Embedded Boundary C++ Implementation

Marissa Ramirez de Chanlatte

January 3, 2020

1 Overview

Based on the algorithm in [2], I wrote a program to generate embedded boundary geometry information. This document is a collection of notes on the theory and implementation of this work.

2 Literature Review

2.1 Robust and Efficient Cartesian Mesh Generation for Component-Based Geometry - Aftosmis, Berger, and Melton

2.1.1 Summary

Aftosmis et. al. present a new method for Cartesian mesh generation. They first intersect the components to obtain the geometry and then perform the volume mesh generation. Importantly, the intersection algorithm is robust, using adaptive precision arithmetic. Involved in the intersection algorithm are many common problems of computational geometry: proximity queries, non-convex polyhedron intersection, constrained triangulation, robust inside/outside predicate, rapid mesh traversal, and robust handling of geometric degeneracies.

Proximity Queries

Important to the intersection algorithm is determining all candidate triangles that could intersect with a particular triangle. Naively implemented, this could have a complexity of $O(N^2)$. The authors use an alternating digital tree (ADT) developed by Bonet and Peraire which reduces this time to $O(logN)$.

Intersection of Triangles in R^3

For two triangles to insersect in 3D, two edges of one triangle must cross the plane of the other and there must be a total of two edges that pierce within the boundaries of the triangle. These are the conditions the authors care to test. One approach is to directly compute the pierce points of the edge of the triangle in the plane of the other, but it is not robust numerically (requires floating point division). Instead, they use a Boolean check of the three-dimensional intersection of an edge and a triangle. It is based on the signed volume of a tetrahedron in R^3. This signed volume is positive when the points forming the tetrahedron (a, b, c) are counterclockwise and negative when they are clockwise. It is zero if they are coplanar. Using this predicate with an edge ab and a triangle $(0, 1, 2)$, ab intersects the plane of the triangle if and only iff the signed volume of T_{012a} and T_{012b} are opposite.

Retriangulation of Intersected Triangles

To handle the retriangulation, the authors employ Green and Sibson's incremental constrained Delaunay triangulation algorithm. This algorithm depends on the in circle predicate (available in Jonathan Schewchuck's robust predicate library).

Inside/Outside Determination

The next step in their algorithm requires deleting all the triangles that are internal to the surface. Determining which triangles are internal can be thought of as a point in polyhedron problem. They use a ray casting approach. To reduce the number of rays that must be cast, each triangle communicates it's status to the three triangles that share its edges and the algorithm recurses on the neighboring triangles. The recursive algorithm is implemented using a stack.

Geometric Degeneracies

The geometric predicates depend heavily on computing the sign of a determinant. Robustly and efficiently computing this quantity is essential. This operation can produce a negative, positive, or zero, where zero is the degenerate case. The difficulty then becomes distinguishing a true zero from floating point error. To do this, the authors compute the determinant value and then make an a posteriori estimate of the maximum error. If the error is larger than the value, they redo the calculation using Jonathan Shewchuck's adaptive precision exact arithmetic method. If it is exactly zero, they go to their tie breaking approach. The tie breaking algorithm is a virtual pertubation method known as "simulation of simplicity (SoS)." The idea is to assume all input data go through an ordered, unique perturbation so that there exist no ties. Because the perturbations are unique and constant ties are resolved in a topologically consistent manner. The geometric data are not altered because the pertubations are virtual.

Volume Mesh Generation

Volume mesh generation is performed after the intersection algorithm, meaning that problems of internal geometry have already been handled, decreasing the complexity of the task. The geometry is divided into a cartesian mesh and then cells that intersect the body and their neighbors are refined making an adaptive mesh. This mesh is treated as unstructured rather than using an octree. To determine cut cells, they use a highly specialized algorithm for use with coordinate aligned regions that comes from the computer graphics literature.

Results

To demonstrate the ability of their algorithm they simulated a few different aircraft with highly complex geometries.

2.1.2 Relevant Points

The intersection of triangles problem is reduced to a question of intersection of an edge and a plane. The authors demonstrate an approach to answering this question that is robust and could be useful in our applications.

2.1.3 Areas of Difference

Octtree vs unstructured. Phil notes driven by platform where memory is at a premium.

2.2 Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms

2.3 A New Adaptive Mesh Refinement Data Structure with an Application to Detonation - Ji, Lien, Yee 2010

2.3.1 Summary

CSAMR [1] is a new data structure designed to improve memory usage and parallelizability for adaptively refined meshes.

Octtrees The standard way to store these meshes is a quad/octtree. While very intuitive in construction, octtrees have a large memory overhead. Removing or adding a cell while refining affects all the neighboring cells making the process also difficult to parallelize. To find the nearest neighbor requires two levels of traversal on average, but may require traversing the entire tree ($O(N)$ using depth first search). A full traversal may also pose problems in parallelization.

Fully Threadded Tree To reduce the memory overhead in octtrees, fully threadded trees (FTT) were developed. They are basically the same concept, with a few optimizations thrown in. Most notably, not all intermediate levels are stored. This ensures the number of traversed levels to find a nearest neighbor never exceeds one. Cells are organized into octs. Each oct has pointers to eight child cells and a parent cell. It also stores information about its level, position, and pointers to the parent cells of the six neighboring octs. Each cell of the oct has a pointer to its child oct (null if there are no children). This comes out to a memory requirement of 19 words per oct or $2\frac{3}{8}$ words per cell. While memory usage is significantly reduced and finding nearest neighbors is easier, it is still difficult to locate an

arbitrary cell potentially requiring a complete traversal that is not easily parallelizable.

Cell-based Structured Adaptive Mesh Refinement The authors propose a new data structure they call Cell-based Structured Adaptive Mesh Refinement (CSAMR). Cells are given cartesian like indices and those indices are used as keys to a hash table containing useful information on parent, children, and neighbors. The usage of indices reduces memory overhead to $\frac{5}{8}$ words per cell. In addition, the AMR process is more easily parallelizable with data structure as we are no longer concerned with whether the parent of a cell resides on the same processor. In two dimensions, the ID is calculated in the following way:

$$ID = \sum_{l=0}^{level-1} (2^l \cdot 2^l) + i \cdot 2^{level} + j \tag{1}$$

where (i, j) is the oct index and level denotes the oct level. Using this ID allows the creation or destruction of an oct during the mesh refinement process to be much simpler. There is no need to explicitly update information concerning parent, children, or neighbors, only to delete or insert the oct from the hash table. This grid partitioning (and mapping) is achieved via a Hilbert space-filling curve. The Hilbert curve has three key properties: mapping, locality, and compactness. The compactness property implies that only the coordinates of the point are necessary for the determination of the location on the one dimensional curve. This property is not shared with other space filling curves, such as the Morton curve, and is important because it allows the curve to be constructed in parallel without needing to exchange any information between processors.

3 Theory

The basic algorithm is as follows (restated from [2]):

1. Compute all derivatives of the normal, $\partial^s n_d$.

$$\hat{n} = \frac{\nabla\psi}{L}, \tag{2}$$

where $L = |\nabla\psi|$.

For $d = 0 \ldots D$ in order calculate

$$\partial^p L = \sum_d \sum_{r \le q} \binom{q}{r} \partial^r n_d \partial^{q-r+e_d+e^i} \psi \tag{3}$$

where $q = p - e^i$ and using $\partial^p L$ calculate

$$L\partial^p n_d = \psi^{(p+e_d)} - \sum_{q \le p, q \ne p} \binom{p}{q} \partial^{p-q} L \partial^q n_d \tag{4}$$

storing all derivatives of the normal.

2. Compute one-dimensional moments $M^q_{V,1}$ using root finding and integration.

$$M^q_v = m^q_v + O(h^R) \tag{5}$$

$$m^q_v = \int_V (x - \bar{x})^q dV \tag{6}$$

3. Irregular one-dimensional moments $M^q_{B,1}$ are zero.

4. Loop through other dimensions as follows:

   (a) for $D = 2, 3$

      i. for $|\mathbf{q}| = \{Q, Q-1, \ldots, 0\}$

         A. Calculate $M^q_{d\pm,D-1}$.

$$M^q_{d\pm} = m^+_{d\pm} O(h^R) \tag{7}$$

$$m^q_{d\pm}(\bar{x}) = \int_{A\pm} (x - \bar{x})^p dA \tag{8}$$

B. Set ρ using the values calculated above:

$$\rho = M^q_{d+,D-1} - M^q_{d-,D-1} + \sum_{1 \leq |s| < S} \frac{\partial^s n_d}{s!} M^{q+s}_{B,D}$$

C. Solve the resulting set of linear equations for $M^{q-e_d}_{V,D}$ and $M^q_{Bd,D}$

$$q_d M^{q-e_d}_{V,D} - n_d M^q_{B,D} = \rho$$

4 Implementation

To best understand the algorithm, I prototyped a 2D version of the software in an iPython notebook. I ran two test problems, the line $y = x$ and a circle $x^2 + y^2 = 1$. I calculated the volume fractions of each cell in the geometry and showed that as the cell size decreased, the total estimated volume approached the true volume.

Next, I implemented a more general, although still 2D, version in C++. This version (along with the Python prototype) is stored at `https://github.com/mzweig/EmbeddedBoundary`. The main differences between this implementation and the earlier one is that arbitrary inputs are now supported and only information about boundary cells is stored, as opposed to all cell information.

4.1 Dependencies

The code has three main dependencies, gtest and gmock, for testing purposes, and Eigen, which provides the least squares linear solver used in step four of the algorithm. Doxygen is used to generate documentation. Besides these specific functions, the code relies exclusively on the standard template library.

4.2 Coding Abstractions

The code is structured into four main classes: `Point`, `Normal`, `Boundary`, `Input`. The point class defines a point object which is a two dimensional array describing a point (x, y). The points can be added and subtracted from each other. The normal class calculates all information pertaining to the vectors normal to the boundary and their derivatives.

The boundary class houses all geometric information pertaining to the boundary cells in an unordered map. Points, representing a cell center, are used as the map keys and the elements returned are structs containing a flag indicating that the cell is on the boundary, an id, a vector of vectors storing information regarding the normal and its derivatives (this is a std::vector as opposed to an array because its size depends on the order of accuracy specified in the input), the one dimensional volume fractions that must be calculated first, and finally the volume and boundary moments. The functions in the boundary class are the main parts of the embedded boundary algorithm that use the normal information to calculate the volume and boundary moments.

The input class provides a base class from which all inputs are derived from. There are several functions that every new input must override and redefine in order for the geometry information to be calculated. They include, a function describing the boundary, a function describing the derivatives of the boundary (up to order Q which must also be defined), the inverse of the boundary function, a function giving an orientation to the boundary (stating whether a given point is "inside" or "outside" the boundary), the desired domain of the calculation in terms of a maximum and minimum in x and y, and the desired cell size (the code assumes square cells).

4.3 Testing

The code is united tested throughout, but as a comprehensive test, I used the input with the geometry $x = y$. I expected the algorithm to be able to find the volume exactly, and it was able to do so. I next intended to test with a circle geometry as I did with the prototype,

but as the C++ version is almost an exact replica of the Python version, I don't expect it should have too many problems.

4.4 Building and Running

Before building, an input class must be specified in main.cc. See the next section for examples. The code can be built with either cmake or GNU make.

4.4.1 Building with cmake

To build with cmake run the following commands

1. `mkdir build`
2. `cd build`
3. `cmake ..`
4. `make`

Run the `makeGeometry` executable to generate volume fractions. Results can be plotted using the python scripts in the plotting directory.

4.4.2 Building with GNU make

To build with GNU make run `make` in the root directory. Run the executable `build_geometry.out` in the build directory. To build and execute unit tests use the command `make test`.

4.4.3 Input Files

Inputs take the form of C++ classes specified in .cc files derived from `input_base.h`. To create an input file, create a new class that override the methods in `input_base.h` with methods that describe the desired geometry. The necessary functions are

- `BoundaryFunction`: A function taking in a quantity x and returning a quantity y that describes the boundary.
- `BoundaryDerivatives`: A function taking in a point and order and returning the value of the derivative of the given order at the given point.
- `BoundaryInverse`: A function taking a quantity y and returning a quantity x that describes the boundary.
- `Inside`: A predicate returning whether or not a given point is inside or outside the boundary.
- `Max` and `Min`: A series of functions returning the maximums and minimums of the domain of interest.
- `CellSize`: Starting cell size.
- `QOrder`: Desired order of accuracy for the moment approximations.

4.5 Example Problems

4.5.1 Line Boundary

This example problem uses the line $x = y$ as the boundary. The volume fractions at the boundary are calculated over the domain $[-1, 1]x[-1, 1]$ with cell size $h = 0.25$ and approximation order 1. The example input can be found in the `inputs/line` directory. The following is a plot of the volume fractions at the boundary. Note that cells that are entirely inside or outside are not stored. Their volume fractions can easily be calculated after the fact by using the inside/outside predicate.

4.5.2 Circle Boundary

This example uses the circle of radius 1 $x^2 + y^2 = 1$ as the boundary. The cell size is again $h = 0.25$ and the order of approximation 1. Below is a plot of the volume fractions along

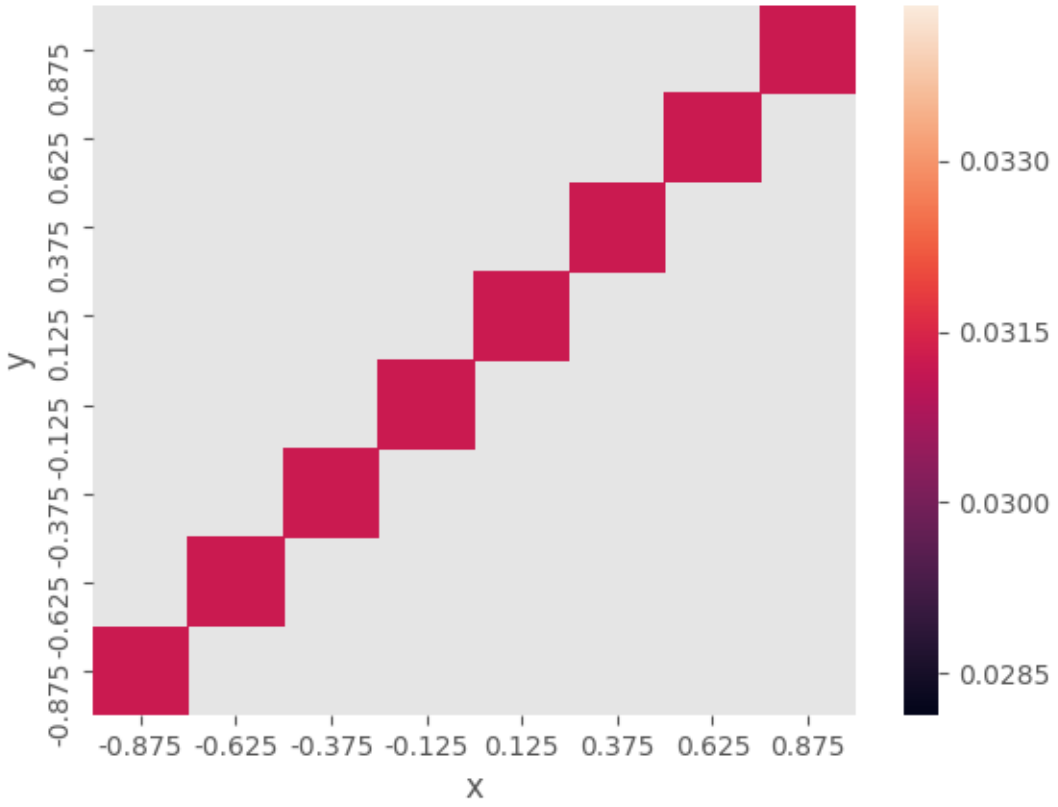

Figure 1: Volume Fractions Along Line Boundary

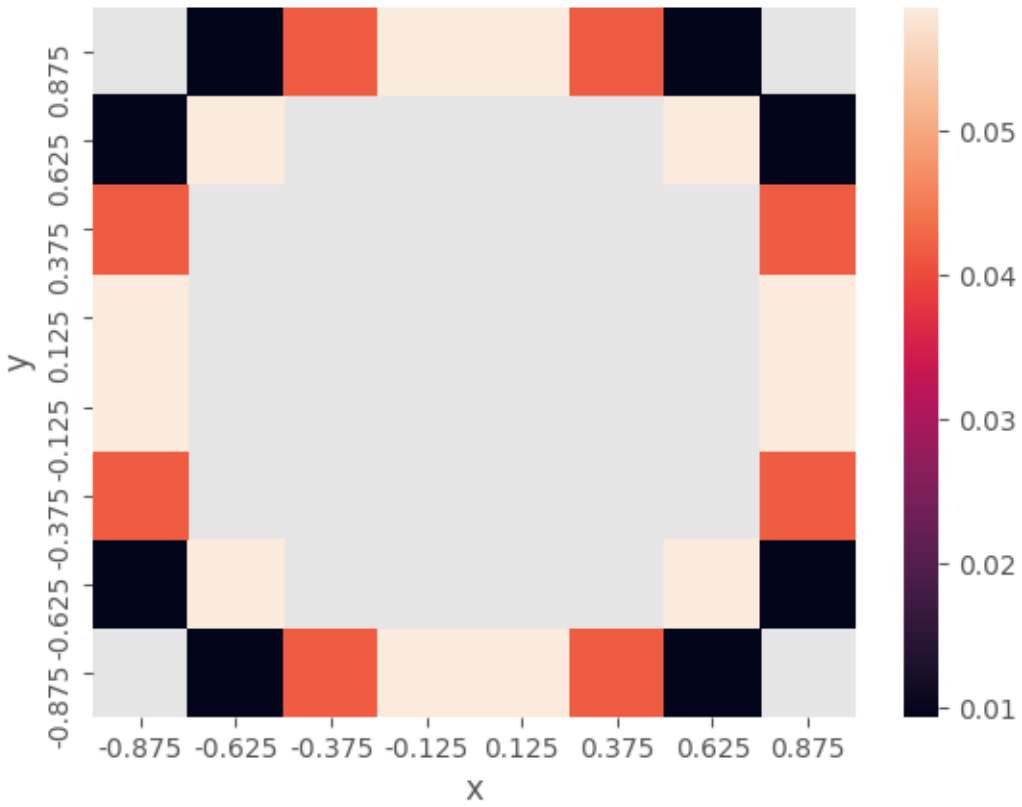

Figure 2: Volum Fractions Along Circl Boundary

the boundary.

4.6 Limitations/Future Work

Currently the code only supports two dimensions and does not support adaptive mesh refinement. The output and visualization system is not there yet. In the python prototype output the volume fractions as a numpy array and plotted them using matplotlib. One approach would be to have my code dump out volume fractions as a txt file and then use python for visualization and post processing. I like this approach in 2D, but in 3D I would probably like to make the code functional with Visit.

References

[1] H. Ji, F.-S. Lien, and E. Yee, *A new adaptive mesh refinement data structure with an application to detonation*, Journal of Computational Physics, 229 (2010), pp. 8981–8993.

[2] P. SCHWARTZ, J. PERCELAY, T. J. LIGOCKI, H. JOHANSEN, D. T. GRAVES, D. DEVENDRAN, P. COLELLA, and E. ATELJEVICH, *High-accuracy embedded boundary grid generation using the divergence theorem*, Communications in Applied Mathematics and Computational Science, 10.