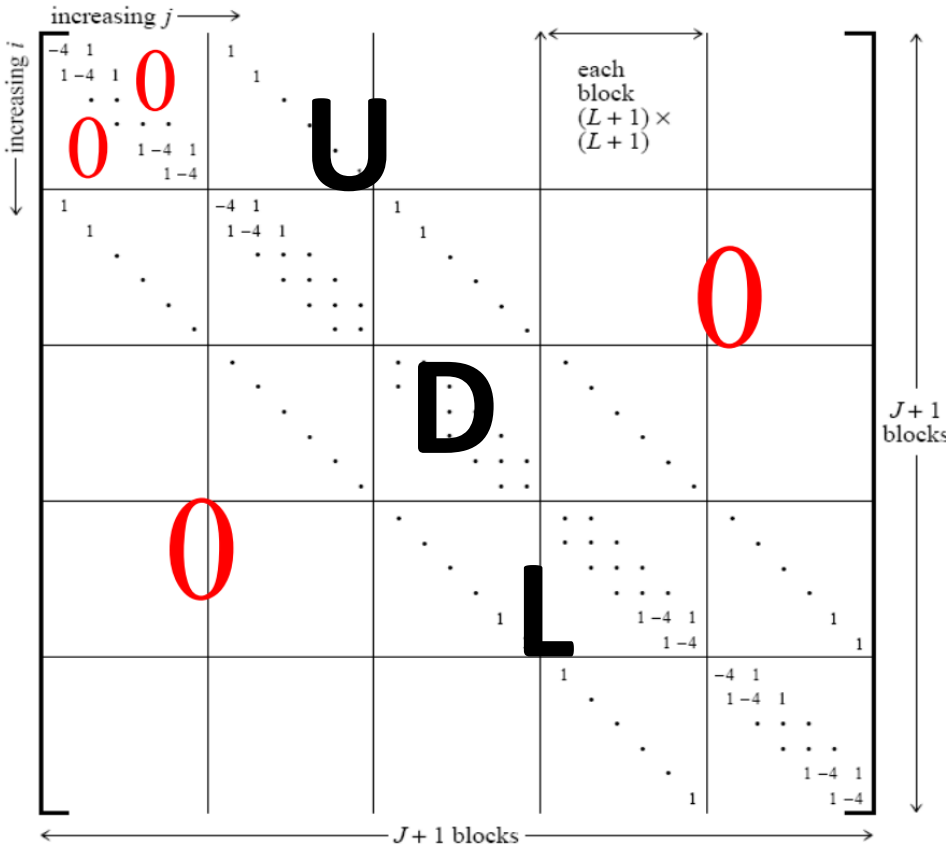
A has (2.7*10^7)^2 =7.29*10^14 entries!

⇒ Memory requirement for 300-cube to store

⇒ φ ~100 MB, A~3Million GByte

$$A\varphi = b$$

Here A is a large sparse matrix of $(N_x-2)^2 x (N_y-2)^2$ points.



$$\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$$

Lower Triangle | Diagonal Elements | Upper Triangle

$$A\boldsymbol{\varphi} = (D+L+U)\boldsymbol{\varphi} = \boldsymbol{b} \quad \rightarrow \quad \boldsymbol{\varphi}^{(k+1)} = D^{-1}\left[\boldsymbol{b} - (L+U)\boldsymbol{\varphi}^{(k)}\right]$$

Relaxation: Jacobi method

Carl Jacobi
1804-1851

From $\dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = \rho(x,y)$

we derived the algebraic equations:

$$u_{i+L+1} + u_{i-(L+1)} + u_{i+1} + u_{i-1} - 4u_i = \Delta^2 \rho_i$$

Assume any initial value, say **u**=0 on all grid points (except the specified boundary values of course) and compute:

$$u_{j,l}^{n+1} = \frac{1}{4}\left(u_{j+1,l}^{n} + u_{j-1,l}^{n} + u_{j,l+1}^{n} + u_{j,l-1}^{n}\right) - \frac{\Delta^2}{4}\rho_{j,l}$$

Use the new values of **u** as input for the right side and repeat the iteration until **u** converges. (n: iteration step)

$$A\varphi = (D+L+U)\varphi = \boldsymbol{b} \quad \rightarrow \quad \varphi^{(k+1)} = (D+L)^{-1}\left[\boldsymbol{b} - U\varphi^{(k)}\right]$$

Gauss Seidel method

- Similar as Jacobi method.
- Difference: Use on the right-hand site already the new (and assumed to be better) approximation u^{n+1}, as soon as known.

C.F. Gauss
1777-1855

$$u_{j,l}^{n+1} = \frac{1}{4}\left(u_{j+1,l}^{n} + u_{j-1,l}^{n+1} + u_{j,l+1}^{n} + u_{j,l-1}^{n+1}\right) - \frac{\Delta^2}{4}\rho_{j,l}$$

Successive Over Relaxation (SOR)

Both Jacobi and Gauss-Seidel do not use the value of $\phi_{i,j}$ at the same lattice point during the update step.
The convergence of the iteration can be improved considerably by using a linear combination of the new and old solutions as follows:

$$\varphi_{i,j}^{(k+1)} = (1-\omega)\varphi_{i,j}^{(k)} + \frac{\omega}{4}\left(\varphi_{i-1,j}^{(k+1)} + \varphi_{i+1,j}^{(k)} + \varphi_{i,j-1}^{(k+1)} + \varphi_{i,j+1}^{(k)} - h^2 S_{i,j}\right)$$

In matrix notation, this is the same as

$$A\boldsymbol{\varphi} = (D+L+U)\boldsymbol{\varphi} = \boldsymbol{b} \quad \rightarrow \quad \boldsymbol{\varphi}^{(k+1)} = (D+\omega L)^{-1}\left[\omega\boldsymbol{b} - (\omega U - (\omega-1)D)\boldsymbol{\varphi}^{(k)}\right]$$

The preconditioner matrix is still in triangular form.

Successive Over Relaxation (SOR)

The over-relaxation parameter ω can be tuned to optimize the convergence. It can be shown that

- SOR converges only for $0<\omega<2$;
- It is faster than Gauss-Seidel only if $1<\omega<2$;
- It converges fastest for a square lattice if $\omega \approx 2/(1+\pi/N)$, where N is the number of points in the x or y directions.

It can be shown that the eigenvalues of the SOR matrix are

$$\mu^{1/2} = \frac{1}{2}\left[\lambda\omega + \sqrt{\lambda^2\omega^2 - 4(\omega - 1)}\right]$$

where λ is an eigenvalue of the Jacobi matrix.

The minimum occurs at $\omega_{\text{opt}} = \dfrac{2}{1+\sqrt{1-\lambda_{\max}^2}}$

Computing time for relaxation methods

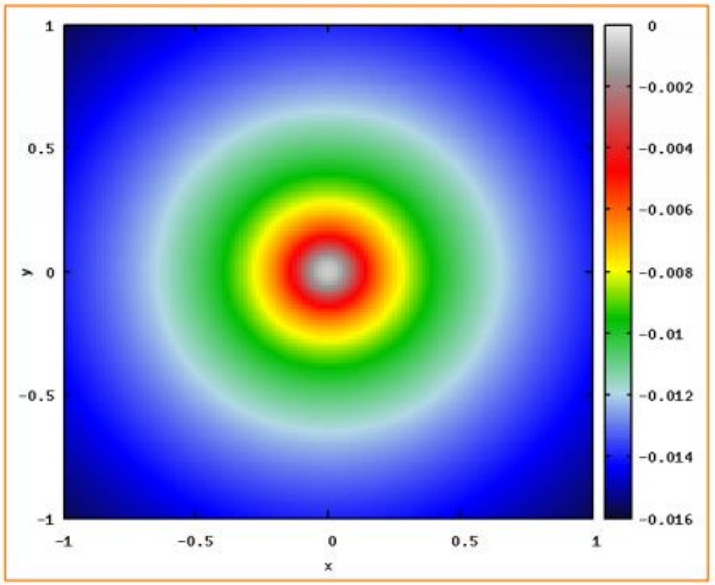
- For a **J** x **J** 2D-PDE the number of iteration steps is ~$\mathbf{J^2}$ (Jacobi GS) or ~**J** (SOR)
- But: Each iteration step takes ~$\mathbf{J^2}$
- Total computing time: ~$\mathbf{J^4}$ (Jacobi, Gauss Seidel)
  ~$\mathbf{J^3}$ (SOR-method)

- Computing time depends also on other factors:
  -required accuracy
  -computational implementation
  -IDL is much slower as C or Fortran
  -Hardware and parallelization

Compute the potential of an infinitely long charged cylinder by solving the Poisson equation

$$\nabla^2\varphi = -\rho\,, \quad \text{with} \quad \rho = \begin{cases} \rho_0 & \text{for} \quad r \le a \\ 0 & \text{otherwise} \end{cases}$$

use a=0.1 and ρ_o=1.



As a boundary condition use the exact solution:

$$\varphi(r) = \begin{cases} -\dfrac{\rho_0 r^2}{4} & \text{for} \quad 0 \le r \le a \\ -\dfrac{\rho_0 a^2}{2}\left[\log\left(\dfrac{r}{a}\right) + \dfrac{1}{2}\right] & \text{otherwise} \end{cases}$$

Solve the equation on the square domain $-1 \le x, y \le 1$, using 128^2 nodes.

$$\epsilon_r = \sum_{ij} \left| \delta_x^2 \varphi_{ij} + \delta_y^2 \varphi_{ij} - h^2 S_{ij} \right|$$

tol = 10^{-7},

Jacobi	Gauss-Siedel	SOR
≈ 46224	≈ 23113	≈ 447

For more details:

http://personalpages.to.infn.it/~mignone/Numerical_Algorithms/ch10_elliptic_pde.pdf