Deterministic dimension independent mesh generation

December 10, 2015

1 Design

In this section, we describe the basic design of the meshing algorithm.

1.1 Requirements

When using finite element method, one usually use an adaptive variable size discretization of the space in order to handle different geometries with different precision requirement. This is very important in order to avoid using the highest grid resolution for the whole problem and to reduce the size of the final linear system to solve. In order to support this, we would like our meshing algorithm to handle arbitrary geometry constraint with arbitrary spacially varying meshing density. Since separate space can be descretized independently, we can assume the space is connected. However, it is not restricted to be singly connected in order to support problems with inner surfaces.

In additional to these basic requirements, we would like to handle meshing of curved space with the same algorithm. This is useful to solve problems on a geometry embedded in a higher dimensional space, e.g. a curved surface in three-dimensional space.

1.2 Abstraction of the model

In order to generate a mesh, it is necessary to specify the problem first. Although we would like to handle arbitrary configurations mentioned in the previous section, we do not want to write specialized code for each problems. Therefore, before being able to generate the mesh, we first need a standardized interface to specify the properties of the problem. The interface needs to be flexible enough to cover different kinds of problems and it should also provide enough information to fully specify the problem. Moreover, the information provided by the interface should be relatively easy to generate during the construction of the problem and it should also be easy for the meshing algorithm to process.

Due to the flexible requirement, we decided that the most important properties of the space is the constraint of allowed values (when the problem is embedded in a higher dimensional space) and the relation between different elements. These informations are important to make sure that the mesh we generate is an accurate description of the problem and is not over sampling the space. Therefore, in additional to the interfaces that provide basic informations (e.g. problem dimension, local mesh density) we define the following two functions to specify how to find the next point in the problem space from the current point, and if a new mesh element is intersecting with existing ones.

```
get_next_point{N,V}(model::AbstractModel{N,V}, point::V,
                    step::V, section::Int, clip::Bool)
check_crossing{N,V}(model::AbstractModel{N,V}, orig_points::NTuple{N,V},
                    point2::V, tileset::TileSet{N,V})
```

With a reasonably small grid size and slow variation of space parameters, these informations can be determined from only local informations of the model and are therefore easy to generate.

2 Implementation

In this section, we describe our implementation of the algorithm. We do not cover all the technical detail about the implementation, which can be more clearly explained by the code itself. Instead, we will describe the important steps in the algorithm and a breif outline how they are implemented.

Since our abstraction of the model only contain local information of the space, the algorithm we use to generate the mesh is naturally an iterative method that covers the whole space by locally extending the current mesh. In each iteration, we pick an edge/surface we previously generateed and try to construct the next one out of it. When generating it, in additional to making sure we are using the right grid size, we also need to make sure that it is within the valid region and does not cross with existing ones. In order to do this, we use the following steps,

- Check if the current point is close enough to the boundary.

  If it is closer than a certain threshold (10% of the grid size), we simply treat it as a point on the boundary and do not extend it.

- Determine if we should simply combine with a neighbering edge.

  In order to do this, we check all of the mesh elements that are directly connected and check if the angle between the two is suitable for generating the next element.

- Look for other existing mesh elements nearby.

  If there are any other nearby elements that are not directly connected with the current one, we pick one of the point among them to construct the next element. We look for nearby elements by calculating the distance from each element to a certain point near or on the current element and therefore does not require any interaction with the model itself. This is a generalization of the previous step but requires more arithmetics and is more expensive.

- Now we can add a new mesh element independent of other ones.

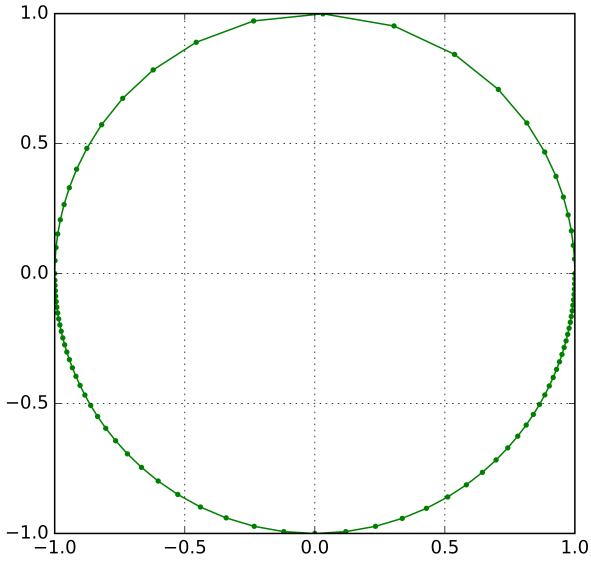
  The only thing we still need to be careful about is the boundary of the area, which is handled by the check_crossing function.

During this process, we keep track of the new elements addes and in which directions they should be extended. We terminate the iteration when there is no element left to be extended and the whole space is covered.

3 Results

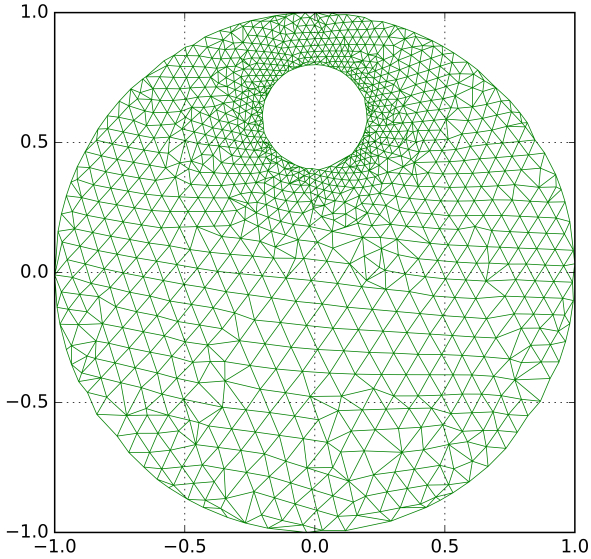
Due to time constraint, we have only implemented and debugged the meshing algorithm for one and two dimensional systems. However, we could demonstrate that the algorithm works unmodified for meshing a curved space embedded in a higher dimention, for example, the edge of a circle or the surface of a sphere. We also demonstrated that we can use post-generation optimization passes to improve the homogeneity of the mesh.

3.1 Meshing the edge of a circle



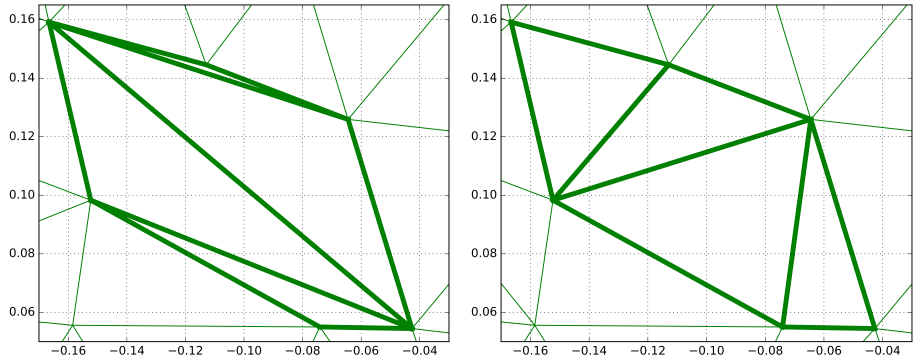
The coordinates are expressed directly with a two dimensional vector instead of a single number and the algorithm can handle this very well. We can also see the variation of the meshing density follows what we specified.

3.2 Meshing a circle with a hole and variable density



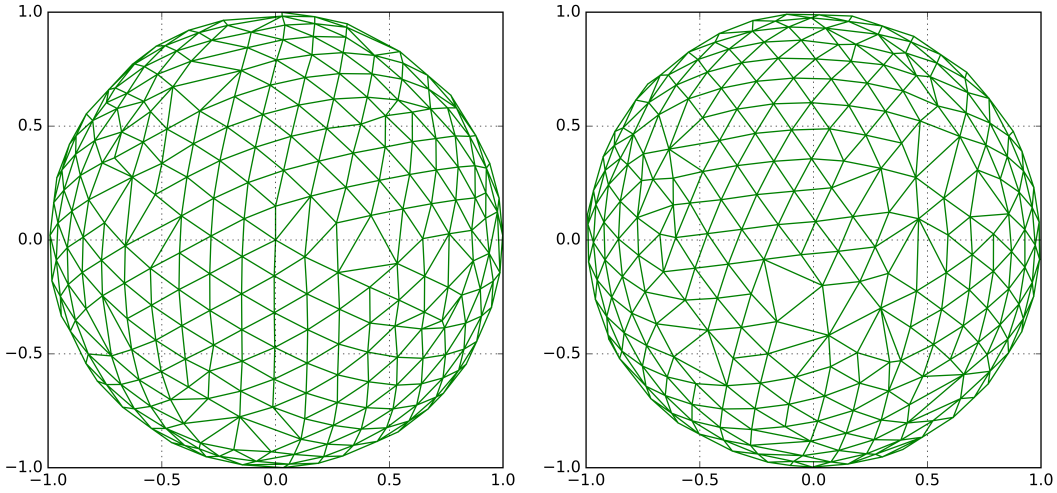
The algorithm smoothly connect the region near the hole with higher meshing density to the region near the center with lower density. It also correctly avoided the hole during the meshing process.

3.3 Post meshing optimization



Here we show a part of the mesh before and after we apply the optimization pass. We clearly see that by rearranging the connection, we could remove many thin and unevenly sided triangles we generate.

3.4 Meshing a three dimenstional curved surface



This is the result of meshing the surface of a sphere in three dimenstion using the same algorithm. The plot shows the projection of the mesh in the front and back side of the sphere onto a two dimensional plane. The mesh spacing are smaller closed to the edge of the edge because of the projection.

4 Future improvements

While the algorithm works well for two dimenstional space of arbitrary shape and curvature, there are still many place that can be improved. The algorithm currently uses a naive approach to look for neighbering points and loop through all the elements currently generate. In practice, this is not necessary since elements that are too far away does not need to be checked. Even though the space is continious, we could still pre-discreteize it into regular square grid and use a hash table to map between the mesh element and the regular grid. This way, we can check only the elements that might be closed enough and ignoring all the rest.

We also note that there are other optimization passes that can be implemented. We could reposition the grid point towards the center of the surrounding ones. If we happens to generate a edge that is too short, we could also simply shrink it and delete unnecessary elements. If we further tweak the parameters of finding closed by elements, we could improve the quality of the mesh directly generated without relying too much on optimizations.