Path Planning and Maze-Solving using Laplace Equation

Thomas Chen

New York University Abu Dhabi

Numerical Methods

Professor Francesco Paparella

May 10, 2024

Abstract

This paper will discuss a piece of software that utilizes the Laplace Equation to find an efficient path to traverse the maze. This software have been tested with mazes of various dimensions and complexities, demonstrating its ability to consistently find the correct path. This path-planning capability is crucial not only in finding the solutions of the maze, but is also tremendously important in the field of robotic motion planning. By integrating theoretical numerical methods with practical computational algorithms, the software provides valuable insights into complex navigational challenges, which are critical in the development of autonomous robotic systems.[1]

Contents

1 Introduction

Path planning challenges, particularly those involving complex environments(eg. mazes), have been a pivotal research direction in the field of robotics and computational research. The ability to solve mazes consistently sheds light on solving the more complex path-planning problems.

This paper focuses on solving basic two-dimensional mazes using the Laplace Equation - a second-order partial differential equation widely used in various fields of science and engineering for modeling the behavior of electric, gravitational and fluid potentials.

The software's core functionality is to derive the efficient(shortest) path through mazes. The maze-solving algorithm implemented in this software is backed-up by rigorous mathematical calculations, which enables the software to tackle mazes of varying dimensions and complexities. It have been rigorously tested across multiple scenarios and edge cases to demonstrate its reliability and efficiency.

This paper is structured as follows: following the introduction, we will discuss the numerical basis on which the software is built and why it works. We then detail the architecture of our software and the specific implementations of the numerical methods used. Subsequent sections present various test cases and their results. Finally, we conclude with the implications of our findings for the development of autonomous systems.

2 Numerical Basis

A harmonic function on a domain $\Omega \subset \mathbb{R}^n$ is a function which satisfies Laplace's equation:

$$\nabla^2 \phi = 0$$

where Δ represents the Laplacian, which in two dimensions involves second derivatives with respect to both spatial dimensions(x and y), i.e.,

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

In the context of a maze, $\phi(x, y)$ can be thought of as a potential function. By solving the Laplace equation over the maze matrix with appropriate boundary conditions, we can generate a potential field across the maze.

2.1 Boundary Condition and Min-Maximum Principle

In our implementation, we will impose the boundary conditions on the specified starting and ending points. Specifically, we will set the potential at the starting point(S) to be 0 and the potential at the ending point(E) to be 1.

Now, according to the textbook [2], we have the definition of Maximum principle:

Theorem 3.14 (Maximum principle for harmonic functions). *Let Ω be bounded open in $\mathbb{R}^n$. Assume $u \in C^2(\omega) \cap C(\bar{\Omega})$ is harmonic in Ω.*

(i) (**Weak maximum principle**) *We have that $\max_{\bar{\Omega}} u = \max_{\partial\Omega} u$.*

(ii) (**Strong maximum principle**) *If, in addition, Ω is connected and there exists a point $x_0 \in \Omega$ such that $u(x_0) = \max_{\bar{\Omega}} u$, then $u(x) \equiv u(x_0)$ for all $x \in \Omega$.*

The Weak maximum principle states that the maximum value that the function u achieves in the domain Ω lies on the boundary $\partial\Omega$. And the Strong maximum principle reinforces by stating that not only does the maximum value occur on the boundary, but if the same maximum value is also found at any point inside Ω, then u must be constant over the entire domain.

Since if u is harmonic, then $-u$ is also harmonic, the Minimum principle also hold in the similar way.

These principles ensure that the minimum and maximum value in our maze potential field occurs at the starting and ending point S and E, where we set the potential $\phi(S) = 0$ and $\phi(E) = 1$.

2.2 Solving the Laplace Equation

To generate a potential field for a maze, we first define a domain Ω representing the maze where the passageways are bound to changes in potential and the walls are boundaries where the potential remains undefined.

We then discretize Ω into a grid where each point (i, j) on the grid corresponds to a point in the maze and let h represent the distance between each grid point, assumed uniform in both the x and y direction for simplicity.

Recall from Section 2, the Laplace equation in two dimensions is given by:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0 \tag{1}$$

We approximate the second derivatives in the Laplace equation with finite difference approximations. For a function $\phi(x, y)$ evaluated at the grid point (i, j), the second derivatives are approximated as follows:

$$\frac{\partial^2 \phi}{\partial x^2} \approx \frac{\phi(i+1, j) - 2\phi(i, j) + \phi(i-1, j)}{h^2} \tag{2}$$

$$\frac{\partial^2 \phi}{\partial y^2} \approx \frac{\phi(i, j+1) - 2\phi(i, j) + \phi(i, j-1)}{h^2} \tag{3}$$

Substituting (2), (3) into Laplace Equation (1), we get:

$$\frac{\phi(i+1, j) + \phi(i-1, j) + \phi(i, j+1) + \phi(i, j-1) - 4\phi(i, j)}{h^2} = 0 \tag{4}$$

Simplifying (4), we get:

$$\phi(i, j) = \frac{\phi(i+1, j) + \phi(i-1, j) + \phi(i, j+1) + \phi(i, j-1)}{4} \tag{5}$$

This equation means that the potential at any point(i, j) is the average of the potentials at the neighboring points.

Now to solve the Laplace Equation and generate a potential field, we would need to iteratively solve the discrete equations at every point until the solution converges to a steady state. In our implementation, we will be using SOR(Successive Over Relaxation) method with $\omega = 1.5$.

The SOR method to solve $Ax = b$ is defined as follows[3]:

$$\begin{aligned} x_0 &= \text{initial vector} \\ x_{k+1} &= (\omega L + D)^{-1}[(1-\omega)D_{x_k} - \omega U_{x_k}] + \omega(D + \omega L)^{-1}b \quad \textbf{for } k = 0, 1, 2, ... \end{aligned}$$

where L is the lower triangular part of A, D is the diagonal of A, U is the upper triangular part of A and ω is the relaxation parameter.

For the Laplace equation on a grid, the matrix A largely represents the connections between each point and its neighbors; therefore, L and U are sparse and only contain elements directly below or above the diagonal, respectively.

Then, the term $(\omega L + D)$ becomes trivial to handle and the term $(1-\omega)D_{x_k} - \omega U_{x_k}$ simplify to directly involving only neighboring values.

Then, in our case, the SOR method simplifies to:

$$potential[i, j] = oldPotential[i, j] + \omega(newPotential[i, j] - oldPotential[i, j])$$

where $newPotential[i, j]$ is equal to (5).

2.3 Path-finding using Potential Field

By the Gradient Theorem[4], which states that moving in the direction of the gradient at any point will lead to the quickest increase in potential, which in this context translates to the most direct route towards the region of highest potential (the end of the maze).

We have found the potential field, which is the solution to the Laplace equation imposed on the domain Ω(maze), and we know from Section 2.1 that the minimum and maximum value of our potential field happens at the starting and ending point of our maze, respectively. Subsequently, the potential increases from the start to the end of the maze.

To navigate the maze from start to end, we can follow the steepest Gradient Ascend. At each step, the direction of movement is determined by the gradient of potential field $\nabla\phi$. However, since ϕ is calculated at discrete points, we approximate by moving to the neighboring grid point with the highest potential value.

3 Implementation

The implementation of the maze-solving problem is done in Python using the libraries $pylab$ and $matplotlib$.

There are three main functions that implements the main functionality of the software.

- `laplaceTransform`

- `mazeSolver`
- `plot_maze`

3.1 laplaceTransform

```
def laplaceTransform(maze, start, end, omega=1.5, iterations = 1000,tol = 1
    e-5):
    potential = full(maze.shape, nan)
    potential[start] = 0
    potential[end] = 1

    n = len(maze)
    m = len(maze[0])

    for i in range(n):
        for j in range(m):
            if maze[i, j] == 0 and (i, j) not in [start, end]:
                potential[i, j] = (potential[start] + potential[end]) / 2


    for _ in range(iterations): #Iteration limit
        old_potential = potential.copy()
        for i in range(n):
            for j in range(m):
                if maze[i,j] == 0 and (i, j) not in [start, end]:
                    neighbors = [
                        potential[i-1, j] if i > 0 else nan,
                        potential[i+1, j] if i < n - 1 else nan,
                        potential[i, j-1] if j > 0 else nan,
                        potential[i, j+1] if j < m - 1 else nan
                    ]
                    valid_neighbors = [n for n in neighbors if not isnan(n)
    ]
                    if valid_neighbors:
                        new_value = mean(valid_neighbors)
                        potential[i, j] = old_potential[i, j] + omega * (
    new_value - old_potential[i, j])

        if allclose(potential, old_potential, atol = tol):
            break

    return potential
```

The `laplaceTransform` function starts by initializing a potential field with the same dimensions as the maze it is trying to solve and filling it with undefined values except at the specified starting and ending points, at which the values are set to 0 and 1, respectively.

Then, it initializes all cells in the potential field corresponding to passage ways of the maze to an initial value. In this case, I have chosen to initialize it to be the average of the potential values at the starting and the ending points of the maze, which is 0.5.

It then iterates using the SOR method with $\omega = 1.5$, maximum iteration limit set to 1000, and a convergence tolerance to stop the iteration when the difference between iterations is less than $1e^{-5}$.

At the end, it returns the resulting potential field ϕ.

3.2 mazeSolver

```
    def mazeSolver(potential_field, start, end):
    path = [start]
    current = start

    n = len(potential_field)
    m = len(potential_field[0])

    while current != end:
        i, j = current
        neighbors = [
            (i-1, j), (i+1, j),
            (i, j-1), (i, j+1)
        ]

        neighbors = [p for p in neighbors if 0 <= p[0] < n
                     and 0 <= p[1] < m
                     and not isnan(potential_field[p])]

        current = max(neighbors, key = lambda p: potential_field[p])
        if current in path:
            print("Unsolvable Maze!")
            return None
        path.append(current)

    return path
```

This function takes the computed potential field and starts at the starting point of the maze(array `path` is initialized to contain the starting point).

It then checks the neighboring points of the potential field and moves to the neighbor with the highest potential value. In the meantime, it records the point that it have just advanced to in the array `path`.

If the algorithm ever encounters a situation where it must revisit an already visited point to proceed, it is forced into a decision that goes against the principle of always moving towards higher potential. This backtracking indicates a fundamental problem with either the potential field's configuration or the maze's structure (such as unsolvable loops or isolated sections), leading to the reasonable conclusion that the maze is unsolvable under the current pathfinding strategy.

If the maze is solvable, then this function will iterate until it reaches the ending point and return a list of points that form the most efficient path from the starting to the ending point.

3.3 plot_maze

```
def plot_maze(maze,path,start,end):
    START_COLOR = 2
    END_COLOR = 3

    maze[start] = START_COLOR
    maze[end] = END_COLOR

    cmap = mcolors.ListedColormap(['white', 'black', 'red', 'green'])

```

```
    imshow(maze, cmap=cmap)
    if not path:
        print("Unsolvable Maze!")
        return
    path_y, path_x = zip(*path)
    scale = max(len(path_y), len(path_x))
    for i in range(1,len(path_y)+1):
           imshow(maze, cmap=cmap)
           plot(path_x[0:i], path_y[0:i], color='blue')
           pause(0.1*(1/scale))
```

This function plots the maze and the path(if the maze is solvable) continuously, showing each step of the path until the ending point. The color of the wall is set to be black, passage way to be white, starting point to be red, and the ending point to be green. The path is represented by a thin blue line.

4 Results

For the maze shown in Figure 1, we can see that the software is able to correctly find the solution path with 31 steps.

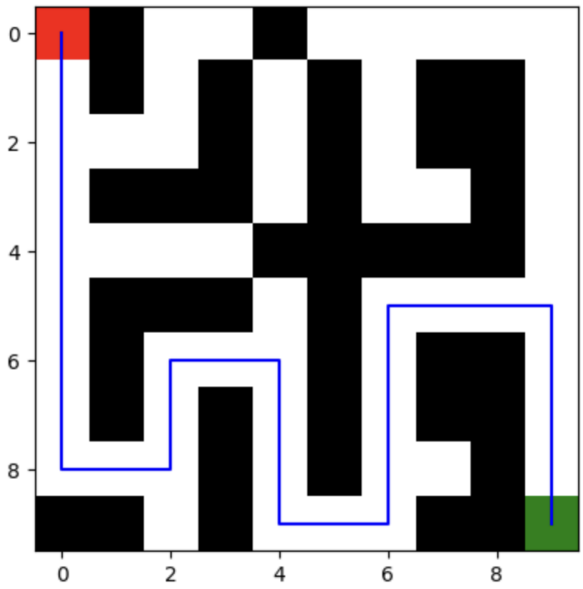

Figure 1

Now if we remove some walls from the same maze to create a shorter solution path while keeping the original path available, we can verify if the software is able to correctly identify the most efficient path.

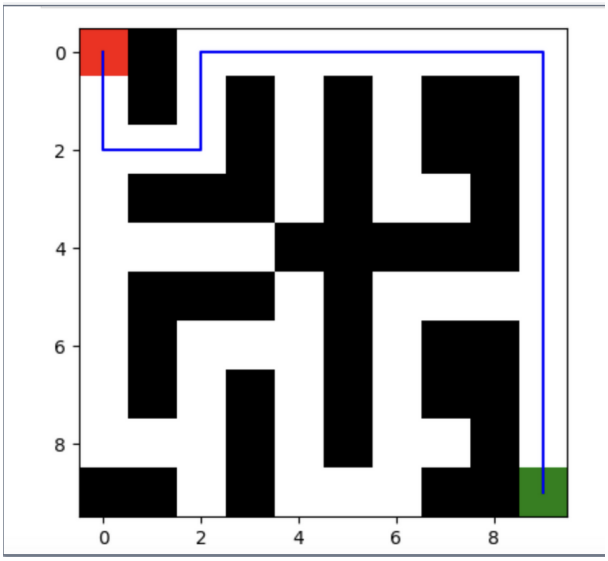

Figure 2

We can see from Figure 2 that the software has chosen a new path with 23 steps instead of the original path with 31 steps, effectively establishing its efficiency.

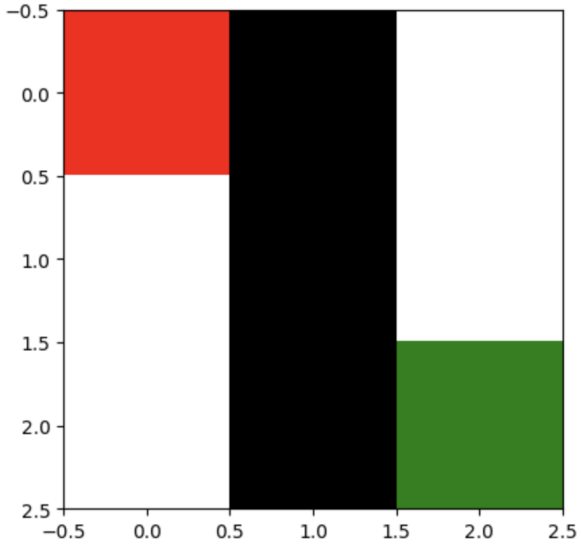

Figure 3

As shown in Figure 3, the wall completely blocks the path between the starting and ending points, and therefore should be no solution. The software is able to identify that there is solution as there is no blue line shown.

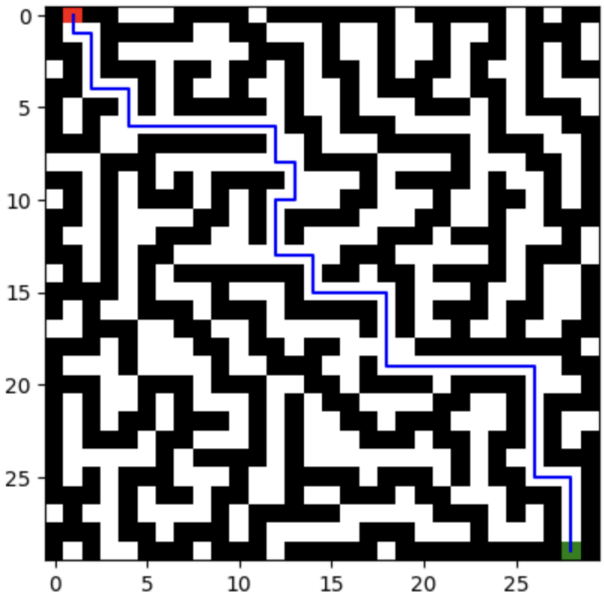

Figure 4

Figure 4 shows a more complex maze of dimension 30x30, and the software's solution.

5 Conclusions

This paper has demonstrated a robust approach to solving two-dimensional mazes using the Laplace Equation, showing the importance of numerical models in computational research and robotics. The algorithm presented in this paper may seem trivial and limited in its purpose; however, it serves as a gateway to tackling more complex path-planning challenges that are common in the development of autonomous systems.

When compared to traditional maze-solving algorithms like backtracking, it is often not so advantageous in terms of running speed, but its ability to accurately find the most efficient path is invaluable. In the real world, accuracy and efficiency can prevail over the slim advantages of running speed. Therefore, the implications of this software extend far beyond the confines of theoretical exercises and into the practical realms where real-world problems persist.

One of the most compelling applications of this work is in robotics, where autonomous machines are often required to navigate complex environments under uncertain conditions. The ability of this software to discern paths through mazes translates directly to the capability of robots to find optimal routes in unstructured or dynamically changing environments. Whether it is a rescue robot navigating through rubble in a disaster-stricken area or a service robot maneuvering through busy city streets, the principles demonstrated by the maze-solving algorithm can be adapted to enhance the robot's decision-making processes, thereby improving operational efficiency and safety.

References

[1] J.B.Burns C.I.Connolly and R.Weiss. Path Planning Using Laplace's Equation. 1990.

[2] *PDE Textbook Chapter 3 Laplace's Equation*.

[3] Timothy Sauer. *Numerical Analysis*. Pearson, third edition, 2019.

[4] Wikipedia contributors. Gradient theorem. 2024.