GridEditor

Design document

Technical Reference Manual

Luca Carniato

Version: 001
SVN Revision: 00

2 September 2020

GridEditor, Technical Reference Manual

Published and printed by:
Deltares
Boussinesqweg 1
2629 HV Delft
P.O. 177
2600 MH Delft
The Netherlands

telephone: +31 88 335 82 73
fax: +31 88 335 85 82
e-mail: info@deltares.nl
www: https://www.deltares.nl

For sales contact:
telephone: +31 88 335 81 88
fax: +31 88 335 81 11
e-mail: software@deltares.nl
www: https://www.deltares.nl/software

For support contact:
telephone: +31 88 335 81 00
fax: +31 88 335 81 11
e-mail: software.support@deltares.nl
www: https://www.deltares.nl/software

Contents

List of Figures

1 Introduction

The Gridgeom library is a C++ dynamic library that performs generation and manipulations of 2D grids. The library originates from the D-Flow FM Fortran code base (interactor). The code was re-written in C++ for the following reasons:

- ◇ Introduce some design in the existing algorithms by separating concerns.
- ◇ Introduce unit testing.
- ◇ Enabling the visualization of the meshes during creation (interactivity).
- ◇ Simplify the connection with C like languages, such C#. C++ and C# have the same array memory layout and pointers can be passed seamlessly, without the need for pointer conversion.

This document describes the development strategies (section 2), the Gridgeom design (section 2), and the responsibilities of the main classes (section 4 and ahead).

2 Gridgeom development strategies

The following strategies were adopted when developing the Gridgeom in C++. Note that the following strategies are not immutable and can be revised. Performance:

- ◇ Avoid abstractions of low-level components: when iterating over millions of nodes, edges or faces polymorphic calls are expensive. Compile-time static polymorphism could be used as an alternative to run-time polymorphism but has not been applied yet.
- ◇ to achieve the same level of performance as the Fortran code, contiguous arrays are extensively used (std::vectors).
- ◇ Computed quantities are often cached in private class members.

Coding:

- ◇ Internal class members are prefixed with m_ to distinguish them from non-class members.
- ◇ Classes and class methods start with capital letters.
- ◇ Input parameters in methods or functions come first, output parameters come second.
- ◇ Whenever possible methods or functions return a successful state, to allow the calling methods to act based on the return value.
- ◇ Pass by reference non-primitive types, to avoid deep copies.
- ◇ Use the const qualifier whenever possible.
- ◇ Avoid overloading copy constructors and assignment operators because the compiler can generate them automatically.

Re-usability:

- ◇ Promote reusability by separating code logic in separate functions or classes.
- ◇ Promote reusability by using templates (the same logic can be used for different types).

Resource management:

- ◇ Avoid the use of new or malloc to allocate heap memory. Use containers instead, such as std::vectors that auto-destroy when out of scope.
- ◇ Use resource acquisition is initialization (RAII).

Usage of external libraries:

- ◇ Promote the use of libraries when possible in this order: standard template library (stl algorithms operating on containers), boost, and then others.

3 Gridgeom design

The design of a geometrical library is influenced by the choices made for the representation of the mesh entities. In Gridgeom an unstructured mesh is uniquely defined by two entities:

- ◇ The nodes vector: represented using an std::vector<Point>, where Points is a structure containing the point coordinates (cartesian or spherical).
- ◇ The edges vector: represented using an std::vector<std:: pair<int,int», containing the start and the end index of the edges in the Nodes vector.

All other mesh properties are computed from these two entities, such as the face nodes, the face edges, the faces mass centers, and the faces circumcenters. See section 5 for some more details.

The library is separated in an API namespace (GridGeomApi) used for communication with the client and a backend namespace (GridGeom), where the classes implementing the algorithms are included (Figure 1). The API namespace contains the library API methods (Gridgeom.cpp) and several structures used for communicating with the clients. These structures are mirrored in the client application and filled with appropriate values.

An example of the mesh refinement algorithm execution is shown in Figure 2. When the client application creates a new mesh two API calls are required: in the first call (ggeo_new_mesh) a new entry in the meshInstances vector is pushed, in the second call (ggeo_set_state) the Mesh class is created and assigned to the entry of the meshInstance vector pushed before. Now the mesh is stored in the library and ready to be used by the algorithms.

The client now calls the ggeo_refine_mesh_based_on_samples function. In the local scope of the function an instance of the MeshRefinement class is created, the Refine method executed and the resulting mesh saved in the meshInstances vector. Finally, the last state of the mesh is retrieved using the ggeo_get_mesh function, whereby all information required for rendering the new mesh (nodes, edges, and faces) are retrieved from the meshInstances vector and returned to the client.

By using this design only the mesh instances are saved throughout the executions and all other algorithm classes (modifiers) are automatically destroyed when the algorithm execution is complete. Exceptions are the algorithms that support interactivity. In these cases, the algorithms are divided into several API calls, and their instances survive until an explicit ”delete” API call.

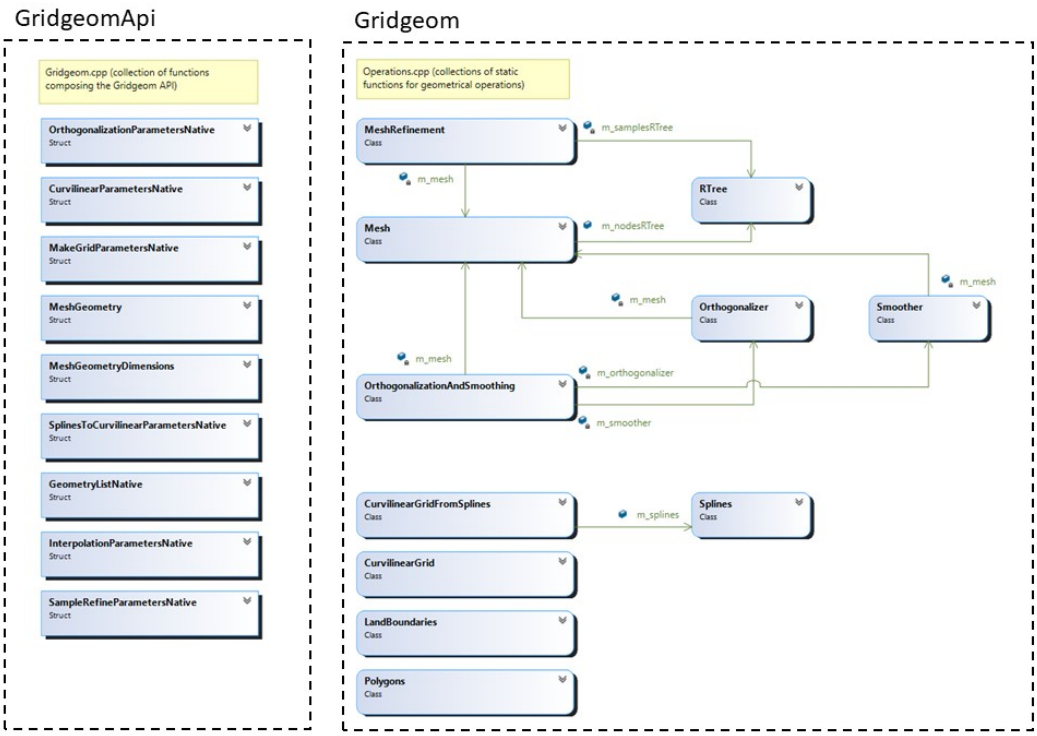

Figure 3.1: *Gridgeom library simplified class diagram. Rectangles with right corners represent structures, rectangles with round corners represent classes and yellow rectangles represent collections of methods (e.g. the API interface or collections of static functions).*

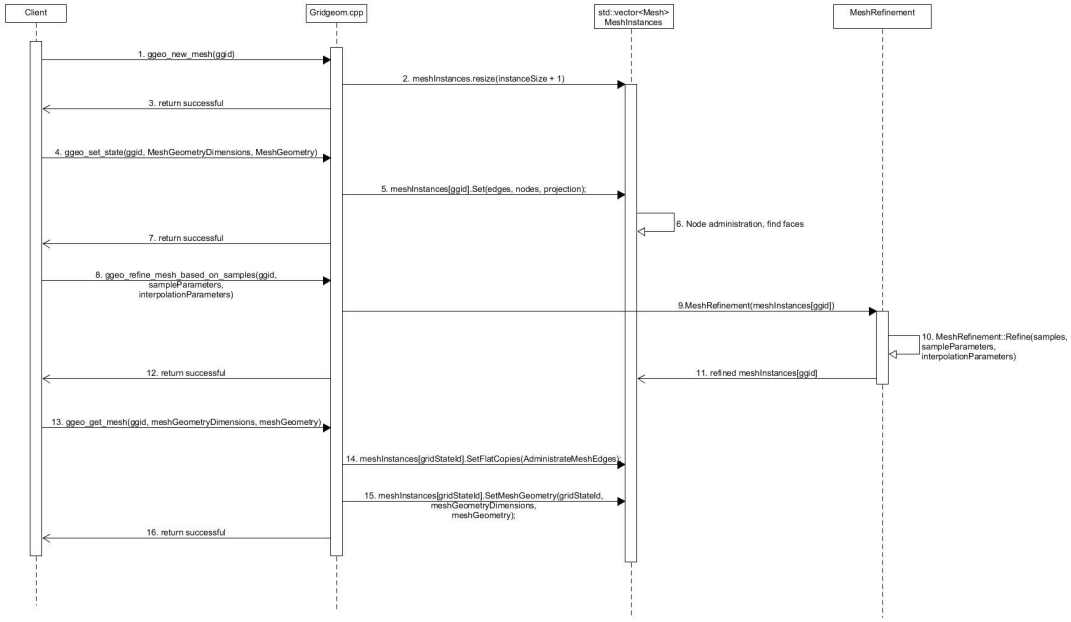

Figure 3.2: *Sequence diagram for creating a new grid and performing mesh refinement.*

4 The Operations file

In the current implementation, several geometrical methods acting on arrays or simple data structures (e.g. Points) are collected in the Operation.cpp file. This choice was made for reusing such methods in several other classes. Operations include:

- ◇ Vector dot product.
- ◇ Resize and filling a vector with a default value.
- ◇ Find the indexes in a vector equal to a certain value.
- ◇ Sort a vector returning its permutation array.
- ◇ Root finding using the golden section search algorithm.
- ◇ Performing coordinate transformations.
- ◇ Inquire if a point is in a polygon.
- ◇ The outer and inner products of two segments.
- ◇ Compute the normal vector of a segment.
- ◇ Compute the distances and the squared distances.
- ◇ Compute the circumcenter of a triangle.
- ◇ Compute if two lines are crossing.
- ◇ Interpolating values using the averaging algorithm.

All operations reported above supports cartesian, spherical, and spherical accurate coordinate systems.

5 The mesh class

The mesh class represents an unstructured mesh. When communicating with the client only unstructured meshes are used. Some algorithms generate curvilinear grids (see section 9), but these are converted to a mesh instance when communicating with the client. The mesh class has the following responsibilities:

- ◇ Construct the mesh faces from the nodes and edges and other mesh mappings required by all algorithms (Mesh::FindFaces). Mesh::FindFaces is using recursion to find faces with up to 6 edges. This is an improvement over the Fortran implementation, where several functions were coded for each face type by repeating code (triangular face, quadrilateral face, etc..).
- ◇ Enabling mesh editing, namely:
  - □ Node merging
  - □ Node insertion
  - □ Moving a node
  - □ Inserting edges
  - □ Deleting edges
  - □ Merging nodes (merging two nodes placed at very close distance).

  These algorithms can be separated into different classes.
- ◇ Converting a curvilinear grid to an unstructured mesh (converting constructor).
- ◇ Holding the mesh projection (cartesian, spherical, or spherical accurate).
- ◇ Making a quad mesh from a polygon or from parameters (this algorithm can be separated into different classes).
- ◇ Making a triangular mesh from a polygon (this algorithm can be separated to a different class).

This algorithm introduces a dependency on the Richard Shewchuk Triangle.c library. The mesh class stores a reference to an RTree instance. RTree is a class wrapping the boost::geometry::index::rtree code, implemented in SpatialTree.hpp. This is required for inquiring adjacent nodes in the merging algorithm.

6 The mesh OrthogonalizationAndSmoothing class

This class implements the mesh orthogonalization and smoothing algorithm as described in D-Flow FM technical manual (consult this manual for the mathematical details on the equations). The algorithm is composed of two differential equations: the first equation maximizes orthogonalization between edges and flow links and the second equation reduces the differences of the internal mesh angles (mesh smoothness). For this reason, the OrthogonalizationAndSmoothing class is composed of a smoother and an orthogonalizer, where the nodal contributions are computed by separate classes, as opposed to the original Fortran implementation. Essentially, the algorithm executes two loops:

- ◇ An outer loop, which itself is composed of the following steps:
  1. Computation of the orthogonalizer contributions.
  2. Computation of the smoother contributions.
  3. Allocation of the linear system to be solved.
  4. Summation of the two contributions (matrix assembly). The two contributions are weighted based on the desired smoothing to orthogonality ratio. OpenMP thread parallelization is used when summing the terms (loop iterations are independent).
- ◇ An inner iteration: the resulting linear system is solved explicitly. The nodal coordinates are updated and the nodes moving on the mesh boundary are projected to the original mesh boundary. If the project to land boundary flag is activated, the boundary nodes are projected to the land boundaries. Also in this case OpenMP parallelization is used in OrthogonalizationAndSmoothing::InnerIteration() because the update of the nodal coordinates has been made iteration-independent.

Gridgeom API has five functions to enable the client to display the mesh during the computations (interactivity). These functions are:

- ◇ ggeo_orthogonalize_initialize
- ◇ ggeo_orthogonalize_prepare_outer_iteration
- ◇ ggeo_orthogonalize_inner_iteration
- ◇ ggeo_orthogonalize_finalize_outer_iteration
- ◇ ggeo_orthogonalize_delete

The execution flow of these functions is shown in Figure A1 of the Appendix. Additional details about these functions can be retrieved from the API documentation.

7 The MeshRefinement class

Mesh refinement is based on iteratively splitting the edges until the desired level of refinement or the maximum number of iterations is reached. Refinement can be based on samples or based on a polygon. The refinement based on samples uses the averaging interpolation algorithm to compute the level of refinement from the samples to the centers of the edges. At a high level, the mesh refinement is performed as follow:

- ◇ Flag the nodes inside the refinement polygon.
- ◇ Flag all face nodes of the faces not fully included in the polygon.
- ◇ Execute the refinement iterations
    1. For each edge store the index of its neighboring edge sharing a hanging node (the so-called brother edge). This is required for the following steps because edges with hanging nodes will not be divided further.
    2. Compute edge and face refinement masks from the samples.
    3. Compute if a face should be divided based on the computed refinement value.
    4. Split the face by dividing the edges.
- ◇ Connect the hanging nodes if required, thus forming triangular faces in the transition area.

As with OrthogonalizationAndSmoothing, MeshRefinement modifies an existing mesh instance.

8 The spline class

The spline class stores the corner points of each spline. Besides the corner points, the derivatives at the corner points are also stored. The coordinates of the points between the corner points are computed in the static method Splines::Interpolate.

9 The CurvilinearGridFromSplines class

In this class, the algorithm to gradually develop a mesh from the centreline of the channel towards the boundaries is implemented. It is the most complex algorithm in the library. The curvilinear mesh is developed from the center spline by the following steps:

- Initialization step
  - The splines are labelled (central or transversal spline) based on the number of corner points and the intersecting angles.
  - The canal heights at a different position along the central spline are computed from the crossing splines.
  - The normal vectors of each m part are computed, as these determine the growing front directions.
  - The edge velocities to apply to each normal direction are computed.
- Iteration step, where the mesh is grown of one layer at the time from the left and right sides of the central spline:
  - Compute the node velocities from the edge velocities.
  - Find the nodes at the front (the front might miss some faces and be irregular).
  - Compute the maximum growth time to avoid faces with intersecting edges.
  - Grow the grid by translating the nodes at the front by an amount equal to the product of the nodal velocity by the maximum grow time.
- Post-processing
  - Remove the skewed faces whose aspect rati0 exceeds a prescribed value.
  - Compute the resulting CurvilinearGrid from the internal table of computed nodes (m_gridPoints).

to support interactivity with the client, the original Fortran algorithm was divided into separate API calls:

- ggeo_curvilinear_mesh_from_splines_ortho_initialize, corresponding to the initialization step above.
- ggeo_curvilinear_mesh_from_splines_iteration, corresponding to the iteration step above.
- ggeo_curvilinear_mesh_from_splines_ortho_refresh_mesh, corresponding to the post-processing above, plus the conversion of the CurvilinearGrid to an unstructured mesh.
- ggeo_curvilinear_mesh_from_splines_ortho_delete, necessary to delete the CurvilinearGridFromSplines instance used in the previous API calls.

A Appendix

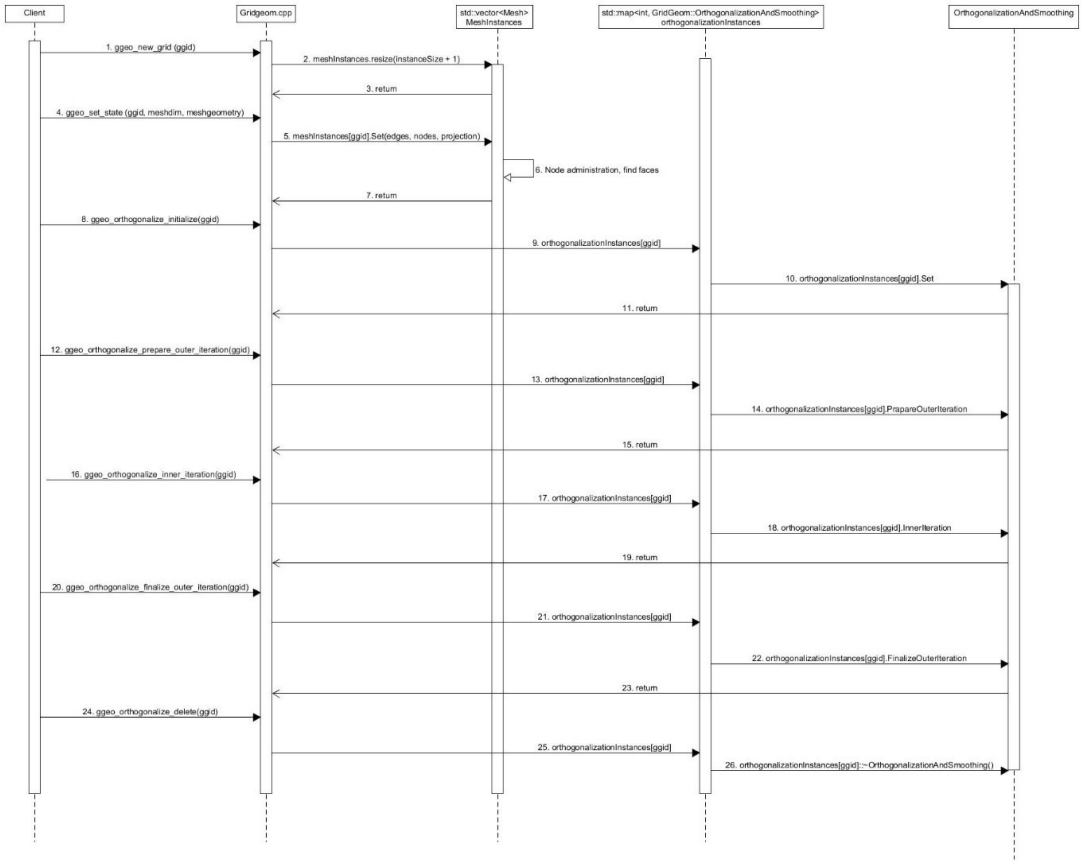

Figure A.1: *Sequence diagram for orthogonalization and smoothing.*