Linear System Solution of Poisson Equation on Dirichlet Boundary

Daniel Underwood

July 21, 2018

Introduction

The Dirichlet boundary value problem is a boundary value problem in which the function on the boundary is defined by a function; that is

$$\hat{\mathcal{D}}u(x_1, ..., x_n) = f(x_1, ..., x_n) \tag{1a}$$
$$u(x_1, ..., x_n) = g(x_1, ..., x_n) \text{ for } (x_1, ..., x_n) \in \partial\Omega \tag{1b}$$

where $\hat{\mathcal{D}}$ is a linear differential operator consisting of a sum of derivatives of any order including mixed derivatives and u, f, and g are functions of n variables from $\mathbb{R}^n$ to $\mathbb{R}$. The problem is solved on a domain $\Omega \subset \mathbb{R}^n$ with a boundary $\partial\Omega$. In simpler terms, a Dirichlet problem is a boundary value problem in which the solution value is given in terms of a function, rather than other conditions such as the value of the solution's derivative on the boundary that is given in a Neumann problem.

The Dirichlet problem is very useful in a variety of fields. Differential equations frequently used with Dirichlet boundary conditions include the Laplace equation, Poisson equation, heat/diffusion, and wave equation. These equations are seen quite often and the latter three are actually all more general forms of the Laplace equation; the Poisson equation is a non-homogeneous form while the heat/diffusion and wave equations add first and second time derivative terms, respectively.

The Laplace equation

$$\Delta u = 0 \tag{2}$$

is one of the most basic partial differential equations used with Dirichlet boundary conditions. The solutions to this equations are harmonic functions. The Laplace equation has uses in various fields of physics to describe potentials due to forces such as gravity, the electromagnetic force, and fluid forces. It it also the steady-state heat or diffusion equation as well as the steady-state wave equation by taking $u_t = 0$ in eq. (4) or $u_{tt} = 0$ in the $\Box$ operator of eq. (5). The solutions of the Laplace equation are harmonic functions.

The Poisson equation

$$\Delta u = f \tag{3}$$

Is the non-homogeneous form of eq. (2). The Poisson equation has uses throughout physics as f can be used to represent forces such as gravity, electromagnetism, and fluid forces. In these cases, u represents a potential, which can be related to a force via Newton's second law or can be used in energy analysis. The f term comes from a source. In the case of an electrostatic field, we have $f = -\frac{\rho(r)}{\epsilon_0}$, where ρ is the radial charge density; the gravitational source term is $f = 4\pi^2 G\rho(r)$, where G is the universal gravitational constant and ρ is the radial mass density.

The heat or diffusion equation

$$u_t - a\Delta u = 0 \tag{4}$$

is a frequently used partial differential equation to represent the transfer of heat on the domain of a piece of material or the diffusion of a fluid through another fluid. a is either the heat transfer coefficient

or the diffusion coefficient depending on the usage case. It is frequently constant as a simplification, but can vary with position or time in more complicated cases. In these cases, the divergence term affects the transfer function and the equation will take on a slightly different form.

The wave equation

$$\Box u = 0 \tag{5}$$

where $\Box \equiv \frac{1}{c^2}\frac{\partial^2}{\partial t^2} - \Delta$ is the d'Alembert operator with c being the wave speed. The wave equation has usage throughout physics. In classical mechanics, the wave equation is used to represent oscillations and is the start to the large field of wave mechanics. Electromagnetism and quantum mechanics have their own wave equations, being Maxwell's equations for electromagnetism and the Schrödinger equation in quantum mechanics.

Mathematical Theory

There are many ways to solve eqs. (2)–(5). Analytically, they may be solved by separation of variables for many cases and in the case of Dirichlet boundary conditions may be solved by using Green's function; however, both of these techniques can become very difficult or even impossible depending on the domain on which the problem is being solved. Due to this difficulty, numerical methods start becoming highly favorable. While numerical methods do not yield a solution in terms of analytic functions, they can give a good idea of the behavior of the solution and parameters can be adjusted to make the error within an acceptable range. Additionally, once written, numerical methods can be used on a variety of variations of the problem and are often solved much quicker than a human could solve the same equation if a solution exists. Numerical data can even be fitted by functions to make guesses of analytic solutions or approximations.

Numerical solutions represent discrete relations rather than continuous ones and can be formulated by transforming the differential equation into a linear system. In this case, we have Poisson's equation with a Dirichlet boundary on a domain $\Omega \subset \mathbb{R}^2$

$$-u_{xx} - u_{yy} = f(x, y) \tag{6a}$$
$$u(x, y) = 0 \text{ for } (x, y) \in \partial\Omega \tag{6b}$$

The domain Ω is shown in fig. 1. The domain could be scaled to arbitrary sizes, so it is easiest to take the squares making up Ω to be unit squares.

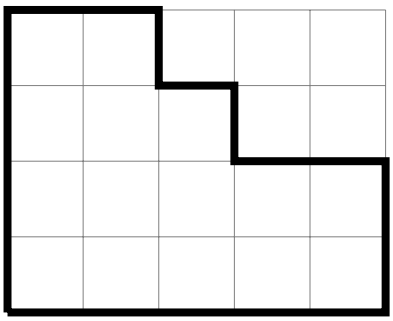


Figure 1: Domain Ω

To solve a differential equation numerically, we must discretize the differential operator into steps. A starting point for this is the limit definition of a derivative

$$f'(a) = \lim_{h \to 0} \frac{f(a + h) - f(a)}{h} \tag{7}$$

To discretize this limit, we take $h \to \Delta h$, where Δh is small and taken to be smaller to achieve more accurate results. This relation can be used with a Taylor series to find relations for higher-order

derivatives. Applying this to find the Laplace operator for $\mathbb{R}^2$ and indexing values yields the algebraic relation

$$-\Delta u(x_i, y_j) = \frac{4u_{ij} - u_{i+1,j} - u_{i+1,j} - u_{i,j+1} - u_{i,j-1}}{(\Delta h)^2} \tag{8}$$

With eq. (8), an operator matrix $\hat{A}$ may be formed such that $\hat{A}\vec{u}$ is a discretized form of Δu for $\vec{u}$ is a vector formed by $u(x_i, y_j)$, where x_i and y_j are points that are created when the boundary Ω is separated into a mesh with step size Δh. We then obtain a linear system

$$\hat{A}\vec{u} = \vec{f} \tag{9}$$

where $\vec{f}$ is the vector created from $f(x_i, y_j)$ evaluated at the same mesh points as $\vec{u}$.

This system can be solved by several different methods. To simplify the computation, the Δh term is factored out of $\hat{A}$ and placed with the function term

$$\tilde{\hat{A}}\vec{u} = (\Delta h)^2 \vec{f} \tag{10}$$

where $\tilde{\hat{A}} \equiv (\Delta h)^{-2} \hat{A}$. With this system defined by the operator in eq. (8), we have an error of order $(\Delta h)^4$.

Large Step Sizes

The system in eq. (10) can be explicitly solved relatively easily for small step sizes. This was done for step sizes of $h = 1$ and $h = \frac{1}{2}$ in listings 1–2, respectively. In these cases, the operator matrices were manually assigned with values of -1 for interacting points.

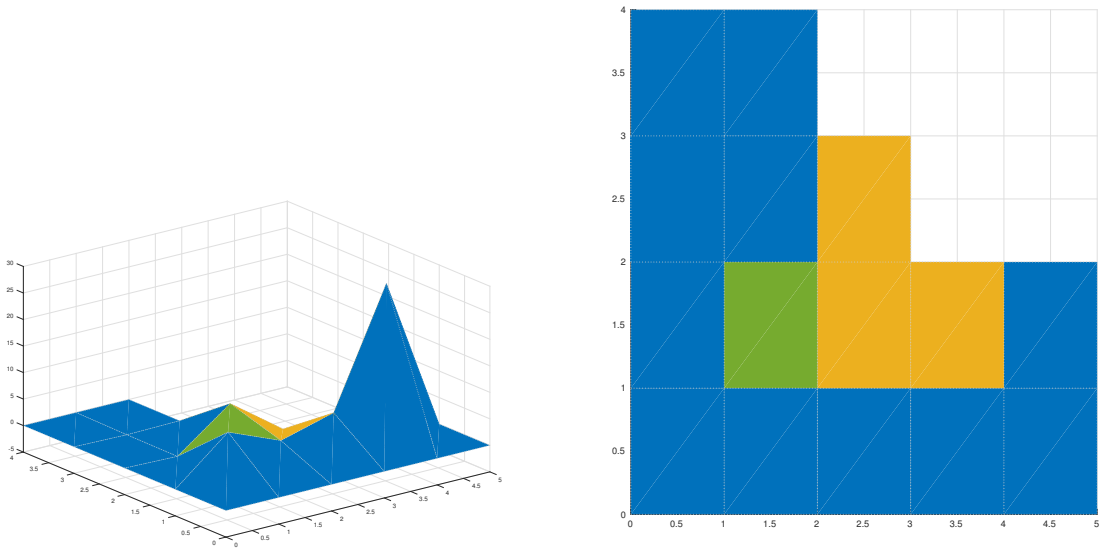


Figure 2: Solution with $h = 1$

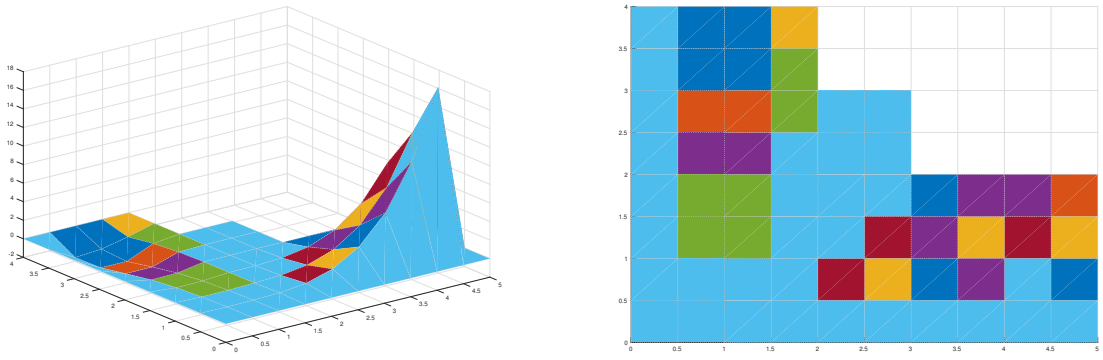


Figure 3: Solution with $h = \frac{1}{2}$

Figures 2–3 respectively show the solutions with the explicitly solved values of h.

Computational Details

To solve this problem numerically, MATLAB code was used. Relevant code is shown and the entire function used is attached in listing 3.

The first step to solving this problem numerically is to create the domain and split it into a mesh based on a step size, which is Δh in eq. (8). In the code, this was done by using a matrix to represent the points inside the domain. For example, the matrix that represents the domain of interest shown in fig. 1 is given by

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

As can be seen, the matrix is in $\mathbb{R}^{m \times n}$ where m is the maximum y value of the domain and n is the maximum x value of the domain if the domain is split into unit squares. Within the matrix, a 1 represents a unit square while a 0 represents a place that is not in the domain. Using this method, it is easy to use an arbitrary domain. The matrix can be of any dimension and the size can be scaled to more accurately represent features such as curves. This way of representing the domain can even deal with domains with features such as holes.

When the domain is split up into smaller pieces, the unit squares of the domain simply repeat $\frac{1}{\Delta h}$ times in both their column and the next row. For example, when a step size of $\Delta h = \frac{1}{2}$ is used, the above matrix undergoes the transformation

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

For this reason, we must choose $\Delta h \leq 1$, although it is already known that a smaller Δh yields better results. This expansion of matrices is handled by using a kronecker product to expand the rows and columns by the step size followed by adding a row and column for the missing points, as the first expanded row and column will have an extra for each step.

```
% Expand domain using the kronecker product
expandedDomain = kron(domainMatrix, ones(steps));

% Fix x axis missing expanded values and boundaries
leftSide = kron(expandedDomain(:,1), ones(1, steps-1));
leftBoundary = zeros(size(leftSide,1), 1);
rightBoundary = leftBoundary;
expandedDomain = [leftBoundary, leftSide, expandedDomain, rightBoundary];

% Fix y axis missing expanded values and boundaries
bottom = kron(expandedDomain(1,:), ones(steps-1, 1));
topBoundary = zeros(1, size(bottom,2));
bottomBoundary = topBoundary;
expandedDomain = [bottomBoundary; bottom; expandedDomain; topBoundary];
```

This section of the code is very fast, even on quite large matrices. It would likely have greater performance if the for loops were changed to parfor loops, although memory and performance limitations cause issues with solving eq. (10) much more quickly than the formation of the matrices in the above code.

The next step is to create the operator matrix $\tilde{\tilde{A}}$. Due to the $4u_{ij}$ term in eq. (8), it is known that the diagonals of $\tilde{\tilde{A}}$ are 4, so an identity matrix is created and multiplied by 4. The interacting points are found by finding points in the expanded domain with MATLAB's find function and inspecting the points above, below, left, and right, of the current point. This is the slowest portion of the code other than solving the system. As with the expansion of the domain, the performance of this code could be increased by using more parallel code, but solving the system is a much more pressing performance issue.

```
% Get x and y points by finding nonzero elements
[yPoints, xPoints] = find(expandedDomain);

% Scale x and y points based on step size
xPoints = stepSize * xPoints;
yPoints = stepSize * yPoints;

% Form operator matrix for equation
% Start with fours on diagonals
operatorMatrix = sparse(1:numel(xPoints), 1:numel(xPoints), ...
    4 * ones(numel(xPoints), 1));

% Check surrounding points and use -1 where necessary
for i=1:numel(xPoints)
    currentPoint = [xPoints(i), yPoints(i)];

    % Set up coordinates of surrounding points
    pointAbove = [currentPoint(1), currentPoint(2) + stepSize];
    pointBelow = [currentPoint(1), currentPoint(2) - stepSize];
    pointLeft = [currentPoint(1) - stepSize, currentPoint(2)];
    pointRight = [currentPoint(1) + stepSize, currentPoint(2)];

    % Vector of surrounding points for easy iteration
    surroundingPoints = [pointAbove; pointBelow; pointLeft; pointRight];

    % Get single indices of surrounding points
    for j=1:numel(surroundingPoints) / 2
        idx = findPoint(xPoints, yPoints, [surroundingPoints(j, 1);
            surroundingPoints(j,2)]);
        operatorMatrix(i, idx) = -1;
    end
```

The interior structure of the matrix can be seen using MATLAB's spy command and is found to be what is shown in fig. 4. From this, it is clear that the matrix is sparse; that is, most of its elements are 0. This allows an optimization in MATLAB by using a sparse matrix. Using a sparse matrix to represent the operator matrix for a step size of $h = \frac{1}{128}$ reduced memory usage from 450GB to 400MB – roughly a difference of three orders of magnitude! This allows nearly any machine to compute solutions for small step sizes, although processing time can be significant.

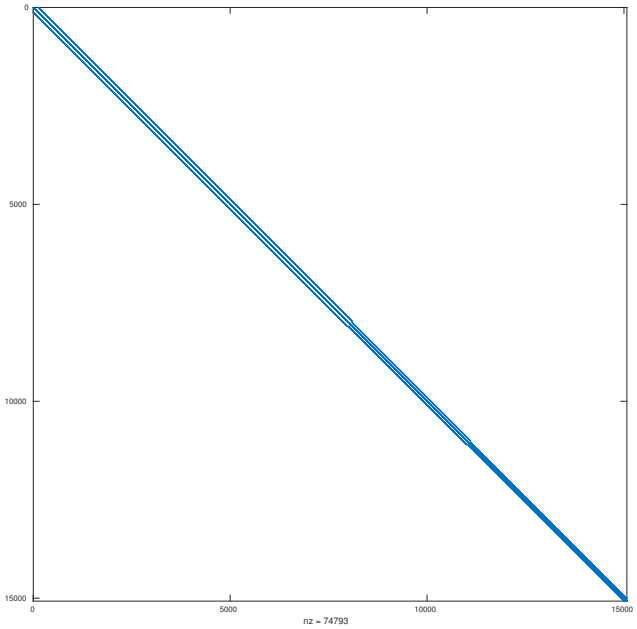


Figure 4: Operator Matrix Structure

The final step before solving the linear system with MATLAB's built-in functions is to form the right-hand side of eq. (10). This is simply done by evaluating $f(x, y)$ at each of the points in the domain and put the results in an array and multiply that array by $(\Delta h)^2$. This is done in the code by a vector operation

```
55
56 % Create RHS vector
```

Now, the linear system is ready to be solved using built-in functions.

```
55
56 % Solve System
57 solutionMatrix = operatorMatrix \ rhsVector;
58
59 % Coordinates for Return
60 [X, Y] = meshgrid(0:stepSize:size(domainMatrix, 2)+1, ...
61     0:stepSize:size(domainMatrix, 1)+1);
62
63 Z = expandedDomain;
64 for i=1:numel(solutionMatrix)
65     Z(find(Z == 1, 1)) = solutionMatrix(i);
66 end
```

```
67
68 % Get rid of x and y coordinates not inside domain or boundary
69 zeroIndices = find(~conv2(Z, [1 1 1; 1 0 1; 1 1 1], 'same'));
70 X(zeroIndices) = NaN;
```

In addition to solving the system, the code snippet above finds points that are out of the domain and removes them from the variables for the x and y points so the domain is plotted properly.

Conclusion

The code presented in this report was used to solve the Poisson equation with the right hand sides

$$f(x,y) = xe^{-x^2-y^2} \tag{11a}$$

$$f(x,y) = x\,(x-y)^3 \tag{11b}$$

with $\Delta h = \frac{1}{64}$. The solution plots were then compared with the solutions found by MATLAB's PDE toolbox. These plots are shown in figs. 6–9 on the following pages. The plots in fig. 5 show the effect of step sizes of $1, 2, 4, 8$ and the rapid approach to the expected solution.

Overall, this method generates very good results with the differential equation and boundary in this case. Due to the flexible nature of how the domain was implemented, it is expected that this method would work for many other domains. Compared to MATLAB's PDE toolbox, the results are slightly worse although the code is likely much simpler. This method takes a similar amount of time to MATLAB's PDE toolbox to run, which is a good sign. The performance could likely be improved by creating more parallel code and carefully profiling the code.

An additional approvement could be made in the representation of the domain. Although the matrix representation of domain points works very well and allows for features such as holes in the domain, it is not easily tested and new bugs may be found when a different domain is used. An improvement to this would be a more advanced system of using a graph data structure to represent the domain. In this system, the nodes would represent points in space. These points could be inside or outside of the domain or be on the boundary of the domain. The edges in the graph would be used to split points to subdivide the points. This would allow the domain to no longer be restricted to rectangles or two dimensions; however, this increase in generality would also require the derivation of a more general operator matrix.

Solution Plots

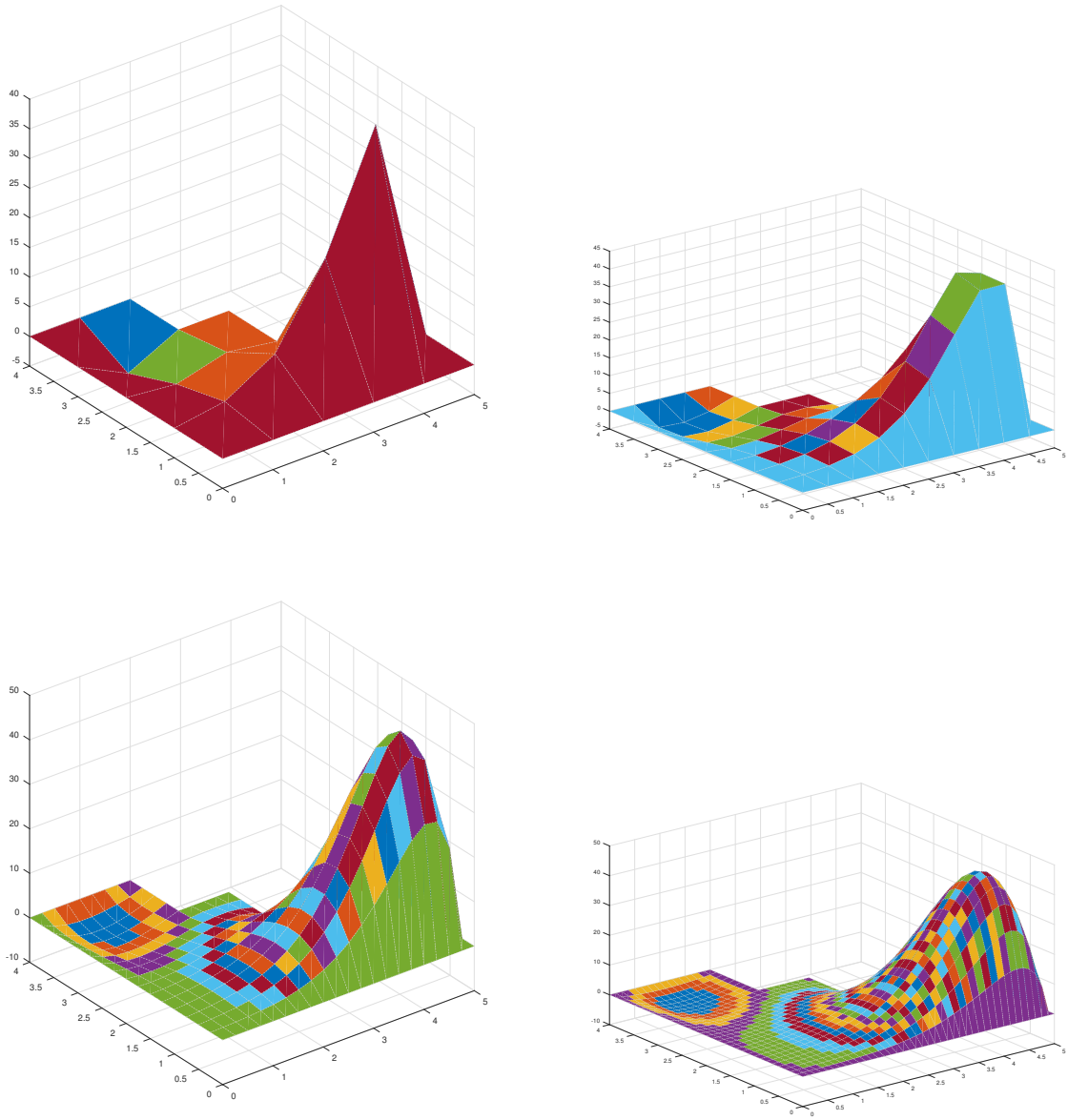
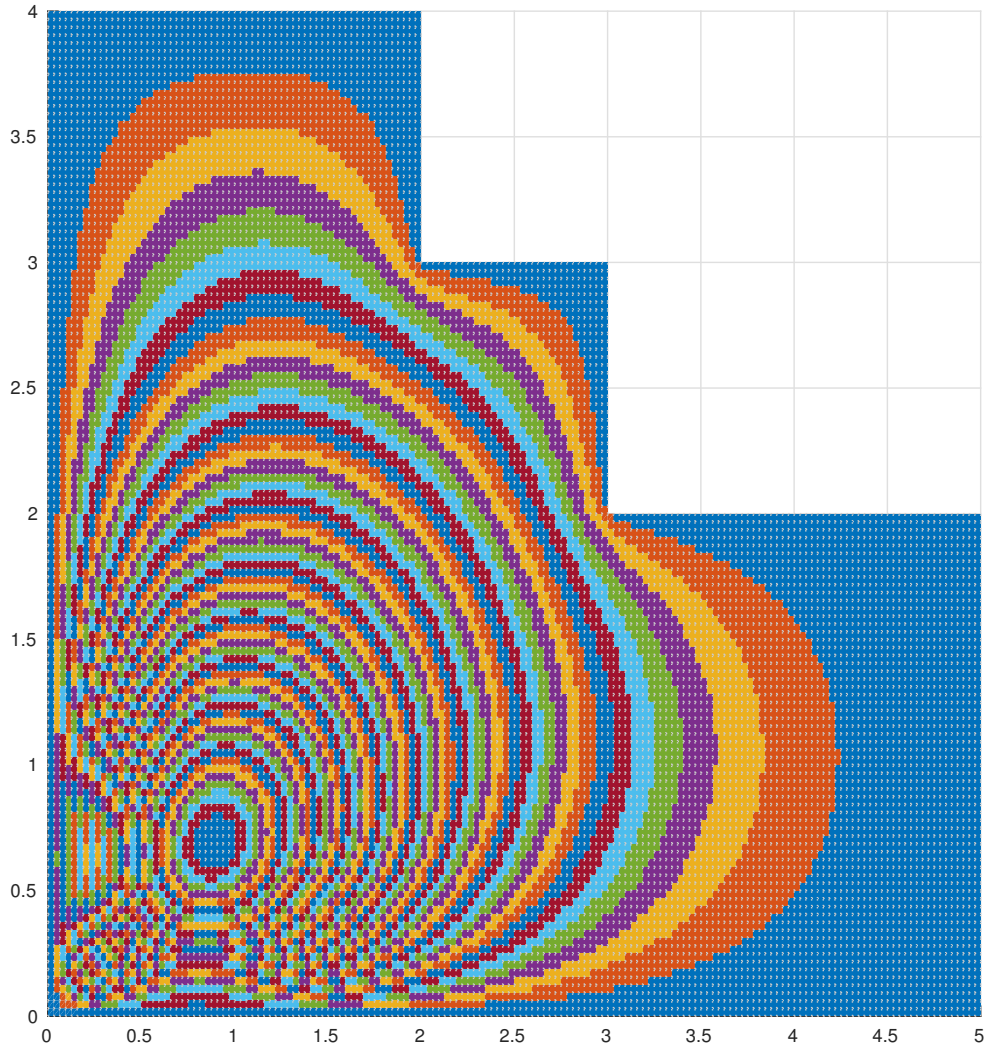
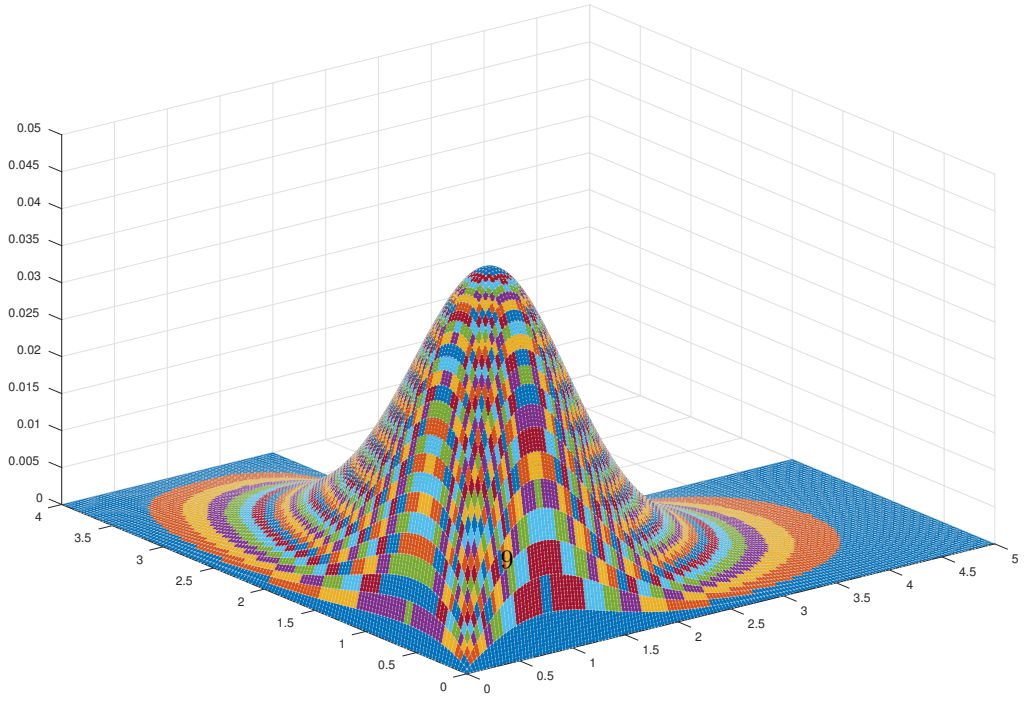


Figure 5: Solutions to Equation (11b) with $\Delta h = 1, 2, 4, 8$


4
3.5
3
2.5
2
1.5
1
0.5
0
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5


0.05
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
4 3.5 3 2.5 2 1.5 1 0.5 0
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5

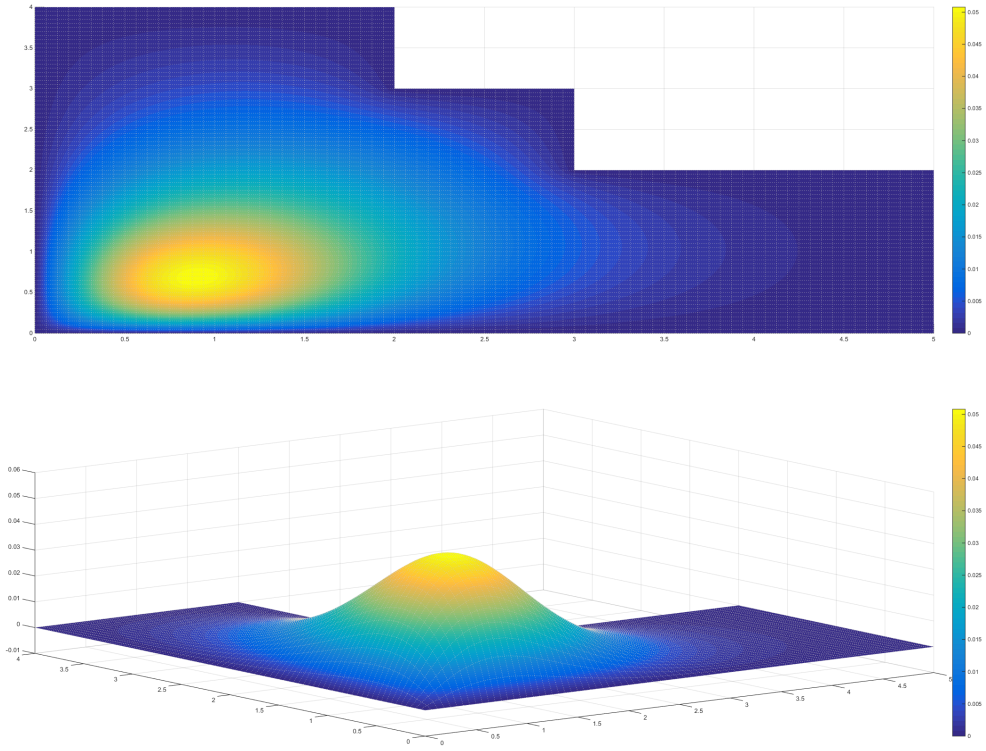


Figure 7: PDE Toolbox Solutions to Equation (11a)

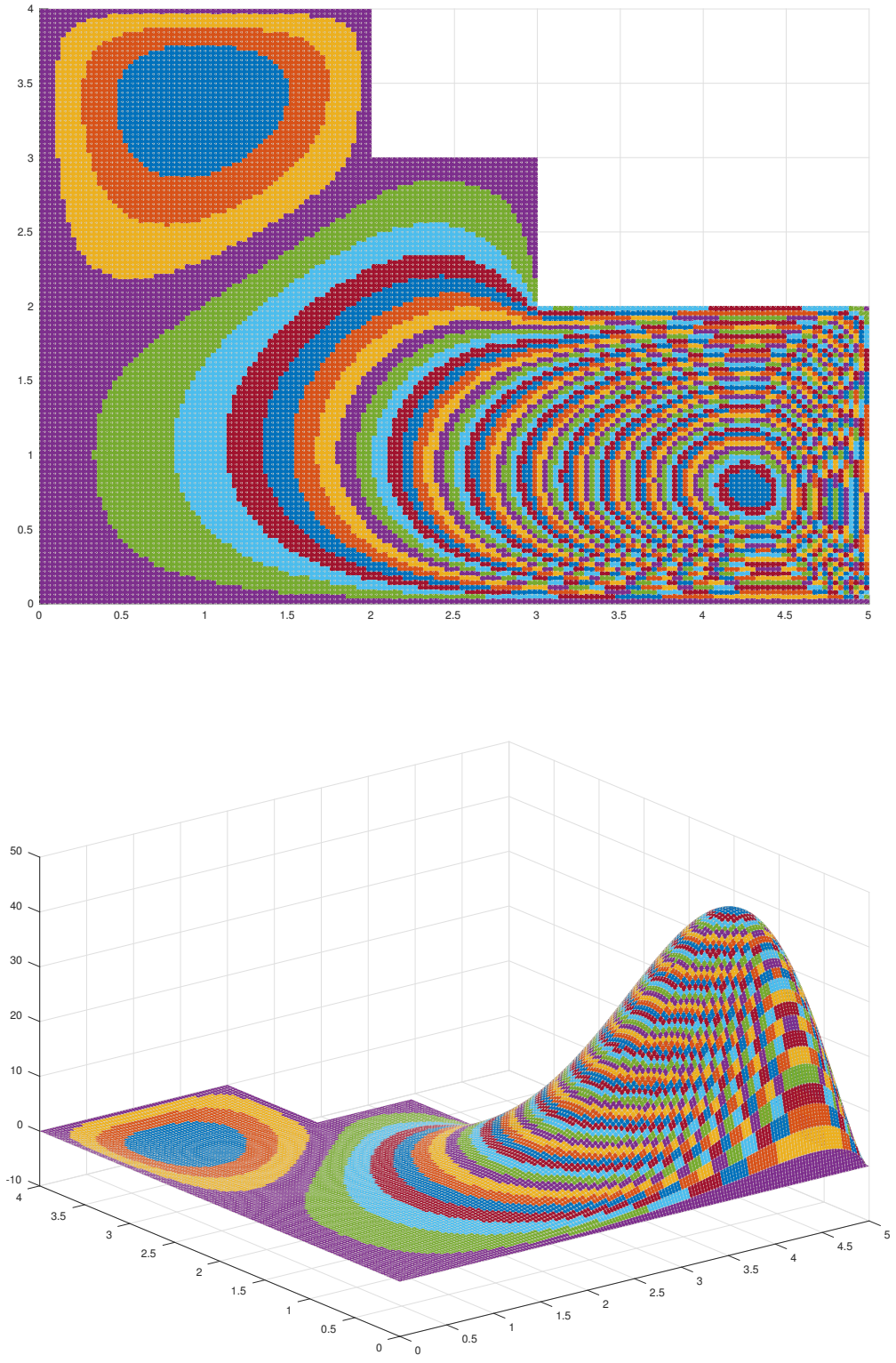


Figure 8: Solutions to Equation (11b)

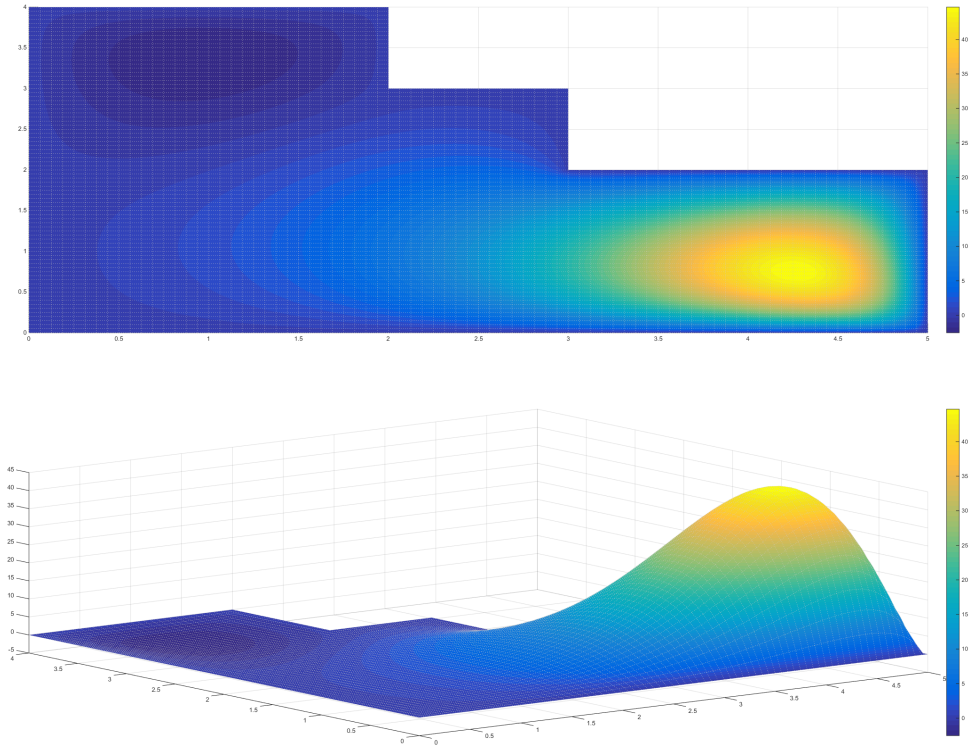


Figure 9: PDE Toolbox Solutions to Equation (11b)

Full Code Listing

Below is a listing of the code used for this report. The most recent code is available in the source repository at `https://github.com/danielunderwood/poisson-central-difference` and the most recent compiled version of the report is available at `https://www.sharelatex.com/github/repos/danielunderwood/poisson-central-difference/builds/latest/output.pdf`.

Listing 1: explicit_h1_solver.m

```
function [ X, Y, Z ] = explicit_h1_solver( rhs )
%EXPLICITH1_SOLVER Explicitly solves the poisson equation on domain
%   Explicitly solves the poisson equation on the given domain with step
%   size of 1. rhs Is a function pointer with two arguments

% Differential Operator Matrix
D = 4 * eye(7);
D(1,2) = -1;
D(1,4) = -1;
D(2,1) = -1;
D(2,5) = -1;
D(2,3) = -1;
D(3,2) = -1;
D(4,1) = -1;
D(4,5) = -1;
D(6,4) = -1;
D(7,6) = -1;

% X and Y Coordinates
X = [1; 2; 3; 4; 1; 2; 1];
Y = [1; 1; 1; 1; 2; 2; 3];

% Evaluate rhs at points
rhsVector = rhs(X,Y);

% Solve for solution vector
solutionVector = D \ rhsVector;

[X,Y] = meshgrid(0:5, 0:4);

Z = [ 0, 0, 0, 0, 0, 0;
    0, solutionVector(1), solutionVector(2), solutionVector(3),
        solutionVector(4), 0;
    0, solutionVector(5), solutionVector(6), 0, 0, 0;
    0, solutionVector(7), 0, 0, 0, 0;
    0, 0, 0, 0, 0, 0];

% Get rid of x and y coordinates not inside domain or boundary
zeroIndices = find(~conv2(Z, [1 1 1; 1 0 1; 1 1 1], 'same'));
X(zeroIndices) = NaN;
Y(zeroIndices) = NaN;

end
```

Listing 2: explicit_h12_solver.m

```
function [ X, Y, Z ] = explicit_h12_solver( rhs )
%EXPLICITH12_SOLVER Explicitly solves the poisson equation on domain
%   Explicitly solves the poisson equation on the given domain with step
```

```
%   size of 1/2. rhs Is a function pointer with two arguments

% --- Differential Operator Matrix ---
% Start with 4 on main diagonal
D = 4 * eye(43);
% Add -1 for interactions
i = [1;1;2;2;2;3;3;3;4;4;4;5;5;5;6;6;6;7;7;8;8;8;9;9;9;9;10;10;10;10;
     11;11;11;11;12;12;12;12;13;13;13;13;14;14;14;15;15;15;16;16;16;16;
     17;17;17;17;18;18;18;18;19;19;19;19;20;20;20;21;21;22;22;22;23;23;
     23;23;24;24;24;24;25;25;25;25;26;26;26;27;27;27;28;28;28;28;29;29;
     29;29;30;30;30;31;31;32;32;32;33;33;33;33;34;34;34;35;35;35;36;36;
     36;36;37;37;37;38;38;38;39;39;39;39;40;40;40;41;41;42;42;42;43;43];

j = [2;8;1;9;3;2;10;4;3;11;5;4;12;6;5;13;7;6;14;1;9;15;2;10;16;8;3;11;
     17;9;4;12;18;10;5;13;19;11;6;14;20;12;7;21;13;8;16;22;9;17;23;15;10;
     18;24;16;11;19;25;17;12;20;26;18;19;13;21;20;14;15;23;27;16;24;28;22;
     17;25;29;23;18;26;30;24;19;31;25;22;28;32;23;29;33;27;24;30;34;28;
     25;31;29;26;30;27;33;35;28;34;36;32;29;37;33;32;36;38;33;37;39;35;
     34;40;36;35;39;41;36;40;42;38;37;43;39;38;42;41;39;43;42;40];

for k=1:43
    D(i(k),j(k)) = -1;
end

domain = [1 1 1 1 1 1 1 1 1;
     1 1 1 1 1 1 1 1 1;
     1 1 1 1 1 1 1 1 1;
     1 1 1 1 1 0 0 0 0;
     1 1 1 1 1 0 0 0 0;
     1 1 1 0 0 0 0 0 0;
     1 1 1 0 0 0 0 0 0];

% Get x and y points
[yPoints, xPoints] = find(domain);
xPoints = 0.5 * xPoints;
yPoints = 0.5 * yPoints;


% Evaluate rhs at points
rhsVector = 0.5^2 * rhs(xPoints, yPoints);

% Solve for solution vector
solutionVector = D \ rhsVector;

% X and Y coordinates
[X,Y] = meshgrid(0:0.5:5, 0:0.5:4);

Z = [zeros(7,1), domain, zeros(7,1)];
Z = [zeros(1,11); Z; zeros(1,11)];
for i=1:numel(solutionVector)
    Z(find(Z == 1, 1)) = solutionVector(i);
end

% Get rid of x and y coordinates not inside domain or boundary
zeroIndices = find(~conv2(Z, [1 1 1; 1 0 1; 1 1 1], 'same'));
X(zeroIndices) = NaN;
```

```
Y(zeroIndices) = NaN;

end
```

Listing 3: centralDifferencePoisson.m

```
function [X, Y, Z] = centralDifferencePoisson(stepSize, domainMatrix, rhs)
% Solve poisson equation with domain given by unit squares in
% matrix. The differential equation is then represented by a
% matrix system of equations to solve.

% Number of steps
steps = round(1 / stepSize);

% Expand domain using the kronecker product
expandedDomain = kron(domainMatrix, ones(steps));

% Fix x axis missing expanded values and boundaries
leftSide = kron(expandedDomain(:,1), ones(1, steps-1));
leftBoundary = zeros(size(leftSide,1), 1);
rightBoundary = leftBoundary;
expandedDomain = [leftBoundary, leftSide, expandedDomain, rightBoundary];

% Fix y axis missing expanded values and boundaries
bottom = kron(expandedDomain(1,:), ones(steps-1, 1));
topBoundary = zeros(1, size(bottom,2));
bottomBoundary = topBoundary;
expandedDomain = [bottomBoundary; bottom; expandedDomain; topBoundary];

% Get x and y points by finding nonzero elements
[yPoints, xPoints] = find(expandedDomain);

% Scale x and y points based on step size
xPoints = stepSize * xPoints;
yPoints = stepSize * yPoints;

% Form operator matrix for equation
% Start with fours on diagonals
operatorMatrix = sparse(1:numel(xPoints), 1:numel(xPoints), ...
    4 * ones(numel(xPoints), 1));

% Check surrounding points and use -1 where necessary
for i=1:numel(xPoints)
    currentPoint = [xPoints(i), yPoints(i)];

    % Set up coordinates of surrounding points
    pointAbove = [currentPoint(1), currentPoint(2) + stepSize];
    pointBelow = [currentPoint(1), currentPoint(2) - stepSize];
    pointLeft = [currentPoint(1) - stepSize, currentPoint(2)];
    pointRight = [currentPoint(1) + stepSize, currentPoint(2)];

    % Vector of surrounding points for easy iteration
    surroundingPoints = [pointAbove; pointBelow; pointLeft; pointRight];

    % Get single indices of surrounding points
    for j=1:numel(surroundingPoints) / 2
```

```
51          idx = findPoint(xPoints, yPoints, [surroundingPoints(j, 1);
                surroundingPoints(j,2)]);
52          operatorMatrix(i, idx) = -1;
53      end
54  end
55
56  % Create RHS vector
57  rhsVector = stepSize.^2 * rhs(xPoints, yPoints);
58
59  % Solve System
60  solutionMatrix = operatorMatrix \ rhsVector;
61
62  % Coordinates for Return
63  [X, Y] = meshgrid(0:stepSize:size(domainMatrix, 2)+1, ...
64      0:stepSize:size(domainMatrix, 1)+1);
65
66  Z = expandedDomain;
67  for i=1:numel(solutionMatrix)
68      Z(find(Z == 1, 1)) = solutionMatrix(i);
69  end
70
71  % Get rid of x and y coordinates not inside domain or boundary
72  zeroIndices = find(~conv2(Z, [1 1 1; 1 0 1; 1 1 1], 'same'));
73  X(zeroIndices) = NaN;
74  Y(zeroIndices) = NaN;
75
76
77  end
```