FEA_Mesher2D Documentation
Finite Element Analysis Boundary Layer Triangular Mesh Generator

JULIETTE PARDUE* and ANDREY CHERNIKOV, Old Dominion University

ACM Reference format:
Juliette Pardue and Andrey Chernikov. 2019. FEA_Mesher2D Documentation. 1, 1, Article 1 (September 2019), 19 pages.
DOI: N/A

1 INTRODUCTION

FEA_Mesher2D is a parallel triangular mesh generator that is capable of generating boundary layer meshes, suitable for finite element analysis simulations, and isotropic mesh regions. FEA_Mesher2D generates a high-fidelity, anisotropic boundary layer mesh from a user-defined growth function, generates a globally Delaunay, graded, isotropic mesh region in parallel, resolves potential interpolation errors in the boundary layer caused by the local mesh density, resolves self intersections and multi-element intersections in the boundary layer, is a push-button mesh generator so no human interaction is required after startup, and is scalable and efficient.

Information regarding the implementation of FEA_Mesher2D is available in the paper titled, "An Efficient Parallel Anisotropic Delaunay Mesh Generator for Two-Dimensional Finite Element Analysis" by Juliette Pardue and Andrey Chernikov.

2 INSTALLATION

The following packages are required to be properly configured before building the application.

- C++11 compiler or later
- C11 compiler or later
- MPI Version 3 Implementation
- POSIX Threads Implementation
- CMake Version 3 or later

*The corresponding author

Author's addresses: J. Pardue and A. Chernikov, Computer Science Department, Old Dominion University, Norfolk, VA 23529.

To compile the application, run the *compile.sh* script
The build script assumes that you have gcc and g++ set to the C and C++ compiler of your choice. The build script will find your system's MPI libraries if they are correctly installed and loaded. The application will be built in the FEA_Mesher2D directory where the compilation script is. If you want to change the directory where the application is built, you can set the location in src/CMakeLists.txt as the *CMAKE_RUNTIME_OUTPUT_DIRECTORY*

3 USAGE

The class MeshGenerator is the object that the end user will interact with. An example of a sample calling procedure is shown in main.cpp. The functions the end user should be concerned with are:

- readInputModel(string filename)
  Reads the input model referenced by the path *filename*
- setBoundaryLayerGrowthFunction(double first_thickness, double growth_rate, int initial_layers)
  Sets the growth function used to generate the anisotropic boundary layer.
  first_thickness is the distance from the geometry that the points in the first layer will be placed.
  growth_rate is the geometric rate at which each layer's thickness will grow from the previous layer. A value of 1 causes all layers to have the same thickness at *first_thickness*. A value greater than 1 cause subsequent layers to grow in thickness, which is the intended usage.
  initial_layers is the number of anisotropic layers that will be originally grown from the geometry. Some triangles in some layers will be removed due to intersections or a poor quality shape.
- setFarFieldDistance(double chord_lengths)
  Sets the distance for the domain away from the geometry that will be meshed. A square is generated that encloses the domain and each side is *chord_lengths* in each direction $(+x, -x, +y, -y)$ from the center of the mesh. One chord length is the length of the input geometry in the x-direction.
- useUniformTrianglesForInviscidRegion()
  This is an optional command. By default, the inviscid region uses smaller triangles near the geometry and grades to larger triangles farther away from the geometry. This function is called if you want the same sized triangles to be used for the entirety of the inviscid region.
- meshDomain()
  This is where the action happens. This function creates all the mesh vertices and mesh triangles using the number of processes launched by MPI.
- collectFinalMesh()
  This function collects all of the mesh entities on the root process so that the mesh can be output.

Four output formats are available, and the user may implement their own output format by adding and implementing a function in the class MeshGenerator. The "output" folder is where mesh files will be written to. The four provided output formats are:

- outputTecplot() - used for Tecplot
- outputShowMe() - used for Shewchuk's "Show Me" application
- outputFUN3D() - .msh file used for the flow solver FUN3D
- outputVTK() - .vtk format, can be used with ParaView

File	$first\_thickness$	$growth\_rate$	$initial\_layers$	$chord\_lengths$	Uniform Inviscid Region
naca0012.poly	0.0001	1.15	30	2	Yes
wright1903.poly	0.00005	1.10	50	30	No
airfoil30p30n.poly	0.000075	1.10	30	10	No

Table 1. Suggested Input Parameters

3.1 Input Geometry File Format

The following describes how a user should format their input files.

- First line: $number\_of\_points, number\_of\_elements$
  The number of elements is the number of polygons that are in the input geometry. The points should be ordered counter-clockwise for each element.
- Next $number\_of\_points$ lines: $id, x_coordinate, y_coordinate, element_id$
  id's should start at zero and increment. $element_id$ is the corresponding element that the point is part of. The first element should be 0 and all points that are part of element 0 should appear before any points of element 1.
- Next line: $number\_of\_edges$
  This should be the same as $number\_of\_points$ since each polygon should be water tight and simple, meaning each point is incident upon two edges.
- Next $number\_of\_edges$ lines: $id, point\_a\_id, point\_b\_id$
  id's should start at zero and increment.
- Next line: $number\_of\_holes$
  This should be the same as the $number\_of\_elements$. One hole should be defined for each element. A hole is a point inside the element's polygon, not on an edge of the element's polygon.
- Next $number\_of\_holes$ lines: $id, x_coordinate, y_coordinate$
  id's should start at zero and increment.

Three properly formatted input files are provided in the geometry folder with this package:

- naca0012.poly - Standard benchmark airfoil
- wright1903.poly - Wright Brother's 1903 airfoil used for the first flight
- airfoil30p30n.poly - Complex airfoil with three elements

The output meshes in the VTK format of the suggested input parameters of the three provided geometries are located in the output folder.

4 FEA_MESHER2D OBJECTS

This section details the different classes, structs, and namespaces used for data structures in the application.

4.1 ADT2D

The ADT2D class is used to efficiently determine if there are intersections while generating the boundary layer. Each ray segment or border segment has their axis-aligned bounding box projected to a 4-dimensional halfspace. The 4D points are stored in an alternating digital tree (ADT) and recursive searching is used to check for overlaps in the axis-aligned bounding boxes, or extent boxes.

Class members:

- ADT2DElement* root - Root element of the tree, default initialized to nullptr

Public member functions:

- ADT2D() - Sets *root* to *nullptr*
- ADT2D(ADT2D&& other)
- ~ADT2D() - Cascades deletion of all *ADT2DElement*s in the tree by starting with *root*
- void removeFirst(int id, const double* extent) - Removes the first occurrence of a 4D point in the tree that overlaps with a provided extent box
  Input:
  id - the identifier of the 4D point to remove
  extent - starting memory address for the testing extent box
- std::vector<int> retrieve(const double* extent) const - Retrieves the 4D points that overlap with a provided extent box
  Input:
  extent - starting memory address for the testing extent box
  Output:
  vector of ids of the 4D points that overlap
- void store(int id, const double* x) - Stores an extent box as a 4D point
  Input:
  id - non-unique identifier
  x - starting memory address for the 4D point

4.2 ADT2DElement

The ADT2DElement class is used to store individual nodes of the alternating digital tree.

Class members:

- int *id* - Non-unique identifier
- ADT2DElement* *left_child* - Initialized to *nullptr*
- int *level* - Used to determine which child to pick
- std::array<double, 4> *object* - The 4D point that the user provided
- ADT2DElement* *right_child* - Initialized to *nullptr*
- std::array<double, 4> *x_max* - The upper bound of this element's domain
- std::array<double, 4> *x_min* - The lower bound of this element's domain

Public member functions:

- ADT2DElement(int adt_level, const std::array<double, 4>& x_minimum, const std::array<double, 4>& x_maximum, int element_id, const double* object_coordinates)
- ~ADT2DElement()

- bool containsHyperRectangle(const std::array<double, 4>& a, const std::array<double, 4>& b) const - Returns true if the hyper-rectangle is inside this element's domain
  Input:
  a - the lower point of the hyper-rectangle
  b - the upper point of the hyper-rectangle
- bool hyperRectangleContainsObject(const std::array<double, 4>& a, const std::array<double, 4>& b) const - Returns true if the user-provided 4D point lies inside the hyper-rectangle Input:
  a - the lower point of the hyper-rectangle
  b - the upper point of the hyper-rectangle

4.3 ADT2DExtent

The ADT2DExtent class is used so that the user can interact with the ADT without having to worry about the coordinate transformations.

Class members:

- ADT2D *adt* - The underlying ADT used to store the elements
- ADTSpaceTransformer *space_transformer* - Performs the coordinate transformations to unit space or real space

Public member functions:

- ADT2DExtent(const Extent& domain) - Input:
  domain - the 2D bounding box that will be projected to a 4D point and used as the root of the ADT
- ~ADT2DExtent()
- void removeFirst(int id, const Extent& extent) - Removes the first occurrence of a 4D point in the ADT that overlaps with a provided bounding box
  Input:
  id - the identifier of the 4D point to remove
  extent - the 2D bounding box that will be projected to a 4D point
- std::vector<int> retrieve(const Extent& domain) const - Searches the ADT for elements that contain the provided bounding box
  Input:
  domain - the 2D bounding box that will be projected to a 4D point and checked for overlaps with other elements
  Output:
  vector of ids of the 4D points that overlap
- void store(int id, const Extent& extent) - Stores a 4D point in unit space in the ADT
  Input:
  id - the non-unique identifier for the new element
  extent - the 2D bounding box that will be projected to a 4D point

4.4 ADT2DSpaceTransformer

The ADTSpaceTransformer class is used as a helper class with the *ADT2D* class. This class provides the coordinate transformations from unit space to real space and real space to unit space

Class members:

- Extent *extent* - The real domain
- double *over_scale* - Used to convert from real space to unit space
  Equal to $1/scale$
- double *scale* - Used to convert from unit space to real space
  The length of the longest side of the domain

Public member functions:

- ADTSpaceTransformer(const Extent& domain) - Input:
  domain - the 2D bounding box that will represent the real domain
- ~ADTSpaceTransformer()
- inline const Extent& getDomain() const - Returns the domain in real space
- inline std::array<double, 2> toRealSpace(const std::array<double, 2>& unit_point) const
- inline std::array<double, 2> toUnitSpace(const std::array<double, 2>& real_point) const

4.5 Application

The Application namespace is used as set of helper functions to perform floating-point comparisons and basic mathematical point and Euclidean vector operations.
std::array<double, 2> is used to represent point and vector types, *point_t* and *vector_t*, respectively.

Namespace members:

- static const double *precision* - Constant used as the floating-point precision for floating-point comparisons
- static const double $degree_{r}adian_{r}atio$ - Constant used to convert between radians and degrees

Namespace functions:

- inline double angleBetweenEdgeAndAxis(const point_t& a, const point_t& b, bool axis) - Computes the angle between edge *ab* and the x-axis or y-axis
- inline double angleBetweenVectors(const vector_t& u, const vector_t& v)
- inline bool areEqual(const double& a, const double& b) - Returns true if the difference is less than *precision*
- inline bool areEqual(const point_t& a, const point_t& b)
- inline double calculateDistance(const point_t& a, const point_t& b)
- inline double calculateMagnitude(const vector_t& v)
- inline std::array<point_t, 2> computeBoundingBox(std::vector<point_t>::iterator first, std::vector<point_t>::iterator last)
- std::array<point_t, 2> computeBoundingBox(std::vector<Vertex>::iterator first, std::vector<Vertex>::iterator last)
- inline double crossProduct(const vector_t& a, const vector_t& b)
- inline double degreesToRadians(double angle)
- inline double dotProduct(const vector_t& a, const vector_t& b)
- inline double elapsedMsecs(const timeval& start, const timeval& stop)
- bool isClockwise(std::vector<Vertex>::iterator first, std::vector<Vertex>::iterator last)
- inline bool isZero(const double& value) - Returns true if value is less than *precision*

- inline point_t midPoint(const point_t& a, const point_t& b)
- inline bool pointRightOfEdge(const point_t& a, const point_t& b, const point_t& t)
- inline double radiansToDegrees(double radians)
- inline int relativeQuadrantToControlPoint(const point_t& p, const point_t& control) - The *control* point is treated as the origin

  The top-right quadrant is 0, the top-left quadrant is 1, the bottom-left quadrant is 2, and the bottom-right quadrant is 3
- inline double triangleArea(const point_t& a, const point_t& b, const point_t& c) - The triangle defined by *abc* should be wound counter-clockwise for a positive area
- inline double vectorDifference(const vector_t& a, const vector_t& b)

4.6 BoundaryLayerMesh

The BoundaryLayerMesh class is responsible for generating the high-fidelity anisotropic boundary layer points, edges, and initial subdomains that will be triangulated. This class checks for intersections in the boundary layer and resolves them, smooths poor quality ray regions, adds points to ensure a smooth transition to the inviscid region, and removes points that would result in poor quality triangles.

Class members:

- std::vector<long> *boundary_layer_element_start* - The id of the first vertex of each element that is not on the model's surface
- double *chord_length* - The distance along the x-axis of the input model
- double *cusp_angle_tolerance* - Ray angles larger than this value, but less than *trailing_edge_angle_tolerance* will be marked as a cusp
- std::vector<int> *cusps*
- std::vector<Edge> *edges*
- double *growth_rate*
- std::vector<double> *holes*
- std::vector<bl_subdomain_t> *initial_subdomains*
- InviscidRegionMesh* *inviscid_region* - The neighboring inviscid region
- std::vector<std::tuple<int, int, double» *large_angles*
- double *last_thickness* - The thickness of the final layer of the boundary layer
- std::vector<double> *layer_offsets* - The thicknesses of each layer
- std::vector<Vertex> *local_vertices* - A process' subset of the boundary layer vertices that it created
- long *max_boundary_layer_vertex_id*
- std::vector<AABB> *max_element_extent_boxes* - The largest extent box of each element
- AABB *max_extent_box* - The extent box that contains the entire boundary layer
- int *max_layers* - The maximum number of layers that can exist in the boundary layer. This value is used for gradation control and is set to $1.25 * num\_layers$
- std::array<double, 2> *mesh_center*
- MeshGenerator& *mesher* - The owning MeshGenerator
- int *next_edge_id*

- int *next_recv_process* - Used by the root process to distribute initial subdomains
- long *next_vertex_id*
- int *num_elements*
- int *num_enclosing_edges*
- int *num_layers*
- int *num_model_edges*
- int *num_model_vertices*
- double *ray_angle_tolerance* - If the angle between two rays is greater than this value, then those rays will be added to *large_angles*
- std::vector<int> *ray_element_start* - The index of the first ray of each element
- std::vector<Ray> *rays*
- std::vector<int> *sharp_trailing_edges*
- std::vector<int> *surface_element_start* - The id of the first vertex of each element
- int *total_initial_subdomains*
- double *trailing_edge_angle_tolerance* - Ray angles larger than this value will be marked as being a sharp trailing edge
- std::vector<std::vector<Vertex*» *transition_vertices* - The vertices that are on the enclosing border of the boundary layer
- std::vector<Vertex> *vertices*

Public member functions:

- BoundaryLayerMesh(MeshGenerator& owning_mesher)
- ~BoundaryLayerMesh()
- void createBoundaryLayerSubdomains() - Creates the initial boundary layer subdomains
- void decomposeInitialSubdomains() - Decomposes the initial subdomains of the boundary layer until each process has a subdomain or a subdomain cannot be further decomposed
- void initializeModelSurface(std::string filename) - Reads the input model referenced by the path filename
- void insertBoundaryLayerPoints()
- void receiveInitialSubdomains()
- void setGrowthFunction(double first_layer_thickness, double layer_growth_rate, int initial_layers) - Sets the growth function used to generate the anisotropic triangles
  Input:
  first_layer_thickness - the distance from the geometry that the points in the first layer will be placed
  layer_growth_rate - the geometric rate at which each layer's thickness will grow from the previous layer. A value of 1 causes all layers to have the same thickness as *first_layer_thickness*. A value greater than 1 causes subsequent layers to grow in thickness
  initial_layers - the number of anisotropic layers that will be originally grown from the geometry. Some triangles in some layers will be removed due to intersections or a poor quality shape

4.7 BoundaryLayerSubdomain

The BoundaryLayerSubdomain class is responsible for performing the paraboloid and lower convex hull decomposition steps in order to split a BoundaryLayerSubdomain into two new subdomains. The BoundaryLayerSubdomain class is also responsible for calling Triangle to triangulate its vertices.

Class members:

- bool *axis* - The coordinate-axis orthogonal to the cut axis
- int *decomposition_level* - Represents how many times this subdomain has been decomposed
- static int *decomposition_threshold* - The maximum number of times a subdomain can be decomposed
- std::vector<std::array<long, 2» *edges*
- std::vector<Vertex> *lower_convex_hull* - The vertices that lie on the lower convex hull of the flattened paraboloid in the vertical plane
- static long *max_vertex_id* - The maximum id for all of the boundary layer vertices
- Vertex* *median_vertex* - The median vertex along the coordinate-axis specified by *axis*
- std::shared_ptr<BoundaryLayerSubdomain> *sub_subdomain* - The other subdomain that is formed by a decomposition step
- std::array<std::vector<Vertex>*, 2> *vertices* - The memory addresses of *x_vertices* and *y_vertices*
- std::vector<Vertex> *x_vertices* - The vertices sorted lexicographically by their x-coordinates
- std::vector<Vertex> *y_vertices* - The vertices sorted lexicographically by their y-coordinates

Public member functions:

- BoundaryLayerSubdomain()
- ~BoundaryLayerSubdomain()
- Decomposition decompose() - Decomposes this subdomain into two new subdomains
- void mesh(BoundaryLayerMesh& owning_mesh) - Triangulates this subdomain and returns the output to *owning_mesh*
- void recvSubdomain(int source)
- void sendSubdomain(int destination)

4.8 Edge

The Edge class is responsible for storing information about mesh edges.

Class members:

- int *id* - Unique identifier
- int *type* - Classification
  0 for constrained
  1 for geometry
  2 for farfield boundary
  3 for boundary layer outer border
- std::array<long, 2> *vertices* - Identifiers of its endpoints

Public member functions:

- Edge() - Default constructor
  Sets everything to -1
- Edge(int edge_id, const Vertex& vertex_a, const Vertex& vertex_b, int edge_type) - Input:
  edge_id - unique identifier
  vertex_a - reference to starting endpoint
  vertex_b - reference to ending endpoint
  edge_type - classification of the type of edge
- Edge(int edge_id, long vertex_a, long vertex_b, int edge_type) - Input
  edge_id - unique identifier
  vertex_a - id of starting endpoint
  vertex_b - id of ending endpoint
  edge_type - classification of the type of edge
- ~Edge()
- static MPI_Datatype createMPIDatatype() - Defines the memory layout used for MPI communications
- int getId() const
- int getType() const
- long getVertexId(int index) const
- std::array<long, 2> getVertices() const
- void print() const
- void setType(int edge_type)
- void setVertices(Vertex& vertex_a, Vertex& vertex_b) - Sets the endpoint ids
  Input:
  vertex_a - reference to starting endpoint
  vertex_b - reference to ending endpoint
- bool shouldBeInFinalMesh() const - Returns true if the edge is part of the geometry or farfield boundary
  Used to apply boundary conditions

4.9 GeoPrimitives

This set of classes are used as helper objects for performing the boundary layer intersections tests.

The AABB class is used to prune the search space of candidate rays when checking for boundary layer intersections. This class is also used for the sizing function for the inviscid region triangles.

Class members:

- std::array<double, 2> *high* - The upper point of the bounding box
- std::array<double, 2> *low* - The lower point of the bounding box

Public member functions:

- AABB()
- AABB(std::array<double, 2> lo, std::array<double, 2> hi)
- AABB(std::array<std::array<double, 2>, 2> bounding_box)
- ~AABB()

- bool containsPortionOf(Segment s) const
- Extent getExtent() const
- std::array<double, 2> getHighPoint() const
- std::array<double, 2> getLowPoint() const
- void inflateDomain(double inflation) - Expands the domain by $inflation$ units in each direction $(+x, -x, +y, -y)$
- bool intersects(const AABB& other) const
- void setDomain(std::array<std::array<double, 2>, 2> bounding_box)

The Extent class is used with the ADT when performing the boundary layer intersection checks.
Class members:

- std::array<double, 2> hi - The upper point of the extent box
- std::array<double, 2> lo - The lower point of the extent box

Public member functions:

- Extent()
- Extent(const std::array<double, 2>& low, const std::array<double, 2>& high)
- ~Extent()
- bool contains(const Extent& extent) const

The Segment class is used for the boundary layer intersection checks.
Class members:

- std::array<double, 2> a - The starting endpoint
- std::array<double, 2> b - The ending endpoint

Public member functions:

- Segment()
- Segment(const Segment& s)
- Segment(std::array<double, 2> p1, std::array<double, 2> p2)
- ~Segment()
- bool doesIntersect(const Segment& s) const
- Extent getExtent() const
- std::array<double, 2> getPointA() const
- std::array<double, 2> getPointB() const
- std::vector<std::array<double, 2» intersectsAt(const Segment& s) const
- Orientation orientation2D(const std::array<double, 2>& t) const - Determines if a test point lies on, left, or right of the directed line ab
- void setA(std::array<double, 2> p)
- void setB(std::array<double, 2> p)

4.10 InviscidRegionMesh

The InviscidRegionMesh class is responsible for creating the isotropic nearbody and inviscid subdomains. It also contains tunable parameters to control the function that determines the desired triangle size for a point in space. These parameters are $fast\_growth$ which controls the distance where the size of triangles will grow at a more rapid rate and *uniform* which uses the size of the triangles in the nearbody region for the entirety of the inviscid region. The last parameter is *decoupling_work_threshold*. If you want smaller subdomains during the initial decoupling procedure, then decrease this number. The smaller the subdomains, the more subdomains there are, and the more concurrency that can be exploited.

Class members:

- std::vector<std::array<double, 2» *aabb_centers* - The center points of each input geometry element's bounding box
- const int *decoupling_work_threshold* - The largest an inviscid subdomain can be in terms of estimated number of triangles
- double *farfield* - The distance in chord lengths for the domain away from the geometry that will be meshed
- std::vector<Edge> *farfield_edges* - The edges that make up the outer border of the domain
  Used to apply boundary conditions for the flow solver
- double *fast_growth* - Number of chord lengths where triangles past this distance will grow at a faster rate
- std::priority_queue<std::shared_ptr<InviscidRegionSubdomain>, std::vector<std::shared_ptr<InviscidRegionSubdomain», std::less<std::shared_ptr<InviscidRegionSubdomain»> *initial_subdomains* - Stores the subdomain with the largest estimated number of triangles on top
  Yes std::less should be used for this because std::greater would put the smallest estimated subdomain on top
- double *isotropic_area* - The average area of triangles in the nearbody region.
  Used as the base size to grow from when creating a graded inviscid region, where triangles are larger the further they are away from the boundary layer
- MeshGenerator& *mesher* - The MeshGenerator that this inviscid region mesh belongs to
- std::array<std::array<double, 2>, 4> *nearbody_box* - The bounding box of the nearbody region
- int *next_edge_id*
- long *next_vertex_id*
- std::vector<double> *triangle_aabb_centers* - Used to call Triangle, contains the same data as *aabb_centers*
- bool *uniform* - True to generate uniform triangles with size *isotropic_area* in the entirety of the inviscid region

Public member functions:

- InviscidRegionMesh(MeshGenerator& owning_mesher)
- ~InviscidRegionMesh()
- void createInitialSubdomains() - Creates the initial nearbody subdomain and four inviscid subdomains that extent all the way to the farfield
- void decoupleInitialSubdomains() - Decouples all subdomains larger than *decoupling_work_threshold*
  Processes will send and receive subdomains from each other until everyone has some subdomains
- void receiveInitialSubdomains() - Called by all other processes except the root
- void setFarFieldDistance(double chord_lengths) - A square is generated that encloses the domain and each side is *chord_lengths* in each direction $(+x, -x, +y, -y)$ from the center of the mesh. One chord length is the length of the input geometry in the x-direction.
- void setNearBodyBoundingBox(std::vector<std::array<std::array<double, 2>, 2»& extents) -
- void synchronizeInviscidParameters() - Called by all processes so everyone has the same values of the parameters used for the triangle sizing functions

- void useUniformTriangles()

4.11 InviscidRegionSubdomain

Class members:

- int *cost* - Estimated number of triangles that will be in this subdomain
- std::vector<std::array<long, 2» *edges*
- bool *nearboy* - True if the subdomain contains a portion of the input geometry
- std::vector<Vertex> *vertices* - Wound counter-clockwise

Public member functions:

- InviscidRegionSubdomain() - Sets *nearbody* to false
- InviscidRegionSubdomain(bool nearbody_subdomain)
- ~InviscidRegionSubdomain()
- void createBorder() - Creates the counter-clockwise edges
- MPI_Datatype createMPIDataType() - Defines the memory layout used for MPI communications
- std::array<double, 2> getCenterPoint() const - Gets the center point of this subdomain's bounding box
- std::array<std::array<double, 2>, 2> getExtentBox() const - Computes the bounding box, used to compute the estimated number of triangles in the subdomain and to determine the center point
- void mesh(InviscidRegionMesh& owning_mesh) - Refines this subdomain and passes the resulting mesh back to *owning_mesh*
  This subdomain can be destroyed after this function returns
- void recvSubdomain(int source)
- void sendSubdomain(int destination)

4.12 MeshGenerator

Class members:

- BoundaryLayerMesh *boundary_layer*
- std::vector<std::shared_ptr<BoundaryLayerSubdomain» *boundary_layer_subdomains*
- int *final_edges* - Number of edges on the input geometry plus the edges on the farfield
  Used to apply boundary conditions
- std::vector<std::array<long, 3» *final_triangles* - The endpoint vertex ids of the resulting mesh triangles
- std::vector<std::array<double, 2» *final_vertices* - The x and y coordinates of the resulting mesh vertices
- std::unordered_map<long, long> *global_to_final* - Maps the non-sequential global id of a mesh vertex to its id in the resulting mesh
- std::string *input_name* - The name of the input file
  Used to name the output files
- InviscidRegionMesh *inviscid_region*
- std::priority_queue<std::shared_ptr<InviscidRegionSubdomain>, std::vector<std::shared_ptr<InviscidRegionSubdomain», std::less<std::shared_ptr<InviscidRegionSubdomain»> *inviscid_subdomains* - Holds all of the inviscid subdomains for a process
  std::less makes it so the subdomain at the top of the queue is the most expensive

- MeshingManager *manager*
- bool *meshing* - Flag that denotes if the worker thread is currently triangulating or refining a subdomain
- long $next\_final\_id$ - Unique sequential identifier to assign to the next final vertex that will be registered
- std::queue<TriangleData> *outbox* - Holds the triangulated or refined subdomain meshes
- const int *processes* - Number of distributed MeshGenerators working
- const int *rank* - Unique id for this process
- pthread_mutex_t $subdomain\_mutex$ - Used to synchronize access to $inviscid\_subdomains$ between the manager and worker thread
- std::vector<long> $subdomain\_num\_triangles$ - Number of triangles in each resulting subdomain mesh

Public member functions:

- MeshGenerator()
- ~MeshGenerator()
- void collectFinalMesh() - Collects all of the mesh entities on the root process so that the mesh can be output
- void meshDomain() - This function creates all the mesh vertices and mesh triangles using the number of processes launched by MPI
- void outputFUN3D() - Outputs the final mesh as a .msh file used for the flow solver FUN3D
- void outputShowMe() - Outputs files for Shewchuk's "Show Me" application
- void outputTecplot() - Outputs the final mesh as a Tecplot file
- void outputVTK() - Outputs the final mesh in the .vtk format, suitable for ParaView
- void readInputModel(std::string filename) - Reads the input model referenced by the path $filename$
- void setBoundaryLayerGrowthFunction(double first_thickness, double growth_rate, int initial_layers) - Sets the growth function used to generate the anisotropic boundary layer
  Input:
  $first\_thickness$ - the distance from the geometry that the points in the first layer will be placed
  $growth\_rate$ - the geometric rate at which each layer's thickness will grow from the previous layer. A value of 1 causes all layers to have the same thickness as $first\_thickness$. A value greater than 1 causes subsequent layers to grow in thickness, which is the intended usage
  $initial\_layers$ - the number of anisotropic layers that will be originally grown from the geometry. Some triangles in some layers will be removed due to intersections or a poor quality shape
- void setFarFieldDistance(double chord_lengths) - Sets the distance for the domain away from the geometry that will be meshed
  Input:
  $chord\_lengths$ - the distance in each direction $(+x, -x, +y, -y)$ from the center of the mesh. One chord length is the length of the input geometry in the x-direction
- void useUniformTrianglesForInviscidRegion() - Uses the same sized triangles to mesh the entirety of the inviscid region
  By default, the inviscid region uses smaller triangles near the geometry and grades to larger triangles farther away from the geometry

4.13 MeshingManager

The MeshingManager class is responsible for managing the progress of the MeshGenerator it is associated with. The MeshingManager will periodically update MeshGenerator's *work_units* estimate, check for messages, send and request work to and from other processes for load balancing. The MeshingManager of the root process keeps track of all of the finished processes and notifies everyone once all processes are finished meshing their subdomains. There are some tunable parameters in this class for the load balancing. You may need to tweak *low_work_threshold* and/or *min_work_threshold* for how aggressive you want the load balancing.

Class members:

- bool *all_finished*
- bool *finished* - True if the MeshGenerator has no subdomains remaining, and all other processes have a low amount of work
- int *finished_processes* - The root keeps track of this value
- int *low_work_threshold* - A process will request work if their *work_units* fall below this value
- MeshGenerator& *mesher* - The mesher that will be managed
- int *min_work_threshold* - The minimum value that *low_work_threshold* can reach
- std::vector<std::array<int, 2» *work_loads* - Contains candidate processes to request work from if this process' *work_units* falls below *low_work_threshold*
- std::atomic_int *work_units* - The current work load estimate for the number of remaining subdomains Concurrent accesses are well-defined
- int* *work_units_memory* - The underlying storage used by *work_units_window*
- MPI_Win *work_units_window* - The MPI object that facilitates RMA operations for checking how much work each process has

Public member functions:

- MeshingManager(MeshGenerator& owning_mesher) - Input:
  owning_mesher - The MeshGenerator that this object will manage
- ~MeshingManager()
- void manageMeshingProgress() - The main loop to manage progress once the MeshGenerator starts meshing its subdomains

4.14 MPICommunications

The MPICommunications namespace provides a templated wrapper for many of the MPI operations used in FEA_Mesher2D.

Namespace functions:

- void initialize() - Wrapper for MPI_Init
- void finalize() - Wrapper for MPI_Finalize
- int myRank()
- int numberOfProcesses()
- MPI_Datatype getType(int value) - Returns MPI_INT
- MPI_Datatype getType(size_t value) - Returns MPI_LONG
- MPI_Datatype getType(long value) - Returns MPI_LONG

- MPI_Datatype getType(double value) - Returns MPI_DOUBLE

Templated namespace functions:
All of these functions are wrappers for their corresponding MPI call and are templated by template<typename T>

- void Send(std::vector<T>& send_buffer, int size, int destination
- void Send(MPI_Datatype datatype, T& value, int destination)
- void Recv(std::vector<T>& recv_buffer, int size, int source)
- void Recv(MPI_Datatype datatype, T& value, int source)
- void Scatter(std::vector<T>& send_buffer, T& recv_value, int root)
- void Scatterv(std::vector<T>& send_buffer, std::vector<T>& recv_buffer, int root)
- void Gather(T value, std::vector<T>& recv_buffer, int root)
- void Gatherv(MPI_Datatype datatype, const std::vector<T>& send_buffer, std::vector<T>& recv_buffer, int root)
- void Gatherv(const std::vector<T>& send_buffer, std::vector<T>& recv_buffer, int root)
- void Broadcast(T& value, int root)
- void Broadcast(std::vector<T>& buffer, int buffer_size, int root)
- void Broadcast(std::vector<T>& buffer, int root)

4.15 Ray

The Ray class is responsible for storing information about the normals emitting from the input geometry. Rays are used to generate the anisotropic boundary layer and are smoothed, clipped, and grown to create valid and high-fidelity triangles.

Class members:

- bool can_grade - Flag to denote if this ray can have more points past *last_layer*
- int element - The input geometry element that this ray is incident upon
- int endpoint_id - The mesh id for the surface vertex that this ray is emitted from
- int last_layer - The index of the last layer, essentially, the number of layers
- long last_vertex_id - The id of the vertex at *last_layer* of this ray
- std::array<double, 2> normal_vector - The unit vector where points will be inserted along
- std::array<double, 2> point - The endpoint or base of the ray

Public member functions:

- Ray() - Only used to allocate memory for MPI communications
- Ray(const Vertex& end_point, std::array<double, 2> normal, int layers, int element_id) - Used when creating the fans at trailing edges
  Input:
  end_point - the input geometry vertex that will be the base of the ray
  normal - the unit vector that points will be inserted along
  layers - the number of points to insert along *normal_vector*
  element_id - the input geometry element that the ray is incident upon
- Ray(const Vertex& end_point, int layers, const BoundaryLayerMesh& owning_mesh) - Used when creating the initial rays from the input geometry

Input:
end_point - the input geometry vertex that will be the base of the ray
layers - the number of points to insert along *normal_vector*
owning_mesh - the mesh that this ray is a part of, used to calculate *normal_vector*

- ~Ray()
- void calculateNormalVector(const std::array<double, 2>& prev_point, const std::array<double, 2>& next_point) - Calculates the unique, topological normal that points outwards from the model
  Input:
  prev_point - the neighboring point before this ray's endpoint
  next_point - the neighboring point after this ray's endpoint
- void decreaseLayers(int desired) - Bounds checking to make sure the new value of *last_layer* is less than the previous value, but not less than zero Input:
  desired - the new value of *last_layer*
- double getMagnitude() const - Gets the length of *normal_vector*
- long layerVertexID(int layer) const - Gets the vertex id at the requested layer
- std::array<double, 2> pointAtDistance(double distance) const - Calculates the location of a point inserted along *normal_vector* from *point*
  Input:
  distance - the distance along *normal_vector* from *point*

4.16 Triangle

The Triangle package by Jonathan Shewchuk is used as the off-the-shelf Delaunay triangulator and refiner for meshing the subdomains. Only one function is used by *FEA_Mesher2D*, the triangulate function.

void triangulate(char* triswitches, struct triangulateio* in, struct triangulateio* out, struct triangulateio* vorout, double iso_area, int u, int comps, double* centers)
Input:
triswitches - The command that controls how Triangle functions
in - The input data structure used which stores the vertices, edges, and holes of a subdomain
out - The output data structure used which stores the mesh vertices and mesh triangles
vorout - Not used with our application
iso_area - The isotropic area to use for the sizing function
u - A flag to represent if uniformly-sized triangles should be used in the inviscid region
comps - The number of elements in the input geometry
centers - The center point of each of the input geometry's elements

4.17 TriangleData

The TriangleData struct is responsible for holding the output from calls to Triangle.
Class members:

- std::vector<long> *global_ids* - Unique global identifiers for the vertices

- int *num_triangles*
- int* *triangles* - Starting memory address for the endpoints of the mesh triangles
  Length is equal to three times *num_triangles*
- double* *vertices* - Starting memory address for the coordinates of the vertices
  Length is equal to twice the size of *global_ids*

Public member functions:

- TriangleData(double*& vertices_in, int num_triangles_in, int*& triangles_in, std::vector<long>& global_ids_in) - Acquires ownership of *vertices_in*, *triangles_in*, and *global_ids_in* and sets the arguments to a null state
- TriangleData(TriangleData&& td)
- ~TriangleData()

4.18 Vertex

The Vertex class is responsible for storing information about mesh vertices.

Class members:

- bool *boundary* - Flag for if the vertex is on a boundary edge
- std::array<double, 2> *coordinates* - The x and y coordinates of the point
- static MPI_Datatype *datatype* - Defines the memory layout used for MPI communications
- long *id* - Unique identifier
- bool *lower_convex_hull* - Flag for if the vertex is on the lower convex hull when decomposing the boundary layer
- std::array<double, 2> *projected* - The coordinates of the projected point on the vertical plane for decomposing the boundary layer

Public member functions:

- Vertex() - Sets *id* to -1
- Vertex(std::array<double, 2> point, bool boundary, long id) - Input:
  point - the x and y coordinates
  boundary - true if the vertex is on a boundary edge
  id - unique identifier
- ~Vertex()
- void calculateProjected(Vertex* median, bool axis) - Calculates the project coordinates of this vertex on the paraboloid
  Input:
  base - the base of the paraboloid
  axis - the non-z axis that is used in the vertical plane
- static void createMPIDataType() - Sets *datatype*
  Only needs to be called once by each process
- bool getBoundary() const
- double getCoordinate(int index) const - x-coordinate is *index* 0 & y-coordinate is *index* 1
- long getID() const
- static MPI_Datatype getMPIDataType()

- std::array<double, 2> getPoint() const
- double getProjected(int index) const - Projected x-coordinate is *index* 0 & projected y-coordinate is *index* 1
- bool orientation2D(const Vertex* a, const Vertex& b) const - Used to determine if a vertex should be part of the lower convex hull when decomposing the boundary layer
  Answers the question: which side of the directed line a->b does this vertex lie towards
  Input:
  a - pointer to the starting vertex of the directed line
  b - reference to the ending vertex of the directed line
- void print() const - Prints to std::cout
- void setBoundary(bool b)
- void setLowerConvexHull(bool exist)
- void setCoordinate(int index, double coordinate) - Sets the x-coordinate or y-coordinate to *coordinate* if *index* is 0 or 1, respectively
- bool useLowerConvexHull() - Returns true and sets to false if this vertex is on the lower convex hull when decomposing the boundary layer