BGLgeom library

Speranza Ilaria (matr. 854196)
Tantardini Mattia (matr. 858603)

February 14, 2017

Contents

1 Introduction

The purpose of the project is to extend the Boost Graph Library (BGL) providing it more functionalities and making it capable to handle graphs with geometric properties, that BGL currently does not support. This mainly means to provide a graph from BGL, that already implements all topological operations, a way to describe its vertices as points and its edges as generic curves in the space (2 or 3 dimensional). Moreover, classes which implement these kind of geometrical properties must be able to carry out geometric and analytic operations, such as computation of first and second derivative of the curves describing edges and creation of numerical meshes on them.
All the geometric functionalities we developed in the library are aimed at solving numerical problems which can be modelled using a graph, but which also are in a geometric setting or need it: for instance, to compute flows in a network of blood vessels, meshes are required to solve finite element problems, and so it can be very useful to couple the graph description with the geometric one.
A lot of different applications can take advantage from this library; in particular, we provide two examples: one that creates a graph computing the intersections among fractures in a fractured porous media, and the other one that solves a diffusion problem on a vascular network embedded in biological tissue.

2 The library

2.1 Briefs on BGL

The BGL is a header-only template library able to create, handle and operate on graphs. It implements some different classes of graphs, all the topological operations concerning them, and a wide variety of algorithms.

Among all these functionalities, we decided to focus on a single class to describe a graph and on how to implement in an easy to use way the geometrical properties. To explain in more details our implementation choices, we need to spend some words on how that class and the graph properties work.

2.1.1 The adjacency_list class

`boost::adjacency_list` is a template class (described in the header *boost/graph/adjacency_list.hpp*) which represents a graph with a two dimensional structure: a `VertexList` and an `OutEdgeList` container. The first one stores all the vertices of the graph, and each vertex contains the other one-dimensional structure which is the list of all the out-edges leaving that vertex (so only out-edges if the graph is directed, all the edges if the graph is undirected).
The class has five template parameters and the full protoytpe is: `boost::adjacency_list< OutEdgeList, VertexList, Directed, VertexProperties, EdgeProperties >`.
The first two template parameters allow to choose the types of the underlying containers for the bidimensional structure. Choosing them affects space complexity of the graph and efficiency of some operations, such as inserting and removing edges. In our work we always set them to the selector `boost::vecS` (that stands for `std::vector`) for ease of use (all the vertex descriptors become `unsigned int`) and since, from BGL documentation, it is on average the best performing choice for every topological operation.
The third template parameter obviously allows to choose between a directed or undirected graph

through the selectors `boost::directedS` or `boost::undirectedS`.
The last two template parameters are the most interesting ones: they allow to choose what are the vertex and the edge properties. The BGL provides an easy way to handle them: the so called *bundled properties*. They simply are structs, which can be passed as template parameter to the `adjacency_list` class, and they will become its vertex and edge properties, along with all their attributes and memeber functions. This gives a lot of flexibility: for instance, if we want all the vertices to have as properties three doubles, an int, two strings and a member that returns a random number, we only have to define them all in the same struct and pass it as the fourth template parameter of the `adjacency_list` class. The same for the edges, passing the corresponding struct as fifth template parameter. Moreover, the usage of a struct, instead of a class, enables public member access, and this comes out in a slight more ease of use when accessing the properties.
Concluding this description, we underline that choosing different values for the template parameters consists in changing the type of the graph, with consequences on some other tools provided by BGL that we are going to explain.

2.1.2 Vertex and edge descriptors and iterators

Two specific handles are provided to manipulate vertices and edges: the vertex and the edge descriptors. They may be of different types, depending on the graph type. They in general 'refer' to a particular vertex or edge, thus allowing for instance to access their properties, or to specify in a very readable way topological operations.
Two more tools are provided to access a graph: vertex and edge iterators. As the descriptors, the iterators can be of different types depending on the graph type. Specific function return the iterators to the first and the last vertex and to the first and the last edge in the graph, thus allowing graph traversal. Dereferencing an iterator, the descriptor of the vertex or edge pointed by that iterator is obtained.
Both descriptors and iterators are always accessible through the `boost::graph_traits` class, defined both in *boost/graph/adjacency_list.hpp* and in *boost/graph/graph_traits.hpp* headers. We also provide, in the *graph_access.hpp* header, some aliases for descriptors and iterators templated directly on the graph's type, which are easier to handle.

2.1.3 Accessing properties

Using *bundled properties*, BGL provides an easy way to acces vertex or edge properties: just use the `operator[]` on a graph object, with index the descriptor of the vertex or edge of interest; this consists in accessing the struct used as vertex or edge property. Since a struct has public access, with the dot operator we can immediately read from or write in each attribute or call a member function.

2.2 BGLgeom

This is the library we developed to meet the goals of the project. The name is the union of "BGL", since this wants to be an extension of it, and of "geom", that stands for "geometry", to suggest the aim of the main functionalities implemented.

The library comes from the need to have a common environment where to build and run very different types of applications, but with a geometrical setting and an underlying graph structure in common. This implies, besides the development of the geometric properties, the implementation of input/output utilities, to make this library produce useful outputs for other

softwares. From this point of view, this library can be a common starting point for different projects and applications.

The library consists in a few source files to be compiled in a real library, and in some more header files which contain all the template classes and functions. Thus it is not a header-only template library.

2.2.1 What is inside

The library contains:

- **Adapters for BGL**: some layers and additional functions to hide the most used native BGL ones and to improve readibility and ease of use.
- **Classes to build graph properties**: all the classes needed to create a vertex and an edge property including the basic geometric requirements.
- **Geometrical and numerical utilities**: we included in this library some code to compute integrals, generate meshes, compute intersections between linear edges.
- **I/O utilities**: we provide one reader class to read tabular ASCII files, and three writer classes to produce three different types of output: ASCII, .pts and .vtp files.
- **Tests**: source code examples to show how the main classes and writers work.

2.2.2 Geometric properties

In this library we do not provide any predefined class of graph. We decided to use the `adjacency_list` class directly from the BGL, since this already gives a lot of flexibility. We focused on implementing the geometrical properties we were required as *bundled properties* (that are really flexible too), so that we could simply pass them as template parameters to create a graph with the desired features. Moreover, we chose to put in our properties only the few functionalities that are really needed for a geometrical description, leaving the user the freedom to add its own characteristics and functionalities.
The geometric properties we were asked to developed are:

- Coordinates for points.
- Boundary conditions.
- Parameterization for the edges, along with computation of the main characteristics: evaluation, first and second derivative, curvature, curvilinear abscissa.
- Meshes on the edges.

All these elements (but the boundary conditions) are templated on the dimension of the space.

Points. To represent points we used `Eigen` matrices, with 1 column and n rows. We chose `Eigen` because it provides high efficiency and a lot of built-in linear algebra methods.

Boundary conditions. We implemented a class to store a boundary condition described as a type and a value. We provided the `enum class` called `BGLgeom::BC_type` to represent the most common types of boundary conditions, labeled as DIR, NEU, MIX, NONE, INT ("internal", between two edges), OTHER.

Edge geometries. Concerning the geometry and parameterization of the edges, we were asked to be as generic as possible, that is the library should have been able to manage any possible parameterization of the type

$$f : \mathbb{R} \to \mathbb{R}^n \quad , \quad n = 2, 3,$$

i.e. from a line to a generic curve in the plane or in the space. We met this goal by developing three different classes: the `linear_geometry` class for the straight lines, and the `generic_geometry` and `bspline_geometry` classes for a generic curve.

Meshes. We provide a class that is at the same time a mesh generator and a container for two types of mesh: the parametric one, generated from the parameterization of the edge, and the real one, that is the evaluation of the parametric mesh on the same edge. We decided to store also the parametric mesh since it may be useful to evaluate it to get other quantities, such as curvature, in correspondence of the points of the real mesh.

2.2.3 Edge geometries

All geometry classes derive from an abstract class, `edge_geometry`, which specifies the functionalities that the geometry of an edge should have: in our case, they are the evaluation of the curve, the computation of first and second derivative, curvature and curvilinear abscissa at given value (or a vector of values) of the parameter. All the classes hold a parameterization of the curve between 0 and 1, so we actually have parameterizations of the type

$$f : [0, 1] \to \mathbb{R}^n \quad , \quad n = 2, 3.$$

for each geometry. We made this choice to have a common interface for the three geometries. All classes perform controls on the value of the parameter: if it is out of bounds, the class will show an error message and abort the program. All the three classes are templated on the dimension of the space.

Linear geometry. The `linear_geometry` class obviously describes straight lines. It stores as internal attributes the coordinates of the source and the target of the edge it describes. This may be memory inefficient in large graphs, since the information about the coordinates are already stored in the vertices, but otherwise recovering that information would have required a lot of operations and a more complex structure . In this way we give directly access inside the class. The coordinates of source and target are used to rescale the parameterization from [0,1] to the real position in the space. It is a very efficient class for straigth lines, since it performs only simple algebraic operations to compute evaluation, first derivative and curvilinear abscissa, and returns directly zero for the second derivative and the curvature.

Generic geometry. The `generic_geometry` class can store the exact parameterization of any curve. We implemented it using the `std::function` wrapper, and it requires to provide the exact analytic expression of the curve and of its first and second derivative, each one parameterized between 0 and 1 (it is care of the user to do this). The evaluation members just evaluate the functions stored in the private attributes.

Bspline geometry. The `bspline_geometry` class stores a description of the curve as a bspline, giving it all the analytic properties of the bspline curves. The class stores as private attributes the control points and the knot vectors for the curve and for its first and second derivatives (that are bsplines as well by property); this implies that the class is more memory consumptive. To

evaluate the curve and the derivatives, the class performs some more computation than in the other classes.
Two ways to build the representation are provided, both requiring a vector of points to be built, coded in the `enum class` called `BSP_type`. Specifying in the constructor the argument `BSP_type::Approx`, the given vector of points will be considered as a vector of control points for the curve (so that the curve smooths the control points), while specifying `BSP_type::Interp` the same vector of points will be considered as a vector of points to be interpolated by the bspline. In the first case the control points are simply copied in the inner private attribute and a uniform knot vector is built. Also in the second case a uniform knot vector t is built by default; after that, Greville abscissae x are computed in the following way:

$$x_j = \frac{1}{3}\sum_{k=1}^{3} t_{j+k} \qquad j = 1, ..., n$$

These are the sites where the equality $B(\mathbf{x}) = \mathbf{p}$ must hold, being $\mathbf{p}$ the input vector of points and B the bspline to be found. This interpolation problem can be easily transferred into the linear system $V\mathbf{d} = \mathbf{p}$, where V is called Vandermonde matrix of the intepolation problem, $\mathbf{d}$ is the vector of the unknown control points, and $\mathbf{p}$ is the vector of the given points to be interpolated. We use Eigen matrices to store all these elements, and the `lu()` Eigen method to solve the system. Two last remarks: in the approximating case, a constructor taking an arbitrary (i.e. non-uniform) knot vector in addition to the control points is available; concerning the interpolating case, the method we implemented works only with uniformly distributed knots and for cubic bsplines.

2.2.4 Building a geometric graph

Geometric base properties. In the *base_properties.hpp* header file we developed two structs which are thought to be the base vertex and edge property for a graph which needs a geometric description: the `Vertex_base_property` and `Edge_base_property` structs. They are substantially two containers, which gather the previous described classes. In the `Vertex_base_property` we included:

- The coordinates of the point in the space representing the vertex.
- A `std::array` to store the boundary conditions on that vertex. We templated the array on the number of the boundary conditions the user needs, since it may happen for some applications that more than one FEM problem has to be solved on the graph, thus requiring more than one boundary condition for each vertex.

In the `Edge_base_property` we included:

- One among the three geometry classes.
- A mesh class.

The `Edge_base_property` is templated on the edge geometry and on the dimension of the space. Both properties are also provided with two more attributes: an index and a label. This may be useful to give a specific ordering to vertices and edges and to keep track of particular parts of the graph.

A first example. To create a geometric graph with the properties mentioned above, we only have to declare an object of class `adjacency_list` with `Vertex_base_property` as fourth template parameter and with `Edge_base_property` as fifth template parameter. For instance:

```
#include <boost/graph/adjacency_list.hpp>
#include "base_properties.hpp"

using Edge_prop =
  BGLgeom::Edge_base_property< BGLgeom::linear_geometry<3>, 3 >;
using Vertex_prop = BGLgeom::Vertex_base_property<3>;
using Graph =
  boost::adjacency_list< boost::vecS, boost::vecS,
  boost::directedS, Vertex_prop, Edge_prop >;
Graph G;
```

Note that we simply created an empty graph: all the properties, especially geometries for edges, have to be set up while building the graph. Moreover, note that, since the chosen geometry for the edge properties `Edge_prop` is `linear_geometry`, and since this `Edge_prop` is passed as template argument to the `adjacency_list` class, the whole graph will have edges with linear geometry.

Extending base properties. We created our first geometric graph with the properties provided by our library. But in the applications a lot of other quantities and parameters may be necessary, for instance diameter and pressure for a blood vessel, or permeability for fractures, or any other kind of variable. How to expand the basic geometric properties of vertices and edges we provide to contain the new variables? Just publicly inherit from our base properties, and define at least a default constructor, since this is needed for the BGL functions that insert vertices and edges in the graph. A generic example is:

```
#include "base_properties.hpp"

struct My_edge_prop : public BGLgeom::Edge_base_property<3> {
  double variable1;     // may be pressure
  int varaible2;        // some flag
  std::string a_string; // a description
  my_class object;      // may be solver for FEM problem on edge

  My_edge_prop() : BGLgeom::Edge_base_property<3>(),
    variable1(.0),
    variable2(0),
    a_string(),
    object() {};
};
```

The use of inheritance is a fast way to derive your own vertex or edge property with a lot of freedom, and also allows access in a fast and easy way to the inherited geometric properties.

Adding vertices and edges. In our library we created some layers of functions already existing in BGL, handling the insertion and the deletion of edges and vertices: `BGLgeom::new_vertex` and `BGLgeom::new_edge` instead of `boost::add_vertex` of `boost::add_edge`. We did so in order to implement some useful overloads which allow to add a vertex or an edge along with its properties and to perform additional checks during the insertion of a new element (for instance, when adding a new vertex with its coordinates already set, there is the possibility to check whether a vertex with the same coordinates already exists; if so, its vertex descriptor is returned, instead of creating a new vertex). Furthermore, we added three new functions (`BGLgeom::new_linear_edge`,

`BGLgeom::new_generic_edge` and `BGLgeom::new_bspline_edge`), to provide an easy way to add a new edge to the graph already with its geometry set up in the proper way.
Here we present a simple example, in which we create a graph G with two vertices and one edge connecting them; we also show how to access the properties of edges and vertices, once they have been inserted in the graph. For further details, see the source code *test_BGL_vs_BGLgeom.cpp*.

```
#include "graph_access.hpp"
#include "graph_builder.hpp"
#include "base_properties.hpp"
#include "point.hpp"

using Edge_prop =
  BGLgeom::Edge_base_property< BGLgeom::linear_geometry<3>, 3 >;
using Vertex_prop = BGLgeom::Vertex_base_property<3>;
using Graph =
  boost::adjacency_list< boost::vecS, boost::vecS,
  boost::directedS, Vertex_prop, Edge_prop >;

Graph G;
BGLgeom::Vertex_desc<Graph> src, tgt;
BGLgeom::Edge_desc<Graph> e;

src = BGLgeom::new_vertex(G); // add a new vertex
tgt = BGLgeom::new_vertex(G); // add another vertex
G[src].coordinates = BGLgeom::point<3>(1,1,1); // set src coordinates
G[tgt].coordinates = BGLgeom::point<3>(2,2,2); // set tgt coordinates

// add an edge connecting src to tgt
e = BGLgeom::new_linear_edge(src, tgt, G);
// set edge's index
G[e].index = 1;
```

2.2.5 Reader class

We provided in this library a generic ASCII reader to read topology and properties of the graph from a tabular file. The reader is a template abstract class. It is templated on the vertex and edge properties of the graph, thus making the same reader suitable for every kind of vertex and edge property a graph will have. There is a third template parameter, defaulted to an empty struct type, that represents the topological information one wants to read about the graph; it can be omitted in particular formats of file (for instance, specifying coordinates of source and target of an edge already tells us the topology; while if indexes for vertices and edges are provided, the topology can be handled in an easier way).
We left as pure virtual those members devoted to read data from file and to return the just read data as vertex or edge properties. While deriving his own concrete reader, the user has to specify:

- all the attributes to be stored while reading;
- in the member `get_data()`, how to read (in the right order) from the file the data for each single edge, using the inner stream attribute ;
- in the members `get_source_data()` and `get_target_data()`, how to pack the data read into a vertex property which will be assigned to the source and the target of the edge he is

currently reading;

- in the member `get_edge_data()`, how to pack the data read into the edge property which will be assigned to the new edge ;
- if topological information are provided, how to pack them in a struct that will contain this kind of information, otherwise he has to define the `get_topological_data()` as a member that returns an empty struct (the default `BGLgeom::no_topological_data` is provided in the same header).

Note that the reader is thought to read one edge at a time from the file and to return the vertex and edge properties related to the edge just read and to its source and target. In this way, it is easy to build a graph in a loop, reading one edge at a time from the file, adding it to the graph and immediately update its properties (and the properties of its source and target), before reading and adding a new edge. To check if a vertex that we want to insert is already present in the graph or not (this is useful not to have different vertex descriptors with the same coordinates), we provided the function `BGLgeom::new_vertex()` that, if the second argument is set as `true`, performs a check over all the vertices in the graph and if there is already a vertex with the same coordinates (within a tolerance set by the user, defaulted to 1^{-10}), then the function returns that vertex descriptor and does not insert a new one.
As final observation, we remember that the reader is able to read all kind of tabular files which have as delimiter among the various columns spaces or tabulations, but not commas (unless they are read as more data that will be discarded).

2.2.6 Writer classes

Three kinds of writers are provided to output the graph and its properties; each one produces a different format of output.

Writer_ASCII. The `writer_ASCII` class produces as output a tabular format (.txt, .dat, etc). It is an abstract class with the same philosophy as the `reader_ASCII` class: the user has to define its concrete writer, specifying which properties of the graph to print, and how. The pure virtual member devoted to this is `write_data()`, which wants as arguments the graph and the edge descriptor of the edge we want to print information about. Remember that from an edge descriptor there is the possibility to recover the source and target properties through their vertex descriptors, using the `BGLgeom::source()` and `BGLgeom::target()` functions. Also in this case, it will be easy to print information about the graph in a loop over all the edges.

Writer_pts. The `writer_pts` class produces a particular kind of output (.pts), with a particular format: it expresses the graph as a sequence of arcs, and for each arc the boundary conditions on source and target and all points of the mesh defined on it (if present, otherwise only source and target coordinates) are printed. We developed such a writer since we use the GetFem++ library in the application on diffusion in a vascular network, and the .pts is the input format for GetFem to read meshes, and then solve FEM problems.
To meet some requests from people that will use our library with the GetFem for further applications, we also developed some particular features for the writer. We provided it with an additional template parameter, the number of boundary conditions (defaulted to one): if specified different from 1, it will produce different output files, each one with the same structure but with the different boundary conditions. Another feature allows to print in a separate file some geometric information (first and second derivative, curvature) in correspondence of the points of

the mesh (if present) on each edge. Just specify `true` as second argument of the `export_pts()` method.

Writer_vtp. The `writer_vtp` class produces a .vtp output, i.e. a VTK polygonal data format. It is a particular format provided by the wider VTK library. It consists in an XML file in which all the topology of the graph is coded. The output consists into two files: one coding the edges, and one coding the vertices of the graph. This kind of output is thought to visualization purpouse: using for instance software such as Paraview, the output files can be easily displayed.

2.2.7 Future developments

Edge base property. The `Edge_base_property` we provide in the library is a kind of 'static' one: it contains one of the three geometries, and the type is chosen passing it through a template parameter. Then the `Edge_base_property` itself is passed as template parameter to create the graph. This implies that the geometry of all the edges in the graph will be fixed to the one used as template parameter. This behaviour may be inefficient for some applications, for instance when the underlying graph has almost all linear edges and only few edges with complex geometry. At this stage, the geometry of such a graph will be fixed to the generic or bspline one for all edges, thus giving some memory and computational inefficiencies on the edges that could have been described by the linear geometry. For this reason, it may be interesting to implement a kind of 'dynamic' version of the `Edge_base_property`, to make the user be able to select a different geometry for each different edge in the graph.

B-spline interpolation. We described in the previous section how we implemented the interpolation of given points with the bspline geometry. The knot vector is made uniform by default, and the system that allows to recover the control points is solved with the Eigen `lu()` method. Choosing different solutions for the knot vector or a different method to solve the system may improve efficiency.

3 Applications

In this section we present two examples from real situations, using tools from BGLgeom library to model and build the underlying graph and then to solve them. The first one concerns fractures in fractured porous media, while the second one concerns microcirculation in blood vessels.

3.1 Fracture Intersection

In this example a list of fractures, all lying on the same plane, is given as input. The goal is to build a graph representing the network, with vertices in correspondence of the fractures' extremities and of the intersection points and edges representing the fractures.

3.1.1 Intersection types

To compute the intersections between two given edges we used a piece of code, developed by Professor Formaggia, which we included as a utility in our libary (*intersections2D.hpp*), after adapting it to our specific data structures. In particular, the struct `Intersection` stores the number of intersections, the intersection points (if the intersection is not just a point but a segment, its two extremes are returned), two boolean variables, whose value depend respectively on the parallelism and the collinearity of the evaluated edges, the intersection type and some

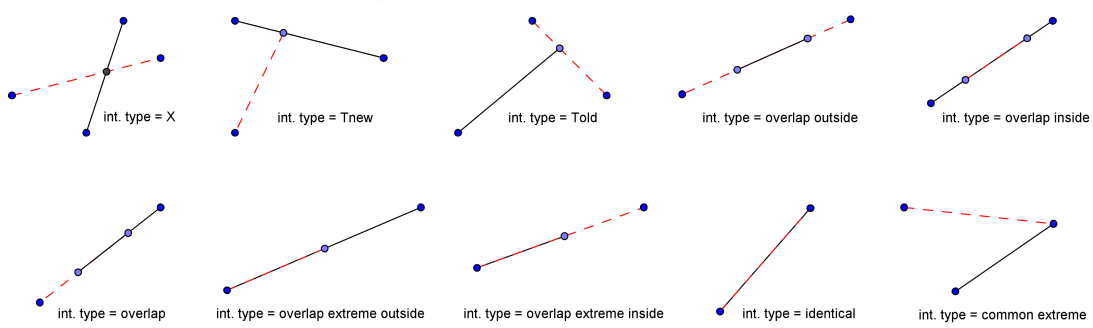


Figure 1: Intersection types. In black the edge already in the graph, in red the new one.

other information.
The intersection type is described by an `enum class` with 12 different values, 10 corresponding to real intersections and listed in figure 1, plus two handling the case with no intersections and the case of some unexpected behaviour. Depending on the intersection type, different actions will be performed during the graph construction.
For this specific application we also created in the *intersections2D_utilities.hpp* header file a struct called `Int_layer`, whose constructor takes as input an `Intersection` struct and computes from it some variables of interest:

- `how`: intersection type, directly copied from `Intersection`;
- `int_edge`: edge descriptor of the intersected edge;
- `int_pts`: intersection point(s), directly copied from `Intersection`
- `swapped_comp`: boolean equal to 1 when there are two intersection points and they are swapped while ordering the intersection vector;
- `intersected_extreme_old`: boolean equal to 0 if the source of the "old" edge (that already is in the graph) is an intersection point, equal to 1 if the target of the "old" edge is involved. This variable is used only when we already know from the intersection type that one and only one extreme of the "old" edge is an intersection point;
- `intersected_extreme_new`: boolean equal to 0 if the source of the "new" edge (the fracture currently added) is an intersection point, equal to 1 if the target of the "new" edge is involved. This variable is used only when we already know from the intersection type that one and only one extreme of the "new" edge is an intersection point.

3.1.2 The algorithm

Here we illustrate, through pseudocode and with a detailed description, how the function `create_graph()` works.

```
while(!eof){
  fracture = getline()
  create fracture edge_property
  int_vect.clear()
  for e in edges(G){
    compute_intersection(e, fracture)
    if intersection exists{
      create Int_Layer with intersection information
      int_vect.push_back(Int_Layer)
    }
  }
  if int_vect.empty(){
    add_edge to graph with fracture edge properties
  }
  else {
    reorder int_vect
    remove_duplicates inside int_vect
    refine_graph(fracture, int_vect)
  }
}
```

The file is read fracture by fracture: for each of them the physical properties are stored in the `Fracture::Edge_prop` struct and the line number is used as identifier; a vertex descriptor called `src` is associated to the first extreme of the fracture, another called `tgt` to the second one. These vertex descriptors can be either associated to already existing vertices in case of already existing coordinates or to new ones otherwise. After the initialization part, we loop over the edges in the graph: for each of them the function `compute_intersection()` checks for intersections with the current fracture; if so, the corresponding `Int_Layer` struct is created and pushed back in a vector which, by the end of the loop, will therefore contain all the intersections with the fracture we are currently reading.
Once the loop over the edges is completed, if the intersection vector is empty, a new edge is created between `src` and `tgt` and the edge_property is bundled to it, otherwise the vector is first reordered and then some "duplicates" are removed. The reordering phase consists in ordering the vector from the intersection point nearest to the source to the farthest one; also the two intersections contained in the same struct (in those cases in which the intersection is a segment and not a single point) are reordered according to the same criterion (and if they are swapped the boolean variable `swapped` in the `Int_layer` is switched from 0 to 1). The duplicates removal, performed by the function `remove_duplicates()`, consists in keeping just one single struct among all of those related to the same intersection point; in figure 2 an example is shown: before adding the new fracture (the red one), the graph has 5 edges and 6 vertices. Looping on the edges, the algorithm correctly detects one intersection with each edge, but there is only one intersection point, E, and no edge has to be modified by the insertion of the new fracture, therefore all the "duplicates" are discarded but one. Note that this happens just in case of multiple intersections of type "Told" or "common extreme".
Now the vector is ready to be processed. We start passing `src` and the first struct of the vector to the function `refine_graph()`: this function performs different actions depending on the intersection type; the common goal is to resolve all the changes due to the insertion of the new fracture and related to that specific segment. The function returns as output the vertex descriptor corresponding to the just created vertex in correspondence of the first intersection point; this descriptor will be passed as source in `refine_graph()` at the next iteration together with the

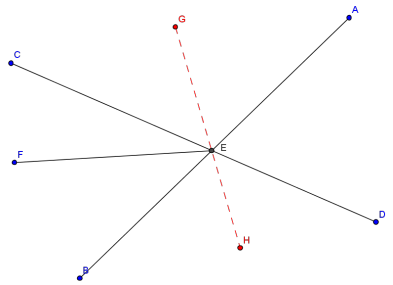


Figure 2: Example on duplicates removal

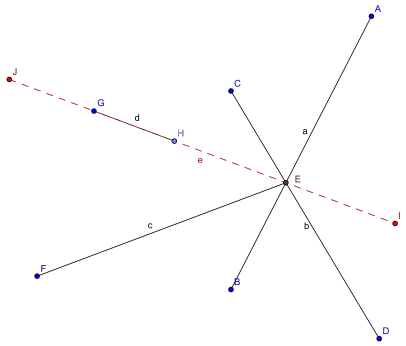


Figure 3: Fracture in red is going to be added to the graph

second intersection struct in the vector. This procedure goes on until the end of the intersection vector is reached; at the end, the vertex descriptor returned by the last call to `refine_graph()` is connected to `tgt`.
In case of partial or total overlapping between the new fracture and an existing edge, a conflict between the corresponding edge properties happens: it is up to the user to decide how to solve it, and this is possible passing as optional parameter to `create_graph()` a lambda function `update_edge_properties()` which will be used inside `refine_graph()` to merge old and new properties. The default behaviour is simply to keep the old property, totally discarding the new one.

3.1.3 A simple example

This example refers to figure 3 and it aims at illustrating how the algorithm works. We consider the insertion of the fifth fracture (e): the previous four (a,b,c,d) have been already added and the graph they created is made of 8 vertices and 6 edges (AE, BE, CE, DE, FE, GH). Since the

extremities of the new fracture have coordinates different from all the other existing points, two new vertices I and J are created. Looping over the edges in the graph, we obtain a vector of six intersection structs; we reorder it and we remove four duplicates in point E (see table 1). Now we are left with a vector of two components: first of all we pass to `refine_graph()` the vertex descriptor I (which is the source of the new fracture) and the first intersection struct (related to point E); being the intersection of type T_old, this function simply adds an edge between I and E and returns as next source E. At the second step, `refine_graph()` takes as input the current source (i.e. the output of the previous call to the function, E in this example) and the second intersection struct: in case of overlap_outside the source is connected to the first intersection (EH), the properties of the old edge (HG) are updated and the second intersection is connected with the target (GJ). In conclusion, the insertion of this new fracture led to the creation of three new edges (all labelled with letter e in order to keep track of the fact they all are part of the fifth fracture) and to the property update of an existing edge.

Int. edge	Type	Int. point(s)
AE	T_old	E
BE	T_old	E
FE	T_old	E
GH	overlap_outside	**G and H**
DE	T_old	E
CE	T_old	E

Int. edge	Type	Int. point(s)
AE	T_old	E
BE	T_old	E
FE	T_old	E
DE	T_old	E
CE	T_old	E
GH	overlap_outside	**H and G**

Initial setting

Ordered

Int. edge	Type	Int. point(s)
AE	T_old	E
GH	overlap_outside	G and H

Ordered and without duplicates
(Final setting)

Table 1: Simplified representation of the intersection vector

3.1.4 A more complex example

In the file `test_all_int_cases.txt` we put a list of fractures with no property associated: the main goal of this example is to show that the code performs correctly whatever is the intersection type it has to handle. Indeed, the sequence of fractures is built in such a way that almost every intersection type appears during the construction of the graph. In figure 4 we show how the graph is built step by step, one fracture at a time.

3.1.5 Handling "real" data

In order to test our algorithm, professor Formaggia provided us two lists of fractures. The input files' formats were different, so we implemented two specific readers, inherited by our `BGLgeom::reader_ASCII` class, to acquire the fractures' extremities and their properties in the right way. The source code is in file `main_fractures.cpp` and the graphical results are shown in figure 5.

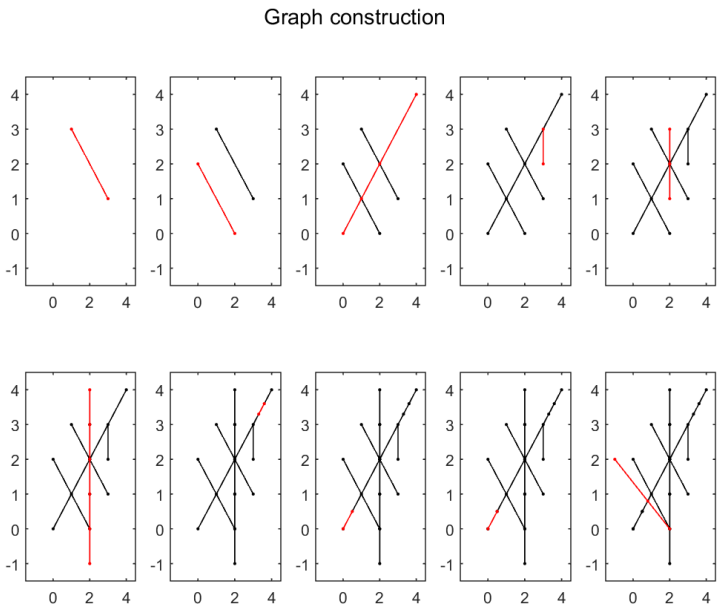
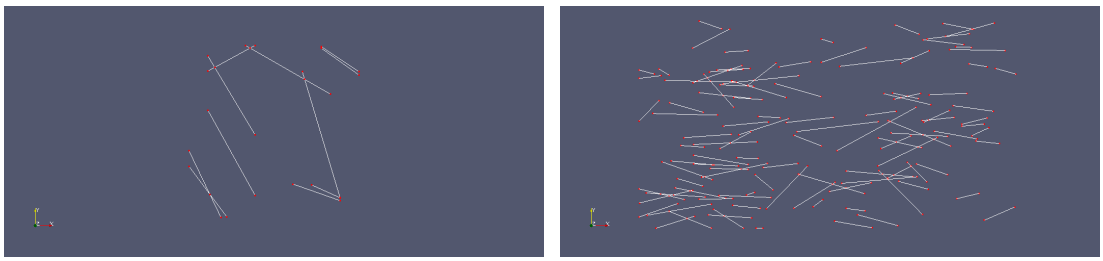


Figure 4: Fracture in red is going to be added to the graph



Input file: fractureElevenVF.dat Input file: fractures.txt

Figure 5: Plots of resulting graph on Paraview

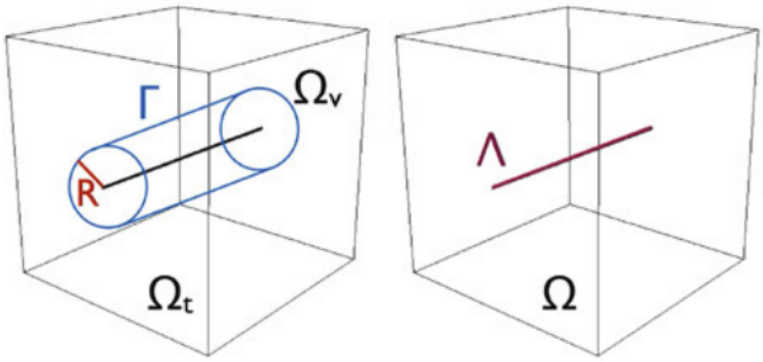


Figure 6: On the left the interstitial tissue slab with one embedded capillary; on the right reduction from 3D to 1D description of the capillary vessel.

3.2 Coupled problems on one-dimensional networks with application to microcirculation

3.2.1 Mathematical formulation

We present a prototype model for fluid transport in a permeable biological tissue perfused by a capillary network. The domain where the model is defined is composed by two parts, Ω_t and Ω_v (where t stands for *tissue* and v stands for *vessel*), denoting the interstitial volume and the capillary bed respectively. We assume that the capillaries can be described as cylindrical vessels and Λ denotes the centerline of the capillary network. The capillary radius, R, is for simplicity considered to be constant. We decompose the network Λ into individual branches Λ_i. Branches are parameterized by the arc length s_i; a tangent unit vector $\boldsymbol{\lambda}_i$ is also defined over each branch, defining in this way an arbitrary branch orientation. In particular, the domain Ω_f can be split into cylindrical branches Ω_i defined as follows

$$\begin{aligned}\Omega_i =&\{\boldsymbol{x} \in \mathbb{R}^3;\ \boldsymbol{x} = \boldsymbol{s} + \boldsymbol{r}\};\\ \boldsymbol{s} \in \Lambda_i =&\mathcal{M}_i(\Lambda' \subset \mathbb{R}^1)\};\\ \boldsymbol{r} \in \mathcal{D}_{\Lambda_i}(\boldsymbol{s}, R) =&\{r\boldsymbol{n}_{\Lambda_i};\ r \in (0, R)\};\end{aligned}$$

where $\mathcal{M}_i$ is a mapping from a reference domain Λ' to the manifold $\Lambda_i \subset \mathbb{R}^3$ and $\boldsymbol{n}_{\Lambda_i}$ denotes a unit normal vector to Λ_i. We denote with Γ_i the lateral surface of Ω_i.

The physical quantity of interest is the concentration of transported solutes c. It is defined as a field depending on time t and space, being $x \in \Omega$ the spatial coordinates. Furthermore, we denote with the subscript v their restriction to the capillary bed (vessels), and with t the restriction to the interstitial tissue. The derivation of our model stems from fundamental balance laws regulating the flow in the capillary bed, the extravasation of plasma and solutes and their transport in the interstitial tissue.

Mass transport is modeled by means of diffusion equations, both in the tissue and in the capillary bed. Furthermore, we want to couple the two problems with a leakage in the vessel walls; then, under the assumption that capillaries can be modeled as cylindrical channels, the magnitude of the mass flux exchanged per unit length between the network of capillaries and the interstitial volume at each point of the capillary vessels is the following:

$$J := k_i\,(c_v - c_t) \quad \text{on } \Gamma_i, \tag{1}$$

being k_i the permability of the vessel wall Γ_i.
Finally, using this notation, the problem becomes:

$$\begin{cases} -\Delta c_t = 0 & \text{in } \Omega_t, \\ -\Delta c_v = 0 & \text{in } \Omega_v, \\ -\nabla c_t \cdot n_t = \kappa_i \left(c_t - c_v\right) & \text{on } \Gamma_i, \\ -\nabla c_v \cdot n_v = \kappa_i \left(c_v - c_t\right) & \text{on } \Gamma_i, \\ \nabla \cdot c_t = 0 & \text{on } \partial\Omega_p \setminus \cup_{i=1}^{N} \Gamma_i, \\ c_v = g & \text{on } \Gamma_{inflow} \\ \nabla \cdot c_v = 0 & \text{on } \Gamma_{outflow} \end{cases}$$

The problem on Ω_v is a prototype of flow or mass transport problem in a network of cylindrical channels, surrounded by the domain Ω_v where another flow and transport problems will be defined, according to porous media equations. It is assumed that the interface between these regions is permeable, namely it is crossed by a normal flux proportional to $\kappa_i \left(u_t - u_v\right)$.
The boundary of Ω_v is divided as follows: $\partial\Omega_v = \Gamma_{inflow} \cup \Gamma_{outflow} \cup_{i=1}^{N} \Gamma_i$; then, we impose boundary conditions on Ω_v such that the mass goes from Γ_{inflow} to $\Gamma_{outflow}$.

Applying model reduction techniques, the equations defined on the network of inclusions Ω_v can be restricted to the one dimensional network Λ. Differentiation over the branches is defined using the tangent unit vector as $\partial_{s_i} = \nabla \cdot \boldsymbol{\lambda}_i$ on Λ_i, i.e. ∂_{s_i} represents the projection of ∇ along $\boldsymbol{\lambda}_i$.
As a result, the 3D-1D counterpart of the problem becomes

$$\begin{cases} a(c_t, q_t) + b_\Lambda(\overline{c_t}, \overline{q_t}) = b_\Lambda(c_v, \overline{q_t}) & \forall q_t \in H_0^1(\Omega_t), \\ A(c_v, q_v) + b_\Lambda(c_v, q_v) = b_\Lambda(\overline{c_t}, q_v) + \mathcal{F}(q_v) & \forall q_v \in H_0^1(\Lambda), \end{cases} \tag{2}$$

where $c_v, q_v \in H_0^1(\Lambda)$ denote trial and test functions relative to the weak reduced problem, with the following bilinear forms:

$$a(w, v) = (\nabla w, \nabla v)_\Omega, \quad A(w, v) = \sum_{i=1}^{N} (\nabla_{\Lambda_i} w, \nabla_{\Lambda_i} v)_{\Lambda_i}, \quad b_\Lambda(w, v) = \sum_{i=1}^{N} \kappa_i (w, v)_{\Lambda_i}.$$

A central role in equation (2) is played by the restriction operator $\overline{(\cdot)}$ that is defined as

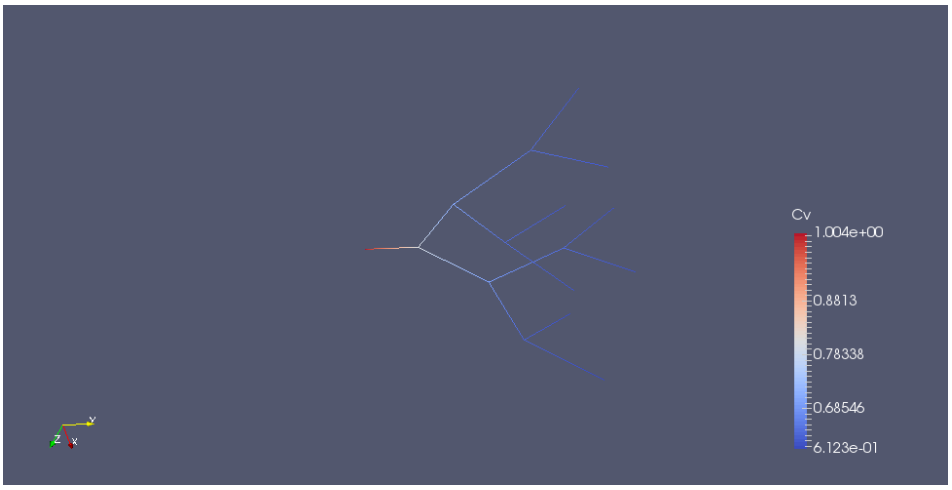
$$\overline{w}^{(i)} = \frac{1}{|\partial\mathcal{D}_{\Lambda_i}|} \int_{\partial\mathcal{D}_{\Lambda_i}} w|_{\Lambda_i \times \partial\mathcal{D}_{\Lambda_i}} \, dr \, d\theta \,.$$

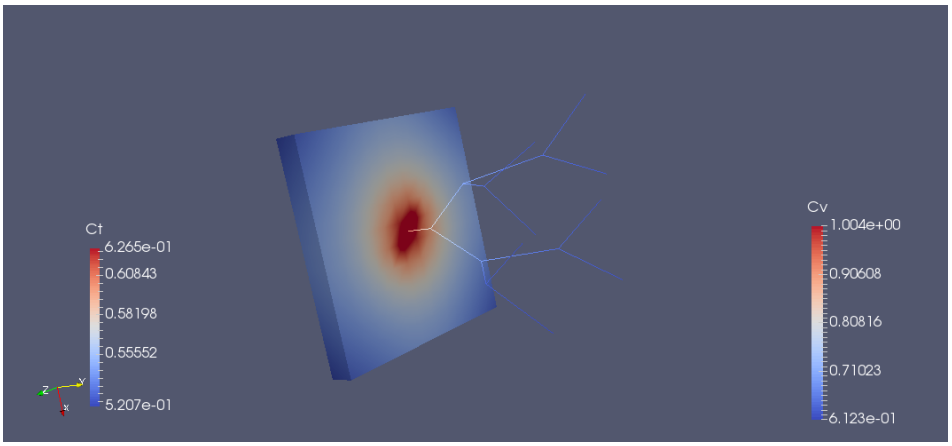
3.2.2 Code implementation and results

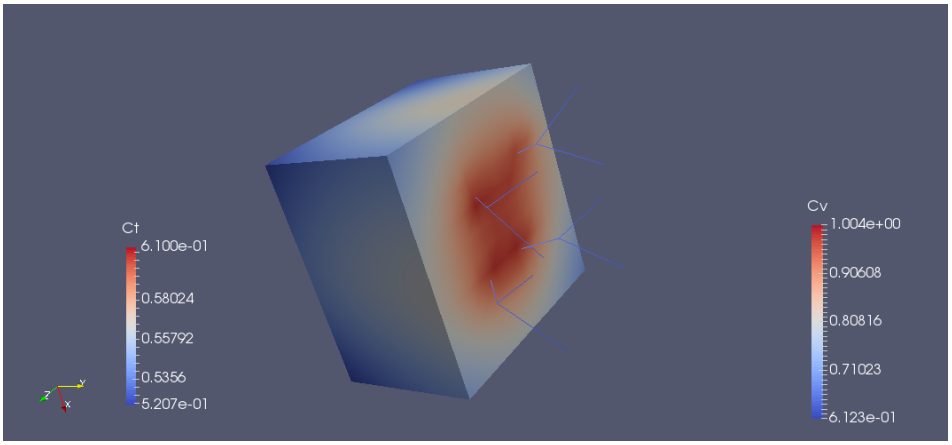
We use a C++ code that exhibits one-to-one correspondence with respect to the mathematical models derived. Specifically, we assemble the generic linear system using GetFEM++, an open-source general-purpose finite element library. For the results presented in this report we applied the direct solver SuperLU 3.03 to solve the monolithic linear system. The code has been written by Stefano Brambilla and it is part of another PACS project; the code can be found at the following GitHub page: `https://github.com/stefano-brambilla-853558/MANworks`.

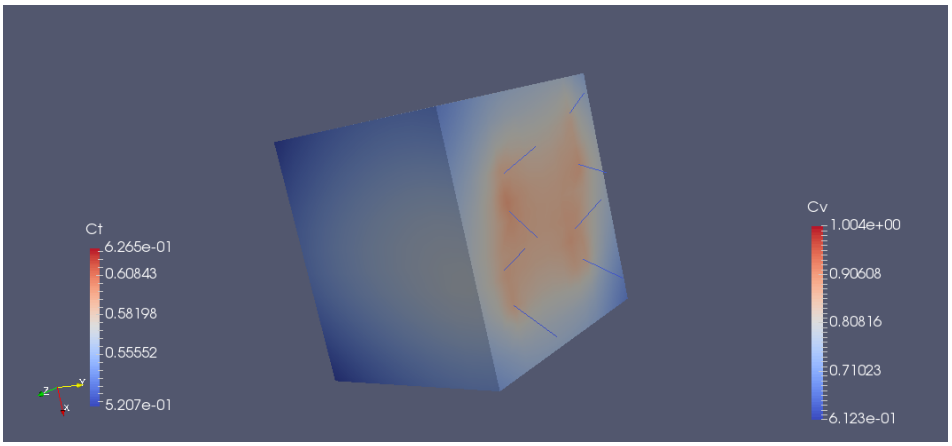
The code needs to read from a file containing the structure of the network: so we first developed a Matlab script to generate the coordinates of all the vertices of a network with successive

bifurcations on orthogonal planes. We used our library to read this file and construct the relative graph, then we created a 1D mesh on each edge, and in the end we used the graph and its properties to generate the `bifurcation_network.pts` file, containing the details of the 1D mesh and of the boundary conditions.
The results show that the mass diffuses into the network and then flows thwrough the vessel walls into the surrounding tissue.









4 References

- Jeremy Siek/Lie-Quan Lee/Andrew Lumsdaine, *The Boost Graph Library - User guide and reference manual*, Addison-Wesley, 2002.
- Tomas Sauer, *Splines in Industrial Applications*, 2007, concerning b-spline interpolation.
- *http://www.boost.org/doc/libs/1_61_0/libs/graph/doc/*, BGL website.
- *http://www.gnu.org/software/make/manual/make.html*, for some support on Makefile.
- *http://eigen.tuxfamily.org/dox*, Eigen documentation.
- *http://www.stack.nl/ dimitri/doxygen/manual/index.html*, Doxygen documentation.
- *http://www.vtk.org/Wiki/VTK/Examples*, some documentation and examples on usage of VTK library.
- *www.stackoverflow.com*.

5 Acknowledgements

We would like to thank our mate Stefano for his support, both practical and moral, in our PACS project. In particular, since he is working on another PACS project on that topic, he helped us a lot in writing the source code for the vascular microcirculation application, and he also gave us large support in writing the report part concerning that application, expecially the mathematical formulation.

We want also to thank Dr. Pasquale Africa for its technical support when nothing was compiling on our virtual machine. Always helpul, he gave us a lot of advice on how to write a good Makefile and some suggestions on how to design our code and the library structure.