BGL Interface

Generated by Doxygen 1.6.1

Mon Oct 24 17:21:34 2016

Contents

Chapter 1

Todo List

Class Geometry::Intersection It can be bettered by adding another attribute that indicates, in the case of two edges end which coincides the relative position on the edge. It requires a simple modification of the function segmentIntersect

Chapter 2

Class Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Label_map_t: the type of a std::map which key is a vertex descriptor and the value is an unsigned int which has the meaning of the current label of the component to which that vertex belongs to.
Components_map_t: the type of a std::map which key is an unsigned int used as label for the group and the value is a std::set containing all the vertex descriptor of the vertices that have that label, i.e that belong to the same component)**??**

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 circular_edge_geometry Class Reference

Public Member Functions

- **circular_edge_geometry** (point center_, double start_angle_, double end_angle_)
- point **value** (double parameter)

The documentation for this class was generated from the following file:

- include/circular_edge_geometry.hpp

5.2 data_structure< Edge_data_structure, Vertex_data_structure > Class Template Reference

An abstract class to handle the user definition data structure The users has to specify, in the derived class, all variables he need in order to store information read from the input file. Then, through the definition of Edge_data_structure and Vertex_data_structure, he can get separately all the information to put on edges and vertices.

`#include <data_structure.hpp>`

Public Member Functions

- virtual Edge_data_structure get_edge_data ()=0

  A method to get the right data to append to an edge.

- virtual Vertex_data_structure get_vertex_data ()=0

  A method to get the right data to append to a vertex.

- virtual void get_data_from_line (std::ifstream &, data_structure &)=0

  The way the data from the input file are read User has to specify how to read data from input file.

5.2.1 Detailed Description

template<typename Edge_data_structure, typename Vertex_data_structure> class data_structure< Edge_data_structure, Vertex_data_structure >

An abstract class to handle the user definition data structure The users has to specify, in the derived class, all variables he need in order to store information read from the input file. Then, through the definition of Edge_data_structure and Vertex_data_structure, he can get separately all the information to put on edges and vertices.

Parameters:

Edge_data_structure A struct where the user has to define type and name of the variables he needs to append to vertices as vertex bundled property

Vertex_data_structure A struct where the user has to define type and name of the variables he needs to append to edge as edge bundled property

Precondition:

It may be useful to declare a friend operator>> to help the reader read the data

The documentation for this class was generated from the following file:

- include/data_structure.hpp

5.3 edge_data_structure< dim > Struct Template Reference

Public Attributes

- unsigned int **edge_id** = 0
- std::string **label** = ""
- double **lenght** = 0
- double **capacity** = 0
- double **diameter** = 0
- double **weight** = 0
- generic_edge_geometry< dim > **edge_geo**

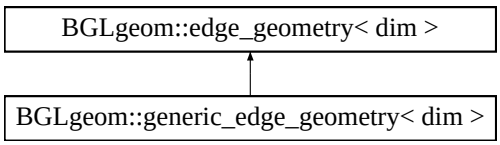
template<unsigned int dim> struct edge_data_structure< dim >

The documentation for this struct was generated from the following file:

- include/data_structure.hpp

5.4 BGLgeom::edge_geometry< dim > Class Template Reference

Inheritance diagram for BGLgeom::edge_geometry< dim >::



Public Member Functions

- virtual BGLgeom::point< dim > **value** (const double parameter)=0
- virtual std::vector< double > **first_derivatives** (const double x)=0
- virtual std::vector< double > **second_derivatives** (const double x)=0

template<unsigned int dim> class BGLgeom::edge_geometry< dim >

The documentation for this class was generated from the following file:

- include/edge_geometry.hpp

5.5 edge_parametrization Class Reference

This class holds the parametrization of the edge.

`#include <Forma_edge_property.hpp>`

5.5.1 Detailed Description

This class holds the parametrization of the edge.

The documentation for this class was generated from the following file:

- include/Forma_edge_property.hpp

5.6 edge_prop_max_flow_t Struct Reference

Public Attributes

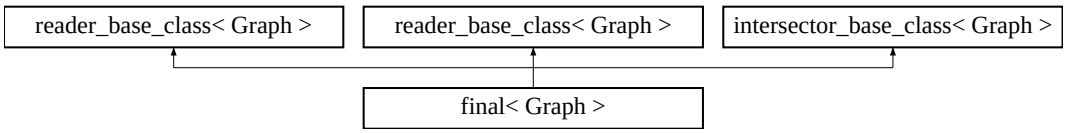
- double **capacity**
- double **residual_capacity**
- bool **original_edge**

The documentation for this struct was generated from the following file:

- include/edge_property_max_flow.hpp

5.7 final< Graph > Class Template Reference

Inheritance diagram for final< Graph >::



Public Types

- typedef reader_base_class< Graph >::Vertex_desc **Vertex_desc**
- typedef reader_base_class< Graph >::Edge_desc **Edge_desc**
- typedef intersector_base_class< Graph >::Edge_iter **Edge_iter**
- typedef intersector_base_class< Graph >::Intersections_type **Intersections_type**
- typedef reader_base_class< Graph >::Vertex_desc **Vertex_desc**
- typedef reader_base_class< Graph >::Edge_desc **Edge_desc**

Public Member Functions

- reader_Formaggia (Graph &_G)

  Default constructor (we need however to initialize the reference to the graph).

- reader_Formaggia (Graph &_G, std::string _file_name, unsigned int _num_dummy_lines)

  Constructor.

- reader_Formaggia (reader_Formaggia const &)

  Default copy constructor.

- reader_Formaggia & operator= (reader_Formaggia const &)

  Default assignment operator.

- virtual ~reader_Formaggia ()

  Destructor.

- void set_e_to_be_removed (Edge_desc const &_e_to_be_removed)

  It allows to set e_to_be_removed.

- void set_split_edge (Edge_desc const &_split_edge)

  It allows to set split_edge.

- void set_intersection_new (Vertex_desc const &_intersection_new)

  It allows to set intersection_new.

- void set_intersection_old (Vertex_desc const &_intersection_old)

  It allows to set intersection_old.

- virtual void read_data_from_line (std::istringstream &temp)

This is the way to interpret the data form Formaggia data file.

- virtual void give_new_source_properties ()

  It assigns properties to new_source in the right way.

- virtual void give_new_target_properties ()

  It assigns properties to new_target in the right way.

- virtual void give_new_edge_properties ()

  Overriding of the abstract method. It assigns properties to new_edge in the right way.

- virtual void build_graph ()

  The set of instruction for one single step in the building of the graph.

- void give_new_intersection_properties ()

  Overriding of the abstrac method. It assigns properties to a new intersection point in the right way.

- void give_split_edge_properties ()

  It assigns properties to split_edge in the right way.

- virtual bool are_intersected ()

  It checks if edges are intersected (only vertical or horizontal).

- virtual void refine_graph ()

  Boh.

- virtual void order_intersections ()

  Bohboh.

- reader_Zunino (Graph &_G)

  Default constructor (we need however to initialize the reference to the graph).

- reader_Zunino (Graph &_G, std::string _file_name, unsigned int _num_dummy_lines)

  Constructor: it assigns value only to the variables in reader_base_class, the others in reader_Zunino are defaulted.

- reader_Zunino (reader_Zunino const &)

  Default copy constructor.

- reader_Zunino & operator= (reader_Zunino const &)

  Assignment operator.

- virtual ~reader_Zunino ()

  Destructor.

- virtual void read_data_from_line (std::istringstream &temp)

  This is the way we read and interpret a file from a Zunino input file format.

- virtual void build_graph ()

  It build 8the graph one edge at a time.

- virtual void give_new_source_properties ()

  It assign the right properties to new_source just added.

- virtual void give_new_target_properties ()

  It assign the right properties to new_target just added.

- virtual void give_new_edge_properties ()

  It assign the properties to the edge just added.

template<typename Graph> class final< Graph >

The documentation for this class was generated from the following files:

- include/reader_Formaggia_class.hpp
- include/reader_Zunino_class.hpp

5.8 Forma_edge_property_t Struct Reference

Public Attributes

- unsigned int frac_num

  This track which fracture this edge belongs to.

- edge_parametrization param

  It describes the parametrization of the real structure of the edge.

The documentation for this struct was generated from the following file:

- include/Forma_edge_property.hpp

5.9 Forma_vertex_property_t Struct Reference

This struct contains the vertex property for Formaggia's example.

`#include <Forma_vertex_property.hpp>`

Public Attributes

- point< 2 > coord

  It contains the vertex coordinates.

- bool is_external

  It tracks if this is an external point (that is: it was in the input file, or it has degree = 1).

5.9.1 Detailed Description

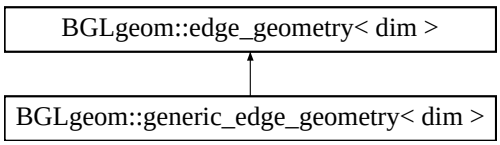
This struct contains the vertex property for Formaggia's example.

The documentation for this struct was generated from the following file:

- include/Forma_vertex_property.hpp

5.10 BGLgeom::generic_edge_geometry< dim > Class Template Reference

Inheritance diagram for BGLgeom::generic_edge_geometry< dim >::



Public Member Functions

- generic_edge_geometry (std::function< BGLgeom::point< dim >(double)> value_)

  s:[0,1] -> value_fun(s):[0,1]^dim

- generic_edge_geometry ()

  default constructor: linear edge (oppure defaultizzo già il fatto di chiamare sempre il linear_edge se non altrimenti specificato?)

- virtual std::vector< double > first_derivatives (const double x)

  first derivative

- virtual std::vector< double > second_derivatives (const double x)

  second derivative

- virtual BGLgeom::point< dim > value (const double parameter)

  curvilinear abscissa

template<unsigned int dim> class BGLgeom::generic_edge_geometry< dim >

5.10.1 Constructor & Destructor Documentation

5.10.1.1 template<unsigned int dim> BGLgeom::generic_edge_geometry< dim >::generic_edge_geometry (std::function< BGLgeom::point< dim >(double)> *value_*) [inline]

s:[0,1] -> value_fun(s):[0,1]^dim stores the function which takes in input the "normalized" parametrization of the edge constructor

5.10.2 Member Function Documentation

5.10.2.1 template<unsigned int dim> virtual std::vector<double> BGLgeom::generic_edge_geometry< dim >::first_derivatives (const double *x*) [inline, virtual]

first derivative

declare the point that will contain the result

Implements BGLgeom::edge_geometry< dim >.

5.10.2.2 template<unsigned int dim> virtual BGLgeom::point<dim> BGLgeom::generic_edge_geometry< dim >::value (const double *parameter*) [inline, virtual]

curvilinear abscissa returns value fun (parametrized between 0 and 1) in s between 0 and 1

Implements BGLgeom::edge_geometry< dim >.

The documentation for this class was generated from the following file:

- include/generic_edge_geometry.hpp

5.11 Geometry::Intersection Struct Reference

A simple struct that contains the result of the intersection test.

`#include <intersector_base_class.hpp>`

Public Attributes

- bool intersect = false

  Segments intersects.

- unsigned int numberOfIntersections = 0u

  Number of intersections (max 2).

- std::array< point< 2 >, 2 > intersectionPoint = std::array<point<2>,2>{point<2>(), point<2>()}

  Intersection points coordinates.

- std::array< std::array< bool, 2 >, 2 > endPointIsIntersection
- std::array< std::array< int, 2 >, 2 > otherEdgePoint
- bool parallel = false

  Edges are parallel.

- bool identical = false

  Edges are identical.

- bool collinear = false

  Edges are collinear (and thus also parallel).

- bool good = true

  Something is not ok.

- double distance = 0.0

  Distance, makes sense only if parallel=true.

5.11.1 Detailed Description

A simple struct that contains the result of the intersection test. To be able to treat the most general case each segment is allowed to have up to two intersections. It happens if the segments overlaps

Todo

It can be bettered by adding another attribute that indicates, in the case of two edges end which coincides the relative position on the edge. It requires a simple modification of the function segmentIntersect

Note:

Piece of code provided by prof. Formaggia

5.11.2 Member Data Documentation

5.11.2.1 std::array<std::array<bool,2>, 2> Geometry::Intersection::endPointIsIntersection

Initial value:

```
      std::array<std::array<bool,2>, 2>{std::array<bool,2>{false,false}, st
d::array<bool,2>{false,false} }
```

Intersection may be end point

endPointIsIntersection[i][j]=true then end j of edge i is at the intersection

5.11.2.2 bool Geometry::Intersection::intersect = false

Segments intersects. True is there is any intersection

5.11.2.3 std::array<std::array<int,2>, 2> Geometry::Intersection::otherEdgePoint

Initial value:

```
      std::array<std::array<int,2>, 2>{std::array<int,2>{-1,-1}, std::array
<int,2>{-1,-1}}
```

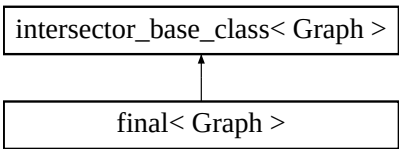
EdgeS join at the end. In that case endPointIsIntersection will be true and the corresponding entry will indicate the numering of the end of the other edge. -1 indicates that the end is not joined. So if endPointIsIntersection[i][j]=true we have otherEdgePoint[i][j]=-1 //End point is not joined with the other edge otherEdgePoint[i][j]= k //End point j of edge j is joined with end point k of other edge

The documentation for this struct was generated from the following file:

- include/intersector_base_class.hpp

5.12 intersector_base_class< Graph > Class Template Reference

Inheritance diagram for intersector_base_class< Graph >::



Public Types

- typedef boost::graph_traits< Graph >::edge_descriptor **Edge_desc**
- typedef boost::graph_traits< Graph >::edge_iterator **Edge_iter**
- typedef boost::graph_traits< Graph >::vertex_descriptor **Vertex_desc**
- typedef std::pair< point< 2 >, point< 2 > > **Line**
- typedef std::vector< std::pair< point< 2 >, Edge_desc > > **Intersections_type**
- typedef std::pair< point< 2 >, Edge_desc > **Intersections_value_type**

Public Member Functions

- intersector_base_class ()

  Default constructor (initialization of the reference to the graph needed).

- intersector_base_class (intersector_base_class const &)

  Copy constructor.

- virtual ~intersector_base_class ()

  Destructor.

- intersector_base_class & operator= (intersector_base_class const &)

  Assignment operator.

- virtual void set_Edge1 (point< 2 > const &P1, point< 2 > const &P2)

  It sets Edge1 from two points.

- virtual void set_Edge1 (Line _L)

  It sets Edge1 from another Line.

- virtual void set_Edge2 (point< 2 > const &P1, point< 2 > const &P2)

  It sets Edge2 from two points.

- virtual void set_Edge2 (Line _L)

  It sets Edge1 from another Line.

- virtual void set_Edge2_descriptor (Edge_desc _Edge2_desc)

  It allows to set Edge2_descriptor.

- virtual bool are_intersected ()=0

It checks if Edge1 and Edge2 are actually intersected If the two edge are intersecating, this method must store in the class variable intersection_point the coordinates of the intersection found.

- virtual void store_intersection ()

  It pushes back a new intersection point between Edge1 and Edge2, remembering the edge descriptor of Edge2.

- virtual void **clear_intersections** ()
- virtual void refine_graph ()=0

  It explains how to rebuilt graph after the intersections were computed It has to inteface with private attributes of the derived class in order to set edge and vertex properties in the right way.

- virtual void order_intersections ()=0

  This reorders the vector intersections according to some order, defined by the user It will consist of a call to "sort" algorithm, in which the compare function must be user defined, choosing a possible ordering in the 2D space.

Protected Attributes

- Line Edge1

  The first of the two edge that are (maybe) intersecating If the user has to perform multiple intersection between a fixed edge and all the other edges in the graph, Edge1 is thougth to be the fixed edge.

- Line Edge2

  The second of the two edge that are (maybe) intersecating If the user has to perform multiple intersection between a fixed edge and all the other edges in the graph, Edge2 is thougth to be the variable edge.

- Intersections_type intersections

  Vector that will contains the intersection point and the edge descriptor of the edge with which the current edge is intersecating.

- point< 2 > intersection_point

  The intersection point between Edge1 and Edge2 (if present).

- Edge_desc Edge2_descriptor

  Edge descriptor of Edge2. If the user has to perform multiple intersections between Edge1 (fixed) and Edge2 (variable), this tracks the edge descriptor of the edges in the graph that are intersecating Edge1 (one at a time).

template<typename Graph> class intersector_base_class< Graph >

The documentation for this class was generated from the following file:

- include/intersector_base_class.hpp

5.13 Geometry::Linear_edge Class Reference

A simple class that hanlde a linear edge This class is thought to manage the description of the geometry of a linear edge, in order to compute intersections.

`#include <intersector_base_class.hpp>`

Public Member Functions

- Linear_edge ()

  Default constructor.

- **extremes_are_set** (false)
- Linear_edge (point< 2 > const &SRC, point< 2 > const &TGT)

  Constructor.

- **extremes_are_set** (true)
- Linear_edge (Linear_edge const &)

  Copy constructor.

- Linear_edge & operator= (Linear_edge const &)

  Assignment operator.

- void set (point< 2 > const &SRC, point< 2 > const &TGT)

  Setting the two end points (extremes) of the edge.

- point< 2 > operator[ ] (std::size_t i)

  Overloading of operator[] to access each of the two end points. Usefull in algorithms extremes[0] = source, extremes[1] = target of the edge.

- point< 2 > **operator[ ]** (std::size_t i) const

5.13.1 Detailed Description

A simple class that hanlde a linear edge This class is thought to manage the description of the geometry of a linear edge, in order to compute intersections.

Remarks:

The class must have an overload of operator[] in order to run in the function that computes intersections

The documentation for this class was generated from the following file:

- include/intersector_base_class.hpp

5.14 my_edge Struct Reference

The documentation for this struct was generated from the following file:

- include/data_structure.hpp

5.15 new_reader_class< Source_data_structure, Target_data_structure, Edge_data_structure, Topological_data_structure > Class Template Reference

Abstract class that implements the functionality to read a file and get data from it The users has to specify, in the derived class, all variables he need in order to store information read from the input file. Then, through the definition of Edge_data_structure and Vertex_data_structure, he can get separately all the information to put on edges and vertices.

`#include <new_reader_class.hpp>`

Public Member Functions

- new_reader_class ()

  Default constructor.

- new_reader_class (std::string _filename)

  Constructor.

- new_reader_class (new_reader_class const &)

  Copy constructor.

- new_reader_class &new_reader_class const &virtual void set_input (std::string _filename)

  Assignment operator.

- virtual void ignore_dummy_lines (unsigned int const &n)

  Ignore n lines of the input code that the user knows he has not to read.

- virtual void read_line ()

  Reads one line and put it into a istringstream.

- virtual bool is_eof ()

  To know outside the class if we have reached the end of file.

- virtual void get_data_from_line ()=0

  Reads the data from one single line. It has to be specified by the user It reads data form the istringstream iss_line that is defined as an attribute of the class and it is updated after every call of read_line().

- virtual Edge_data_structure get_edge_data ()=0

  A method to get the right data to append to an edge.

- virtual Source_data_structure get_source_data ()=0

  A method to get the right data to append to the source.

- virtual Target_data_structure get_target_data ()=0

  A method to get the right data to append to the target.

- virtual Topological_data_structure get_topological_data ()=0

  A method to get the right topological data from a line.

Protected Attributes

- std::string filename

  The name of the file to be read.

- std::ifstream in_file

  File stream to handle the input file.

- std::string line

  String in which the data read from a line are put.

- std::istringstream iss_line

  Data put in line are converted in istringstream to be got by the user.

5.15.1 Detailed Description

template<typename Source_data_structure, typename Target_data_structure, typename Edge_data_structure, typename Topological_data_structure = no_topological_data> class new_reader_class< Source_data_structure, Target_data_structure, Edge_data_structure, Topological_data_structure >

Abstract class that implements the functionality to read a file and get data from it The users has to specify, in the derived class, all variables he need in order to store information read from the input file. Then, through the definition of Edge_data_structure and Vertex_data_structure, he can get separately all the information to put on edges and vertices.

Parameters:

Edge_data_structure A struct where the user has to define type and name of the variables he needs to append to vertices as vertex bundled property

Vertex_data_structure A struct where the user has to define type and name of the variables he needs to append to edge as edge bundled property

Precondition:

It may be useful to declare a friend operator>> to help the reader read the data

5.15.2 Member Function Documentation

5.15.2.1 template<typename Source_data_structure, typename Target_data_structure, typename Edge_data_structure, typename Topological_data_structure = no_topological_data> virtual void new_reader_class< Source_data_structure, Target_data_structure, Edge_data_structure, Topological_data_structure >::ignore_dummy_lines (unsigned int const & *n*) [inline, virtual]

Ignore n lines of the input code that the user knows he has not to read.

Remarks:

It sets the file stream n lines after the previous position

5.15.2.2 template<typename Source_data_structure, typename Target_data_structure, typename Edge_data_structure, typename Topological_data_structure = no_topological_data> new_reader_class& new_reader_class const& virtual void new_reader_class< Source_data_structure, Target_data_structure, Edge_data_structure, Topological_data_structure >::set_input (std::string *_filename*) `[inline, virtual]`

Assignment operator. Set the input file to read

The documentation for this class was generated from the following file:

- include/new_reader_class.hpp

5.16 no_topological_data Struct Reference

An empty struct to handle the case the user do not need to store topological data Inside this the user may put data as vertex and edge descriptor for the connettivity of the graph.

`#include <new_reader_class.hpp>`

5.16.1 Detailed Description

An empty struct to handle the case the user do not need to store topological data Inside this the user may put data as vertex and edge descriptor for the connettivity of the graph.

The documentation for this struct was generated from the following file:

- include/new_reader_class.hpp

5.17 our_disjoint_sets< Graph > Class Template Reference

Template class to handle disjoint sets The template parameters are:

Label_map_t: the type of a std::map which key is a vertex descriptor and the value is an unsigned int which has the meaning of the current label of the component to which that vertex belongs to.

Components_map_t: the type of a std::map which key is an unsigned int used as label for the group and the value is a std::set containing all the vertex descriptor of the vertices that have that label, i.e that belong to the same component.

`#include <our_disjoint_sets.hpp>`

Public Types

- typedef boost::graph_traits< Graph >::vertex_iterator **Vertex_iter**
- typedef boost::graph_traits< Graph >::vertex_descriptor **Vertex_desc**
- typedef std::map< Vertex_desc, unsigned int > **Label_map_t**
- typedef std::map< unsigned int, std::list< Vertex_desc > > **Components_map_t**
- typedef Label_map_t::key_type **Label_key_t**
- typedef Label_map_t::mapped_type **Label_mapped_t**
- typedef Components_map_t::key_type **Components_key_t**
- typedef Components_map_t::mapped_type **Components_mapped_t**
- typedef Components_mapped_t::value_type **Comp_mapped_vertex_t**

Public Member Functions

- our_disjoint_sets (Graph &_G)

  Default constructor.

- our_disjoint_sets (our_disjoint_sets const &)

  Copy constructor.

- our_disjoint_sets & operator= (our_disjoint_sets const &)

  Assignment operator.

- ~our_disjoint_sets ()

  Destructor.

- void make_label_map ()

  It creates the label map starting form the Graph The label_map is set up by associating to each vertex descriptor a progressive unsigned int as label, that indicates to which component the vertex belongs to. In other words, label_map is set up by assuming that each vertex is a separated component.

- Label_mapped_t get_label (Label_key_t const &vertex)

  It returns, from the label_map, the label of the component which the vertex belongs to.

- void set_label (Label_key_t const &vertex, Label_mapped_t const &label)

  It allows to set the label of that vertex in label map.

- bool is_present_component (Components_key_t const &label_of_the_component)

Checks if a particular component (i.e its label) is already present in the components_map.

- std::pair< typename Components_mapped_t::iterator, typename Components_mapped_t::iterator > get_iterator (Components_key_t const &label_of_the_component)

  It returns a pair containing the iterator to begin and end of the list that contains all the verteices of the given component.

- void new_component (Components_key_t const &label_value)

  It creates a new component with the given label value as key in components_map.

- void insert_vertex_in_component (Comp_mapped_vertex_t const &vertex, Components_key_t const &label_value)

  It add the given vertex descriptor to the component with that label.

- void insert_tgt_comp_in_src_comp (Components_key_t const &tgt_label_value, Components_key_t const &src_label_value)

  It insert the target component in the source component.

- void erase_component (Components_key_t const &label_value)

  It removes from components_map the component with the given key (=label of the component).

- Components_map_t get_components_map ()

  It returns the components_map outside the class.

Friends

- std::ostream & operator<< (std::ostream &out, our_disjoint_sets &dsets)

  Overloading of operator<< to view components_map.

5.17.1 Detailed Description

template<typename Graph> class our_disjoint_sets< Graph >

Template class to handle disjoint sets The template parameters are:

Label_map_t: the type of a std::map which key is a vertex descriptor and the value is an unsigned int which has the meaning of the current label of the component to which that vertex belongs to.

Components_map_t: the type of a std::map which key is an unsigned int used as label for the group and the value is a std::set containing all the vertex descriptor of the vertices that have that label, i.e that belong to the same component.

The documentation for this class was generated from the following file:

- include/our_disjoint_sets.hpp

5.18 BGLgeom::point< dim, Storage_t > Class Template Reference

Class template for storing the vertex coordinates in n-dimentional space.

`#include <generic_point.hpp>`

Public Member Functions

- point ()
  Default constructor.
- point (std::initializer_list< Storage_t > args)
  Constructor.
- point (std::array< Storage_t, dim > const &P)
  Constructor from a std::array<Storage_t,dim>.
- point (point< dim, Storage_t > const &)
  Copy constructor.
- point< dim, Storage_t > & operator= (point< dim, Storage_t > const &)
  Assignement operator.
- point< dim, Storage_t > & operator= (std::array< Storage_t, dim > const &P)
  Overload of assignment operator to create conversion directly form std::array<Storage_t, dim>.
- Storage_t x ()
  Gets the first coordinate.
- Storage_t **x** () const
- Storage_t y ()
  Gets the second coordinate.
- Storage_t **y** () const
- Storage_t z ()
  Gets the third coordinate.
- Storage_t **z** () const
- std::size_t get_dim ()
  Gets the dimension of the point, and so the number of the coordinates.
- std::size_t **get_dim** () const
- void set_x (double const &x)
  Set coordinate x of the point, corresponding to coord[0].
- void set_y (double const &y)
  Set coordinate y of the point, corresponding to coord[1].
- void set_z (double const &z)
  Set coordinate z of the point, corresponding to coord[i].

- void set (std::initializer_list< Storage_t > args)

  Set method to assign coordinates to an already existing point.

- Storage_t & operator[ ] (std::size_t i)

  Overloading of operator[], to get the i-th coordinate or write in it.

- Storage_t & **operator[ ]** (std::size_t i) const
- bool operator< (point< dim, Storage_t > const &P2) const

  Operator< overloading Point1 < Point2 if Point1.x is smaller than Point2.x; if they are equal, compare in the same waythe y coordinate, and so on.

- bool operator> (point< dim, Storage_t > const &point2) const

  Operator> overloading It is the negation of operator<.

Friends

- std::ostream & operator<< (std::ostream &out, point< dim, Storage_t > const &P)

  Overload of operator<<.

- std::istream & operator>> (std::istream &in, point< dim, Storage_t > &P)

  operator>> overloading

- point< dim, Storage_t > operator- (point< dim, Storage_t > const &P, point< dim, Storage_t > const &Q)

  Overloading of operator- for points.

- point< dim, Storage_t > operator- (point< dim, Storage_t > const &P, std::array< Storage_t, dim > const &a)

  Overload of operator- It defines difference between points and std::array, to define conversion between this two similar classes.

- point< dim, Storage_t > **operator-** (std::array< Storage_t, dim > const &a, point< dim, Storage_t > const &P)
- point< dim, Storage_t > operator+ (point< dim, Storage_t > const &P, point< dim, Storage_t > const &Q)

  Overloading of operator+ for points.

- point< dim, Storage_t > operator+ (point< dim, Storage_t > const &P, std::array< Storage_t, dim > const &a)

  Overload of operator+ It defines sum between points and std::array, to define conversion between this two similar classes.

- point< dim, Storage_t > **operator+** (std::array< Storage_t, dim > const &a, point< dim, Storage_t > const &P)
- point< dim, Storage_t > operator* (double const &k, point< dim, Storage_t > const &P)

  Overloading of operator It represents the multiplication of the coordinates of a point for a scalar.*

- point< dim, Storage_t > **operator*** (point< dim, Storage_t > const &P, double const &k)

- point< dim, Storage_t > operator/ (point< dim, Storage_t > const &P, double const &k)

  Overloading of operator/ It represents the division of the coordinates of a point for a scalar. Implemented using operator∗.

5.18.1 Detailed Description

template<unsigned int dim, typename Storage_t = double> class BGLgeom::point< dim, Storage_t >

Class template for storing the vertex coordinates in n-dimentional space.

Note:

Constructors and set method are implemented with std::initializer_list, so they have to be called with: method({args})

Parameters:

dim Template argument that specifies the dimension of the space

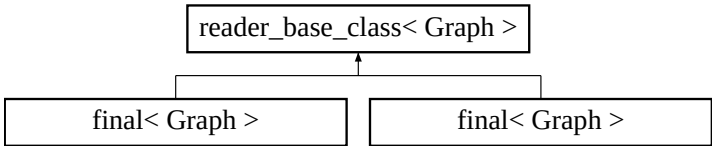
Storage_t Template argument that specifies the precision type for the coordinates

The documentation for this class was generated from the following file:

- include/generic_point.hpp

5.19 reader_base_class< Graph > Class Template Reference

Inheritance diagram for reader_base_class< Graph >::



Public Types

- typedef boost::graph_traits< Graph >::vertex_descriptor **Vertex_desc**
- typedef boost::graph_traits< Graph >::edge_descriptor **Edge_desc**

Public Member Functions

- reader_base_class (Graph &_G)

  Default constructor (we need however to initialize the reference).

- reader_base_class (Graph &_G, std::string _file_name, unsigned int _num_dummy_lines)

  Constructor: assign only num_dummy_lines, empty graph.

- reader_base_class (reader_base_class const &)

  Default copy constructor.

- reader_base_class & operator= (reader_base_class const &)

  Assignement operator.

- virtual ~reader_base_class ()

  Destructor (needed?).

- virtual void set_input_file (std::string _file_name)

  It allows to set the input file.

- virtual void set_num_dummy_lines (unsigned int const &_num_dummy_lines)

  It allows to set both num_dummy_lines and current_line_number (they must have the same value).

- virtual void read_input_file ()

  Read the input file.

- virtual void ignore_dummy_lines (std::ifstream &file)

  It ignores the first n lines, that are headers, in the input file.

- virtual void read_data_from_line (std::istringstream &temp)=0

  It describes how to read each line in the input file, and in which variables to store the data.

- virtual void build_graph ()=0

It build the graph one edge at a time, called many times from an external loop.

- virtual void give_new_source_properties ()=0

  It assigns properties to new_source in the rigth way. It has to be called in build_graph()!

- virtual void give_new_target_properties ()=0

  It assigns properties to new_target in the right way. It has to be called in build_graph()!

- virtual void give_new_edge_properties ()=0

  It assigns properties to new_edge in the rigth way. It has to be called in build_graph()!

- virtual void if_edge_not_inserted ()

  It deals with wrong insertion of an edge. It can be called only after a call to boost::add_edge with the pair <Edge_desc, bool> as return value. In this way the value of edge_inserted is set up.

Protected Attributes

- Graph & G

  A reference is used to represent the Graph. Using a reference allows us not to copy the whole graph outside the class once finished to read data and to build the graph. In this way we build the pre-existent graph (created in the main) while reading the input file. We do not use extra memory.

- std::string file_name

  The string in which is stored the name of the input file to read.

- unsigned int num_dummy_lines

  The numbers of initial lines (headers) that the reader has to skip to read useful data.

- std::string line

  This will contain one line of the input file, to parse.

- Vertex_desc new_source

  The vertex descriptor for the source of the new edge added.

- Vertex_desc new_target

  The vertex descriptor for the target of the new edge added.

- Edge_desc new_edge

  The edge descriptor for the new edge.

- bool edge_inserted

  Bool returned in a pair with an edge descriptor by add_edge function. The value is assigned when trying to insert an edge in the graph. It is set to true if the edge was succesfully inserted, false otherwise.

- unsigned int current_line_number

  It tracks the current line of the input file.

template<typename Graph> class reader_base_class< Graph >

The documentation for this class was generated from the following file:

- include/reader_base_class.hpp

5.20 vertex_data_structure< dim > Struct Template Reference

Public Attributes

- unsigned int **vertex_id** = 0
- std::string **label** = ""
- point< dim > **coord**

template<unsigned int dim> struct vertex_data_structure< dim >

The documentation for this struct was generated from the following file:

- include/data_structure.hpp

5.21 Zunino_edge_data Struct Reference

Public Attributes

- double **capacity**
- double **length**

The documentation for this struct was generated from the following file:

- include/new_reader_Zunino.hpp

5.22 Zunino_edge_property_t Struct Reference

Public Attributes

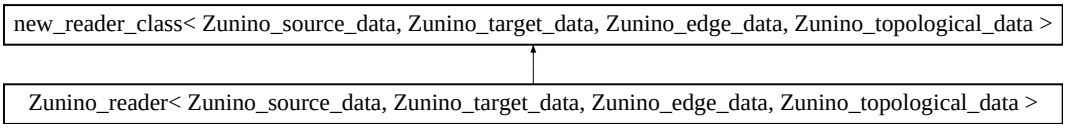
- double **capacity**
- double **length**

The documentation for this struct was generated from the following file:

- include/Zunino_edge_property.hpp

5.23 Zunino_reader< Zunino_source_data, Zunino_target_data, Zunino_edge_data, Zunino_topological_data > Class Template Reference

Inheritance diagram for Zunino_reader< Zunino_source_data, Zunino_target_data, Zunino_edge_data, Zunino_topological_data >::



Public Member Functions

- virtual void get_data_from_line ()

  Reads the data from one single line. It has to be specified by the user It reads data form the istringstream iss_line that is defined as an attribute of the class and it is updated after every call of read_line().

- virtual Zunino_source_data get_source_data ()

  A method to get the right data to append to the source.

- virtual Zunino_target_data get_target_data ()

  A method to get the right data to append to the target.

- virtual Zunino_edge_data get_edge_data ()

  A method to get the right data to append to an edge.

- virtual Zunino_topological_data **get_topologica_data** ()

template<typename Zunino_source_data, typename Zunino_target_data, typename Zunino_edge_data, typename Zunino_topological_data> class Zunino_reader< Zunino_source_data, Zunino_target_data, Zunino_edge_data, Zunino_topological_data >

The documentation for this class was generated from the following file:

- include/new_reader_Zunino.hpp

5.24 Zunino_source_data Struct Reference

Public Attributes

- BGLgeom::point< 3 > **SRC**

The documentation for this struct was generated from the following file:

- include/new_reader_Zunino.hpp

5.25 Zunino_target_data Struct Reference

Public Attributes

- BGLgeom::point< 3 > **TGT**

The documentation for this struct was generated from the following file:

- include/new_reader_Zunino.hpp

5.26 Zunino_topological_data Struct Reference

Public Attributes

- unsigned int **src**
- unsigned int **tgt**

The documentation for this struct was generated from the following file:

- include/new_reader_Zunino.hpp

Chapter 6

File Documentation

6.1 include/compute_euclidean_distance.hpp File Reference

Computes the euclidean distance between two given vertices. `#include "compute_euclidean_distance_imp.hpp"`

Functions

- template<typename Graph >
  double **compute_euclidean_distance** (typename boost::graph_traits< Graph >::vertex_descriptor a, typename boost::graph_traits< Graph >::vertex_descriptor b, Graph const &G)

6.1.1 Detailed Description

Computes the euclidean distance between two given vertices.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sep 2016.

6.2 include/data_structure.hpp File Reference

Declaration of generic data structure to represent vertex and edge properties. `#include "user_-data_structure.hpp"`

```
#include <string>
#include "generic_edge_geometry.hpp"
#include "generic_point.hpp"
```

Classes

- class data_structure< Edge_data_structure, Vertex_data_structure >

  An abstract class to handle the user definition data structure The users has to specify, in the derived class, all variables he need in order to store information read from the input file. Then, through the definition of Edge_data_structure and Vertex_data_structure, he can get separately all the information to put on edges and vertices.

- struct my_edge
- struct edge_data_structure< dim >
- struct vertex_data_structure< dim >
- class data_structure< Edge_data_structure, Vertex_data_structure >

  An abstract class to handle the user definition data structure The users has to specify, in the derived class, all variables he need in order to store information read from the input file. Then, through the definition of Edge_data_structure and Vertex_data_structure, he can get separately all the information to put on edges and vertices.

Functions

- **get_data_from_line** (std::ifstream &in, data_structure &D)
- virtual read_file my_edge **get_edge_data** ()

Variables

- struct my_edge **length**
- point **coord**

6.2.1 Detailed Description

Declaration of generic data structure to represent vertex and edge properties.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.3 include/dijkstra.hpp File Reference

Solves the single-source shortest-paths problem on a weighted, directed graph with non-negative edge weights. This function takes in input the graph, the source vertex and two vectors, one for the distance map and the other for the predecessor map, which will be filled with the results of the algorithm. `#include "boost/graph/dijkstra_shortest_paths.hpp"`

```
#include "Zunino_edge_property.hpp"
```

```
#include "dijkstra_imp.hpp"
```

Functions

- template<typename Graph >
  void **dijkstra** (Graph const &G, typename boost::graph_traits< Graph >::vertex_descriptor const &v, std::vector< int > &distances, std::vector< typename boost::graph_traits< Graph >::vertex_-descriptor > &predecessors)

6.3.1 Detailed Description

Solves the single-source shortest-paths problem on a weighted, directed graph with non-negative edge weights. This function takes in input the graph, the source vertex and two vectors, one for the distance map and the other for the predecessor map, which will be filled with the results of the algorithm.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Oct 2016.

6.4 include/dijkstra_imp.hpp File Reference

Solves the single-source shortest-paths problem on a weighted, directed graph with non-negative edge weights. .

Functions

- template<typename Graph >
  void **dijkstra** (Graph const &G, typename boost::graph_traits< Graph >::vertex_descriptor const &v, std::vector< int > &distances, std::vector< typename boost::graph_traits< Graph >::vertex_-descriptor > &predecessors)

6.4.1 Detailed Description

Solves the single-source shortest-paths problem on a weighted, directed graph with non-negative edge weights. .

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Oct 2016.

6.5 include/disjoint_components.hpp File Reference

Identifies if there are fully disconnected subgraphs.. . `#include <map>`

```
#include <tuple>
#include <iostream>
#include <boost/graph/graph_traits.hpp>
#include "our_disjoint_sets.hpp"
#include "disjoint_components_imp.hpp"
```

Functions

- template<typename Graph >
  void disjoint_components (Graph &G)

6.5.1 Detailed Description

Identifies if there are fully disconnected subgraphs.. .

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sep 2016.

6.5.2 Function Documentation

6.5.2.1 template<typename Graph > void disjoint_components (Graph & *G*) [inline]

Given a graph, this function checks whether there are fully disconnected subgraphs, i.e. subgraphs with no edge connecting each other. It returns a map which associates each vertex with an integer identifying the subgraph it belongs to.

6.6 include/disjoint_components_imp.hpp File Reference

Identifies if there are fully disconnected subgraphs.

Functions

- template<typename Graph >
  void disjoint_components (Graph &G)

6.6.1 Detailed Description

Identifies if there are fully disconnected subgraphs.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sep 2016. Note: the algorithm is not very well. It can be bettered, in particular when changing all the labels of a big component that we have to move in another small component. Better to move the smaller into the bigger.

6.6.2 Function Documentation

6.6.2.1 template<typename Graph > void disjoint_components (Graph & *G*) [inline]

Given a graph, this function checks whether there are fully disconnected subgraphs, i.e. subgraphs with no edge connecting each other. It returns a map which associates each vertex with an integer identifying the subgraph it belongs to.

6.7 include/edge_geometry.hpp File Reference

Virtual base class for the geometry of an edge .

Classes

- class BGLgeom::edge_geometry< dim >

6.7.1 Detailed Description

Virtual base class for the geometry of an edge .

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.8 include/Forma_edge_property.hpp File Reference

This contains the struct for edge properties that has to be used for Formaggia's example .

Classes

- class edge_parametrization

  This class holds the parametrization of the edge.

- struct Forma_edge_property_t

6.8.1 Detailed Description

This contains the struct for edge properties that has to be used for Formaggia's example .

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.9 include/Forma_vertex_property.hpp File Reference

This contains the struct for vertex properties that has to be used for Formaggia's example . `#include "generic_point.hpp"`

Classes

- struct Forma_vertex_property_t

  This struct contains the vertex property for Formaggia's example.

6.9.1 Detailed Description

This contains the struct for vertex properties that has to be used for Formaggia's example .

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.10 include/generic_edge_geometry.hpp File Reference

Class for circular geometry of an edge . `#include <array>`

`#include <functional>`

`#include "generic_point.hpp"`

`#include "edge_geometry.hpp"`

Classes

- class BGLgeom::generic_edge_geometry< dim >

6.10.1 Detailed Description

Class for circular geometry of an edge . class for the generic geometry of an edge

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.11 include/generic_point.hpp File Reference

Templete class to handle points in 2D or 3D (or even greater). `#include <array>`

`#include <iostream>`

`#include <initializer_list>`

`#include <type_traits>`

Classes

- class BGLgeom::point< dim, Storage_t >

  Class template for storing the vertex coordinates in n-dimentional space.

6.11.1 Detailed Description

Templete class to handle points in 2D or 3D (or even greater).

Author:

Ilaria Speranza and Mattia Tantardini

Date:

Sept, 2016

6.12 include/graph_builder.hpp File Reference

Utilities to build a graph. `#include <vector>`

`#include <boost/graph/graph_traits.hpp>`

`#include "generic_point.hpp"`

Functions

- template<typename Graph , typename Source_data_structure >
  void give_source_properties (Source_data_structure const &D, boost::graph_traits< Graph >::vertex_descriptor const &v, Graph &G)

  Giving to source node v all properties through assigning the Source_data_structure.

- template<typename Graph , typename Target_data_structure >
  void give_target_properties (Target_data_structure const &D, boost::graph_traits< Graph >::vertex_descriptor const &v, Graph &G)

  Giving to target node v all properties through assigning the Target_data_structure.

- template<typename Graph , typename Edge_data_structure >
  void give_edge_properties (Edge_data_structure const &D, boost::graph_traits< Graph >::edge_descriptor const &e, Graph &G)

  Giving to edge e all properties through assigning the Edge_data_structure.

- template<typename Graph , typename Vertex_data_structure , typename Edge_data_structure , typename Intersections_container = std::vector<point<2>>>
  void **refine_graph** (Graph &G, Intersections_container const &I, boost::graph_traits< Graph >::edge_descriptor edge1, boost::graph_traits< Graph >::edge_descriptor edge2)

6.12.1 Detailed Description

Utilities to build a graph.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.13 include/intersector_base_class.hpp File Reference

Abstract class to handle intersections of edges in a graph with geometrical properties It contains also some utilities needed to compute the intersection between two (linear) edges. `#include <vector>`

```
#include <array>
#include <tuple>
#include <cmath>
#include <limits>
#include <boost/graph/graph_traits.hpp>
#include "generic_point.hpp"
```

Classes

- class Geometry::Linear_edge

  A simple class that hanlde a linear edge This class is thought to manage the description of the geometry of a linear edge, in order to compute intersections.

- struct Geometry::Intersection

  A simple struct that contains the result of the intersection test.

- class intersector_base_class< Graph >

Typedefs

- typedef std::array< double, 2 > Vector

  Ecco il trucco forse!!! Una volta stabilita tutta la geometria, questa classe dovrebbe funzionare sempre allo stesso modo. Cioè, tipo: le intersezioni si troveranno sempre nella stessa maniera, i punti e gli edge saranno tutti descritti alla stessa maniera, ecc.

Functions

- template<typename Edge_rapresentation_t = Linear_edge>
  Intersection Geometry::compute_intersection (Linear_edge const &Edge1, Linear_edge const &Edge2, double const tol=20 *std::numeric_limits< double >::epsilon())

  Function to compute intersection between two linear edges.

- std::ostream & Geometry::operator<< (std::ostream &out, Geometry::Intersection const &i)

  Overload of operator<< to show the information contained in the strcut Intersection.

6.13.1 Detailed Description

Abstract class to handle intersections of edges in a graph with geometrical properties It contains also some utilities needed to compute the intersection between two (linear) edges.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.14 include/io_graph.hpp File Reference

Declaration of functions related to input and output of the graph. `#include <tuple>`

```
#include <boost/graph/adjacency_list.hpp>
#include <iostream>
#include <sstream>
#include <string>
#include <fstream>
#include <set>
#include <utility>
#include <cmath>
#include <algorithm>
#include "Forma_vertex_property.hpp"
#include "Forma_edge_property.hpp"
#include "generic_point.hpp"
#include "io_graph_imp.hpp"
```

Functions

- template<typename Graph , typename Reader >
  void **read_input_file** (Graph &G, Reader R, std::string file_name)
- template<typename Graph >
  void read_zunino_old_format (Graph &G, std::string file_name)

  Reads data about the graph from the input file given by professor Zunino.

- template<typename Graph >
  void **read_Formaggia_format** (Graph &G, std::string file_name)
- template<typename Graph >
  boost::graph_traits< Graph >::vertex_descriptor vertex_insertion_or_identification (Graph &G, point< 2 > const &P)

  Helper function for read_Formaggia_format.

- template<typename Graph >
  void check_for_intersections (std::vector< std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > > &v, point< 2 > const &SRC, point< 2 > const &TGT, Graph const &G)

  Helper function for read_Formaggia_format.

- template<typename Graph , bool src_less_than_tgt>
  bool compare (std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > pair1, std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > pair2)

  Helper function for check_for intersection.

- std::pair< bool, point< 2 > > are_intersected (std::pair< point< 2 >, point< 2 > > line1, std::pair< point< 2 >, point< 2 > > line2)

helper function for check_for_intersection

- template<typename Graph >
  void refine_graph (Graph &G, typename std::vector< std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > > const &vect, int frac_number, typename boost::graph_traits< Graph >::vertex_descriptor src, typename boost::graph_traits< Graph >::vertex_descriptor tgt)

  Helper function for read_Formaggia_format.

6.14.1 Detailed Description

Declaration of functions related to input and output of the graph.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.14.2 Function Documentation

6.14.2.1 std::pair<bool, point<2>> are_intersected (std::pair< point< 2 >, point< 2 > > *line1*, std::pair< point< 2 >, point< 2 > > *line2*)

helper function for check_for_intersection we assume the fractures can only be vertical or horizontal. It compute the coordinates of the intersection point, if present.

6.14.2.2 template<typename Graph > void check_for_intersections (std::vector< std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > > & *v*, point< 2 > const & *SRC*, point< 2 > const & *TGT*, Graph const & *G*) [inline]

Helper function for read_Formaggia_format. This function checks if two lines (fractures) are intersected. If yes, it creates a vector with all the intersection points already ordered with the right direction (from source to target vertex)

Parameters:

vect The vector that will be filled with the intersection points of each new edge

SRC Source vertex of the current edge

TGT Target vertex of the current edge

G Graph

Returns:

void

6.14.2.3 template<typename Graph , bool src_less_than_tgt> bool compare (std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > *pair1*, std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > *pair2*) [inline]

Helper function for check_for intersection. Given a couple of points, this function orders the intersection points according to the template parameter src_less_than_tgt. The ordering is needed in order to craete the new edges in the right way, preserving the direction of the fractures.

Parameters:

pair1 It is the intersection point between the current edge and the edge described by the second component of the pair

pair2 It is the intersection point between the current edge and the edge described by the second component of the pair

Returns:

bool

6.14.2.4 template<typename Graph > void read_zunino_old_format (Graph & *G*, std::string *file_name*) [inline]

Reads data about the graph from the input file given by professor Zunino. The funcitions reads from a file where data is written as:

line1: description of file

line2: description of file

from line 3: line_number - source - target - diameter - length - source_coord - target_coord

6.14.2.5 template<typename Graph > void refine_graph (Graph & *G*, typename std::vector< std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > > const & *vect*, int *frac_number*, typename boost::graph_traits< Graph >::vertex_descriptor *src*, typename boost::graph_traits< Graph >::vertex_descriptor *tgt*) [inline]

Helper function for read_Formaggia_format. This function breaks old edges to create a refined graph according to the intersection points found while inserting the current edge. It preserves the fracture number of each old edge while creating the new ones.

Parameters:

G Graph

vect The vector of the intersection points

frac_number The fracture number of the current edge

src Vertex descriptor of the source of the current edge

tgt Vertex descriptor of the target of the current edge

Returns:

void

6.14.2.6 template<typename Graph > boost::graph_traits<Graph>::vertex_descriptor vertex_insertion_or_identification (Graph & *G*, point< 2 > const & *P*) [inline]

Helper function for read_Formaggia_format. The function checks if the vertex we are trying to insert is already present in the graph, (in this case it will be ignored) or if it isn't already present (in this case it will be inserted).

Parameters:

G Graph we are constructing

P Point we want to check if is present or not

Returns:

vertex_descriptor

6.15 include/io_graph_imp.hpp File Reference

Definition of functions related to input and output of the graph.

Functions

- template<typename Graph >
  void **read_Formaggia_format** (Graph &G, std::string file_name)
- template<typename Graph >
  boost::graph_traits< Graph >::vertex_descriptor vertex_insertion_or_identification (Graph &G, point< 2 > const &P)

  Helper function for read_Formaggia_format.
- template<typename Graph >
  void check_for_intersections (std::vector< std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > > &vect, point< 2 > const &SRC, point< 2 > const &TGT, Graph const &G)

  Helper function for read_Formaggia_format.
- template<typename Graph , bool src_less_than_tgt>
  bool compare (std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > pair1, std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > pair2)

  Helper function for check_for intersection.
- std::pair< bool, point< 2 > > are_intersected (std::pair< point< 2 >, point< 2 > > line1, std::pair< point< 2 >, point< 2 > > line2)

  helper function for check_for_intersection
- template<typename Graph >
  void refine_graph (Graph &G, typename std::vector< std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > > const &vect, int frac_number, typename boost::graph_traits< Graph >::vertex_descriptor src, typename boost::graph_traits< Graph >::vertex_descriptor tgt)

  Helper function for read_Formaggia_format.

6.15.1 Detailed Description

Definition of functions related to input and output of the graph.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.15.2 Function Documentation

6.15.2.1 std::pair<bool, point<2> > are_intersected (std::pair< point< 2 >, point< 2 > > *line1*, std::pair< point< 2 >, point< 2 > > *line2*)

helper function for check_for_intersection we assume the fractures can only be vertical or horizontal. It compute the coordinates of the intersection point, if present.

6.15.2.2 template<typename Graph > void check_for_intersections (std::vector< std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > > & *v*, point< 2 > const & *SRC*, point< 2 > const & *TGT*, Graph const & *G*) [inline]

Helper function for read_Formaggia_format. This function checks if two lines (fractures) are intersected. If yes, it creates a vector with all the intersection points already ordered with the right direction (from source to target vertex)

Parameters:

vect The vector that will be filled with the intersection points of each new edge

SRC Source vertex of the current edge

TGT Target vertex of the current edge

G Graph

Returns:

void

6.15.2.3 template<typename Graph , bool src_less_than_tgt> bool compare (std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > *pair1*, std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > *pair2*) [inline]

Helper function for check_for intersection. Given a couple of points, this function orders the intersection points according to the template parameter src_less_than_tgt. The ordering is needed in order to craete the new edges in the right way, preserving the direction of the fractures.

Parameters:

pair1 It is the intersection point between the current edge and the edge described by the second component of the pair

pair2 It is the intersection point between the current edge and the edge described by the second component of the pair

Returns:

bool

6.15.2.4 template<typename Graph > void refine_graph (Graph & *G*, typename std::vector< std::pair< point< 2 >, typename boost::graph_traits< Graph >::edge_descriptor > > const & *vect*, int *frac_number*, typename boost::graph_traits< Graph >::vertex_descriptor *src*, typename boost::graph_traits< Graph >::vertex_descriptor *tgt*) [inline]

Helper function for read_Formaggia_format. This function breaks old edges to create a refined graph according to the intersection points found while inserting the current edge. It preserves the fracture number of each old edge while creating the new ones.

Parameters:

G Graph

vect The vector of the intersection points

frac_number The fracture number of the current edge

src Vertex descriptor of the source of the current edge

tgt Vertex descriptor of the target of the current edge

Returns:

void

6.15.2.5 template<typename Graph > boost::graph_traits<Graph>::vertex_descriptor vertex_insertion_or_identification (Graph & *G*, point< 2 > const & *P*) [inline]

Helper function for read_Formaggia_format. The function checks if the vertex we are trying to insert is already present in the graph, (in this case it will be ignored) or if it isn't already present (in this case it will be inserted).

Parameters:

G Graph we are constructing

P Point we want to check if is present or not

Returns:

vertex_descriptor

6.16 include/maximum_flow.hpp File Reference

Header file for managing maximum_flow algorithm from BGL. `#include <map>`

```
#include <tuple>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/push_relabel_max_flow.hpp>
#include <boost/property_map/property_map.hpp>
#include <boost/graph/properties.hpp>
#include "generic_point.hpp"
#include "edge_property_max_flow.hpp"
#include "maximum_flow_imp.hpp"
```

Functions

- template<typename Graph , typename Edge_Descriptor_g >
  double maximum_flow (Graph const &G, typename boost::graph_traits< Graph >::vertex_descriptor s, typename boost::graph_traits< Graph >::vertex_descriptor t, std::map< Edge_Descriptor_g, double > &out_residual_capacity)

  It runs push_relabel_max_flow algorithm on graph G.

- template<typename Graph , typename Flow_Graph , typename Edge_fg >
  void build_flow_graph (Graph const &G, Flow_Graph &FG, std::map< Edge_fg, Edge_fg > &rev_map)

  Helper function for maximum_flow.

- template<typename Graph , typename Flow_Graph , typename Edge_Descriptor_g >
  void store_residual_capacity (Graph const &G, Flow_Graph const &FG, std::map< Edge_Descriptor_g, double > &out_residual_capacity)

  Helper function that stores residual capacity on edges after computation of max flow.

6.16.1 Detailed Description

Header file for managing maximum_flow algorithm from BGL.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sep 14, 2016

6.16.2 Function Documentation

6.16.2.1 template<typename Graph , typename Flow_Graph , typename Edge_fg > void build_flow_graph (Graph const & *G*, Flow_Graph & *FG*, std::map< Edge_fg, Edge_fg > & *rev_map*) [inline]

Helper function for maximum_flow. This function build the flow graph associated to the input graph. This is because we want not to modify the original Graph passed as input in maximum_flow, and because the push_relabelmax_flow algorithm requires such a Graph.

Parameters:

G Graph

FG Flow graph that will be built inside the function. We need to build a new graph with a special structure in order to accomplish the requirments of boost::push_relabel_max_flow

rev_map Map that stores the reverse edge for each edge in the graph. It is needed to build Flow Graph.

6.16.2.2 template<typename Graph , typename Edge_Descriptor_g > double maximum_flow (Graph const & *G*, typename boost::graph_traits< Graph >::vertex_descriptor *s*, typename boost::graph_traits< Graph >::vertex_descriptor *t*, std::map< Edge_Descriptor_g, double > & *out_residual_capacity*) [inline]

It runs push_relabel_max_flow algorithm on graph G. This function find the maximum flow that can flow from node s to node t.

Parameters:

G Graph

s Source vertex chosen for the maximum flow problem

t Target vertex chosen for the maximum flow problem

out_residual_capacity Map that stores the residual capacity left in each edge

6.16.2.3 template<typename Graph , typename Flow_Graph , typename Edge_Descriptor_g > void store_residual_capacity (Graph const & *G*, Flow_Graph const & *FG*, std::map< Edge_Descriptor_g, double > & *out_residual_capacity*) [inline]

Helper function that stores residual capacity on edges after computation of max flow. We use a vector. Next step: using a map<Edge_descriptor, residual_capacity_value> This function search in the flow graph which edges have the same sources and target as the edges in G, so that we can associate the right residual capacity to the right original edge of G. This is because FG is a utility in order to run the push_relabel algorithm and it is destroied after exiting this function.

Parameters:

G Graph

FG Flow Graph

out_residual_capacity Map that stores the residual capacity left in each edge

6.17 include/maximum_flow_imp.hpp File Reference

Implementations of the functions defined in maximum_flow.hpp.

Functions

- template<typename Graph , typename Edge_Descriptor_g >
  double maximum_flow (Graph const &G, typename boost::graph_traits< Graph >::vertex_descriptor s, typename boost::graph_traits< Graph >::vertex_descriptor t, std::map< Edge_Descriptor_g, double > &out_residual_capacity)

  It runs push_relabel_max_flow algorithm on graph G.

- template<typename Graph , typename Flow_Graph , typename Edge_fg >
  void build_flow_graph (Graph const &G, Flow_Graph &FG, std::map< Edge_fg, Edge_fg > &rev_map)

  Helper function for maximum_flow.

- template<typename Graph , typename Flow_Graph , typename Edge_Descriptor_g >
  void store_residual_capacity (Graph const &G, Flow_Graph const &FG, std::map< Edge_Descriptor_g, double > &out_residual_capacity)

  Helper function that stores residual capacity on edges after computation of max flow.

6.17.1 Detailed Description

Implementations of the functions defined in maximum_flow.hpp.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sep 14, 2016

6.17.2 Function Documentation

6.17.2.1 template<typename Graph , typename Flow_Graph , typename Edge_fg > void build_flow_graph (Graph const & *G*, Flow_Graph & *FG*, std::map< Edge_fg, Edge_fg > & *rev_map*) [inline]

Helper function for maximum_flow. This function build the flow graph associated to the input graph. This is because we want not to modify the original Graph passed as input in maximum_flow, and because the push_relabelmax_flow algorithm requires such a Graph.

Parameters:

G Graph

FG Flow graph that will be built inside the function. We need to build a new graph with a special structure in order to accomplish the requirments of boost::push_relabel_max_flow

rev_map Map that stores the reverse edge for each edge in the graph. It is needed to build Flow Graph.

6.17.2.2 template<typename Graph , typename Edge_Descriptor_g > double maximum_flow (Graph const & *G*, typename boost::graph_traits< Graph >::vertex_descriptor *s*, typename boost::graph_traits< Graph >::vertex_descriptor *t*, std::map< Edge_Descriptor_g, double > & *out_residual_capacity*) [inline]

It runs push_relabel_max_flow algorithm on graph G. This function find the maximum flow that can flow from node s to node t.

Parameters:

G Graph

s Source vertex chosen for the maximum flow problem

t Target vertex chosen for the maximum flow problem

out_residual_capacity Map that stores the residual capacity left in each edge

6.17.2.3 template<typename Graph , typename Flow_Graph , typename Edge_Descriptor_g > void store_residual_capacity (Graph const & *G*, Flow_Graph const & *FG*, std::map< Edge_Descriptor_g, double > & *out_residual_capacity*) [inline]

Helper function that stores residual capacity on edges after computation of max flow. We use a vector. Next step: using a map<Edge_descriptor, residual_capacity_value> This function search in the flow graph which edges have the same sources and target as the edges in G, so that we can associate the right residual capacity to the right original edge of G. This is because FG is a utility in order to run the push_relabel algorithm and it is destroied after exiting this function.

Parameters:

G Graph

FG Flow Graph

out_residual_capacity Map that stores the residual capacity left in each edge

6.18 include/new_reader_class.hpp File Reference

Base abstract class to read input file. `#include <string>`

`#include <fstream>`

`#include <sstream>`

`#include <iostream>`

`#include <cstdlib>`

`#include <exception>`

Classes

- struct no_topological_data

  An empty struct to handle the case the user do not need to store topological data Inside this the user may put data as vertex and edge descriptor for the connettivity of the graph.

- class new_reader_class< Source_data_structure, Target_data_structure, Edge_data_structure, Topological_data_structure >

  Abstract class that implements the functionality to read a file and get data from it The users has to specify, in the derived class, all variables he need in order to store information read from the input file. Then, through the definition of Edge_data_structure and Vertex_data_structure, he can get separately all the information to put on edges and vertices.

6.18.1 Detailed Description

Base abstract class to read input file.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016 This abstract class provides the user some methods and functionality to read data form the input file and check errors.

6.19 include/new_reader_Zunino.hpp File Reference

Class for reading from Zunino files. `#include "new_reader_class.hpp"`

`#include "generic_point.hpp"`

Classes

- struct Zunino_source_data
- struct Zunino_target_data
- struct Zunino_edge_data
- struct Zunino_topological_data
- class Zunino_reader< Zunino_source_data, Zunino_target_data, Zunino_edge_data, Zunino_topological_data >

6.19.1 Detailed Description

Class for reading from Zunino files.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016 In this header file the user has to implement:

- A struct to handle data which will be put as edge_property
- A struct to handle data which will be put as vertex_property
- A reader class that inherits from new_reader_class, in which the user has to put all variables that will be read from input file and override all abstract methods

6.20 include/our_disjoint_sets.hpp File Reference

Class to handle disjoint sets . `#include <iostream>`

`#include <map>`

`#include <tuple>`

`#include <list>`

`#include <boost/graph/graph_traits.hpp>`

Classes

- class our_disjoint_sets< Graph >

  Template class to handle disjoint sets The template parameters are:
  Label_map_t: the type of a std::map which key is a vertex descriptor and the value is an unsigned int which has the meaning of the current label of the component to which that vertex belongs to.
  Components_map_t: the type of a std::map which key is an unsigned int used as label for the group and the value is a std::set containing all the vertex descriptor of the vertices that have that label, i.e that belong to the same component.

6.20.1 Detailed Description

Class to handle disjoint sets .

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sep 2016.

6.21 include/reader_base_class.hpp File Reference

Base abstract class to read input file and creating the graph. `#include <iostream>`

`#include <string>`

`#include <fstream>`

`#include <sstream>`

`#include <boost/graph/graph_traits.hpp>`

Classes

- class reader_base_class< Graph >

6.21.1 Detailed Description

Base abstract class to read input file and creating the graph.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016 It contains all the variables needed to read an input file and to store a graph. It allows to specify how to read the imput file through the abrstract methods.

6.22 include/reader_Formaggia_class.hpp File Reference

Implementation of the reader_base_class for the Formaggia file format. `#include <algorithm>`

`#include "reader_base_class.hpp"`

`#include "intersector_base_class.hpp"`

`#include "generic_point.hpp"`

Classes

- class final< Graph >

6.22.1 Detailed Description

Implementation of the reader_base_class for the Formaggia file format.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.23 include/reader_Zunino_class.hpp File Reference

Implementation of the reader for Zunino file format. `#include <tuple>`

```
#include <iostream>
#include "reader_base_class.hpp"
#include "generic_point.hpp"
```

Classes

- class final< Graph >

6.23.1 Detailed Description

Implementation of the reader for Zunino file format.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.24 include/topological_distance.hpp File Reference

Computes topological distance. . `#include "topological_distance_imp.hpp"`

Functions

- template<typename Graph >
  void **dijkstra** (Graph const &G, typename boost::graph_traits< Graph >::vertex_descriptor s)

6.24.1 Detailed Description

Computes topological distance. .

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sep 2016.

6.25 include/topological_distance_imp.hpp File Reference

Computes topological distance. . `#include <boost/graph/dijkstra_shortest_paths.hpp>`

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

`#include <vector>`

`#include "edge_property.hpp"`

Functions

- template<typename Graph >
  double **dijkstra** (Graph const &G, typename boost::graph_traits< Graph >::vertex_descriptor s, typename boost::graph_traits< Graph >::vertex_descriptor t, std::vector< typename boost::graph_traits< Graph >::vertex_descriptor > &v)

6.25.1 Detailed Description

Computes topological distance. .

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sep 2016.

6.26 include/Zunino_edge_property.hpp File Reference

Contains the struct for edge properties in Zunino's problem.

Classes

- struct Zunino_edge_property_t

6.26.1 Detailed Description

Contains the struct for edge properties in Zunino's problem.

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.27 src/main_Formaggia.cpp File Reference

Source code for Formaggia's example . `#include <iostream>`

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "Forma_vertex_property.hpp"
#include "Forma_edge_property.hpp"
#include "reader_Formaggia_class.hpp"
```

Functions

- int **main** ()

6.27.1 Detailed Description

Source code for Formaggia's example .

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016

6.28 src/main_Zunino.cpp File Reference

Source code for Zunino example. . `#include <iostream>`

```
#include <string>
#include <vector>
#include <tuple>
#include <boost/graph/adjacency_list.hpp>
#include "new_reader_Zunino.hpp"
#include "graph_builder.hpp"
```

Functions

- int **main** ()

6.28.1 Detailed Description

Source code for Zunino example. .

Author:

Ilaria Speranza & Mattia Tantardini

Date:

Sept, 2016