

Solving maze puzzles

Niels Agertoft, Rasmus L. T. Henriksen, Jonas B. Hubrechts.

Introduction

Path planning is important in many applications. An example of a path planning problem is maze puzzles. One way these can be solved is by using PDE's. By using some of the unique properties of the Laplacian problem, and the finite element method, it is possible to derive a route through a maze.

General idea

The general idea behind solving maze puzzles, is to have a maze with a defined start and finish edge (E_{start} and E_{finish}), as well as a defined set of interior and exterior boundaries. One then uses a Laplace PDE on the domain, as well as a Dirichlet boundary on the start and finish nodes respectively. With the start position having a defined value of 1 and the finish a defined value of 0. What will then be exploited is the function that fulfils the Laplace PDE has no interior local minima, which means it will be continuously descending towards the finish. Given a solution, it is then possible to trace a path through the maze from start to finish.

$$\Delta u = 0, (x, y) \in \Omega$$

$$u_n = 0, (x, y) \in \partial\Omega \setminus \{E_{start}, E_{finish}\}$$

$$u = 1, (x, y) \in E_{start} \wedge u = 0, (x, y) \in E_{finish}$$

Cutting algorithm

In order to solve the problem, the maze must be cut up along the walls in order to allow solutions like the one shown in *Figure 2*. To do this we must define new points at necessary nodes (degree-1) and redefine EToV. To allow for easier type-in of new mazes (or simply using the Amaze package to generate random mazes), we will also define beds from this function. The only inputs are the original nodes, edges (given as noedges x 2 array), original EToV and start/finish edges. To perform the necessary updates and adding the necessary nodes on the edges/boundary, for each tree (i.e. separate collection of edges, see *Figure 3*) we perform a walk always turning right when possible. Using the previous node, the current node, the next node in the walk and some geometry, we can determine whether or not and element needs to be redefined.



Figure 1: Example of edges in a maze

Deriving the route

Idea

From looking at the 3d plot (figure 2), it becomes obvious that to find a route, we can use a variation of a gradient descent. Since our calculated solution output is not a function, we can not use the exact derivatives. Instead, for each node visited, we check its neighbouring points function value, and move to the one that has the lowest.

Algorithm

Our algorithm does two checks, it checks the immediate neighbouring nodes to check for walls, and it checks the neighbouring nodes two steps away. If it detects a wall it will add a penalty, when searching for the minimal value two steps away. The reason we move two nodes in each iteration is to move between each "square" in the maze. We saw that only moving one node could lead to errors since the immediate neighbour nodes would sometimes guide the algorithm the wrong way.

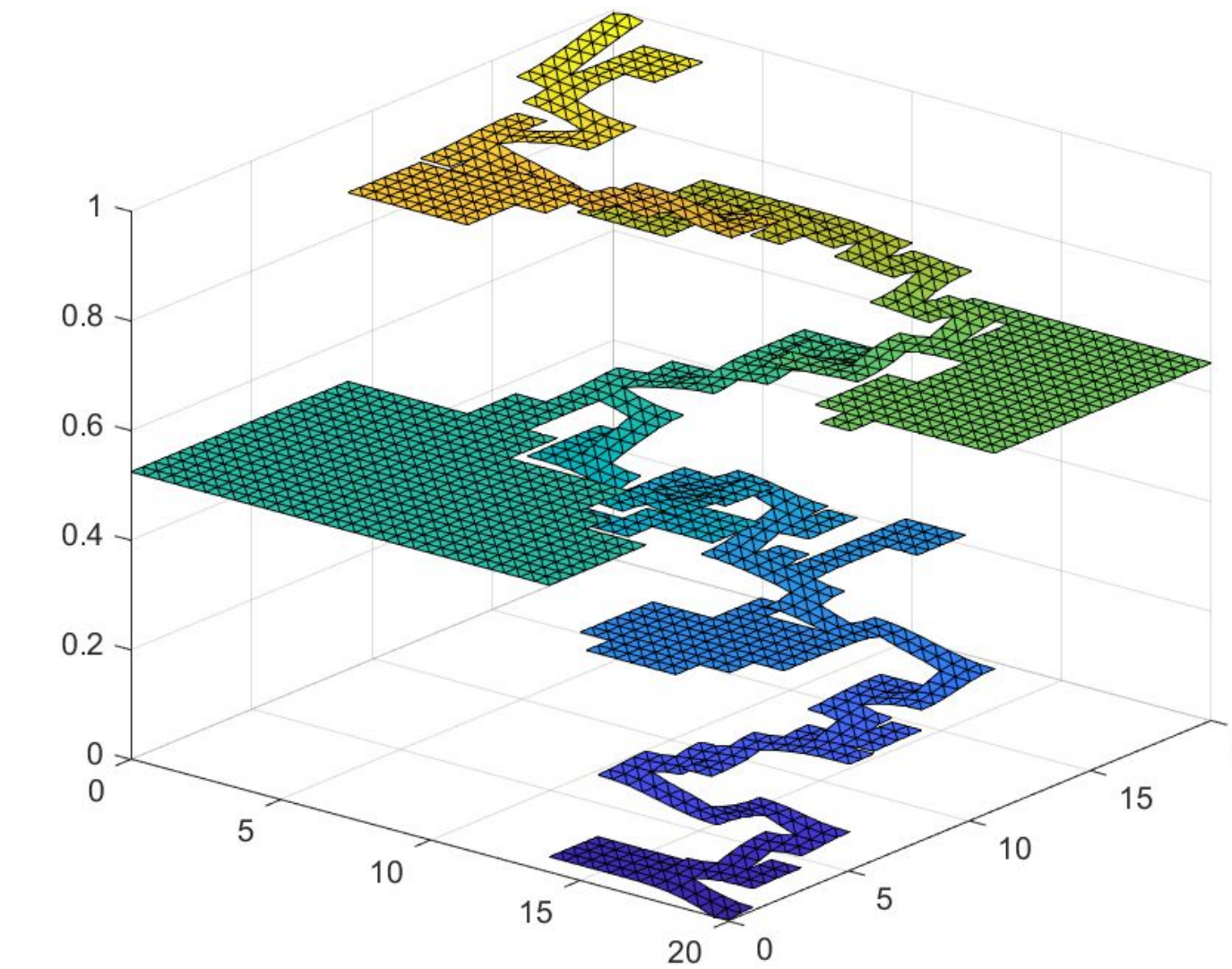


Figure 2: An example maze

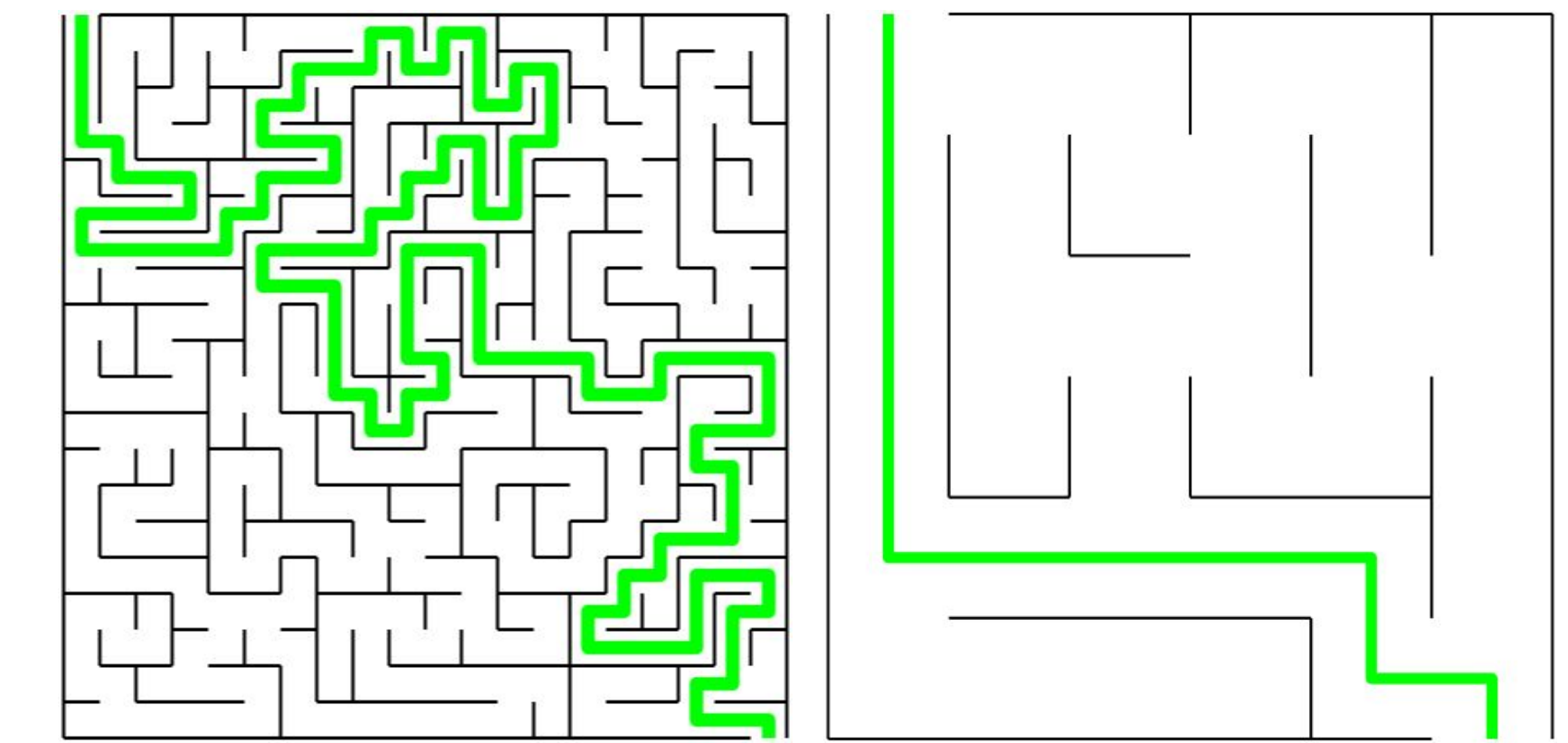


Figure 3: Multiple mazes with (fastest) solution

Multiple solutions, dead ends and where we went wrong

The algorithm solves the mazes by taking the locally steepest path through the network, this oftentimes results in the shortest route. When it then comes to the dead end situations that arise, the algorithm does not get trapped as it just follows the descending gradient towards the finish and the dead end paths will just flat out, as it can never reach the finish. We encountered an issue by having the entire finishing boundary set to 0, as this meant it could sometimes believe there were additional paths towards the finish, that only reached the left corner of the finish edge but not the "real" finish edge. This was solved in our code by outputting the new indices of the start and finish edges.

Scaling, climate and further perspectives

When looking at figure 4, we see that the time used by our solver scales linear with the degrees of freedom, while the gradientdescent scales as the root of the DoF. Hence if we wanted to optimize our code further, we would start by looking at the solver. Even though the solver is the worst at scaling, it is still able to solve large problems fast, as seen from the plot it only takes 0,1726 seconds to solve a maze of size 100x100 (60003 DoF). The speed of the program means that we see a low climate affect. By using some conservative estimates of the power consumption of our pc. From figure 5 we see that for the 100x100 maze a total of 0,00436 g. Of course since the CO2 emitted is proportional to time, it will also scale linearly with DoF. Since the next step would be to implement a solver for 3d mazes, the sizes of the problems would grow significantly. Hence we would see a lot more CO2 emitted, which is another reason to optimize our solver.

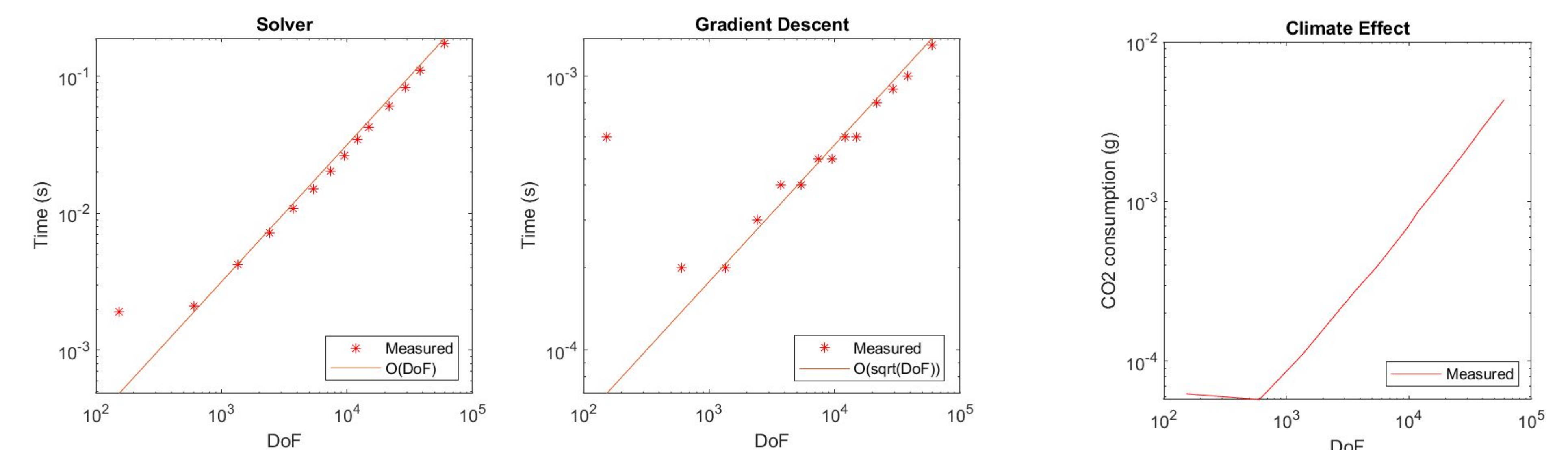


Figure 4: Time plots

Figure 5: Climate plots