

An Introduction to Fast Poisson Solvers

S. Nicholas Barkas

June 8, 2005

1 Introduction

There are many physical phenomena which are described by a class of elliptic PDEs known as Poisson equations. In two dimensions, these equations have the form

$$\nabla^2 u(x, y) = f(x, y).$$

The ability to solve these equations quickly can help us to predict how these physical systems will behave. For example, the following equation describes the electrostatic potential field induced by charges in space:

$$\nabla^2 V = -4\pi\rho, \tag{1}$$

where V is a potential (voltage) field and ρ is a charge density function (this particular equation uses Gaussian, or cgs, units) [2, p. 574]. It should be noted that the equation governing gravitational potential energy of masses in space, and thus describing the many-body problem in celestial mechanics, is nearly identical to (1). In this paper we shall look at fast Poisson methods that can solve this problem in less time than, for example, iterative methods such as Conjugate Gradient. First we'll look at a method based on fast Fourier transforms. Then we'll see how it does at solving (1) in two dimensions for a collection of point charges in a box with grounded walls.

2 Fourier Transforms

Fourier transforms are useful in a wide range of applications, from multiplying large integers to signal processing to what we discuss in this paper—solving partial differential equations. The continuous Fourier transform, which is often referred to just as the Fourier transform, is defined by the following pair of integrals:

$$\begin{aligned} f(x) &= \int_{-\infty}^{\infty} g(\alpha) e^{i\alpha x} d\alpha \\ g(\alpha) &= \frac{1}{2\pi} \int_{-\infty}^{\infty} f(x) e^{-i\alpha x} dx \end{aligned}$$

$g(\alpha)$ is referred to as the Fourier transform of $f(x)$, and $f(x)$ is usually referred to as the inverse Fourier transform of $g(\alpha)$ [2, p. 648]. A requirement for the Fourier transform to exist is that $\int_{-\infty}^{\infty} |f(x)| dx$ must be finite.

By applying Fourier transforms to partial differential equations, we can convert them to ordinary differential equations which are easier to solve. See [4] for further information on analytical solutions to PDEs via Fourier transforms.

3 Discrete and Fast Fourier Transforms

Continuous Fourier transforms are useful for analytical calculations, but when we need to perform a Fourier transform on a computer we can usually more easily use a discrete Fourier transform (DFT). A vector \mathbf{x} with n elements x_0, x_1, \dots, x_{n-1} has a DFT \mathbf{y} , also with n elements, defined by

$$y_k = \sum_{j=0}^{n-1} \omega_n^{kj} x_j,$$

where $\omega_n = \cos(2\pi/n) - i \sin(2\pi/n) = e^{-2\pi i/n}$. This transform can be written as a matrix-vector product

$$\mathbf{y} = F_n \mathbf{x},$$

where F_n is an $n \times n$ matrix with elements $f_{pq} = \omega_n^{pq}$ where $p, q = 0, \dots, n-1$ [8, p. 2-3]. For example,

$$F_4 = \begin{bmatrix} \omega_4^0 & \omega_4^0 & \omega_4^0 & \omega_4^0 \\ \omega_4^0 & \omega_4^1 & \omega_4^2 & \omega_4^3 \\ \omega_4^0 & \omega_4^2 & \omega_4^4 & \omega_4^6 \\ \omega_4^0 & \omega_4^3 & \omega_4^6 & \omega_4^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}.$$

Typically, multiplying an $n \times n$ matrix with an $n \times 1$ vector requires $O(n^2)$ operations. The idea behind the fast Fourier transform (FFT) is to reduce this operation count to $O(n \log n)$ operations by taking advantage of the structure of the matrix F_n . How the FFT is computed is beyond the scope of this paper, but much information is available in [8] and many other references.

4 Discrete Sine Transform

Related to the DFT is the discrete sine transform, or DST. A vector $\mathbf{x} = [x_1, x_2, \dots, x_{m-1}]^T$ has DST \mathbf{y} (also with $m-1$ elements) given by

$$y_k = \sum_{j=1}^{m-1} \sin\left(\frac{kj\pi}{m}\right) x_j.$$

The DST can also be written as a matrix-vector product. The $n \times n$ matrix S_n has j, k -th element ($0 \leq j, k \leq n-1$)

$$[S_n]_{kj} = \sin\left(\frac{2\pi kj}{n}\right),$$

and we can write the DST of \mathbf{x} as

$$\mathbf{y} = S_{2m}(1:m-1, 1:m-1)\mathbf{x}, \tag{2}$$

where $S_{2m}(1:m-1, 1:m-1)$ denotes the $m-1 \times m-1$ block in the upper left corner of S_{2m} , not including the edges where j or $k = 0$ [8, p. 229-230]. Note that $F_n = C_n - iS_n$ where

$$[C_n]_{kj} = \cos\left(\frac{2\pi kj}{n}\right).$$

In other words, $-S_n$ is the imaginary part of F_n .

5 FFT Poisson Solver

While we can use the continuous Fourier transform to aid the analytical solution of PDEs as mentioned above, the FFT can be used numerically to solve Poisson problems. Using the FFT in this way is an example of what is called a fast Poisson solver.

A 2-D Poisson equation $\nabla^2 u(x, y) = f(x, y)$ on domain $\Omega = \{(x, y) \mid a \leq x \leq b, c \leq y \leq d\}$ can be discretized to get

$$\frac{u_{j-1,k} - 2u_{j,k} + u_{j+1,k}}{\Delta x^2} + \frac{u_{j,k-1} - 2u_{j,k} + u_{j,k+1}}{\Delta y^2} = f(x_j, y_k) \quad (3)$$

for the interior points. $1 \leq j \leq m$ and $1 \leq k \leq n$, where $m+1$ and $n+1$ are the number of cells in the x and y directions, Δx and Δy are the mesh widths in the x and y directions, $f(x_j, y_k) = f(a + j\Delta x, c + k\Delta y)$, and $u_{j,k} \approx u(a + j\Delta x, c + k\Delta y)$. The boundary values are at $j = 0, k = 0, j = m+1$, and $k = n+1$.

If our region Ω is a square with equally spaced mesh widths $\Delta x = \Delta y = h$, the number of cells in both directions is the same: $m = n$. For further simplification, we assume we have homogeneous Dirichlet boundary conditions: $u(x, y) = 0$ on $\partial\Omega$. In this case, (3) can be written

$$UT + TU = h^2 B \quad (4)$$

where

$$T = \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \dots & \dots & \dots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}, \quad U = \begin{bmatrix} u_{1,1} & \dots & u_{1,n} \\ \vdots & & \vdots \\ u_{n,1} & \dots & u_{n,n} \end{bmatrix}, \quad \text{and}$$

$$B = \begin{bmatrix} f(x_1, y_1) & f(x_2, y_1) & \dots & f(x_n, y_1) \\ f(x_1, y_2) & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ f(x_1, y_n) & \dots & \dots & f(x_n, y_n) \end{bmatrix}.$$

By Theorem 4.5.2 from [8, p. 250],

$$V^{-1}TV = D, \quad (5)$$

where $V = S_{2(n+1)}(1 : n, 1 : n)$ and D is a diagonal matrix with diagonal entries $\lambda_j = -4 \sin^2\left(\frac{j\pi}{2(n+1)}\right)$, $1 \leq j \leq n$. (5) can be rewritten

$$T = VDV^{-1}$$

by multiplying by V on the left and V^{-1} on the right. This can then be substituted into (4) for T to get

$$VDV^{-1}U + UV DV^{-1} = h^2 B.$$

By doing some further multiplications, we get

$$DV^{-1}UV + V^{-1}UVD = V^{-1}h^2 BV,$$

and we can then write

$$D\bar{U} + \bar{U}D = h^2 \bar{B}$$

where $\bar{U} = V^{-1}UV$ and $\bar{B} = V^{-1}BV$. Since D is a diagonal matrix with entries λ_j , we can solve directly for the j, k -th element of \bar{U} :

$$\bar{U}_{j,k} = \frac{h^2 \bar{B}_{j,k}}{\lambda_j + \lambda_k}.$$

Once we have \bar{U} , it is then easy to compute $U = V\bar{U}V^{-1}$.

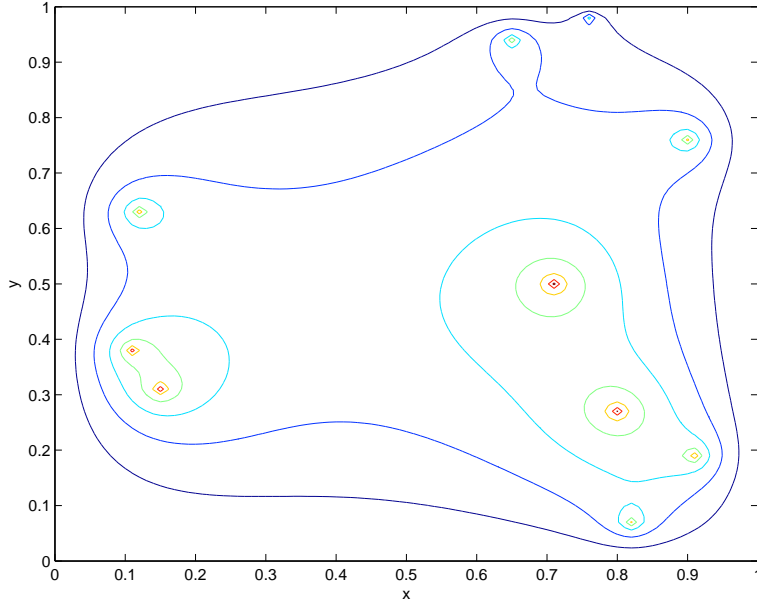


Figure 1: Contour plot of potential field

We have to do several matrix multiplications above to find \bar{B} , and to find U after \bar{U} has been computed. However, these multiplications can be performed very quickly because they in fact only require doing discrete sine transforms. For one thing, it turns out that $V^{-1} = \frac{2}{n+1}V$ [8, p. 240], so we do not need to invert V . Also, multiplying a matrix on the right by V is the same as performing a DST on its columns, while multiplying on the left by V is the same as performing a DST on its rows, which can be seen by comparing V to the matrix in (2). If we have n grid cells in each direction on our domain (n^2 total) we have to perform 4 DST operations on n vectors of length n , or $4n$ DSTs altogether. The fast DST is $O(n \log n)$, so the total cost of our algorithm is then $O(n^2 \log n)$. This is also often written $O(N \log N)$, where N is the total number of grid cells (n^2 in this case).

6 Example Output of Fast Poisson Solver

The plots in Figures 1 and 2 are of the electrostatic potential fields induced by 10 randomly placed point charges. These were generated by Matlab code which solves the Poisson problem (1) using fast Fourier transforms. (code included below).

7 Conclusion

By being able to solve Poisson problems such as (1) quickly on a computer, we can, for example, predict the motion of charges in space over time. By discretizing in time, we can solve our Poisson problem to find a potential field, determine where the charges will move based on the electric force field generated by the potentials, update the charge locations (which changes the charge density function, ρ), then do it all again at the next time step. If solving the Poisson problem is expensive, taking many time steps will require much cpu time.

The fast discrete sine transform method outlined above is not the only type of fast Poisson solver. Recently, a new algorithm known as the fast multipole method was developed, which can also solve Poisson problems in $O(N \log N)$ time. FMM solvers are particularly well suited for solving problems on irregularly shaped

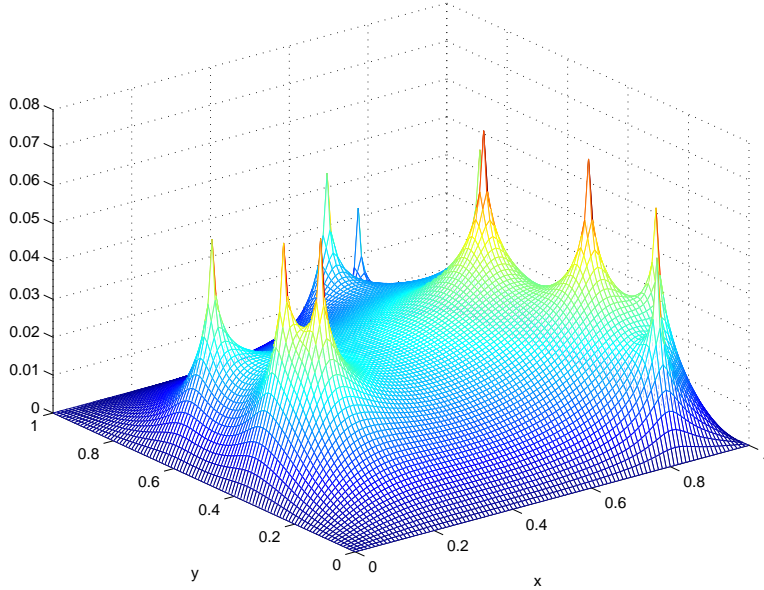


Figure 2: 3-d meshed surface plot of potential field

domains. Some information on the FMM is available in [1, 5, 6]. Additionally, a Matlab fast multipole method toolbox is available for free on the web from <http://www.madmaxoptics.com>, and it is documented in [9].

References

- [1] R. BEATSON AND L. GREENGARD, *A short course on fast multipole methods*, in Wavelets, Multilevel Methods and Elliptic PDEs, Oxford University Press, 1997, ch. 1, pp. 1–37. ISBN 0-19-850190-0.
- [2] M. L. BOAS, *Mathematical Methods in the Physical Sciences*, John Wiley & Sons, second ed., 1983. ISBN 0-471-04409-1.
- [3] J. DEMMEL, *CS 267: Notes for Lectures 15 and 16, Mar 5 and 7, 1996*. <http://www.cs.berkeley.edu/~demmel/cs267/lecture24/lecture24.html>, 1996. [Online; accessed 04-June-2005].
- [4] D. G. DUFFY, *Transform Methods for Solving Partial Differential Equations*, CRC Press, 1994. ISBN 0-8493-7374-3.
- [5] L. GREENGARD AND V. ROKHLIN, *A fast algorithm for particle simulations*, Journal of Computational Physics, 73 (1987), pp. 325–348.
- [6] J. HUANG AND L. GREENGARD, *A fast direct solver for elliptic partial differential equations on adaptively refined meshes*, SIAM Journal of Scientific Computing, 21 (2000), pp. 1551–1566.
- [7] D. KINCAID AND W. CHENEY, *Numerical Analysis: Mathematics of Scientific Computing*, Brooks / Cole, third ed., 2002. ISBN 0-534-38905-8.
- [8] C. V. LOAN, *Computational Frameworks for the Fast Fourier Transform*, SIAM, 1992. ISBN 0-89871-285-8.
- [9] MAXMAX OPTICS, *FMM Toolbox (2D) for MATLAB User's Manual*. <http://www.madmaxoptics.com/technology/products/FMMToolbox.asp>, 2002. [Online; accessed 10-May-2005].

- [10] M. PICKERING, *An Intorduction to Fast Fourier Transform Methods for Partial Differential Equations, with Applications*, Research Studies Press Ltd., 1986.
- [11] WIKIPEDIA, *Fourier transform – Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Fourier_transform, 2005. [Online; accessed 05-June-2005].

A Matlab Code

```
% solve del^2 u = -4 * pi * rho using a fast Poisson solver

% domain: for simplicity, a square {0 <= x,y <= 1}. the edges are grounded, so
% on the boundary u(x,y) = 0. we'll have 100 cells in the mesh of our domain.
a = 0; b = 1;
ncells = 100;
xe = linspace(a,b,ncells+1); ye = linspace(a,b,ncells+1);
[x, y] = meshgrid(xe,ye);

% grid spacing
h = (b-a)/ncells;

% set up right hand side. our charge density will just be 10 point charges on
% random grid points with strength 1. note that the right hand side does not
% include the borders; the charge is always zero on the borders since they are
% have no potential (they are grounded).
rho = zeros(ncells-1);
pts = round(rand(10,2) * (ncells-1));
for i=1:10
    rho(pts(i,1),pts(i,2)) = -4 * pi;
end

% now solve to get a potential field
V = fft_poisson(rho,h);
% pad V with zero boundary values
V = [zeros(ncells-1,1), V, zeros(ncells-1,1)];
V = [zeros(1,ncells+1); V; zeros(1,ncells+1)];

% contour plot of potential field
figure(1)
contour(x,y,V)
xlabel('x'); ylabel('y');

% mesh plot of potential field
figure(2)
mesh(x,y,V)
xlabel('x'); ylabel('y');

function u = fft_poisson(b,h)

% This function solves the 2-d Poisson problem del^2 u = f(x,y) using the fast
% Fourier transformation. The Poisson problem has homogeneous Dirichlet boundary
% conditions, and is defined on a square region with equal sized mesh widths.
%
```

```

% parameters:
% b = matrix of f values evaluated at interior meshpoints
% h = mesh width
%
% returns:
% u = solution to PDE at interior meshpoints

% get dimensions of b
[m, n] = size(b);
b_bar = zeros(n,n);
u_bar = b_bar;
u = u_bar;

% make sure we have a square grid
if (m ~= n)
    error('matrix b is not square');
end

% b_bar = 2/(n+1) * v * b * v
% first do a DST on columns of b, which is the same as multiplying v*b
for j=1:n
    b_bar(:,j) = fast_dst(b(:,j));
end

% then do DST on rows of vb, which is analogous to multiplying v*b*v
for i=1:n
    % have to take transpose of row, since fast_dst needs a column vector
    b_bar(i,:) = fast_dst(b_bar(i,:).').';
end

% now scale by 2/(n+1)
b_bar = b_bar * (2/n+1);

% next we can solve for u_bar
u_bar = zeros(n,n);
lambda = [1:n];
lambda = -4 * (sin((lambda*pi) / (2*n + 2))).^2;
for i=1:n
    for j=1:n
        u_bar(i,j) = (h^2 * b_bar(i,j)) / (lambda(i) + lambda(j));
    end
end

% u = 2/(n+1) * v * u_bar * v
% do a DST on columns of u_bar, which is analogous to multiplying v * u_bar
for j=1:n
    u(:,j) = fast_dst(u_bar(:,j));
end

% then do a DST on rows
for i=1:n
    u(i,:) = fast_dst(u(i,:).').';
end

```

```
% last, multiply by 2/(n+1)
u = u * 2/(n+1);
```

```
return
```

```
function y = fast_dst(x)
```

```
% Perform a fast DST on a vector. Since Matlab does not have a DST function,  
% this just uses the built in FFT function.
```

```
%
```

```
% parameter:
```

```
% x = input column vector
```

```
%
```

```
% returns:
```

```
% y = the DST of x
```

```
n = length(x);
```

```
tmp = zeros(2*n + 2, 1);
```

```
tmp = -imag(fft([0; x; zeros(n+1,1)]));
```

```
y = tmp(2:n+1);
```

```
return
```