Solving Eikonal Partial Differential Equations on Unstructured Triangular Grids

S Baskar, K Kulkarni

February 7, 2017

Abstract

This is not exactly a paper but rather a detailed explanation of how I working on the project. I am trying to give an overview of the tasks I performed every week. This could help me retrace any mistake that I did over the period. I hope I do not come over here to debug the results.(That's a blatant lie! I don't think I will remember what I did in the first week while working during the fourth week and that's the main reason for documenting it properly.)

1 Week 1

The first week I achieved the task of making a library in C++ in order to process the mesh information. I coded the problem trying to be as systematic as possible, by using many intricacies of Object-Oriented Programming. I made classes which are the building blocks of the OOP. The main reason for implementing classes is that it makes the code very readable, and during the further weeks my main code would be just using this pre-processor.

1.1 Requirements for the Code

Although there are various other libraries which share the same purpose to interpret a mesh, but this Finite Difference code has a special requirement which being that one needs to identify all the neighboring elements of a particular node and check for the cardinality condition for each of the element, which is the method for implementing the Fast Marching Algorithm [1].

1.2 Classes developed

Node: The node contains all the information relevant to a node. Figure 1 shows the precise depiction of a node interpreted in the current work.

In the figure(Fig. 1) the elements are named as ϵ_i where $i = 0, 1, \ldots 9$. The characteristic is passing through ϵ_9 and the rear end of the ray is passing through ϵ_4. Hence, while doing the calculations we must use the values from the nodes of ϵ_4 in order to satisfy the upward transmission of wave.

The following information has been chosen to define a 'Node'

- `float x, float y`: These are the co-ordinates of the nodes in a two dimensional domain.
- `int noOfNbgElements`: Variable to store the number of adjacent neighboring elements for the node.
- `Element** nbgElements`: This is interpreted as an array of addresses of the class `Element`. The main reason for storing the addresses is that the time for passing over the information decreases drastically. This stores the addresses of all the neighboring elements for the node. It is very important to know all the neighboring elements in order to apply the Fast Marching Algorithm.
- `float* thetaStart, float* thetaEnd`: These two arrays store the θ_{start} and θ_{end} for each of the neighboring elements, as shown in the figure. Each of these arrays would be of length `noOfNbgElements`.

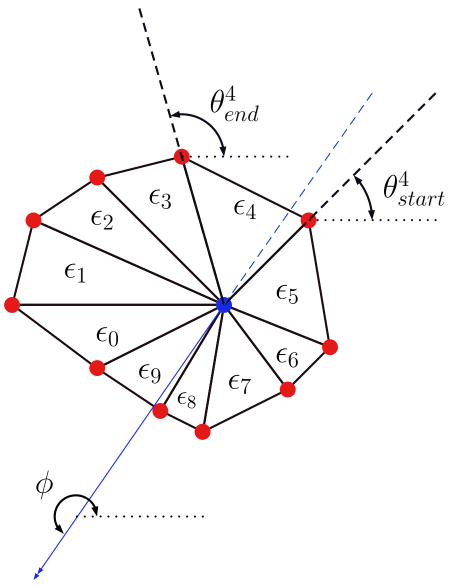


Figure 1: Structure of a Node in the grid. ϕ is the characteristic angle. The Node of reference is labeled blue, rest all nodes are labeled in red.

Element: The element class contains all crucial information which is sufficient to define an element. The current pre-processor implements only grids with triangular elements, and hence the interpreter hard-codes some of the values during its implementation. For ex. Only 3 nodes are needed to be considered to accurately define an element. In the Fig. 2, η_i's are the nodes which

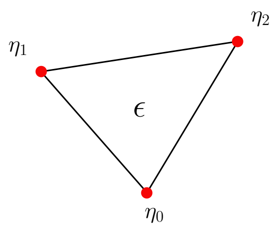


Figure 2: The structure of a Triangular Element(ϵ). The 3 nodes of the element are η_i's

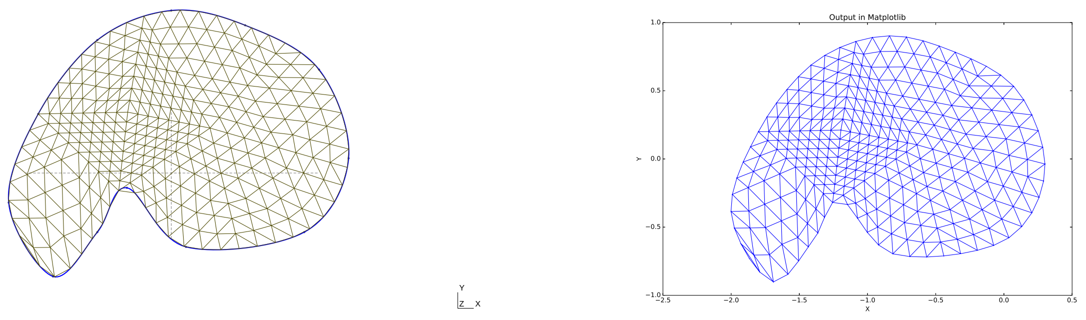
are needed to be stored within the class. The following information has been chosen to define an 'Element'.

- `Node* nodes[3]`: This is an array containing the 'addresses' to the 3 nodes forming the element. Again, it is very important to store the references to the node rather than storing the whole nodes as it would decrease the space complexity and also working around with pointers always helps us to get better running times.

Mesh2D: This class is the collective of the previous two classes. Basically it is a container that has both elements and nodes within it. The `Mesh2D` class contains a very important function '`Mesh2D::readFromFile(string fileName)`' which reads the mesh from a '.msh' file which is obtained from GMSH[2]. This function reads from the file and populates all the variables involved.

It contains the following properties:

- `int noOfNodes, int noOfElements`: These two variables store the number of nodes and elements which are present in the mesh.
- `Node** nodes`: It is and array of size 'noOfNodes', which stores the addresses of all the nodes that are present in the mesh.



(a) The mesh input of an arbitrary shape taken from GMSH.

(b) The output of the mesh as plotted by `‘matplotlib.pyplot’`.

Figure 3: My flowers.

- `Element** elements`: It is and array of size ‘noOfElements’, which stores the addresses of all the elements that are present in the mesh.

1.3 Results

In order to test the code several inputs of unstructured grids have been given as inputs to it and tested with the actual results after plotting. The results matched and the following images demonstrate the same.

The figures Fig. 3a and Fig.3b closely match each other. Several other tests have been performed even to verify the coordinates of the nodes and the structure of elements.

1.4 Plotting in Python

The output generated by the library “GMSH-Interpreter” was then fed to a Python file that would read the information about the nodes and elements. The ‘`matplotlib.tri`’ [3] package was used to achieve the task.

1.5 Conclusion

The GMSH pre-processor is setup and verified. With this work done we can proceed to the next step of actually implementing the FMM algorithm on such domains.

2 Week 2

The main tasks for the week were to generate the driver files, mark the initial conditions and finally mark neighboring nodes of the initial alive points. Throughout the whole work, a structured square mesh has been used for simplicity but they can be changed with a simple change in the initial conditions. The code is developed in a way to tackle any unstructured domains. The only change that needs to be done is to change the way in the initial conditions for the neighboring points are set.

2.1 Initial stages of the solver

A solver class named `EikonalSolver` has been initiated. It is currently capable of taking the inputs given and plotting the current state of the nodes i.e. in a graphical manner it will represent how many nodes are Alive($\mathcal{A}$), Far Away($\mathcal{F}$) and in the Narrow Band($\mathcal{N}$) region.

The variables used to define the solver are as follows

- `Mesh2D *mesh`: This stores the address of the mesh which is involved in the solver. The mesh itself stores the co-ordinates, nodes, elements, etc.
- `function<float(float, float)> F, v1, v2`: These functions store the wave speed and medium velocity fields i.e. $F(x, y)$, $v1(x, y)$ & $v2(x, y)$ respectively. Where the medium velocity is given by $v(x, y) = v1(x, y)\hat{\imath} + v2(x, y)\hat{\jmath}$.
- `priority_queue<Node*, vector<Node*>, Compare> narrowBandHeap`: This is a priority queue which would enable us to implement the heap data structure, which is provided as a part of the Standard Library in C++. The class `Compare` has been designed in order to sort the nodes with respect to their T i.e. their wave arrival time. At each iteration, the head would contain the node in the Narrow Band region with the minimum arrival time.

2.2 Domain of Interest

The domain(Ω) is an (un)structured grid as shown in Fig. 4.

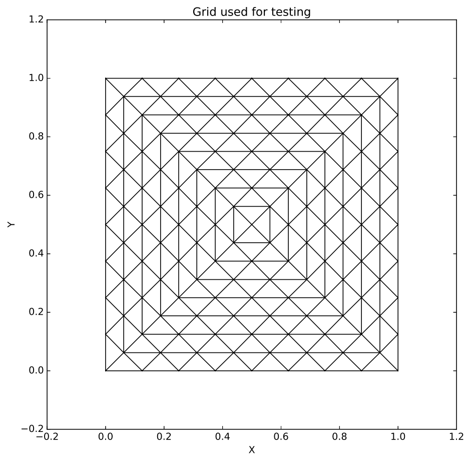


Figure 4: The domain(Ω) on which the code is tested

2.3 Initialization

The whole process of initialization can be broken down into the following steps:

1. The initial conditions are chosen such that the line $x = 0$ would be the initial set of Alive points & the points on the initial wavefront. The code would initially consider all the nodes to be far-away points and the grid would be of the state as shown in Fig. 5.

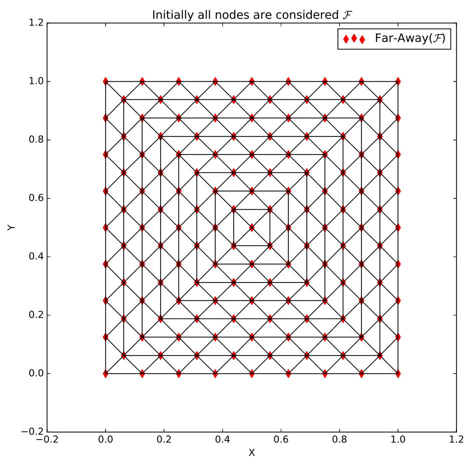


Figure 5: All the nodes are initially marked as $\mathcal{F}$

2. Then, all the points satisfying the condition $x = 0$ would be considered to be Alive and hence, would be removed from the set $\mathcal{N}$, and added to the set $\mathcal{A}$. The state of all the nodes in the grid resembles the state in Fig. 6

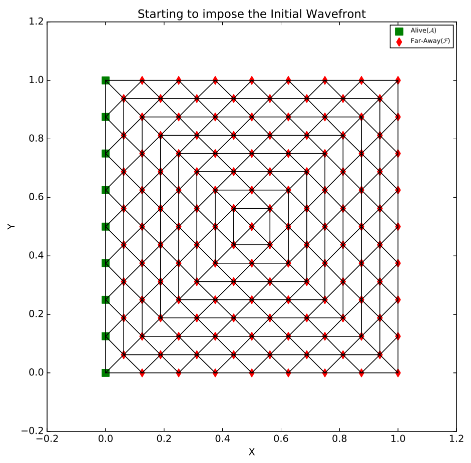


Figure 6: The initial condition and the nodes of initial wavefront are pushed in the set $\mathcal{A}$

3. Finally all the neighboring nodes of the Alive nodes and belonging to the set $\mathcal{N}$, and the state would replicate the state as shown in Fig. 7.

2.4 Conclusion

The initialization has been complete, and the next task would be to trying to implement the Fast Marching Algorithm and the initialized solution. The code used in [1] must be now implemented on unstructured grids.

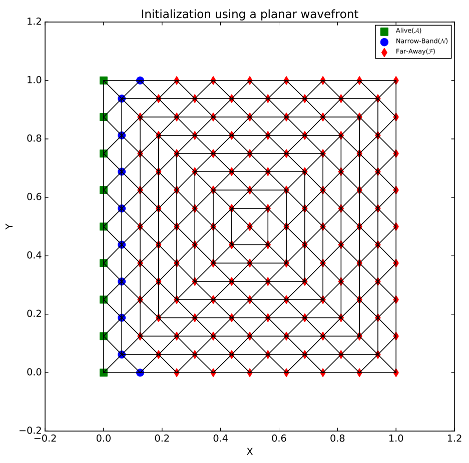


Figure 7: Initialization has been complete and the set $\mathcal{N}$ has been populated successfully.

3 Week 3

This week was a very important week, as the task of pre-processing the mesh and generating the input had already been done and this week I had to start the implementation of the Fast Marching Algorithm[1]. I tried to implement the `scheme()` function which was described as `schemeF()` in the earlier code used by Dahiya et. al. Two new functions have been developed namely: `solve()` and `scheme()`. They have been explained in depth in the further sections.

3.1 Current Implementation Details

1. The initial conditions, including the initial set of Narrow Band nodes have been taken as an input. Using those initial conditions the very first Narrow Band Heap has been generated. The generation of the Narrow Band Heap has been done in the constructor of the `EikonalSolver` itself. All the initial nodes belonging to $\mathcal{N}$ have been marked as accepted.

2. The main function is `solve()` and this must called in order to start the implementation of the program.

3. In solve:

   - The `solve()` at the start itself, it enters a loop which continues until the $\mathcal{N}$ heap is non-empty.
   - During each iteration of the loop, it identifies the topmost node, which indeed corresponds to the node which has the minimum `T`.
   - Then, it identifies all the neighboring Elements of the node, which was already a functionality added in the first week. (During the pre-processing itself all the neighboring elements of each node have been identified.)
   - After identifying the neighboring elements the nodes which are a part of those elements are all added to a `set`. The `set` is a data-structure provided by the C++ Standard Template Library.
   - Why use a `set`? The most important aspect of a `set` is that each element is unique i.e whenever any node is added to the `set` it identifies whether the node is already present in the `set` and only then inserts it. The main problem being, as shown in Fig. 1, consecutive elements share a common node. For ex. ϵ_2 and ϵ_3 share a node, and while iterating through the neighboring elements it is very important to recompute the value `T` only once at each of the "Neighboring Nodes". Hence in order to identify all the "*non-alive*" neighboring nodes `set` must be used.
   - Once all the "*non-alive*" neighboring nodes have been identified the function `scheme()` [See pt. 4] is then run on each element of the `set` which had been previously identified.

- Once the program has returned from `scheme()`, the node is now added to the set of $\mathcal{N}$. Simply put, the state is updated to Narrow Band value.

4. In scheme:
   - As soon as a node enters in the function, its previous state(i.e whether it belongs to $\mathcal{N}$ or $\mathcal{F}$) is stored.
   - Arrays namely `possibleSolutions` and `solutionStates` have been initialized.
     - Both are of size equal to the number of neighboring elements of the given node.
     - All the entities of `possibleSolutions` store the solution T_{ij} which has been computed from that particular neighboring element. Initially all are set to be ∞, which would be useful when we need to take the minimum of all the acceptable solutions.
     - The `solutionStates` store the "*acceptance quality*" of the solution solution which has been found from the appropriate element in the above step.
   - Now, the function loops through all the neighboring elements of the given node. The following are the conditions which maybe faced, related to the other two nodes of the neighboring element:
     - Both $\in \mathcal{F}$: Does not calculate the value here the `possibleSolution` is left to remain ∞ and the `solutionState` is set to 0(zero).
     - One node $\in \mathcal{F}$ and another $\notin \mathcal{F}$: The value of T_{ij} can be calculated from one of the nodes which $\notin \mathcal{F}$. And the `possibleSolution` here is set to 1. Although we could get a value of T_{ij}, the value has not come from acceptable sources, and hence this solution would not be accepted.
     - Both $\notin \mathcal{F}$: We can obtain the value of T_{ij}, using the values of the two corresponding nodes. In this case:
       * `solutionState = 2`: If and only if, both the nodes from which the calculation is done are accepted.
       * `solutionState = 1`: If atleast one of the nodes is not accepted.
   - Once, the above loop covering all the neighboring elements has run. The function then tries to find the solution which is the most appropriate.
   - First it tests all the solutions which have `solutionState = 2` and takes the minimum of solution computed. And mark the node as *"accepted"*. Push it to the $\mathcal{N}$ heap.
   - If in the above step, no neighboring element gave the value `solutionState = 2`, in that case update the `T` of the node and mark it "*not accepted*".

5. Finally, update the state of the node to $\mathcal{N}$.

3.2 Functions Developed

The following functions have been developed:

- `solve()`: This an integral public function which must be called in order to start computing the solution. The working of this function has been explain in Sec. 3.1.

- `scheme()`: This is the function which helps us to calculate the `T` of the node, which is being put into $\mathcal{N}$ for the first time. This function also helps to recompute of the node which is already in the $\mathcal{N}$. The functioning of this is explained in Sec. 3.1.

3.3 Formulae Used

The following formulae have been used, the notation reference can be taken from [1]. The directions and the notation have also been stated in Fig. 8.

$$m_{11} = \frac{x-a_1}{|AX|} \quad (1)$$

$$m_{12} = \frac{y-b_2}{|AX|} \quad (2)$$

$$m_{21} = \frac{x-b_1}{|BX|} \quad (3)$$

$$m_{22} = \frac{y-b_2}{|BX|} \quad (4)$$

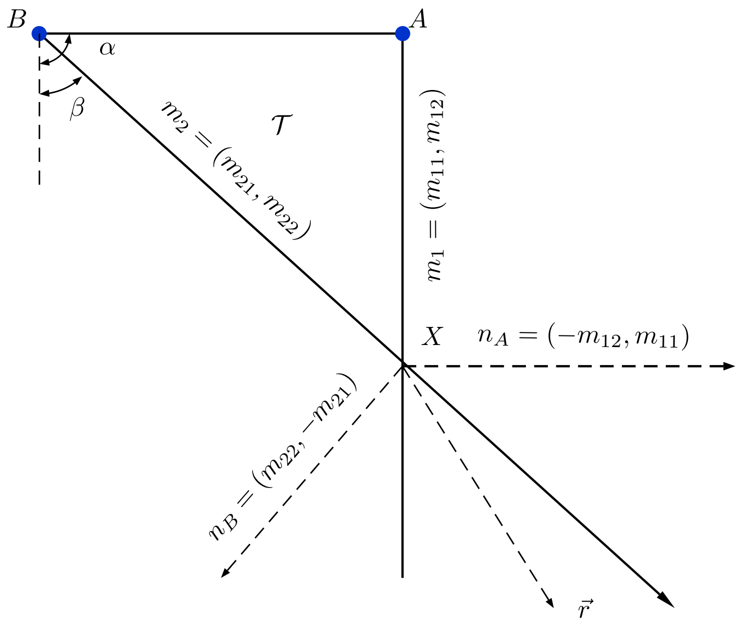


Figure 8: Direction and notations used in the context

The following formulation has been used when the information about both the nodes have been used.

$$
\begin{aligned}
a &= \frac{m_{22}}{(m_{11}m_{22}-m_{12}m_{21})|AX|} - \frac{m_{12}}{(m_{11}m_{22}-m_{12}m_{21})|BX|} && (5)\\
b &= \frac{-m_{22}T(A)}{(m_{11}m_{22}-m_{12}m_{21})|AX|} + \frac{m_{12}T(B)}{(m_{11}m_{22}-m_{12}m_{21})|BX|} && (6)\\
c &= \frac{-m_{21}}{(m_{11}m_{22}-m_{12}m_{21})|AX|} + \frac{m_{11}}{(m_{11}m_{22}-m_{12}m_{21})|BX|} && (7)\\
d &= \frac{m_{21}T(A)}{(m_{11}m_{22}-m_{12}m_{21})|AX|} - \frac{m_{11}T(B)}{(m_{11}m_{22}-m_{12}m_{21})|BX|} && (8)
\end{aligned}
$$

The following formulation has been used when one of the nodes belongs to $\mathcal{F}$. (Say node B belongs to $\mathcal{F}$)

$$
\begin{aligned}
a &= \frac{m_{22}}{(m_{11}m_{22}-m_{12}m_{21})|AX|} && (9)\\
b &= \frac{-m_{22}T(A)}{(m_{11}m_{22}-m_{12}m_{21})|AX|} && (10)\\
c &= \frac{-m_{21}}{(m_{11}m_{22}-m_{12}m_{21})|AX|} && (11)\\
d &= \frac{m_{21}T(A)}{(m_{11}m_{22}-m_{12}m_{21})|AX|} && (12)
\end{aligned}
$$

3.4 Conclusion

In order to complete the algorithm, only checking the causality condition is left. That would mostly be the task for the next week. Most of the conditions which have been currently assumed are subject to change in the coming weeks.

4 Week 4

The objective of this week was to insert the causality condition alongside the previous week's code. The θ_{start} and θ_{end} of each element had been pre-computed while the processing of the mesh. The requirements of the week were as follows:

- Calculate the characteristic direction arising due to T_{ij} from the respective element.
- Then with the assistance of the pre-computed θ_{start} and θ_{start} verify whether the causality condition is satisfied.
- Set the indicator variables accordingly, according to the results obtained, and finally set the acceptance of the nodes.

4.1 Overview of the code

Currently the code has been provided with almost all the functionalities and this might be a good time to summarize the current version of the code.

`→Setting the initial conditions.`

1. From the program `main.cpp`, the initial conditions are supplied.
2. The initial points include all the initial "Alive" nodes and the initial "Narrow-Band" nodes.

`→Entering the EikonalSolver class' constructor.`

1. Both the initial given nodes($\in \mathcal{A}$ `or` $\in \mathcal{N}$) are tagged as "Accepted".
2. Sets the `F, v1, v2` of all the nodes in the mesh, which has been supplied as an input.
3. Then the constructor does a single scan through the mesh in order to identify all the "Narrow Band" nodes and adds all of them to the "`narrowBandHeap`".
4. Now, the Narrow-Band Heap is appropriately initialized.

`→Now, the program enters the function solve().`

1. Identifies the topmost node in the heap(i.e. the node with the minimum 'T'), pops it out and stores it in the variable "`currentNode`".
2. The state of the "`currentNode`" is updated to "Alive".
3. All the neighboring elements of the "`currentNode`" are identified and the "`non-Alive`" nodes of the neighboring elements are added to the set[1] "`set_new`"[I still feel that it should be "`non-Accepted` rather than `non-Alive`"].
4. On all the nodes contained within the `set_new`, the function `scheme()` is run.
5. After the `scheme()` is run the node is marked as "`Narrow-Band`"

`→Overview of the function scheme().`

1. It takes the input – the node on which the 'T' value is to be computed.
2. Then stores the relevant information about the node such as `noOfNbgElements`.
3. Then it creates 4 arrays which have listed in Table 1.

[1]The information about the data-structure set has been illustrated in the previous week.

Name	Type	Size	Initial Value	Purpose
possibleSolutions	`float`	noOfNbgElements	∞	To get the value of the solution computed from the particular Nbg. Element.
calculationIndicator	`bool`	noOfNbgElements	`false`	This indicator is true when the calculation of 'T' results in a finite value.
acceptanceIndicator	`bool`	noOfNbgElements	`false`	This indicator is set true when the other 2 nodes of the element are "accepted".
causalityIndicator	`bool`	noOfNbgElements	`false`	This indicator is set true when the computed 'T' satisfies the causality condition.

Table 1: The four arrays

4. Then the function loops through all the neighboring elements of the input node, and calculates the value arising from each of the neighboring element and stores the computed value(if possible) in "`possibleSolution`".
5. Once the 'T' is computed all the indicators are updated accordingly.
6. After scanning through all the neigboring elements, the solutions along with their indicators are used to check whether the solution should be accepted or not.
7. Currently, the solution is accepted if and only if all the 3 indicators are set to `true`.
8. If a solution is accepted the node is pushed into the $\mathcal{N}$ heap.

4.2 Conclusion

The whole code is now ready, with a bit of controversial statements related to the acceptance of the nodes everything is set and one can start testing using various test cases.

References

[1] D. Dahiya and S. Baskar, “Characteristic fast marching method on triangular grids for the generalized eikonal equation in moving media,” *Wave Motion*, vol. 59, pp. 81–93, 2015.

[2] C. Geuzaine and J.-F. Remacle, “Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities,” *International Journal for Numerical Methods in Engineering*, vol. 79, no. 11, pp. 1309–1331, 2009.

[3] J. D. Hunter, “Matplotlib: A 2d graphics environment,” *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.