DOCUMENTAZIONE PROGETTO DI PCS 2020/2021

* Antonino Emanuele Scurria -N. matricola : 269950
* Andrea Ruglioni -N. matricola : 272481
* Juan Carlos Vega Oliver -N. matricola : 262087

Di seguito verranno riportate brevi descrizioni dei metodi pubblici delle varie classi da noi implementate:

CLASSE “POINT” (DTO):

*Point:*

dati in input: Vector2d coordinates;

dati in output: none;

logica: crea una istanza della classe inizializzando l’attributo “\_coordinates”

al valore di “coordinates”;

*SetCoordinates :*

dati in input: const double x, const double y;

dati in output: none;

logica: setta l’attributo “\_coordinates” al valore di un vettore di componenti

“x” e “y”;

*GetCoordinates :*

dati in input: none;

dati in output: const Vector2d;

logica: getter dell’attributo “\_coordinates”;

*GetPointLabel :*

dati in input: none;

dati in output: int;

logica: getter dell’attributo “\_label”;

*SetPointLabel :*

dati in input: const int label;

dati in output: none;

logica: setter dell’attributo “label”;

*Operator = :*

dati in input: const Point & point;

dati in output: Point&;

logica: copia gli attributi di “point” nell’istanza che chiama il metodo;

*Operator + :*

dati in input: const Point& point;

dati in output: Point;

logica: crea un’istanza della classe che ha come attributo “\_coordinates”

la somma degli attributi “\_coordinates” dell’oggetto che chiama il metodo

e dell’oggetto dato in input;

*Operator - :*

dati in input: const Point& point;

dati in outpute: Point;

logica: crea un’istanza della classe che ha come attributo “\_coordinates”

la differenza degli attributi “\_coordinates” dell’oggetto che chiama il metodo

e dell’oggetto dato in input;

*Operator == :*

dati in input: const Point& p1, const Point& p2;

dati in output: bool;

logica: ritorna 1 se la norma della differenza degli attributi “\_coordinates”

dei due punti è minore o uguale di una certa tolleranza dichiarata

tramite apposita MACRO, 0 altrimenti;

*Operator != :*

dati in input: const Point& p1, const Point& p2;

dati in output: bool;

logica: ritorna 1 se l’operatore “==” ritorna 0, 0 altrimenti;

CLASSE “SEGMENT” (DTO):

*Segment :*

dati in input: const Point& start, const Point& end;

dati in output: none;

logica: inizializza gli attributi “\_start” e “\_end” rispettivamente al valore

di “start” e “end” e inizializza l’attributo “\_tangent” alla differenza tra gli

attributi “\_coordinates” di “end” e “start”;

*SetStart :*

dati in input: const Point& start;

dati in output: none;

logica: setter dell’attributo “\_start” al valore di “start”;

*GetStart :*

dati in input: none;

dati in output: const Point&;

logica: ritorna l’attributo “\_start”;

*SetEnd :*

dati in input: const Point& end;

dati in output: none;

logica: setter dell’attributo “\_end” al valore di “end”;

*GetEnd :*

dati in input: none;

dati in output: const Point&;

logica:ritorna l’attributo “\_end”;

*SetSegment :*

dati in input: const Point& start, const Point& end;

dati in output: none;

logica: setta gli attributi “\_start” e “\_end” rispettivamente al valore

di “start” e “end”;

*ComputeTangent :*

dati in input: none;

dati in output: Vector2d&;

logica: inizializza l’attributo “\_tangent” alla differenza tra gli

attributi “\_coordinates” di “\_end” e “\_start”;

*GetTangent :*

dati in input: none;

dati in output: const Vector2d&;

logica: getter dell’attributo “\_tangent”;

*PointOnTheRightSide :*

dati in input: const Point& point;

dati in output: int;

logica: crea un segmento che ha come origine l’attributo “\_start” dell’oggetto

che chiama il metodo e come fine “point”; successivamente viene effettuato

un prodotto vettoriale tra la tangente del segmento che chiama il metodo

e la tangente del nuovo segmento creato; se il prodotto vettoriale e’ minore

della tolleranza prefissata allora il punto si trova alla sinistra del segmento

chiamante il metodo e viene ritornato 1; se il prodotto vettoriale ha modulo

minore della tolleranza il punto viene considerato sul segmento e viene

ritornato 0; se il prodotto vettoriale ha valore superiore alla tolleranza, il

punto si trova alla destra del segmento e viene ritornato -1.

*Operator = :*

dati in input: const Segment&;

dati in output: Segment&;

logica: crea un’istanza della classe copiando in questa i valori degli attributi

posseduti dal segmento dato in input;

*Operator == :*

dati in input: const Point&;

dati in output: bool;

logica: se gli attributi “\_start” e “\_end” del segmento chiamante il metodo e

del segmento dato in input sono uguali ( tramite l’apposito operatore “==”)

viene ritornato 1, altrimenti 0;

CLASSE “POLYGON”:

Polygon:

dati in input: const vector<Point>& vertices;

dati in output: none;

logica: crea un’istanza della classe “Polygon” inzializzando il valore dell’

attributo “\_numberVertices” al valore ottenuto da “vertices.size()”;

dopo aver effettuato un’apposita “reserve” sull’attributo “\_vertices”

inizializza l’attributo “\_vertices” con i valori di “vertices”; infine

inizializza l’attributo “\_edges” (vector<Segment>) , effettuando

una preventiva resize al valore di “\_numberVertices”, e ,iterando,

inizializza l’i-esimo elemento di “\_edges” con un’istanza della classe

“Segment”, chiamando il costruttore della classe appena citata, e

dandogli come dati di input (effettuando un preliminare controllo sugli

indici per non andare fuori dal range del vettore “vertices”) “vertices[i]”

(inizio del segmento), e “vertices[i+1] (fine del segmento).

*GetNumberVertices:*

dati in input: none;

dati in output: unsigned int;

logica: ritorna il numero dei vertici di un oggetto della classe “Polygon”;

*GetVertices:*

dati in input: none;

dati in output: vector<Point>

logica: ritorna un vettori di oggetti del tipo “Point” corrispondenti ai vertici dell’oggetto di tipo “Polygon” che chiama il metodo

*SetVertexAtPosition:*

dati in input: const Point& vertex, const int& position;

dati in output: none

logica: il metodo opera un’assegnazione dell’attributo

“\_vertices” (vector<Point>) alla posizione indicata da “position”;

*AppendVertex:*

dati in input: const Point& vertex;

dati in output: none;

logica: effettua un’operazione di “push\_back” di “vertex” sull’attributo

“\_vertices” (vector<Point>) e aggiorna l’attributo \_numberVertice

(unsigned int)al valore di \_vertices.size();

*InsertVertexAtPosition:*

dati in input: const Point& vertex, const int& position ;

dati in output : none

logica : effettua un’operazione di “insert” di “vertex” alla posizione

indicata da “position” nell’attributo “\_vertices”(vector<Point>);

*GetVertexAtPosition:*

dati in input: const int& position;

dati in output: const Point&;

logica: getter dell’oggetto “Point”alla posizione “position” in “\_vertices”;

*NextVertexId:*

dati in input: const int id;

dati in output: int;

logica: ritorna “id + i” se quest è entro i limiti della size di “\_vertices”;

*GetLastVertex:*

dati in input: none;

dati in output: const Point&;

logica:ritorna l’ultimo elemento nell’attributo “\_vertices”(vector<Point>);

*ComputeArea:*

dati in input: none;

dati in output: double;

logica: applica il metodo del calcolo di aree di Gauss su tutti i vertici del

poligono;

*PointInPolygon:*

dati in input: const Point&;

dati in output: bool;

logica: itera su tutti i lati del poligono, la cui informazione è contenuta

nell’attributo “\_edges”, il metodo della classe “Segment”

“PointOnTheRightSide” per controllare se un punto stia alla

sinistra di ogni lato e dunque, poiché questo metodo viene usato

esclusivamente per poligoni convessi, all’interno del poligono;

*GetEdges:* dati in input: none;

dati in output: vector<Segment>&;

logica: getter dell’attributo “\_edges”;

*Operator “=”:*

dati in input: const Polygon& polygon;

dati in output: Polygon&;

logica: overloading dell’operatore “=”: i dati della varibile “polygon”

vengono copiati nell’istanza che chiama il metodo;

*SetVertexLabelAtPosition:*

dati in input: const int& position, int label;

dati in output: none;

logica: setta l’attributo “\_label” dell’oggetto della classe “Point” alla

posizione indicata da “position” nell’attributo”\_vertices”

al valore di “label”;

*GetPointLabelAtPosition:* dati in input: const int& position;

dati in output: int;

logica: ritorna in valore dell’attributo “\_label” dell’istanza della classe

“Point” che si trova alla posizione indicata da “position”

nell’attributo “\_vertices”;

*SetVertexLabels:* dati in input: const vector<unsigned int>& polygonVertices;

dati in output: none;

logica: usando il metodo della classe “Point” “SetPointLabel”, setta

il valore dell’attributo “\_label” dell’i-esimo elemento

di “\_vertices” al valore dell’i-esimo elemento di “polygnVertices”;

*SetDefaultVertexLabels:*

dati in input: none;

dati in output: none;

logica: Juan devi farlo;

*IsPolygonOnTheRightSide:*

dati in input: const Segment& line;

dati in output: int;

logica: itera su tutti i vertici del poligono il metodo associato alla classe

“Point” “PointOnTheRightSide: cosí se il numero di vertici del poligono

che si trovano a destra del segmento corrisponde al numero stesso dei

vertici, il poligono si troverá a destra del segmento, mentre se abbiamo

che tutti i vertici si trovano a sinistra del segmento, il poligono si troverá

a destra dello stesso;

*BuildUpPolygonEdges:*

dati in input: none;

dati in output: none;

logica: riempe l’attributo “\_edges” del poligono chiamante il metodo, iterando

sui vertici del poligono e chiamando il costruttore di “Segment” a cui

vengono dati in input il vertice in esame e il successivo.

CLASSE “INTERSECTOR1D1D”:

*Intersector1D1D:*

dati in input: none;  
 dati in output: none ;

logica: crea un’istanza della classe “Intersecor1D1D” inizializzando l’

attributo “resultParametricCoordinates” a un vettore di due

componenti entrambe pari a 0; inizializza inoltre il tipo di intersezione

definito dalla variabile “type” (che è di tipo Type) al valore

“NoIntersection”;

*SetFirstSegment:* dati in input: const Vector2d& origin, const Vector2d& end;

dati di output: none;

logica:

*SetSecondSegment:* dati in input: const Vector2d& origin, const Vector2d& end;

dati di output: none;

logica:

*ComputeIntersection:*

dati in input: none;

dati in output: bool;

logica: algoritmo che riconosce se vi è intersezione tra due segmenti

,prolungando il primo sulla sua retta di giacitura fino a farlo intersecare

(se vi è intersezione) col secondo segmento. Nel

fare ció, classifica l’intersezione in base ai tipi presenti nelle

enumerazioni: vi è il caso “NoIntersection” che indica la non presenza

di un’intersezione; abbiamo poi IntersectionOnSegment che indica

la presenza di un intersezione sul segmento (e non sulla retta in cui

giace); IntersectionOnLine indica che vi è intersezione

al di fuori del segmento (e dunque sul prolungamento di quest’ultimo);

ParalleIntersectionOnSegment indica che vi è un ‘intersezione del tipo

IntersectionOnSegment ma i due segmenti sono paralleli( bisogna

considerare questo caso come se i due segmenti giacessero per una

parte l’uno sull’altro); infine abbiamo ParallelIntersectionOnLine che

indica un’intersezione del tipo IntersectionOnLine quando i due

segmenti sono paralleli. Le posizioni delle intersezioni sono a sua volta

divise in 4 casi: Inner indica che il punto di intersezione è all’interno

del segmento; Outer indica che il punto di intersezione è all’esterno

del segmento; Begin indica che il punto di intersezione coincide con il

punto di partenza del segmento; End indica che il punto di intersezione

coincide con la fine del segmento. L’algoritmo dunque procede grazie

a semplice teoria di geometria analitica, cercando di ridurre al minimo

il costo computazionale delle operazioni. ( essenzialmente si tratta di

risolvere un sistema tra una retta e un segmento)

*ParametricCoordinates:*

dati in input: none;

dati in output: const Vector2d&;

logica: ritorna l’attributo “resultParametricCoordinate”;

*FirstParametricCoordinates:*

dati in input: none;

dati in output: const double&;

logica: ritorna la prima componente di “resultParametricCoordinate”;

*SecondParametricCoordinates:*

dati in input: none;

dati in output: const double&;

logica: ritorna la seconda componente di “resultParametricCoordinate”;

*PositionIntersectionInFirstEdge:*

dati in input: none;

dati in output: Position&;

logica: ritorna la posizione(codificata attraverso un’enumerazione che assume

4 diversi valori) dell’intersezione nel primo segmento ;

*PositionIntersectionInSecondEdge:*

dati in input: none;

dati in output: Position&;

logica: ritorna la posizione(codificata attraverso un’enumerazione che assume

4 diversi valori) dell’intersezione nel secondo segmento ;

*IntersectionFirstParametricCoordinate:*

dati in input: const Vector2d origin, const Vector2d end;

dati in output: Vector2d

logica: ritorna il punto di intersezione rispetto alla retta costruita sulla

tangente del primo segmento, e avente

origine nel punto di origine del segmento stesso (si consideri che il

segmento è parametrizzato in ascissa curvilinea)

*IntersectionFirstParametricCoordinate:*

dati in input: const Vector2d origin, const Vector2d end;

dati in output: Vector2d

logica: ritorna il punto di intersezione rispetto alla retta costruita sulla

tangente del secondo segmento, e avente

origine nel punto di origine del segmento stesso (si consideri che il

segmento è parametrizzato in ascissa curvilinea)

CLASSE “IntersectorPolygonLine”:

*IntersectorPolygonLine:* dati in input: IIntersector1D1D& intersector, IPolygon& polygon, ISegment&

Segment;

dati in output: none;

logica: crea un instanza della classe “intersectorPolygonLine” inizializzando gli

attributi “\_intersector”, “\_polygon” e “\_line” ai rispettivi parametri

passati in input. Inoltre inizializza i vettori di supporto necessari per il taglio del poligono ai loro valori di default.

*FindIntersectionVertices:* dati in input: none;

dati in output: const vector<Point>&;

logica: effettua un ciclo su tutti i segmenti del poligono e tramite l “\_intersector” vengono ricercati i punti di intersezione che vengono ordinati in base alla coordinata parametrica o all’id del segmento di intersezione tramite delle mappe e multimappe. Grazie a queste vengono generati due vettori di supporto per trovare i poligoni tagliati.

Infine viene assegnato ad ogni punto di intersezione e del poligono una label associata univocamente alle coordinate del punto, in modo da definire univocamente un sotto-poligono come un vettore di label.

*FindPolygons:*

dati in input: none;

dati in output: const vector<Polygon>&;

logica: dopo aver trovato i punti di intersezione studia i vertici del poligono in verso antiorario formando i vari sotto-poligono in cui viene tagliato dal segmento. L’algoritmo termina quando tutti i vertici sono stati esplorati e restituisce i poligoni formatisi.

*GetExtendedPolygon:* dati in input: none;

dati in output: const vector<Polygon>&;

logica: ritorna l’attributo “\_extendedNewPolygons”, che rappresenta i poligoni tagliati a cui vengono aggiunti gli estremi del segmento se sono all’ interno del poligono.

*FindNewPoints:*

dati in input: none;

dati in output: const vector<Point>&;

logica:

*GetStartFound:* dati in input: none;

dati in output: bool;

logica: ritorna l’attributo “\_startfound”;

*GetEndFound:* dati in input: none;

dati in output: bool;

logica: ritorna l’attributo “\_endfound”;

CLASSE “PolygonCutter”:

*PolygonCutter:* dati in input: IIntersectorPolygonLine& intersector, IPolygon& polygon, ISegment& segment;

dati in output: none;

logica: crea un instanza della classe “PolygonCutter” inizializzando gli attributi “\_intersector”, “\_polygon” e “\_cutter” necessari al taglio del poligono.

*CutPolygon:* dati in input: none;

dati in output: const vector<Polygon>&;

logica: tramite l “\_intersector” utilizza le funzioni “findIntersectionVertices”,

“findNewPoints” e “findPolygons” per ricavare i sotto-poligoni che restituisce.

*GetCuttedPolygon:* dati in input: none;

dati in output: const vector<Polygon>&;

logica: restituisce i poligoni in cui viene suddiviso “\_polygon” da “\_cutter”

*GetIntersectionVertices:* dati in input: none;

dati in output: const vector<Point>&;

logica: restituisce i punti di intersezione tra “\_polygon” e “\_cutter”

*GetNewPoints:* dati in input: none;

dati in output: const vector<Point>&;

logica: restituisce tutti i punti ottenuti che non fanno parte dei vertici del poligono, quindi i punti di intersezione e gli estremi del segmento “\_cutter”

*ShowPolygon:* dati in input: const string filepath = “showPolygon”;

dati in output: none;

logica: genera uno script matlab per visualizzare “\_polygon” ed i poligoni che vengono formati dall’intersezione.

CLASSE “ReferenceElement”:

*ReferenceElement:*

dati in input: Polygon& polygon;

dati in output: none;

logica: effettua una resize(4) sull’attributo “\_boundingBoxVertices”;

*ComputeBoundingBox:*

dati in input: Polygon& polygon;

dati in output: none;

logica: inizialmente ricerca i piú grandi e piccoli valori assunti in ascissa e

ordinata e con questi inizializza delle variabili ausiliare create

appositamente (minX, maxX ecc…); infine con queste ultime inizializza

i quattro vertici del bounding box (\_boundingBoxVertices[i] sará

inizializzato con l’i-esimo vertice del rettangolo, costruito attraverso

l’apposito costruttore della classe “Point”, considerando che vengono

percorsi in senso antiorario e partendo dal vertice in basso a sinistra).

*ComputeReferenceElement:*

dati in input: Polygon& polygon;

dati in output: none;

logica: il metodo ha l’obiettivo di dividere il poligono richiuso dentro il

bounding box in sottopoligoni, formati dai punti di intersezione tra

i lati del poligono e quelli del bounding box: il primo passo consiste

nel trovare il primo vertice (i vertici sono ordinati secondo

l’indicizzazione originaria del poligono in esame) che ha intersezione

con il lato inferiore del bounding box (che è un rettangolo). In seguito,

vengono visitati tutti i restanti punti del poligono, iterando quindi un

numero di volte pari al numero di vertici - 1, e, ogni qual volta viene

trovato un vertice che abbia intersezione con i lati del bounding box

,rispettando determinate condizioni, si forma un sottopoligono.

Le intersezioni vengono identificate non attraverso l’utilizzo dell’

“Intersector1D1D”, ma si sfrutta il fatto che le intersezioni avvengono

con il bounding box che è per l’appunto un rettangolo: le intersezioni

vengono identificate se i vertici del poligono hanno stessa ascissa o

ordinata dei 4 punti che delimitano e identificano il bounding box.

Le condizioni prima dette, a cui bisogna prestare attenzione, riguardano

casi limite in cui non appena trovato un vertici che abbia intersezione,

prima di formare un nuovo poligono, bisogna verificare che il vertice

precedente abbia intersezione su un lato diverso del bounding box

(si pensi al caso in cui il poligono da circondare col bounding box sia

esso stesso un rettangolo e, dunque, coincida col bounding box) in

quanto si formerebbe un nuovo poligono che è in realtá un segmento.

Una volta trovato dunque un vertice che “chiuda” il sottopoligono

viene costruito un nuovo poligono formato da tutti i vertici compresi

tra il primo vertice (che per costruzione ha intersezione con uno dei

lati) e il successivo vertice che ha intersezione (comprendendo

con opportuni controlli i vertici stessi del bounding box, che vengono

inclusi qualora il sottopoligono non venga formato attraverso vertici

del poligono che hanno intersezione su di uno stesso lato).

*GetBoundingBoxVertices:*

dati in input: none;

dati in output: const vector<Point>&;  
 logica: ritorna l’attributo “\_boundingBoxVertices”;

*GetReferenceElementPolygons:*

dati in input: none;

dati in output: const vector<Polygon>&;

logica: ritorna l’attributo “\_referenceElementPolygons”;

*GetBoundingBoxVertexAtPosition:*

dati in input: unsigned int& I;

dati in output: const Point&;

logica: ritorna \_boundingBoxVertices[i];

*GetBoundingBox:*

dati in input: none

dati in output: const Polygon&

logica: ritorna \_boundingBox

*FindBoundaryVertices:*

dati in input: none;

dati in output: none;

logica: cicla su i poligoni trovando i punti che si trovano sulla bounding box

e riempe le mappe getMapSouthXCoordinates, getMapNorthXCoordinates, getMapEastYCoordinates, getMapWestYCoordinates secondo il lato sul quale si trovano

*MakeConforming:*

dati in input: none;

dati in output: none;

logica: cicla su i poligoni della reference Element identificando per ogni lato

se si trova sulla bounding box. Se è così cicla su i punti della mappa corrispondente al lato opposto e se ne trova qualcuno dentro il lato che non sia un angolo della bounding box e che non coincida con i capi del segmenti, lo proietta nel lato in analisi. Alla fine abbiamo un vettore di poligoni conformi e quindi un reference element conforme

*GetConformingReferenceElementPolygons:*

dati in input: none;

dati in output: const vector<Polygon>&;

logica: ritorna \_conformingReferenceElementPolygons

*GetMapSouthXCoordinates:*

dati in input: none;

dati in output: const map<double, Point>&;

logica: getter dell’attributo “\_mapSouthXCoordinate”;

*GetMapNorthXCoordinate:*

dati in input: none;

dati in output: const map<double, Point>&;

logica: : getter dell’attributo “\_mapNorthXCoordinate”;

*GetMapEastYCoordinate:*

dati in input: none;

dati in output: const map<double, Point>&;

logica: : getter dell’attributo “\_mapEastYCoordinate”;

*GetMapWestYCoordinate:*

dati in input: none;

dati in output: const map<double, Point>&;

logica: : getter dell’attributo “\_mapWestYCoordinate”;

*ShowReferenceElement:*

dati in input: const string filepath = “showReferenceElement”;

dati in output: none;

logica: genera uno script matlab che mostra il bounding box del poligono e tutti i poligoni al suo interno.

CLASSE “Mesh”:

*Mesh:*

*CreateMesh:*

dati in input: const Polygon& domain, ReferenceElement& referenceElement;

dati in output: none;  
logica: Trasla il reference Element conforme all’origine di coordinate.

Utilizzando gli attributi \_maxNumRefElementsPerColumn e \_maxNumRefElementsPerRow per sapere i limiti massimi dei cicli, trasla il referenceElement conforme di un vettore di dimensioni multipli delle dimensioni della bounding box del poligono di referenza usando la funzione *TranslateConformingReferenceElement* prima riempiendo la mesh da sinistra a destra fino a completare una riga e poi dal basso all’alto per le righe successive.

Per ogni elemento traslato utilizza *referenceElementInConvexDomain per* determinare se si trova dentro del dominio convesso, se è così l’aggiunge direttamente al vettore \_meshCells, altrimenti cicla su i lati del dominio cercando un’intersezione con ongi lato della bounding box dell’elemento. Se non c’è intersezione vuol dire che l’elemento è esterno per cui non è aggiunto alla mesh. Se invece c’è intersezione taglia l’elemento e aggiorna i suoi poligoni con i nuovi poligoni generati dal taglio che cadono dentro il dominio usando la funzione *cutAndDiscardPolygonsOnRightSide*. Le celle della mesh contengono poligono conformi, per cui tutta la mesh è conforme.

*TranslateConformingReferenceElement:*

dati in input: double xTranslation, double yTranslation

dati in output: ReferenceElement

logica: trasla il reference element comprendente i punti “nord”, “sud”

“est” e “ovest” che rendono coerente la mesh.

*TranslatePolygon:*

dati in input: double xTranslation, double yTranslation;

dati in output: ReferenceElement;

logica: trasla il poligono dato in input di una quantitá pari a

“xTranslation” e “yTranslation” rispettivamente lungo l’asse x e y;

*SetRectangularDomain:*

dati in input: const double& domainLength, const double& domainHeight;

dati in output: static Polygon;

logica: Riempe gli attributi \_domain, \_domainBoundingBox,

\_domainBoundingBoxLength, \_domainBoundingBoxHeight, \_maxNumRefElementsPerRow, \_maxNumRefElementsPerColumn con cui calcola il numero di celle della mesh e reserva la memoria necessaria. Il tutto è referito a un dominio rettangolare

*SetConvexDomain:*

dati in input: const Polygon & convexDomain;

dati in output: none

logica: Riempe gli attributi \_domain, \_domainBoundingBox,

\_domainBoundingBoxLength, \_domainBoundingBoxHeight, \_maxNumRefElementsPerRow, \_maxNumRefElementsPerColumn con cui calcola il numero di celle della mesh e reserva la memoria necessaria che come massimo si utizzarebbe secondo la bounding box del dominio. Il tutto è referito a un dominio convesso generico

*ReferenceElementInConvexDomain:*

dati in input: ReferenceElement& referenceElement;

dati in output: bool

logica: itera sui vertici dell’attributo “\_boundingBoxVertices” il metodo

“PointInPolygon”: cosí, trattando solo domini convessi, abbiamo che

se tutti i vertici del bounding box sono all’interno del poligono, il

reference element stesso si troverá all’interno del dominio in esame;

*CutAndDiscardPolygonOnRightSide:*

dati in input: ReferenceElement& referenceElement, Segment& line;

dati in output: none;

logica: Crea un vettore di poligoni chiamato newVectorOfPolygons dove

memorizza i nuovi poligoni che calcola nella seguente maniera: Analizza ognuno dei poligoni del vettore di poligoni del reference Element per determinare se si trova a destra, a sinistra o nessuno dei due con rispetto al segmento dato in input. Se si trova a destra, lo butta via, se si trova a sinistra lo aggiunge a newVectorOfPolygons e se non è né a destra né, talgia il poligono , butta via i poligoni risultanti che sono a destra e aggiunge a newVectorOfPolygons i poligoni a sinistra. Alla fine crea un reference Element con questi poligoni e riempe gli attributi corrispondenti

*GetBoundingBoxLenght:*

dati in input: none;

dati in output: double&;

logica: getter dell’attributo “\_boundingBoxLenght”;

*GetBoundingBoxHeight:*

dati in input: none;

dati in output: double&;

logica: getter dell’attributo “\_boundingBoxHeight”

*GetMaxNumRefElementsPerRow:*

dati in input: none;

dati in output: unsigned int;

logica: getter dell’attributo “\_maxNumRefElementsPerRow”;

*GetMaxNumRefElementsPerColumn:*

dati in input: none;

dati in output: unsigned int;

logica: getter dell’attributo “\_maxNumRefElementsPerColumn”;

*GetDomainBoundingBox:*

dati in input: none;

dati in output: const Polygon&;

logica: getter dell’attributo “\_domainBoundingBox”;

*GetDomain:*

dati in input: none;

dati in output: const Polygon&;

logica: getter dell’attributo “\_domain”;

*GetMeshCells:*

dati in input: none

dati in output: const vector<ReferenceElement>&;

logica: getter dell’attributo “\_meshCells”;

*GetReferenceElementBoundingBox:*

dati in input: none;

dati in output: const Polygon&;

logica: getter dell’attributo “\_referenceElementBoundingBox”;

*GetDomainBoundingBoxLenght:*

dati in input: none

dati in output: double&;

logica: getter dell’attributo “\_domainBoundingBoxLenght”;

*GetDomainBoundingBoxHeight:*

dati in input: none

dati in output: double&

logica: getter dell’attributo “\_domainBoundingBoxHeight”;

*GetMeshCells:*

dati in input: none;

dati in output: const vector<ReferenceElement>&;

logica: ritorna l’attributo “\_meshCells”;

*GetReferenceElement:*

dati in input: none;

dati in output: ReferenceElement&;

logica: getter dell’attributo “\_referenceElement”

*ShowMesh:*

dati in input: const string filePath = “showMesh.m”;

dati in output: none

logica: crea uno script Matlab con cui andare a visualizzare graficamente

l’algoritmo;