Embedded Boundary C++ Implementation

Marissa Ramirez de Chanlatte

September 26th, 2019

1 Overview

Based on the algorithm in [1], 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 Theory

The basic algorithm is as follows (restated from [1]):

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

$$\hat{n} = \frac{\nabla \psi}{L}, \tag{1}$$

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

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

$$\partial^p L = \sum_{d} \sum_{r \leq q} \binom{q}{r} \partial^r n_d \partial^{q-r+e_d+e^i} \psi \tag{2}$$

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

$$L\partial^p n_d = \psi^{(p+e_d)} - \sum_{q \leq p, q \neq p} \binom{p}{q} \partial^{p-q} L \partial^q n_d \tag{3}$$

storing all derivatives of the normal.

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

$$M^q_v = m^q_v + O(h^R) \tag{4}$$

$$m^q_v = \int_V (x - \bar{x})^q dV \tag{5}$$

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

$$M^q_{d\pm} = m^+_{d\pm} O(h^R) \tag{6}$$

$$m^q_{d\pm}(\bar{x}) = \int_{A\pm} (x - \bar{x})^p dA \tag{7}$$

         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 $M^q_{Bd,D}$

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

3 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.

3.1 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.

3.2 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).

3.3 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.

3.4 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] 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.