Implementation of the 2-D Solution for the Poisson Equation on the Dirichlet Boundary

Saran Ahluwalia and Mountain Chan

October 9, 2020

Introduction

The Dirichlet boundary value problem is a variant of the boundary value problem in which the function on the boundary is as follows:

$$\hat{\mathcal{S}}u(x_1, ..., x_n) = f(x_1, ..., x_n) \tag{1a}$$

$$u(x_1, ..., x_n) = g(x_1, ..., x_n) \text{ for } \forall(x_1, ..., x_n) \in \partial\Omega \tag{1b}$$

where $\hat{\mathcal{S}}$ is a linear differential operator consisting of the summand of any order derivative including heterogenuous partial derivatives. Moreover, 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 defined boundary $\partial\Omega$. In essence, the Dirichlet problem is a boundary value problem in which the solution value is a function. This is diametrically opposed to the Neumann problem in which the value of the solution's derivative is on the boundary [1].

The Dirichlet problem is very useful in a variety of fields. In particular, differential equations that are often used with Dirichlet boundary conditions include the Laplace equation, Poisson equation and diffusion. The aforementioned are actually all more general forms of the Laplace equation; the Poisson equation is a non-homogeneous form while the diffusion equation adds first and second derivative terms, respectively [2].

The Laplace equation

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

is a partial differential equation used with Dirichlet boundary conditions. The solutions to this equations are harmonic functions. On the other hand, the Laplace equation is leveraged in machine learning. In particular, there is an emerging interest in its exploitation in natural language processing within the context of reducing dimensions for vectorized word embeddings. Moreover, this operator is often used to smooth the gradient during batch gradient descent [3]. The Poisson equation

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

is the non-homogeneous form of eq. (2). The Poisson equation representation for f can be used to represent forces such as gravity, electromagnetism, and fluid forces. Recent developments in its application are also in neural networks in order to solve elliptic PDEs. A network is constructed for the boundary and another for the PDE network. Here an update on the boundary occurs multiple times before a single evaluation on the interior. Thereafter, the loss function is a computed based off of the boundary loss and interior loss[4]. In the former applications, 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

[1] Greenspan, Donald. "Numerical Analysis and the Dirichlet Problem." Mathematics Magazine, vol. 32, no. 4, 1959, pp. 177–188. JSTOR, www.jstor.org/stable/3029112.

[2] Volker John's Course Notes for PDE and numerical analysis: `https://www.wias-berlin.de/people/john/LEHRE/NUM_PDE_FUB/num_pde_fub.pdf`

[3] Stanford CS229: Machine Learning: `http://cs229.stanford.edu/notes/cs229-notes1.pdf`

[4] Deep Learning for PDEs: `http://stanford.edu/~kailaix/files/Deep_Learning_for_Partial_Differential_Equations.pdf`

a source origin. 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[5].

In the latter applications, there have been novel approaches that reliably compute many useful properties of a silhouette for images. It has been previously been show that this function can be used to reliably extract various shape properties including part structure and rough skeleton, local orientation and aspect ratio of different parts, and convex and concave sections of the boundaries. This abets in shape classification algorithms [6].

Mathematical Theory

There are myriad methods to solve eq. (2). Analytically, this equation is often solved through the separation of variables. Specifically, with Dirichlet boundary conditions, the Lapace equation may be solved by using Green's function; however, both of these aforementioned techniques can become infeasible depending on the domain on which the problem is being solved. Hence an exploration utilizing numerical methods is warranted in order to provide a more exhaustive heuristic to solve these equations.

While numerical methods do not necessarily yield a solution in terms of analytic functions, they do provide abstract depictions of the solution. The solution and parameters can be adjusted in order to minimize error within a certain threshold. Additionally, once written, numerical methods can be used on a variety of variations of the problem and are often solved more expeditiously. Numerical data can even be fitted by functions to make approximations of analytic solutions.

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 $\Omega \subset \mathbb{R}^2$

$$-u_{xx} - u_{yy} = f(x, y) \tag{4a}$$
$$u(x, y) = 0 \text{ for } \forall (x, y) \in \partial\Omega \tag{4b}$$

The domain Ω is depicted in fig. 1. The domain could be scaled to arbitrary sizes; however, it is more feasible to utilize Ω to be unit squares.

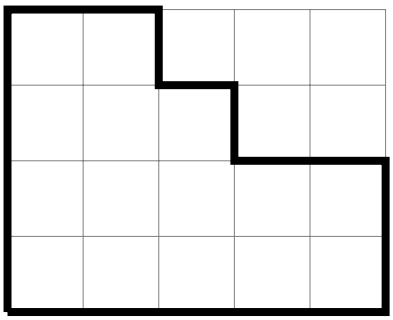


Figure 1: Domain Ω

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 [7]

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h} \tag{5}$$

[5]Lecture Notes by Anthony Pierce: Solving the Heat, Laplace and Wave equations using finite difference methods `https://www.math.ubc.ca/~peirce/M257_316_2012_Lecture_8.pdf`.

[6]Shape Representation and Classification Using the Poisson Equation `http://www.wisdom.weizmann.ac.il/~meirav/TPAMI-0429-0805-1.pdf`.

[7]From Vivi Andasari 2015 Course `http://people.bu.edu/andasari/courses/Fall2015/be503703Fall2015.html`.

To discretize this limit we allow $h \to \Delta h$, where Δh is minimal and decreases in order to improve the accuracy of the result. This relation can be used with a Taylor series to find relations for higher-order derivatives (as Dr.Moody Chu discussed in lecture). Applying this to find the Laplace operator for $\mathbb{R}^2$ and indexing values yields the algebraic relation

$$-\Delta u(x_i, y_j) = \frac{4u_{ij} - u_{i+1,j} - u_{i+1,j} - u_{i,j+1} - u_{i,j-1}}{(\Delta h)^2} \tag{6}$$

With eq. (6), 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

$$\hat{A}\vec{u} = \vec{f} \tag{7}$$

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

$$\hat{A}'\vec{u} = (\Delta h)^2 \vec{f} \tag{8}$$

where $\hat{A}' \equiv (\Delta h)^{-2} \hat{A}$. With this system defined by the operator in eq. (6), we have an error of order $(\Delta h)^4$ [8].

Computational Details

Below is detailed part 1(a) of this assignment which entailed depicting the operator matrix with the 7×7 dimension A in a table format:

4	-1	0	0	-1	0	0
-1	4	-1	0	0	-1	0
0	-1	4	-1	0	0	0
0	0	-1	4	0	0	0
0	0	0	0	4	-1	-1
0	0	0	0	-1	4	0
0	0	0	0	-1	0	4

Programmatically, in order to solve this problem, Python 3.6, with packages NumPy, SciPy and Matplotlib was used. Prototyping and validation of the generic solver and the solutions for (1a) and (1b) were performed leveraging Matlab in order to verify dimension structure and adherence to project-specific requirements. Specifically, the validation was isolated to the construction of the operator matrix, the augmented domain and solution vectors. Subsequently these evaluations were computed with SciPy, transformed into the Matlab API for full double arrays and then plotted using the surf API [9]. Relevant Python snippets are displayed in this section.

Briefly, the first step for solving a given problem is to establish a domain matrix. Subsequently, the step size is used to construct a mesh grid which is Δh in eq. (6). Programmatically a matrix was furnished in order to represent the interior points within the domain. For example, the matrix that represents the domain of interest shown in fig. 1 is given by. This is the base domain that was provided for problem one. This domain was subsequently extrapolated to a generic solution for part two.

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

This matrix that can be any matrix in $\mathbb{R}^{m \times n}$ represents m the maximum y value of the domain and n and where the maximum x value of the domain if the domain is split into unit squares. A 1 represents

[8] From Dr. Robert Hunt's Course at Cambridge University `http://www.damtp.cam.ac.uk/user/reh10/lectures/nst-mmii-chapter2.pdf`.

[9] Mathworks Surf Plotting `https://www.mathworks.com/help/matlab/ref/surf.html`

a unit square while a 0 represents a location in space that is not in the domain. This heuristic enables any solver to extrapolate the solution matrix to any arbitrary domain. Hence, this matrix can be of any dimension and the dimension can be scaled to more accurately represent nonlinear features.

Furthermore, splitting the domain via the unit squares of the domain entails repeating $\frac{1}{\Delta h}$ times in both their column and the next row. For example, when a step size of $\Delta h = \frac{1}{2}$ is used, the aforementioned matrix undergoes the transformation

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

Thus, empirical evidence suggests a delta value of $\Delta h \leq 1$ is optimal and empirically appears to produce the most refined solutions. Empirically, we confirmed that smaller Δh sizes improved solutions, controlling for the domain and function. Unfortunately, due to time constraints and task prioritization accrued results of the runtime are not presented in this section.

The Kronecker product[10] was used in order to expand the rows and columns by the step size. Subsequently, a row and column are concatenated, vertically and horizontally along the dimension in order to normalize the missing interior points in both the domain of x and y.

The aforementioned procedure for creating the eq. (8) is relatively fast, even with relatively large (k dimensional) matrices. Specifically, in Python for a step size of 32, the mean time for this step was 0.009035 seconds with the following operating system specifications: 2 vCPU @ 2.2GHz and 13GB RAM. This is not surprising considering that NumPy arrays and matrices are written in C.

The next step for the algorithm is to construct the actual operator matrix $\tilde{\tilde{A}}$. Due to the $4u_{ij}$ term in eq. (6), it was proven in class that the diagonals of $\tilde{\tilde{A}}$ are 4. Thus initializing an identity matrix is created and multiplied by the scalar, four. For a given point in the domain, the interior points are found by finding points in the expanded domain with a combination of both NumPy's **nonzero** and **where** functions and interrogating the immediate neighbors, above, below, left, and right, of the given point.

The interior structure of the matrix can be seen using Matplotlib's axes library's **spy** command and resulted in the figure below fig. 4. The figure below indicates that the matrix is sparse. We leveraged SciPy's CSR formatted sparse matrix [11] for the generic solver in order to compute more performant solutions for smaller step sizes.

For example, with a step size of $h = \frac{1}{1028}$ (run on Google Compute (GCE) [12] standard machine type with 8 vCPUs and 30 GB of memory) reduced memory usage, on average, from approximately 20000MB to approximately 4000MB after 10 trials.

[10] Mathworks Documentation for Kronecker product: `https://www.mathworks.com/help/matlab/ref/kron.html`.

[11] SciPy CSR Sparse Matrix Documentation `https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html`.

[12] Google Compute Engine `https://cloud.google.com/compute/`

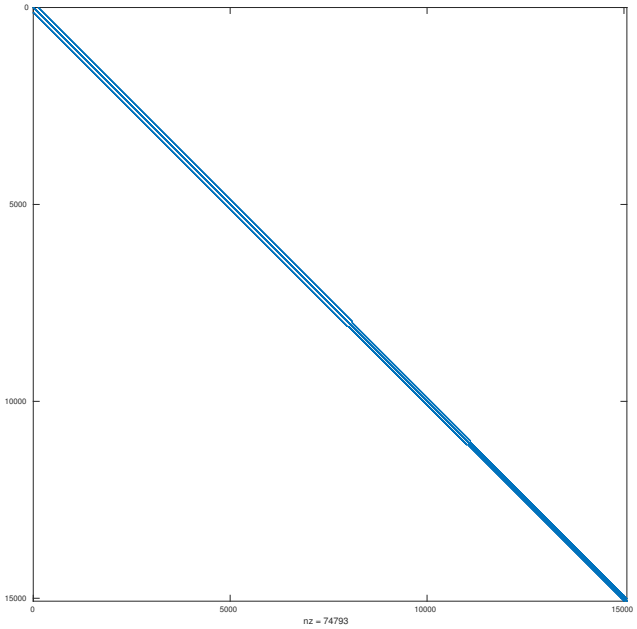
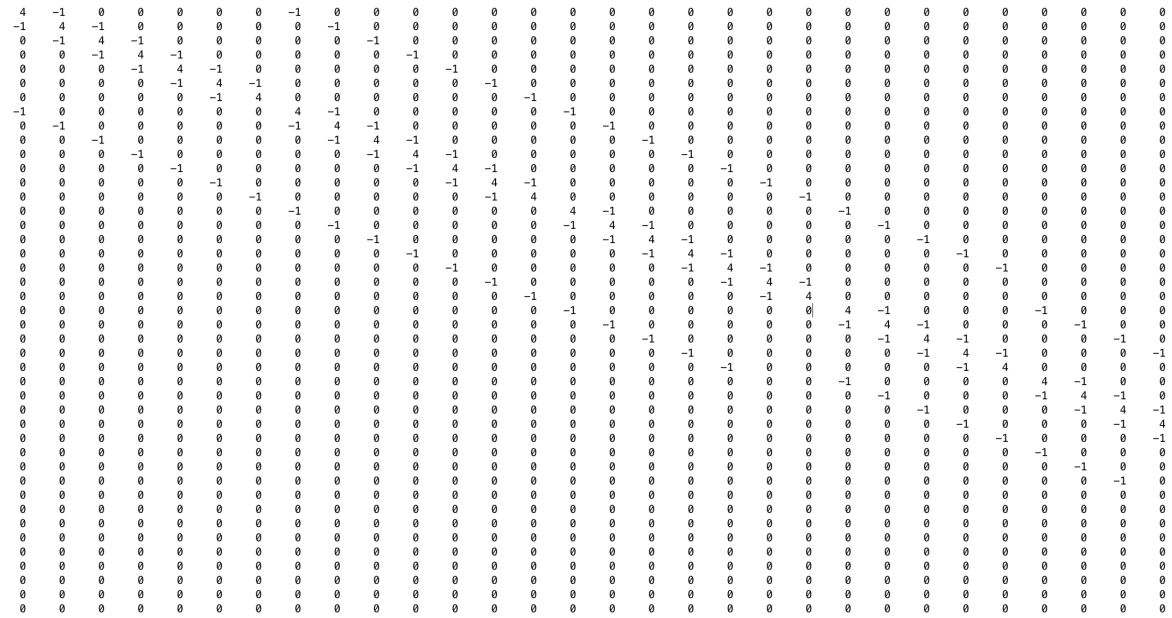
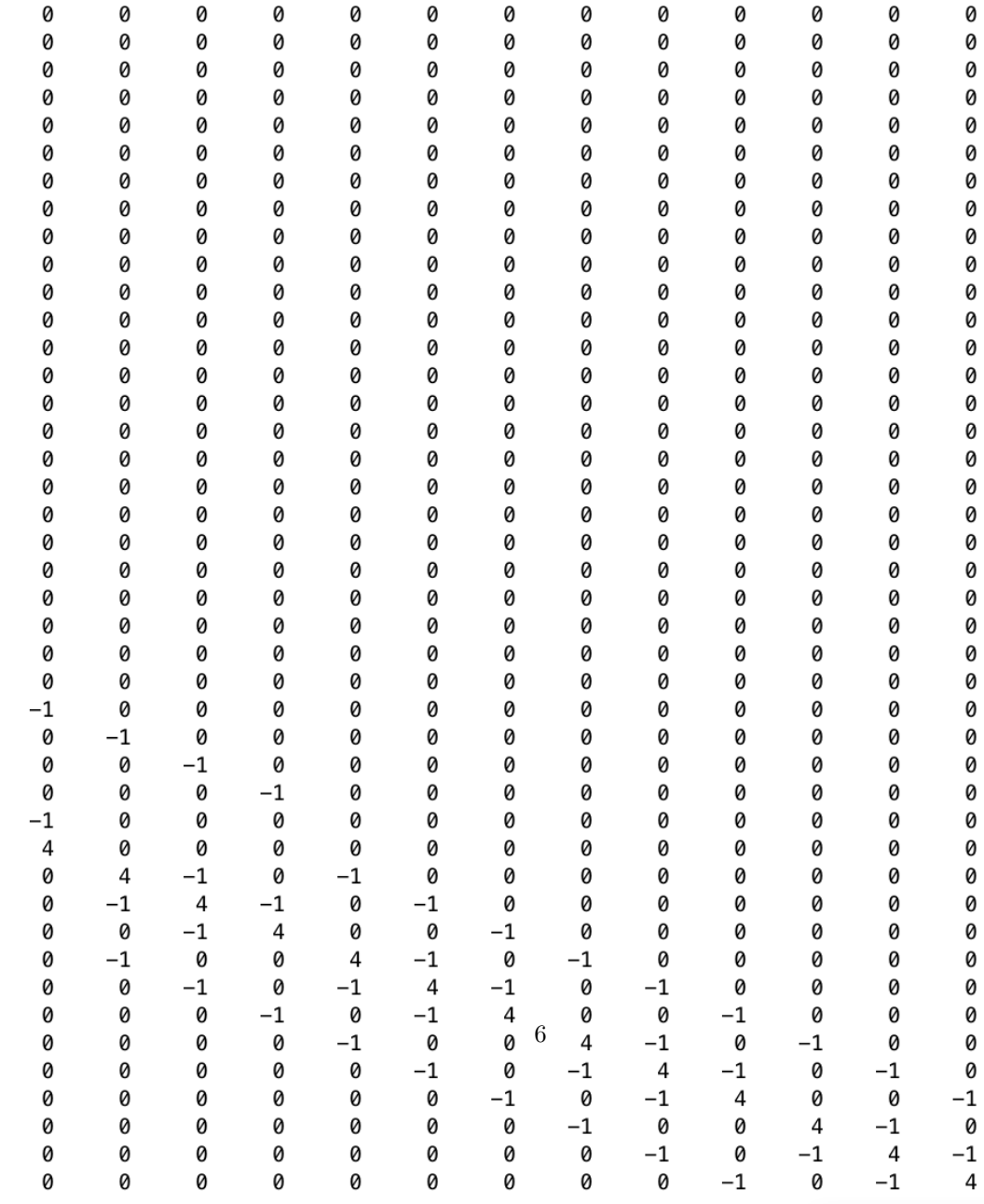


Figure 2: Operator Matrix Structure

Columns 1 through 30



Columns 31 through 43



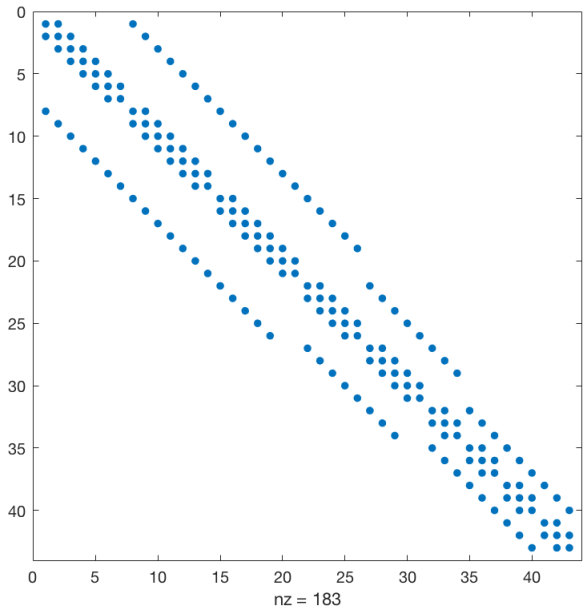


Figure 4: Operator Matrix Spy for 43 X 43

Finally, the right hand side solution vector is constructed before solving the linear system with NumPy's linear algebra module for solving the right-hand side of eq. (8). This is performed by evaluating each $f(x, y)$ at each of the points in the domain, embedding the result in the resulting array and applying the product with $(\Delta h)^2$. Subsequently the linear system is solved.

In addition to solving the system, the generic solver throws out all x and y coordinates that are not in the domain. This part of the algorithm is needed in order to properly visualize all solutions within the constraints of the surface's domain.

Python Code

0.0.1 Part 1a

```
import numpy as np
import scipy.signal as signal

def rhs(x, y):
    # Element-wise multiplication (9b)
    return np.multiply(x, (x - y)**3)

def matprint(mat, fmt="g"):
    col_maxes = [max([len(("{:"+fmt+"}").format(x)) for x in col]) for col in mat.T]
    for x in mat:
        for i, y in enumerate(x):
            print(("{:"+str(col_maxes[i])+fmt+"}").format(y), end="  ")
        print("")

def problem_one_solver(rightHandSide, *args, **kwargs):

    differential_operator = np.dot(4, np.eye(7))
    differential_operator[0,1] = -1
    differential_operator[0,4] = -1
    differential_operator[1,2] = -1
```

```
    differential_operator[1,5] = -1
    differential_operator[2,3] = -1
    differential_operator[4,5] = -1
    differential_operator[4,6] = -1
    differential_operator[1,0] = -1
    differential_operator[2,1] = -1
    differential_operator[3,2] = -1
    differential_operator[5,4] = -1
    differential_operator[6,4] = -1
    # X and Y Coordinates for evaluating right hand side
    X = np.concatenate([[1],[2],[3],[4],[1],[2],[1]], axis = 0)
    Y = np.concatenate([[1],[1],[1],[1],[2],[2],[3]], axis = 0)
    # Actually evaluate the rhs at points
    rhs_return = rightHandSide(X, Y)

    # Solve for solution vector
    solution_vector = np.linalg.solve(differential_operator, rhs_return)

    # define the mesh grid for embedding solution and for visualization
    X, Y = np.meshgrid(np.arange(0,float(6)), np.arange(0,float(5)))

    U = np.vstack([[0,0,0,0,0,0],
        [0, solution_vector[0],
        solution_vector[1],
        solution_vector[2],
        solution_vector[3], 0],
        [0,solution_vector[4],
         solution_vector[5],0,0,0],
        [0,solution_vector[6],0,0,0,0],
        [0,0,0,0,0,0]])

    # Remove x and y coordinates that are not within the domain or boundary
    arr_temp = np.vstack([[1,1,1],[1,0,1],[1,1,1]])
    convolve2d_temp = np.logical_not(signal.convolve2d(U, arr_temp, mode='same'))
    zero_indices = np.where(convolve2d_temp)

    X[zero_indices] = np.nan
    Y[zero_indices] = np.nan

    return X, Y, U

if __name__ == "__main__":

    X, Y, Z = problem_one_solver(rhs)
```

0.0.2 Part 1b

```
import numpy as np
import scipy.signal as signal

def rhs(x, y):
    # Element-wise multiplication
    return np.multiply(x, (x - y)**3)

def matprint(mat, fmt="g"):
    col_maxes = [max([len(("{:"+fmt+"}").format(x)) for x in col]) for col in mat.T]
```

```
    for x in mat:
        for i, y in enumerate(x):
            print(("{:"+str(col_maxes[i])+fmt+"}").format(y), end="  ")
        print("")

def problem_one_part_two_solver(rhs_func, *args, **kwargs):

    # Differential operator
    differential_operator = np.dot(4, np.eye(43))
    row_indices = np.concatenate([[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]], axis = 0)

    column_indices = np.concatenate([[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]], axis = 0)


    for k in range(0, 43):
        differential_operator[row_indices[k], column_indices[k]]= -1

    arbitrary_domain = np.vstack([
        [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]
    ])
#     # Retrieve x and y points that will be evaluated
    y_interior, x_interior = np.nonzero(arbitrary_domain)
    x_interior = np.dot(0.5, x_interior)
    y_interior = np.dot(0.5, y_interior)
#   # Evaluate right hand side at interior points
    return_vector = np.dot(0.5 ** 2, rhs_func(x_interior, y_interior))
#   # Solve the solution vector
    solution_vector = np.linalg.solve(differential_operator, return_vector)
#   # X and Y coordinates
```

```
    X, Y = np.meshgrid(np.arange(0, 4.5, 0.5), np.arange(0, 5.5, 0.5))

    seven_by_one = np.zeros((7, 1))
    one_by_eleven = np.zeros((1, 11))

    U = np.concatenate((seven_by_one, arbitrary_domain, seven_by_one), axis  = 1)
    U = np.concatenate((one_by_eleven, U, one_by_eleven), axis = 0).T

    (i, j) = (U != 0).nonzero()
    for solution in np.arange(0, np.size(solution_vector)).reshape(-1):
      U[i, j] = solution_vector[i]

    # spy for debugging
    fig = plt.figure(figsize=(6, 6))
    ax = fig.add_subplot(111)
    ax.spy(U);

   # Remove x and y coordinates that are not within the arbitrary_domain or boundary
    arr_temp = np.vstack([[1,1,1],[1,0,1],[1,1,1]])
    convolve2d_temp = np.logical_not(signal.convolve2d(U, arr_temp, mode='same'))
    zero_indices = np.where(convolve2d_temp)

    X[zero_indices] = np.nan
    Y[zero_indices] = np.nan

    return X, Y, U

if __name__ == "__main__":
    X, Y, Z = problem_one_part_two_solver(rhs)
```

0.0.3 Generic Solver

```
import numpy as np
import scipy.signal as signal
import scipy
import matplotlib.pyplot as plt

def rhs(x, y):
    # Element-wise multiplication
    return np.multiply(x, (x - y)**3)

def matprint(mat, fmt="g"):
    col_maxes = [max([len(("{:"+fmt+"}").format(x)) for x in col]) for col in mat.T]
    for x in mat:
        for i, y in enumerate(x):
            print(("{:"+str(col_maxes[i])+fmt+"}").format(y), end="  ")
        print("")

def matrix_generator(inputStepSize, rhsEquation, domainMatrix, *args,**kwargs):

    # 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.

    # Stride length
    steps = round(1 / inputStepSize)
```

```
# Expand the domain using the kronecker product
refinedDomain = np.kron(domainMatrix, np.ones(steps))
leftSide = np.kron(refinedDomain[:, 0], np.ones((1, steps - 1)))
leftBoundary = np.zeros((refinedDomain.shape[0], 1))
rightBoundary = np.copy(leftBoundary)
debugging = [(np.shape(arr)) for arr in (leftBoundary, leftSide, refinedDomain, rightBoundary)]
refinedDomain = np.hstack((leftBoundary, refinedDomain, rightBoundary))


# Fix y axis missing expanded values and boundaries
belowboundary = np.kron(refinedDomain[0, ], np.ones( (steps - 1, 1)))
topBoundary = np.zeros( (1, belowboundary.shape[1]))
bottomBoundary = np.copy(topBoundary)
refinedDomain = np.concatenate((bottomBoundary, belowboundary,
                                refinedDomain, topBoundary), axis = 0)

# plot the pattern of the domain to ensure it is correct -
#   used for debugging
fig = plt.figure(figsize=(6, 6))
ax = fig.add_subplot(111)
ax.spy(refinedDomain);

x_interior, y_interior = np.where(refinedDomain != 0)

# Scale x and y points based on step size
x_interior = inputStepSize * x_interior
y_interior = inputStepSize * y_interior

operatorMatrix = 4 * scipy.sparse.eye(x_interior.size).toarray()

#Iterate through each neighborhood and assign -1 where necessary
for i in np.arange(0, x_interior.size).reshape(-1):
  current_x = np.array(x_interior[i])
  current_y = np.array(y_interior[i])

  currentPoint = np.hstack((current_x, current_y))
  above = np.hstack((currentPoint[0], currentPoint[1] + inputStepSize))
  below = np.hstack((currentPoint[0], currentPoint[1] - inputStepSize))
  left = np.hstack((currentPoint[0] - inputStepSize, currentPoint[1]))
  right = np.hstack((currentPoint[0] + inputStepSize, currentPoint[1]))

  neighbors = np.vstack((above, below, left, right))

  for j in np.arange(1, neighbors.size / 2).reshape(-1):
    x_indices = np.where(x_interior == np.vstack((neighbors[int(j), 0],
                                                  neighbors[int(j), 1])))[0][0])

    index = np.where(
      y_interior[x_indices] == np.vstack((neighbors[int(j), 0],
                                          neighbors[int(j), 1])))[1][0])
    operatorMatrix[i, index] = -1

# Create right hand side vector
rhsVector = np.dot(inputStepSize ** 2, rhsEquation(x_interior, y_interior))

# Solve the system
```

```
    solutionMatrix = np.linalg.solve(operatorMatrix, rhsVector)

    # Coordinates plotted for visualization
    X,Y = np.meshgrid(np.arange(0, np.size(refinedDomain, axis = 1) + 1,
                                inputStepSize),
                      np.arange(0, np.size(refinedDomain, axis = 0) + 1, inputStepSize))

    Z = np.copy(refinedDomain)
    (i, j) = (Z != 0).nonzero()
    for solution in np.arange(0, np.size(solutionMatrix)).reshape(-1):
      Z[i, j] = solutionMatrix[i]

    # Remove x and y coordinates that are not within the domain or boundary
    arr_temp = np.vstack([[1,1,1],[1,0,1],[1,1,1]])
    convolve2d_temp = np.logical_not(signal.convolve2d(Z, arr_temp, mode='same'))
    zeroIndices = np.where(convolve2d_temp)
    X[zeroIndices] = 0
    Y[zeroIndices] = 0

    return X, Y, Z


if __name__ == "__main__":
  domain = np.vstack([[1,1,1,1],[1,1,0,0], [1,0,0,0]])
  # example usage
  matrix_generator(1, rhs, domain)
```

Conclusion

$$f(x,y) = xe^{-x^2-y^2} \tag{9a}$$

$$f(x,y) = x\,(x-y)^3 \tag{9b}$$

The generic solver, above, was used to solve the Gaussian and cubic functions, eq. (9a) and eq. (9b), respectively. Due to local OS limitations, conflated with prevarication of heap memory by resource intensive applications such as the Chromium driver, we were limited to solving the solution with a solution of $\Delta h = \frac{1}{32}$ for eq. (9a) and $\Delta h = \frac{1}{64}$ for eq. (9b), respectively. The solution plots were compared with the solutions found by Matlab's built-in PDE toolbox[13]. These plots are shown in figs. 7–10 (below). The plots in fig. 8 display the evolving solution for step sizes for $1, 2, 4, 8$ and the precipitous convergence to the solution.

The generic solver is able to handle any arbitrary domain in the current implementation. This flexibility makes it ideal for solving more complex problems over myriad domains. However, it should be noted that the solutions for the given step sizes are markedly worst than the PDE-Toolbox's solver, and due to it's iterative implementation is extremely expensive and inefficient. In fact the current algorithm's implementation is at worst quadratic running time, $\mathcal{O}(n^2)$ where n is the number of interior points within the $x-$axis domain.

To account for this, a more complex functional data structure, such as a ConcTree [14]. This would allow for the domain into several subsets that can be evaluated upon using separate processes and then joined upon convergence. Another variant is to utilize Voronoi partitions in order to sub-divide the domain into more discrete points and then seek neighbors using k-nearest neighbors. These are of course optimizations on the representation of the domain.

Finally, another variant, specific to kinetic motion, would be an implementation of a the parallel Barnes-Hut algorithm for N-body simulation. N-body simulation is a simulation of a system of N

[13] Mathworks PDE Toolbox `https://www.mathworks.com/products/pde.html`

[14] Conc-Trees for Functional and Parallel Programming `http://aleksandar-prokopec.com/resources/docs/lcpc-conc-trees.pdf`

particles that interact with physical forces, such as gravity or electrostatic force. Given initial positions and velocities of all the particles (or bodies), the N-body simulation computes the new positions and velocities of the particles as the time progresses. It does so by dividing time into discrete short intervals, and computing the positions of the particles after each interval.

To take advantage of this observation, the Barnes-Hut algorithm relies on a quadtree – a data structure that divides the space into cells, and answers queries such as 'What is the total mass and the center of mass of all the particles in this cell?'

An iteration of the Barnes-Hut algorithm is composed of the following steps:

- **1** Construct the quadtree for the current arrangement of the bodies.
- **2** Determine the boundaries, i.e. the square into which all bodies fit.
- **2** Construct a quadtree that covers the boundaries and contains all the bodies.

Then, (i) Update the bodies – for each body: (ii) Update the body position according to its current velocity. (iii) Using the quadtree, compute the net force on the body by adding the individual forces from all the other bodies. (iv) Update the velocity according to the net force on that body.

It turns out that, for most spatial distribution of bodies, the expected number of cells that contribute to the net force on a body is $log(n)$. Hence the overall complexity of the Barnes-Hut algorithm is $\mathcal{O}(n \log n)$!

In conclusion, this exercise affords many new avenues to solve this problem; this brief synopsis provides a solid foundation in order to further explore computational complexity within higher dimensional spaces and non-linear problems.

Solutions

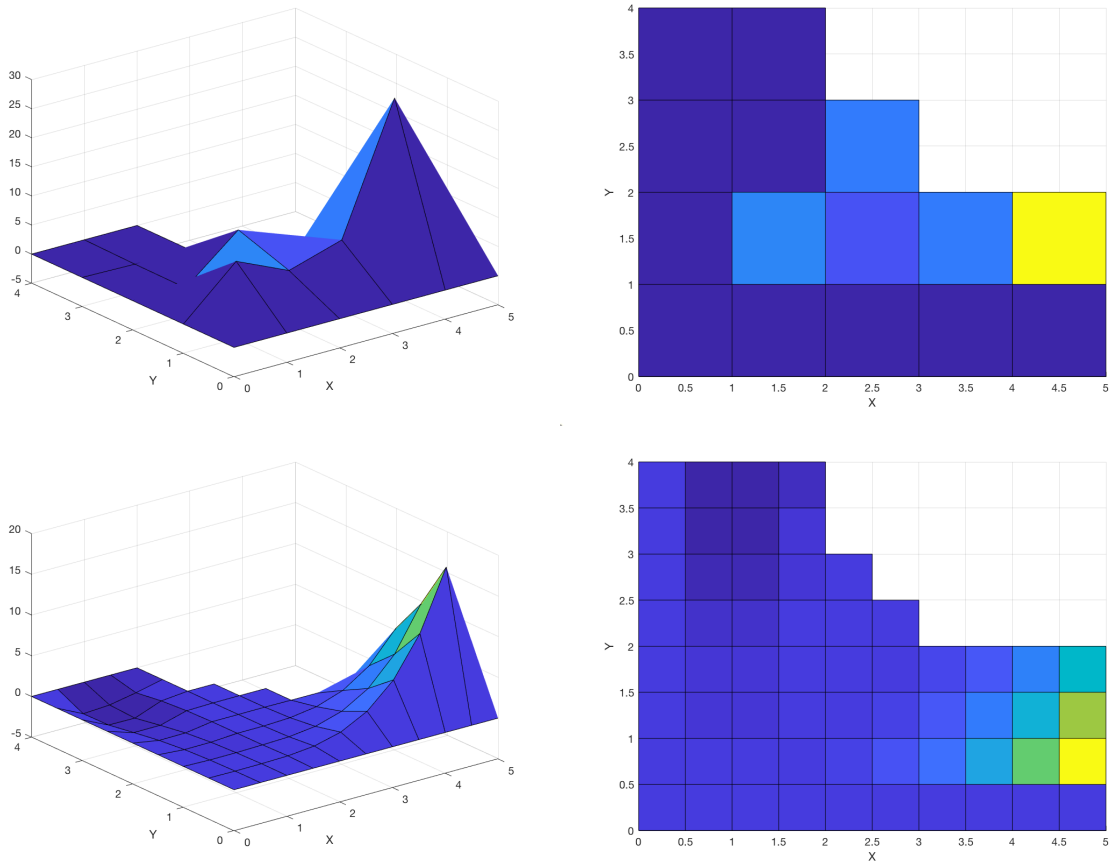


Figure 5: Solutions to Part 1a with step size of 1 (top) and 1b with step size of 1/2 (bottom), respectively

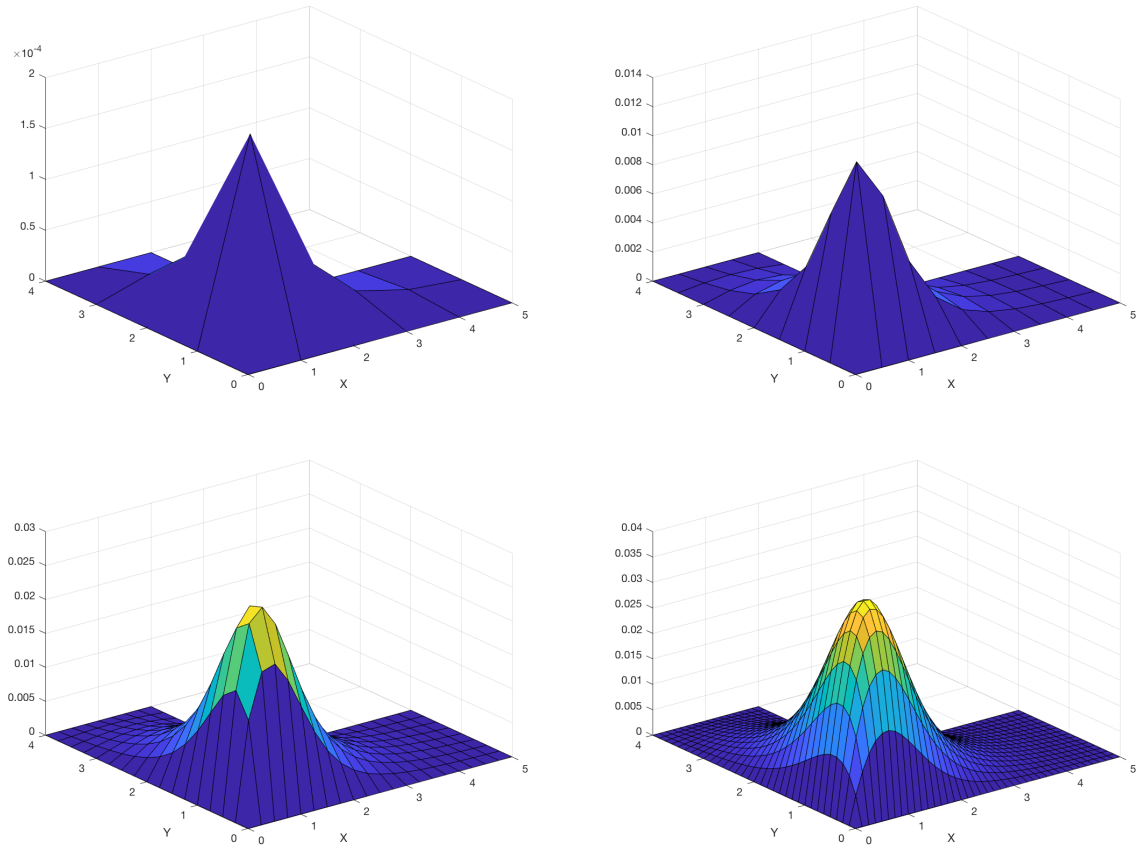
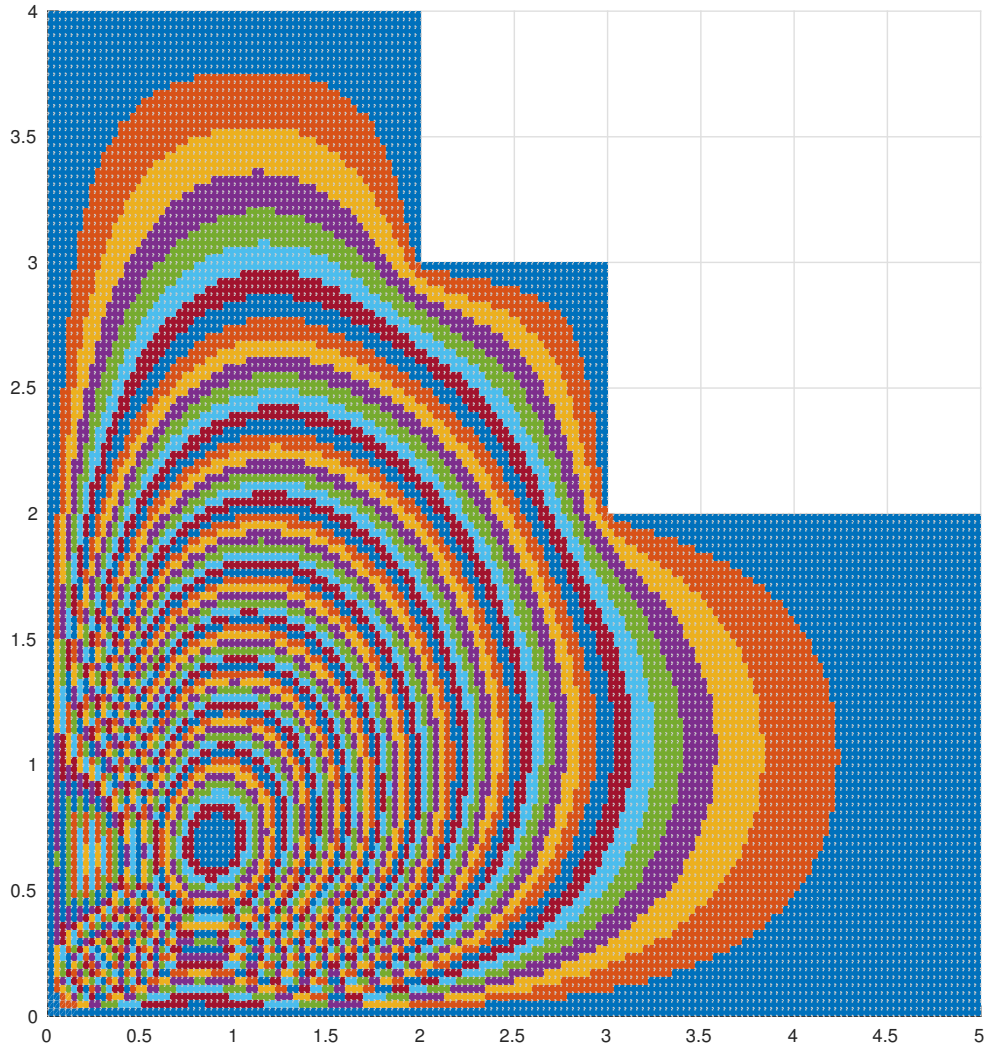
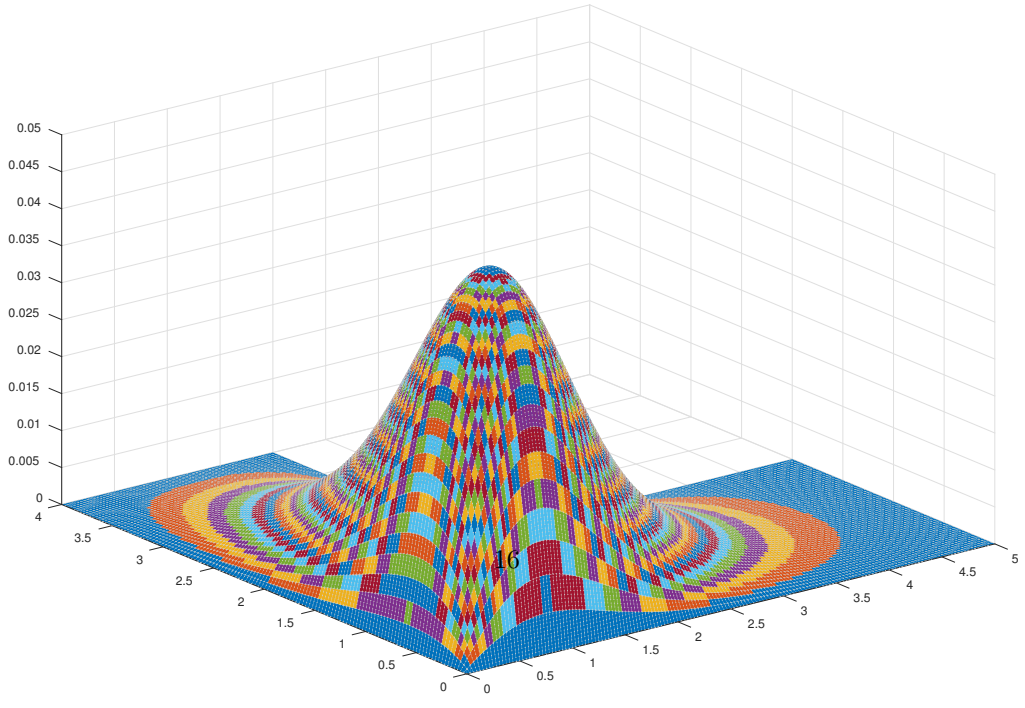


Figure 6: Solutions to eq. (9a) with $\Delta h = 1, 2, 4, 8$ and 32 (below). Color palette adjusted in order to highlight contrast in value approximations at each cell in the domain


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

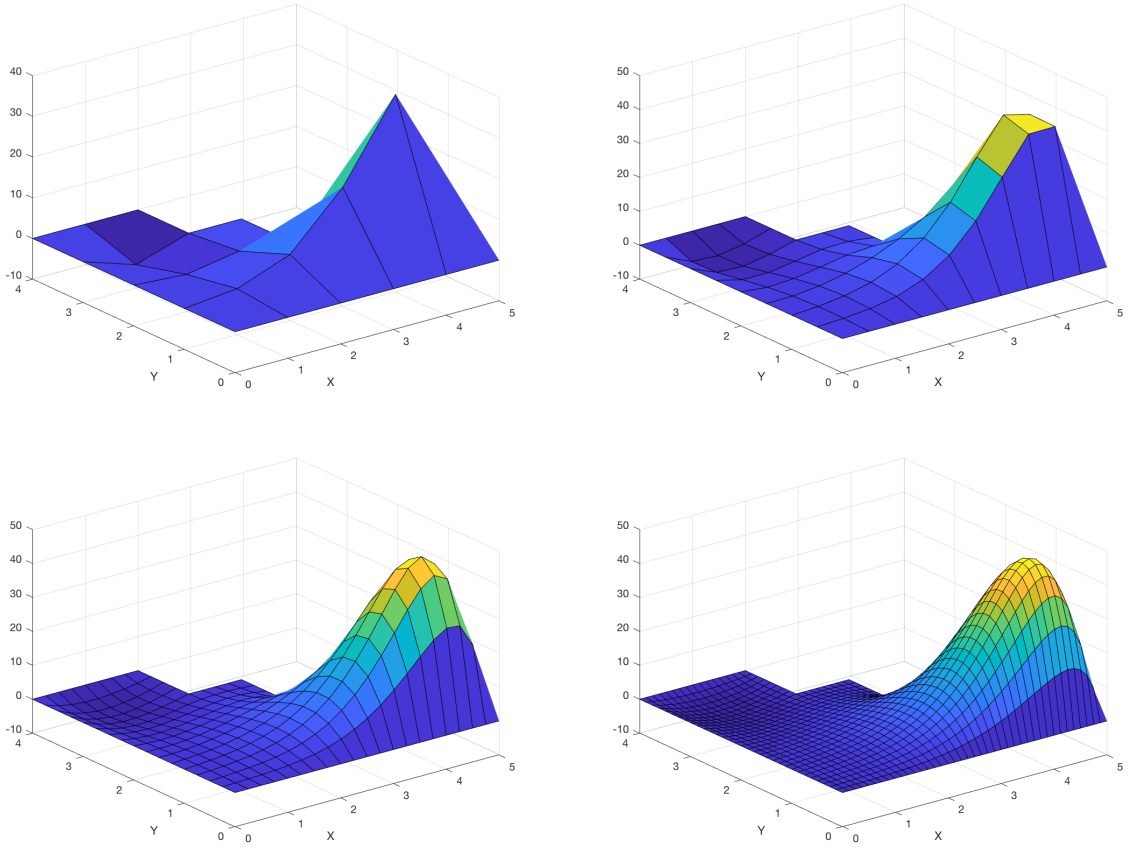


Figure 8: Solutions to eq. (9b) with $\Delta h = 1, 2, 4, 8$

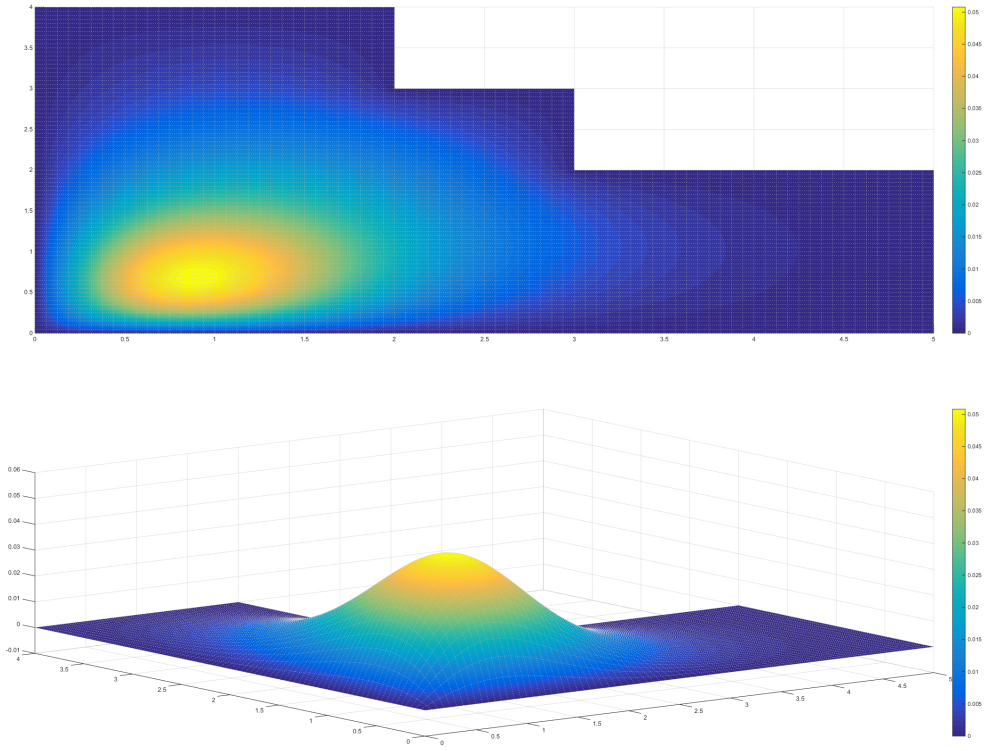


Figure 9: PDE Toolbox Solutions to Equation (9a)

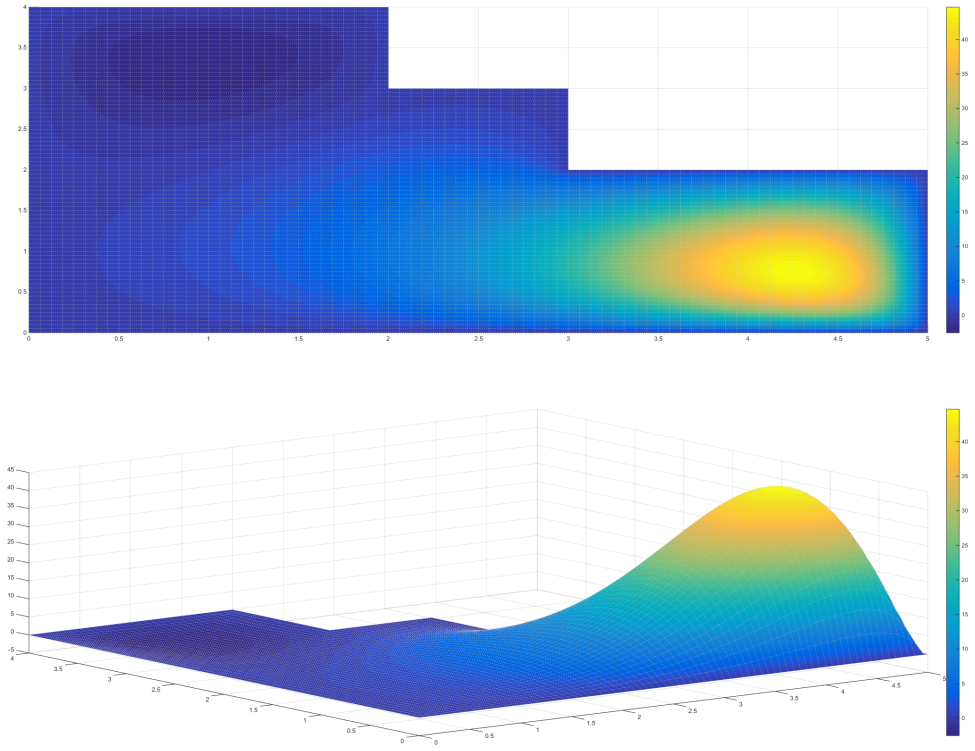


Figure 10: PDE Toolbox Solutions to Equation (9b)