PCS\_PROJECT 07/06/2021:  
INTERSECTOR AND MESH ALGORITHM

C++:

POINT\_CLASS.HPP

#ifndef POINT\_H

#define POINT\_H

#include "Eigen"

#include "MACROS.h"

using namespace Eigen;

class **Point**

{

private:

Vector2d \_coordinates;

int \_label = -2;

public:

**Point**(const double x = 0.0, const double y = 0.0);

**Point**(Vector2d coordinates);

**Point**(const Point& point);

void **setCoordinates**(const double x, const double y) {\_coordinates(x, y);}

const Vector2d& **getCoordinates**() const {return \_coordinates;}

int **getPointLabel**() const{ return \_label;}

void **setPointLabel**(const int label);

Point& operator = (const Point& point);

Point operator + (const Point& point);

Point operator - (const Point& point);

friend bool operator == (const Point& p1, const Point& p2);

friend bool operator != (const Point& p1, const Point& p2);

};

#endif

POINT\_CLASS.CPP

#include "Point.hpp"

#include "Eigen"

using namespace Eigen;

Point::**Point**(const double x, const double y)

: \_coordinates(x, y)

{

}

Point::**Point**(Vector2d coordinates)

{

\_coordinates = coordinates;

}

Point::**Point**(const Point &point)

{

\_coordinates = point.\_coordinates;

\_label = point.\_label;

}

void Point::**setPointLabel**(int label)

{

\_label = label;

}

Point &Point::operator =(const Point &point)

{

\_coordinates = point.getCoordinates();

\_label = point.getPointLabel();

return \*this;

}

Point Point::operator +(const Point &point)

{

return Point(\_coordinates + point.getCoordinates());

}

Point Point::operator -(const Point &point)

{

return Point(\_coordinates - point.getCoordinates());

}

bool operator == (const Point& p1, const Point& p2)

{

double maxnorm= p1.getCoordinates().norm();

double norm2= p2.getCoordinates().norm();

if( norm2 > maxnorm)

maxnorm = norm2;

return ((p1.getCoordinates() - p2.getCoordinates()).norm()<= POINT\_TOLERANCE\*maxnorm );

}

bool operator != (const Point& p1, const Point& p2)

{

return !(p1 == p2);

}

SEGMENT\_CLASS.HPP

#ifndef SEGMENT\_H

#define SEGMENT\_H

#include "Point.hpp"

using namespace Eigen;

class **ISegment**

{

public:

virtual void ***setStart***(const Point& start) = 0;

virtual const Point& ***getStart***() const = 0;

virtual Point& ***getStart***() = 0; // non-const because in some case we have to change the labels

virtual void ***setEnd***(const Point& end) = 0;

virtual const Point& ***getEnd***() const = 0;

virtual Point& ***getEnd***() = 0; // non-const because in some case we have to change the labels

virtual void ***setSegment***(const Point& start, const Point& end) = 0;

virtual const Vector2d& ***computeTangent***() = 0;

virtual const Vector2d& ***getTangent***() const = 0;

virtual int ***pointOnTheRightSide***(const Point& point) const = 0;

};

class **Segment** : public ISegment

{

private:

Point \_start;

Point \_end;

Vector2d \_tangent;

public:

**Segment**(const Point& start = Point(), const Point& end = Point());

void ***setStart***(const Point& start) {\_start = start;}

const Point& ***getStart***() const {return \_start;}

Point& ***getStart***() {return \_start;} // non-const because in some case we have to change the labels

void ***setEnd***(const Point& end) {\_end = end;}

const Point& ***getEnd***() const {return \_end;}

Point& ***getEnd***() {return \_end;} // non-const because in some case we have to change the labels

void ***setSegment***(const Point& start, const Point& end) {\_start = start; \_end = end;}

const Vector2d& ***computeTangent***();

const Vector2d& ***getTangent***() const {return \_tangent;}

int ***pointOnTheRightSide***(const Point& point) const;

Segment& operator = (const Segment& segment);

friend bool operator == (const Segment& s1, const Segment& s2);

};

#endif

SEGMENT\_CLASS.CPP

#include "Segment.hpp"

#include <iostream>

using namespace Eigen;

Segment::**Segment**(const Point &start, const Point &end)

: \_start(start), \_end(end)

{

\_tangent = end.getCoordinates() - start.getCoordinates();

}

const Vector2d &Segment::***computeTangent***()

{

\_tangent = (\_end - \_start).getCoordinates();

return \_tangent;

}

int Segment::***pointOnTheRightSide***(const Point &point) const

{

int val;

double crossProduct;

Segment segment(\_start, point);

//computing cross product

crossProduct =(\_tangent.x() \* segment.*getTangent*().y()) - (segment.*getTangent*().x() \* \_tangent.y());

if (crossProduct < - POINT\_TOLERANCE)

{

val = 1; //si trova a destra

return val;

}

else if(crossProduct > POINT\_TOLERANCE)

{

val = -1; //si trova a sinistra

return val;

}

else

{

val = 0; // il punto è sulla stessa retta del segmento

return val;

}

}

Segment &Segment::operator =(const Segment &segment)

{

\_start = segment.\_start;

\_end = segment.\_end;

\_tangent = segment.\_tangent;

return \*this;

}

bool operator == (const Segment& s1, const Segment& s2)

{

return (s1.*getStart*() == s2.*getStart*() && s1.*getEnd*() == s2.*getEnd*());

}

POLYGON\_CLASS.HPP

#ifndef POLYGON\_H

#define POLYGON\_H

#include "Segment.hpp"

#include <vector>

using namespace std;

class **IPolygon**

{

public:

virtual unsigned int ***getNumberVertices***()const = 0;

virtual const vector<Point>& ***getVertices***() const = 0;

virtual void ***setVertexAtPosition***(const Point& vertex, const int& position) = 0;

virtual void ***appendVertex***( const Point& vertex)=0;

virtual void ***insertVertexAtPosition***(const Point& vertex, const int& position) = 0;

virtual const Point& ***getVertexAtPosition***(const int& position) const = 0;

virtual int ***nextVertexId***(unsigned int id) const = 0;

virtual const Point& ***getLastVertex***() const = 0;

virtual const vector<Segment>& ***getEdges***() const = 0;

virtual void ***buildUpPolygonEdges***() = 0;

virtual void ***setVertexLabelAtPosition***(const int& position, int label) = 0;

virtual int ***getVertexLabelAtPosition***(const int& position) const = 0;

virtual void ***setVertexLabels***( const vector<unsigned int> &polygonVertices) = 0;

virtual void ***setDefaultVertexLabels***() = 0;

virtual double ***ComputeArea***() const = 0;

virtual bool ***pointInPolygon***(const Point& point) = 0;

virtual int ***isPolygonOnTheRightSide***(const Segment& line) const = 0;

};

class **Polygon** : public IPolygon

{

private:

unsigned int \_numberVertices;

vector<Point> \_vertices;

vector<Segment> \_edges;

public:

**Polygon**();

**Polygon**(const vector<Point>& vertices);

unsigned int ***getNumberVertices***()const { return \_numberVertices;}

const vector<Point>& ***getVertices***() const {return \_vertices;}

void ***setVertexAtPosition***(const Point& vertex, const int& position);

void ***appendVertex***( const Point& vertex);

void ***insertVertexAtPosition***(const Point& vertex, const int& position);

const Point& ***getVertexAtPosition***(const int& position) const{return \_vertices[position];}

int ***nextVertexId***(unsigned int id) const;

const Point& ***getLastVertex***() const {return \_vertices[\_numberVertices - 1];};

const vector<Segment>& ***getEdges***() const {return \_edges;}

void ***buildUpPolygonEdges***(); // useful when polygons are created by appending vertices

void ***setVertexLabelAtPosition***(const int& position, int label);

int ***getVertexLabelAtPosition***(const int& position) const { return \_vertices[position].getPointLabel();}

void ***setVertexLabels***( const vector<unsigned int> &polygonVertices);

void ***setDefaultVertexLabels***();

double ***ComputeArea***() const ;

bool ***pointInPolygon***(const Point& point);

int ***isPolygonOnTheRightSide***(const Segment& line) const;

Polygon& operator = (const Polygon& polygon);

};

#endif

POLYGON\_CLASS.CPP

#include "Polygon.hpp"

#include "Eigen"

using namespace std;

Polygon::**Polygon**()

{

\_numberVertices = 0;

\_vertices.reserve(3);

}

Polygon::**Polygon**(const vector<Point>& vertices)

{

\_numberVertices = vertices.size();

\_vertices.resize(\_numberVertices);

\_vertices = vertices;

\_edges.resize(\_numberVertices);

for (unsigned int n = 0; n < \_numberVertices; n++)

{

if (n == \_numberVertices - 1)

\_edges[n] = Segment(\_vertices[n], \_vertices[0]);

else

\_edges[n] = Segment(\_vertices[n], \_vertices[n + 1]);

}

}

void Polygon::***setVertexAtPosition***(const Point& vertex, const int& position)

{

if (position < 0 || position >= (int)\_numberVertices)

throw runtime\_error("Error: Vertex poition is not present");

\_vertices[position] = vertex;

}

void Polygon::***appendVertex***( const Point& vertex)

{

\_vertices.push\_back(vertex);

\_numberVertices = \_vertices.size();

}

void Polygon::***insertVertexAtPosition***(const Point& vertex, const int& position)

{

vector<Point>::iterator PointIt;

PointIt = \_vertices.begin() + position;

\_vertices.insert(PointIt, vertex);

\_numberVertices = \_vertices.size();

//insert newEdge

Segment newSegment(\_vertices[position], \_vertices[(position + 1)%\_numberVertices]);

vector<Segment>::iterator EdgeIt;

EdgeIt = \_edges.begin() + position;

\_edges.insert(EdgeIt, newSegment);

//fix adjacent edges

\_edges[(position - 1)%\_numberVertices].*setEnd*(\_vertices[position]);

\_edges[(position + 1)%\_numberVertices].*setStart*(\_vertices[(position + 1)%\_numberVertices]);

}

int Polygon::***nextVertexId***(unsigned int id) const

{

if (id < 0 || id >= \_numberVertices)

throw runtime\_error("Error: Vertex id is not present");

if (id == \_numberVertices - 1)

return 0;

else

return id + 1;

}

double Polygon::***ComputeArea***() const // algoritmo di Gauss

{

double area = 0;

for (unsigned int i = 0; i < \_numberVertices; i++)

{

if (i == \_numberVertices - 1)

area += \_vertices[i].getCoordinates()(1) \* (\_vertices[i - 1].getCoordinates()(0) - \_vertices[0].getCoordinates()(0));

else if (i == 0)

area += \_vertices[i].getCoordinates()(1) \* (\_vertices[\_numberVertices - 1].getCoordinates()(0) - \_vertices[i + 1].getCoordinates()(0));

else

area += \_vertices[i].getCoordinates()(1) \* (\_vertices[i - 1].getCoordinates()(0) - \_vertices[i + 1].getCoordinates()(0));

}

return abs(area)/2;

}

bool Polygon::***pointInPolygon***(const Point &point) // only for convex polygons because Project's 3 domain is supposed convex

{

unsigned int i;

bool val = 1; //by default the point is inside the polygon

for(i = 0; i < \_numberVertices && val == 1; i++)

{

// if the point point is to the right of some edge, it is not inside the polygon, so val is set to 0

if (\_edges[i].*pointOnTheRightSide*(point) == 1)

val = 0;

}

return val;

}

Polygon &Polygon::operator =(const Polygon &polygon)

{

\_numberVertices = polygon.\_numberVertices;

\_vertices.resize(\_numberVertices);

\_vertices = polygon.\_vertices;

\_edges.resize(\_numberVertices);

\_edges = polygon.\_edges;

return \*this;

}

void Polygon::***setVertexLabelAtPosition***(const int &position, int label)

{

if (position < 0 || position >= (int)\_numberVertices)

throw runtime\_error("Error: Vertex poition is not present");

\_vertices[position].setPointLabel(label);

}

void Polygon::***setVertexLabels***(const vector<unsigned int> &polygonLabels)

{

if (polygonLabels.size() != \_numberVertices)

throw runtime\_error("Error: size dimensions must agree");

for(unsigned int i = 0; i < \_numberVertices ; i++)

{

int label = polygonLabels[i];

\_vertices[i].setPointLabel(label);

}

}

void Polygon::***setDefaultVertexLabels***()

{

for(unsigned int i = 0; i < \_numberVertices ; i++)

{

\_vertices[i].setPointLabel(i);

\_edges[i].*getStart*().setPointLabel(i);

\_edges[i].*getEnd*().setPointLabel((i+1)%\_numberVertices);

}

}

int Polygon::***isPolygonOnTheRightSide***(const Segment &line) const

{

int side = 0;

int numPointsOnTheRight = 0;

int numPointsOnTheLeft = 0;

//controlling if the vertices are on on the right or on the left of the segment

for (unsigned int i = 0; i< \_numberVertices; i++)

{

if ( line.*pointOnTheRightSide*(\_vertices[i]) == 1)

numPointsOnTheRight ++;

if ( line.*pointOnTheRightSide*(\_vertices[i]) == -1)

numPointsOnTheLeft ++;

}

if ( numPointsOnTheRight > 0 && numPointsOnTheLeft == 0) //polygon is on the right

side = 1;

if ( numPointsOnTheRight == 0 && numPointsOnTheLeft > 0) //polygon is on the left

side = -1;

return side;

}

void Polygon::***buildUpPolygonEdges***()

{

\_edges.resize(\_numberVertices);

for (unsigned int n = 0; n < \_numberVertices; n++)

{

if (n == \_numberVertices - 1)

\_edges[n] = Segment(\_vertices[n], \_vertices[0]);

else

\_edges[n] = Segment(\_vertices[n], \_vertices[n + 1]);

}

}

INTERSECTOR1D1D\_CLASS.HPP

#ifndef INTERSECTOR1D1D\_HPP

#define INTERSECTOR1D1D\_HPP

#include "Eigen"

#include "iostream"

#include "MACROS.h"

#include "Segment.hpp"

using namespace std;

using namespace Eigen;

class **IIntersector1D1D**

{

public:

enum **Type**

{

NoIntersection = 0,

IntersectionOnLine = 1,

IntersectionOnSegment = 2,

IntersectionParallelOnLine = 3,

IntersectionParallelOnSegment = 4,

};

enum **Position**

{

Begin = 0,

Inner = 1,

End = 2,

Outer = 3

};

public:

virtual void ***SetFirstSegment***(const Vector2d& origin, const Vector2d& end) = 0;

virtual void ***SetSecondSegment***(const Vector2d& origin, const Vector2d& end) = 0;

virtual bool ***ComputeIntersection***() = 0;

virtual const Vector2d& ***ParametricCoordinates***() = 0;

virtual const double& ***FirstParametricCoordinate***() = 0;

virtual const double& ***SecondParametricCoordinate***() = 0;

virtual const Position& ***PositionIntersectionInFirstEdge***() = 0;

virtual const Position& ***PositionIntersectionInSecondEdge***() = 0;

virtual const Type& ***TypeIntersection***() = 0;

virtual Vector2d ***IntersectionFirstParametricCoordinate***(const Vector2d& origin,const Vector2d& end) const = 0;

virtual Vector2d ***IntersectionSecondParametricCoordinate***(const Vector2d& origin,const Vector2d& end) const = 0;

};

class **Intersector1D1D** : public IIntersector1D1D

{

public:

Type type;

private:

Position positionIntersectionFirstEdge;

Position positionIntersectionSecondEdge;

Vector2d resultParametricCoordinates;

Vector2d originFirstSegment;

//right side of the system: it is equal to the origin of the second segment - the origin of the first segment

Vector2d rightHandSide;

//matrix made up of the 2 segment tangent vectors

Matrix2d matrixTangentVector;

public:

**Intersector1D1D**();

//FirstSegment must be set before the second because then we use the attribute "originFirtsSegment"

void ***SetFirstSegment***(const Vector2d& origin, const Vector2d& end) {matrixTangentVector.col(0) = end - origin; originFirstSegment = origin; }

void ***SetSecondSegment***(const Vector2d& origin, const Vector2d& end) {matrixTangentVector.col(1) = origin - end; rightHandSide = origin - originFirstSegment; }

///Compute the intersections between the line of the first edge and the second edge and returns the parametric coordinates and the intersection type

///The first parametric coordinate refers to the first tangentVector and ...

///@note In case of parallelism both the parametric coordinates refers to the first edge

bool ***ComputeIntersection***();

const Vector2d& ***ParametricCoordinates***() {return resultParametricCoordinates;}

const double& ***FirstParametricCoordinate***() {return resultParametricCoordinates(0);}

const double& ***SecondParametricCoordinate***() {return resultParametricCoordinates(1);}

const Position& ***PositionIntersectionInFirstEdge***(){return positionIntersectionFirstEdge;}

const Position& ***PositionIntersectionInSecondEdge***(){return positionIntersectionSecondEdge;}

const Type& ***TypeIntersection***() {return type;}

Vector2d ***IntersectionFirstParametricCoordinate***(const Vector2d& origin,const Vector2d& end) const {return (1 - resultParametricCoordinates(0)) \* origin + resultParametricCoordinates(0) \* end; }

Vector2d ***IntersectionSecondParametricCoordinate***(const Vector2d& origin,const Vector2d& end) const {return (1 - resultParametricCoordinates(1)) \* origin + resultParametricCoordinates(1) \* end; }

};

#endif // INTERSECTOR1D1D\_H

INTERSECTOR1D1D\_CLASS.CPP

#include "Intersector1D1D.hpp"

Intersector1D1D::**Intersector1D1D**()

{

type = Intersector1D1D::NoIntersection;

resultParametricCoordinates.setZero(2);

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bool Intersector1D1D::***ComputeIntersection***()

{

// parallelism is the wedge product of the 2 tangent vectors, so analyzing it we can say if they are parallel

double parallelism = matrixTangentVector.determinant();

type = NoIntersection;

bool intersection = false;

// check is used to not use square root function and division, lowering the computational cost. In fact we are controlling if the wedge product

// divided by the squared norms of the two tangent vectors multiplied is greater than squared parallelism

double check = INTERSECTION\_TOLERANCE \* INTERSECTION\_TOLERANCE \* matrixTangentVector.col(0).squaredNorm() \* matrixTangentVector.col(1).squaredNorm();

//checking if the vectors are not parallel

if(parallelism \* parallelism >= check)

{

Matrix2d solverMatrix;

//inverting the matrix made up by the 2 tangent vectors multiplied by the determinant

solverMatrix << matrixTangentVector(1,1), -matrixTangentVector(0,1), -matrixTangentVector(1,0), matrixTangentVector(0,0);

/// <li> If the edge and the trace are not parallel look for the intersection with parametric coordinates

resultParametricCoordinates = solverMatrix \* rightHandSide;

//dividing by the determinant here because we are computing only 2 divisions

resultParametricCoordinates /= parallelism;

//checking if the parametric coordinates of the second segment is inside the segment itself

if (resultParametricCoordinates(1) > -INTERSECTION\_TOLERANCE && resultParametricCoordinates(1)-1.0 < INTERSECTION\_TOLERANCE)

{

//now we can say that there is intersection because the line of the first segment intersect the second segment and thus the intesection

//is at least "IntersectionOnLine"

type = IntersectionOnLine;

intersection = true;

//checking if the parametric coordinates of the intersection, in respect to the first segment, is inside the segment itself

if (resultParametricCoordinates(0) > -INTERSECTION\_TOLERANCE && resultParametricCoordinates(0)-1.0 < INTERSECTION\_TOLERANCE)

//if it is true the intersection is inside the segment

type = IntersectionOnSegment;

}

}

//if they are parallel

else

{

//parallelism2 is the wedge product between the tangent vector of the first segment and the vector made up of (origin of the second segment) - (origin of the first segment),represented by rightHandSide

//parallelism2 is used to control if the segments are on the same line: if the wedge product is 0 we can say that they are parallel on the same line because

//rightHandSide will be equal to (0,0), whereas in the other case it will be different from 0

double parallelism2 = fabs(matrixTangentVector(0,0) \* rightHandSide.y() - rightHandSide.x() \* matrixTangentVector(1,0));

/// <li> In case of parallelism check if the segment is the same with parametric coordinates

double squaredNormFirstEdge = matrixTangentVector.col(0).squaredNorm();

//check 2 is used just like the previous check

double check2 = INTERSECTION\_TOLERANCE \* INTERSECTION\_TOLERANCE \* squaredNormFirstEdge \* rightHandSide.squaredNorm();

//checking if the segments are on the same line, if not there is "NoIntersection"

if( parallelism2 \* parallelism2 <= check2 )

{

//dividing here to reduce computational cost

double tempNorm = 1.0/squaredNormFirstEdge;

//inverting the tangent vector to solve the system for the first parametric coordinate

resultParametricCoordinates(0) = matrixTangentVector.col(0).dot(rightHandSide) \* tempNorm ;

// solving for the second parametric coordinate

resultParametricCoordinates(1) = resultParametricCoordinates(0) - matrixTangentVector.col(0).dot(matrixTangentVector.col(1)) \* tempNorm;

// we know that there is intersection

intersection = true;

type = IntersectionParallelOnLine;

//swapping in order to have ascending order

if(resultParametricCoordinates(1) < resultParametricCoordinates(0))

{

double tmp = resultParametricCoordinates(0);

resultParametricCoordinates(0) = resultParametricCoordinates(1);

resultParametricCoordinates(1) = tmp;

}

// if one vertex is in the edge there is the intersection

if( (resultParametricCoordinates(0) > -INTERSECTION\_TOLERANCE && resultParametricCoordinates(0)-1.0 < INTERSECTION\_TOLERANCE) ||

(resultParametricCoordinates(1) > -INTERSECTION\_TOLERANCE && resultParametricCoordinates(1)-1.0 <INTERSECTION\_TOLERANCE) )

type = IntersectionParallelOnSegment;

else

{

//IL PRIMO SEGMENTO DATO IN INPUT E' CONTENUTO NEL SECONDO O VICEVERSA

if( ( resultParametricCoordinates(0) < INTERSECTION\_TOLERANCE && resultParametricCoordinates(1) - 1.0 > -INTERSECTION\_TOLERANCE) )

type = IntersectionParallelOnSegment;

}

}

}

/// </ul>

if(resultParametricCoordinates(0) < -INTERSECTION\_TOLERANCE || resultParametricCoordinates(0) > 1.0 + INTERSECTION\_TOLERANCE)

positionIntersectionFirstEdge = Outer;

else if((resultParametricCoordinates(0) > -INTERSECTION\_TOLERANCE) && (resultParametricCoordinates(0) < INTERSECTION\_TOLERANCE))

{

resultParametricCoordinates(0) = 0.0;

positionIntersectionFirstEdge = Begin;

}

else if ((resultParametricCoordinates(0) > 1.0 - INTERSECTION\_TOLERANCE) && (resultParametricCoordinates(0) < 1.0 +INTERSECTION\_TOLERANCE))

{

resultParametricCoordinates(0) = 1.0;

positionIntersectionFirstEdge = End;

}

else

positionIntersectionFirstEdge = Inner;

if(resultParametricCoordinates(1) < -INTERSECTION\_TOLERANCE || resultParametricCoordinates(1) > 1.0 + INTERSECTION\_TOLERANCE)

positionIntersectionSecondEdge = Outer;

else if((resultParametricCoordinates(1) > -INTERSECTION\_TOLERANCE) && (resultParametricCoordinates(1) < INTERSECTION\_TOLERANCE))

{

resultParametricCoordinates(1) = 0.0;

positionIntersectionSecondEdge = Begin;

}

else if ((resultParametricCoordinates(1) > 1.0 - INTERSECTION\_TOLERANCE) && (resultParametricCoordinates(1) < 1.0 + INTERSECTION\_TOLERANCE))

{

resultParametricCoordinates(1) = 1.0;

positionIntersectionSecondEdge = End;

}

else

positionIntersectionSecondEdge = Inner;

return intersection;

}

INTERSECTOR\_POLYGON\_LINE\_CLASS.HPP

#ifndef INTERSECTORPOLYGONLINE\_H

#define INTERSECTORPOLYGONLINE\_H

#include <vector>

#include <unordered\_map>

#include "Polygon.hpp"

#include "Point.hpp"

#include "Intersector1D1D.hpp"

#include <gtest/gtest\_prod.h>

namespace **IntersectorPolygonLineNamespace** {

class **IIntersectorPolygonLine**

{

private:

virtual bool ***checkInConicCombination***(const Vector2d& line, const Vector2d& v1, const Vector2d& v2, int& flag) const = 0;

virtual void ***extendPolygonIfEndpointsOnEdge***(int& numPoly, const int& firstIntersectionId, const int& secondIntersectionId, const bool& forwardDirection) = 0;

public:

virtual const vector<Point>& ***findIntersectionVertices***() = 0;

virtual const vector<Polygon>& ***findPolygons***() = 0;

virtual const vector<Point>& ***findNewPoints***() = 0;

virtual const vector<Polygon>& ***getExtendedPolygons***() = 0;

virtual bool ***getStartFound***() const = 0;

virtual bool ***getEndFound***() const = 0;

};

class **IntersectorPolygonLine** : public IIntersectorPolygonLine

{

private:

IIntersector1D1D& \_intersector;

IPolygon& \_polygon;

ISegment& \_line;

vector<Point> \_intersectionVertices;

vector<Point> \_newPoints;

vector<Polygon> \_extendedNewPolygons;

vector<Polygon> \_newPolygons;

vector<int> \_intersectionIdToSegmentId;

vector<int> \_segmentIdToIntersectionId;

vector<bool> \_foundVertex;

bool \_startFound = false;

bool \_endFound = false;

vector<int> \_flag;

vector<bool> \_parallelIntersections;

multimap<double, Point> \_mmapParametricCoordinateToIntersectionPoint;

multimap<double, int> \_mmapParametricCoordinateToSegmentId;

map<int, double> \_mapSegmentIdToParametricCoordinate;

map<int,double> \_mapIntersectionIdToParametricCoordinate;

public:

**IntersectorPolygonLine**(IIntersector1D1D& intersector, IPolygon& polygon, ISegment& segment);

const vector<Point>& ***findIntersectionVertices***();

const vector<Polygon>& ***findPolygons***();

const vector<Point>& ***findNewPoints***();

const vector<Polygon>& ***getExtendedPolygons***() {return \_extendedNewPolygons;}

bool ***getStartFound***() const {return \_startFound;}

bool ***getEndFound***() const {return \_endFound;}

private:

bool ***checkInConicCombination***(const Vector2d& line, const Vector2d& v1, const Vector2d& v2, int& flag) const;

//N.B: per poter fare il test di un metodo privato in forma diretta serve avere un test friend che possa accedere ai membri privati della classe

FRIEND\_TEST(TestIntersectorPolygonLine, **TestCheckInConicCombination**);

void ***extendPolygonIfEndpointsOnEdge***(int& numPoly, const int& firstIntersectionId, const int& secondIntersectionId, const bool& forwardDirection);

};

}

#endif

INTERSECTOR\_POLYGON\_LINE\_CLASS.CPP

#include "IntersectorPolygonLine.hpp"

#include "MACROS.h"

using namespace std;

namespace **IntersectorPolygonLineNamespace** {

IntersectorPolygonLine::**IntersectorPolygonLine**(IIntersector1D1D &intersector, IPolygon &polygon, ISegment &segment)

: \_intersector(intersector), \_polygon(polygon), \_line(segment)

{

unsigned int numberVertices = polygon.*getNumberVertices*();

\_segmentIdToIntersectionId.resize(numberVertices);

\_foundVertex.resize(numberVertices);

\_flag.resize(numberVertices);

\_parallelIntersections.resize(numberVertices);

// set default values for these support attributes that helps cutting the polygon

for (unsigned int i = 0; i < numberVertices; i++)

{

\_segmentIdToIntersectionId[i] = -1;

\_foundVertex[i] = false;

\_flag[i] = 0;

\_parallelIntersections[i] = false;

}

// set as the first segment of intersection the line since it remains the same all time

\_intersector.*SetFirstSegment*(segment.*getStart*().getCoordinates(), segment.*getEnd*().getCoordinates());

}

const vector<Point> &IntersectorPolygonLine::***findIntersectionVertices***()

{

const vector<Segment> edges = \_polygon.*getEdges*();

\_polygon.*setDefaultVertexLabels*();

\_line.*getStart*().setPointLabel(\_polygon.*getNumberVertices*());

\_line.*getEnd*().setPointLabel(\_polygon.*getNumberVertices*()+1);

// cycle through all edges to find intersections

for (unsigned int n = 0; n < \_polygon.*getNumberVertices*(); n++)

{

\_intersector.*SetSecondSegment*(edges[n].*getStart*().getCoordinates(), edges[n].*getEnd*().getCoordinates());

if (\_intersector.*ComputeIntersection*()) // true if there is intersection

{

// se il segmento di intersezione ed il lato sono paralleli

if (\_intersector.*TypeIntersection*() == Intersector1D1D::IntersectionParallelOnSegment

|| \_intersector.*TypeIntersection*() == Intersector1D1D::IntersectionParallelOnLine )

{

\_parallelIntersections[n] = true;

}

else

{

// finds intersection point and its parametric coordinate

Vector2d intersection = \_intersector.*IntersectionFirstParametricCoordinate*(\_line.*getStart*().getCoordinates(), \_line.*getEnd*().getCoordinates());

double s = \_intersector.*FirstParametricCoordinate*();

// check if the point has been found before, this would mean there is intersection in a vertex

if (\_mmapParametricCoordinateToIntersectionPoint.find(s) == \_mmapParametricCoordinateToIntersectionPoint.end())

{

// insert values in a map or multimap so that they are ordered based on the key

\_mmapParametricCoordinateToIntersectionPoint.insert(pair<double, Point>(*s*, Point(intersection)));

\_mmapParametricCoordinateToSegmentId.insert(pair<double, int>(*s*, *n*));

\_mapSegmentIdToParametricCoordinate.insert(pair<int, double>(*n*,*s*));

}

else // if there is intersection in a vertex

{

// cerca se l intersezione sul vertice è su (lastVertex)(0) in modo da rendere il tutto ciclico

if ((n == \_polygon.*getNumberVertices*()-1) && \_mmapParametricCoordinateToSegmentId.find(s)->second == 0)

{

// nel caso imposto il \_flag[0] invece di \_flag[n] perche il flag è associato alla seconda intersezione per come è costruito il programma

bool isInConic = *checkInConicCombination*(\_line.*computeTangent*(), edges[n].*getTangent*() , edges[0].*getTangent*(), *\_flag[*0*]*);

// se isInConic = True significa che il vertice va inserito due volte

// le multimap sono utilizzate proprio per questo

if (isInConic)

{

\_mmapParametricCoordinateToIntersectionPoint.insert(pair<double, Point>(*s*, Point(intersection)));

\_mmapParametricCoordinateToSegmentId.insert(pair<double, int>(*s*, *n*));

\_mapSegmentIdToParametricCoordinate.insert(pair<int, double>(*n*, *s*));

}

// in modo da far si che nel caso di intersezione sul vertice da contare una sola voltasia sempre il primo vertice ad avere l intersezione,

// in questo caso abbiamo il vertice (n)(0) e quindi l intersezione va segnata su n

else

\_mmapParametricCoordinateToSegmentId.find(s)->second = (int)n;

}

else

{

bool isInConic = *checkInConicCombination*(\_line.*computeTangent*(), edges[n-1].*getTangent*() , edges[n].*getTangent*(), *\_flag[n]*);

if (isInConic)// se è true, aggiunge il punto alle strutture, duplicandolo, altrimenti non lo aggiunge

{

\_mmapParametricCoordinateToIntersectionPoint.insert(pair<double, Point>(*s*, Point(intersection)));

\_mmapParametricCoordinateToSegmentId.insert(pair<double, int>(*s*, *n*));

\_mapSegmentIdToParametricCoordinate.insert(pair<int, double>(*n*,*s*));

}

}

}

}

}

}

// maps and multimaps have been used because are needed elements ordered by a key

// makes a vector with the Points from the map \_mmapParametricCoordinateToIntersectionPoint<double,Point> and adds labels

// in this way intersectionVertices are ordered by parametric coordinate

int i = 0;

multimap<double, Point>::const\_iterator iterator1;

for (iterator1 = \_mmapParametricCoordinateToIntersectionPoint.begin(); iterator1 != \_mmapParametricCoordinateToIntersectionPoint.end(); iterator1++)

{

\_intersectionVertices.push\_back(iterator1->second);

\_intersectionVertices[i].setPointLabel(\_polygon.*getNumberVertices*()+2+i);

i++;

}

int j = 0;

multimap<double, int>::const\_iterator iterator2;

for (iterator2 = \_mmapParametricCoordinateToSegmentId.begin(); iterator2 != \_mmapParametricCoordinateToSegmentId.end(); iterator2++)

{

// if flag=-1 means that there is a double intersection on a vertex but that they are inserted in reverse order, so we

// have to switch them

if (\_flag[iterator2->second] == -1)

{

int last = \_intersectionIdToSegmentId.back();

// bisogna scambiare i valori associati a parametric e iterator2->second in \_segmentIdToIntersectionId

\_segmentIdToIntersectionId[iterator2->second] = \_segmentIdToIntersectionId[last];

\_segmentIdToIntersectionId[last] = j;

j++;

// bisogna scambiare i valori in \_intersectionIdToSegmentId

unsigned int lastIndex = \_intersectionIdToSegmentId.size() - 1;

\_intersectionIdToSegmentId[lastIndex] = iterator2->second;

\_intersectionIdToSegmentId.push\_back(last);

}

else

{

\_intersectionIdToSegmentId.push\_back(iterator2->second);

\_segmentIdToIntersectionId[iterator2->second] = j;

j++;// posizione dell'intersezione

}

}

//loops through the intersection Points comparing if they are equal to any of the endpoints of the corrisponding segment,

//If yes, it sets the attribute \_label to that of the polygon vertex

for(unsigned int k = 0; k < \_intersectionIdToSegmentId.size(); k++)

{

int segmentPosition = \_intersectionIdToSegmentId[k];

const Point& segmentStartPoint = \_polygon.*getEdges*()[segmentPosition].*getStart*();

if (\_intersectionVertices[k] == segmentStartPoint )

\_intersectionVertices[k].setPointLabel(segmentStartPoint.getPointLabel());

const Point& segmentEndPoint = \_polygon.*getEdges*()[segmentPosition].*getEnd*();

if (\_intersectionVertices[k] == segmentEndPoint )

\_intersectionVertices[k].setPointLabel(segmentEndPoint.getPointLabel());

};

//usando il vettore \_segmentIdToIntersectionId e la mappa \_mapSegmentIdToParametricCoordinate

for ( unsigned int k = 0; k < \_intersectionIdToSegmentId.size(); k++)

{

double parametricCoordinate = \_mapSegmentIdToParametricCoordinate[\_intersectionIdToSegmentId[k]];

// map needed for the method extendPolygonIfEndpointsOnEdge

\_mapIntersectionIdToParametricCoordinate.insert(pair<int, double>(*k*, *parametricCoordinate*));

}

if (\_intersectionVertices.size() % 2 == 1 )

throw runtime\_error ("Error: the number of intersection vertices should be even");

return \_intersectionVertices;

}

const vector<Polygon>& IntersectorPolygonLine::***findPolygons***()

{

// clear just to be sure it is empty

\_newPolygons.clear();

\_extendedNewPolygons.clear();

const vector<Point>& vertices = \_polygon.*getVertices*();

int numPoly = 0;

// cycle through all vertices till every of them has been visited

for (unsigned int n = 0; n < \_foundVertex.size(); n++)

{

if(\_parallelIntersections[n] && \_foundVertex[n] == false)

{

\_foundVertex[n] = true;

\_foundVertex[(n+1)% \_polygon.*getNumberVertices*()] = true;

}

else

{//begin else

if (\_foundVertex[n] == false) // il vertice non è stato ancora visitato

{

\_newPolygons.push\_back(Polygon());

\_extendedNewPolygons.push\_back(Polygon());

unsigned int id = n;

bool forwardDirection;

// each while cycle ends when it returns to the start point, completing a cutted polygon

do

{

// if there is no intersection in that segment the algorithm append the vertex and goes to the next

if (\_segmentIdToIntersectionId[id] == -1)

{

\_newPolygons[numPoly].*appendVertex*(vertices[id]);

\_extendedNewPolygons[numPoly].*appendVertex*(vertices[id]);

\_foundVertex[id] = true;

id = \_polygon.*nextVertexId*(id);

}

else // there is intersection on that segment

{

// append the vertex to the cutted polygon

\_newPolygons[numPoly].*appendVertex*(vertices[id]);

\_extendedNewPolygons[numPoly].*appendVertex*(vertices[id]);

\_foundVertex[id] = true;

// get the id (ordered in parametric coordinates) of the intersection point

int intersectionId = \_segmentIdToIntersectionId[id];

// if the intersection is on the vertex we don t add it since the points are equal

if (\_intersectionVertices[intersectionId] != vertices[id])

{

\_newPolygons[numPoly].*appendVertex*(\_intersectionVertices[intersectionId]);

\_extendedNewPolygons[numPoly].*appendVertex*(\_intersectionVertices[intersectionId]);

}

// intersection are in pair and they creates segment with intersection id even-odd that are inside the polygon

if (intersectionId % 2 == 0) // if intersection id is even, the segment inside is (id),(id+1)

{

forwardDirection = true; // forward from even to odd position

int firstIntersectionId = intersectionId;

int secondIntersectionId = intersectionId + 1;

// check if segment start and and are inside the polygon

*extendPolygonIfEndpointsOnEdge*( *numPoly* ,firstIntersectionId, secondIntersectionId, forwardDirection);

intersectionId++;

}

else // if intersection id is odd the segment inside the polygon is (id-1),(id)

{

forwardDirection = false;// backward from odd to even position

int firstIntersectionId =intersectionId-1;

int secondIntersectionId = intersectionId;

// check if segment start and and are inside the polygon

*extendPolygonIfEndpointsOnEdge*( *numPoly* ,firstIntersectionId, secondIntersectionId, forwardDirection);

intersectionId--;

}

//end

id = \_intersectionIdToSegmentId[intersectionId];

// se il punto d'intersezione d'arrivo != seguente vertice cioè se non cade sul vertice succesivo

// se i due punti sono uguali non o aggiungo perche lo metto al prossimo ciclo e verrebbe un "doppione"

if (\_intersectionVertices[intersectionId] != vertices[\_polygon.*nextVertexId*(id)])

{

// significa che c è un vertice isolato per l intersezione in cui è impossibile passare ed allora lo settiamo

// true ed invece esploriamo l intersezione sul vertice (aggiunta ad inizio ciclo) in quanto hanno le stesse coordinate

// per saperlo bisogna vedere se i due punti di intersezione consecutivi sono lo stesso e quindi coincidente con il vertice

if ( (forwardDirection && \_intersectionVertices[intersectionId] == \_intersectionVertices[intersectionId - 1])

|| (!forwardDirection && \_intersectionVertices[intersectionId] == \_intersectionVertices[intersectionId + 1]))

\_foundVertex[id] = true;

else

{

\_newPolygons[numPoly].*appendVertex*(\_intersectionVertices[intersectionId]);

\_extendedNewPolygons[numPoly].*appendVertex*(\_intersectionVertices[intersectionId]);

}

}

id = \_polygon.*nextVertexId*(id);

}

} while (id != n);

// with each cycle a sub-polygon have been discovered

numPoly++;

}

}//end else

}

// Fills \_edges attribute of newPolygons costructed by cutting as opposed to those constructed by the constructor

for (unsigned int j = 0; j < \_newPolygons.size(); j++)

{

\_newPolygons[j].*buildUpPolygonEdges*();

\_extendedNewPolygons[j].*buildUpPolygonEdges*();

}

return \_newPolygons;

}

const vector<Point> &IntersectorPolygonLine::***findNewPoints***()

{

// adds start and end of the segment if they are inside the polygon

if(\_startFound == true )

{

\_newPoints.push\_back(\_line.*getStart*());

}

if(\_endFound == true)

{

\_newPoints.push\_back(\_line.*getEnd*());

}

//loops through \_intersectionVertices and adds the ones that don't fall in a vertex since they are not newPoints

for(unsigned int k = 0; k < \_intersectionVertices.size(); k++)

{

int segmentPosition = \_intersectionIdToSegmentId[k];

const Point& segmentStartPoint = \_polygon.*getEdges*()[segmentPosition].*getStart*();

const Point& segmentEndPoint = \_polygon.*getEdges*()[segmentPosition].*getEnd*();

if (\_intersectionVertices[k] != segmentStartPoint && \_intersectionVertices[k] != segmentEndPoint)

\_newPoints.push\_back(\_intersectionVertices[k]);

};

return \_newPoints;

}

// per cercare la tipologia di intersezione su vertice

// l output di acos è [0, pi]

bool IntersectorPolygonLine::***checkInConicCombination***(const Vector2d& line, const Vector2d& v1, const Vector2d& v2, int& flag) const

{

const Vector2d nline = line.normalized();

const Vector2d nv1 = v1.normalized();

const Vector2d nv2 = v2.normalized();

// calcolare la bisettrice

Vector2d bisector = nv1 + nv2;

bisector = bisector.normalized();

const double theta = acos(bisector.dot(nv1));

const double alfa = acos(bisector.dot(nline));

// means bisector is perpendicular to the edge, so the vertex has been added to create a conforming mesh

// in this case there is a single intersection point

if ((abs(theta - M\_PI/2)) < INTERSECTION\_TOLERANCE)

{

flag = 0;

return false;

}

// double intersection point

if (alfa < theta)

{

flag = 1;

return true;

}

// double intersection point but in reversed order

else if (alfa > M\_PI/2 && (M\_PI - alfa) < theta)

{

flag = -1;

return true;

}

flag = 0;

return false;

}

void IntersectorPolygonLine::***extendPolygonIfEndpointsOnEdge***(int& numPoly, const int &firstIntersectionId, const int &secondIntersectionId, const bool &forwardDirection)

{

if ( \_mapIntersectionIdToParametricCoordinate[firstIntersectionId] < 0

&& \_mapIntersectionIdToParametricCoordinate[secondIntersectionId] > 0

&& \_mapIntersectionIdToParametricCoordinate[secondIntersectionId] < 1)

{

\_extendedNewPolygons[numPoly].*appendVertex*(\_line.*getStart*());

\_startFound = true;

}

else if ( \_mapIntersectionIdToParametricCoordinate[firstIntersectionId] < 0

&& \_mapIntersectionIdToParametricCoordinate[secondIntersectionId] > 1)//A e B

{

if(forwardDirection == true){

\_extendedNewPolygons[numPoly].*appendVertex*(\_line.*getStart*());

\_startFound = true;

\_extendedNewPolygons[numPoly].*appendVertex*(\_line.*getEnd*());

\_endFound = true;

}

else

{

\_extendedNewPolygons[numPoly].*appendVertex*(\_line.*getEnd*());

\_endFound = true;

\_extendedNewPolygons[numPoly].*appendVertex*(\_line.*getStart*());

\_startFound = true;

}

}

else if ( \_mapIntersectionIdToParametricCoordinate[firstIntersectionId] > 0

&& \_mapIntersectionIdToParametricCoordinate[firstIntersectionId] < 1

&& \_mapIntersectionIdToParametricCoordinate[secondIntersectionId] > 1)

{

\_extendedNewPolygons[numPoly].*appendVertex*(\_line.*getEnd*());

\_endFound = true;

}

else if ( \_mapIntersectionIdToParametricCoordinate[firstIntersectionId ] > 0

&& \_mapIntersectionIdToParametricCoordinate[firstIntersectionId ] < 1

&& \_mapIntersectionIdToParametricCoordinate[secondIntersectionId] < 0)

{

\_extendedNewPolygons[numPoly].*appendVertex*(\_line.*getStart*());

\_startFound = true;

}

else if ( \_mapIntersectionIdToParametricCoordinate[firstIntersectionId] > 1

&& \_mapIntersectionIdToParametricCoordinate[secondIntersectionId] < 0)//B e A

{

if(forwardDirection == true){

\_extendedNewPolygons[numPoly].*appendVertex*(\_line.*getEnd*());

\_endFound = true;

\_extendedNewPolygons[numPoly].*appendVertex*(\_line.*getStart*());

\_startFound = true;

}

else

{

\_extendedNewPolygons[numPoly].*appendVertex*(\_line.*getStart*());

\_startFound = true;

\_extendedNewPolygons[numPoly].*appendVertex*(\_line.*getEnd*());

\_endFound = true;

}

}

else if ( \_mapIntersectionIdToParametricCoordinate[firstIntersectionId] > 1

&& \_mapIntersectionIdToParametricCoordinate[secondIntersectionId] < 1

&& \_mapIntersectionIdToParametricCoordinate[secondIntersectionId] > 0)

{

\_extendedNewPolygons[numPoly].*appendVertex*(\_line.*getEnd*());

\_endFound = true;

}

}

}

POLYGON\_CUTTER\_CLASS.HPP

#ifndef POLYGONCUTTER\_H

#define POLYGONCUTTER\_H

#include "Intersector1D1D.hpp"

#include "IntersectorPolygonLine.hpp"

using namespace IntersectorPolygonLineNamespace;

namespace **PolygonCutterNamespace** {

class **IPolygonCutter**

{

public:

virtual const vector<Polygon>& ***cutPolygon***() = 0;

virtual const vector<Polygon>& ***getCuttedPolygons***() const = 0;

virtual const vector<Point>& ***getIntersectionVertices***() const = 0;

virtual const vector<Point>& ***getNewPoints***() const = 0;

virtual void ***showPolygon***(const string filePath = "showPolygon.m") = 0;

};

class **PolygonCutter**: IPolygonCutter

{

private:

IIntersectorPolygonLine& \_intersector;

IPolygon& \_polygon;

ISegment& \_cutter;

vector<Polygon> \_cuttedPolygons;

vector<Point> \_intersectionVertices;

vector<Point> \_newPoints;

public:

**PolygonCutter**(IIntersectorPolygonLine& intersector, IPolygon& polygon, ISegment& segment);

const vector<Polygon>& ***cutPolygon***();

const vector<Polygon>& ***getCuttedPolygons***() const {return \_cuttedPolygons;};

const vector<Point>& ***getIntersectionVertices***() const {return \_intersectionVertices;}

const vector<Point>& ***getNewPoints***() const {return \_newPoints;}

void ***showPolygon***(const string filePath = "showPolygon.m");

};

}

#endif

POLYGON\_CUTTER\_CLASS.CPP

#include "PolygonCutter.hpp"

#include "fstream"

namespace **PolygonCutterNamespace** {

PolygonCutter::**PolygonCutter**(IIntersectorPolygonLine& intersector, IPolygon &polygon, ISegment &segment)

: \_intersector{intersector}, \_polygon{polygon}, \_cutter{segment}

{

}

const vector<Polygon> &PolygonCutter::***cutPolygon***()

{

\_intersectionVertices = \_intersector.*findIntersectionVertices*();

\_newPoints = \_intersector.*findNewPoints*();

\_cuttedPolygons = \_intersector.*findPolygons*();

return \_cuttedPolygons;

}

void PolygonCutter::***showPolygon***(const string filePath)

{

if (\_cuttedPolygons.empty())

throw runtime\_error("First you need to find sub-polygons");

ofstream of;

of.open (filePath);

const vector<Point>& vertices = \_polygon.*getVertices*();

of << "width = 1.0;" << endl

<< "labs = 0:" << vertices.size()-1 << ';' << endl

<< "labsInt = 0:" << \_intersectionVertices.size()-1 << ';' << endl

<< "points = [" << endl;

for (unsigned int i = 0; i < vertices.size(); i++)

of << "\t" << vertices[i].getCoordinates()[0] << ", " << vertices[i].getCoordinates()[1] << endl;

of << "];" << endl;

of << "polygon = polyshape(points);" << endl

<< "segment = [" << endl

<< "\t" << \_cutter.*getStart*().getCoordinates()[0] << ", " << \_cutter.*getStart*().getCoordinates()[1] << endl

<< "\t" << \_cutter.*getEnd*().getCoordinates()[0] << ", " << \_cutter.*getEnd*().getCoordinates()[1] << endl

<< "];" << endl

<< "intersection = [" << endl;

for (unsigned int i = 0; i < \_intersectionVertices.size(); i++)

of << "\t" << \_intersectionVertices[i].getCoordinates()[0] << ", " << \_intersectionVertices[i].getCoordinates()[1] << endl;

of << "];" << endl;

of << "figure;" << endl

<< "hold on;" << endl

<< "plot(polygon, 'FaceColor', 'w', 'LineWidth', width);" << endl

<< "plot([segment(1,1), segment(2,1)], [segment(1,2), segment(2,2)], 'r--');" << endl

<< "plot(points(:, 1),points(:, 2), 'ko');" << endl

<< "plot(intersection(:, 1), intersection(:, 2), 'rs');" << endl

<< "plot(segment(:, 1), segment(:, 2), 'r.', 'MarkerSize', 9);" << endl

<< "text(points(:,1), points(:,2), string(labs), 'VerticalAlignment', 'top', 'HorizontalAlignment', 'left');" << endl

<< "text(intersection(:,1), intersection(:,2), string(labsInt), 'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'right', 'Color', 'r');" << endl

<< "hold off;" << endl << endl;

of << "figure;" << endl

<< "hold on;" << endl;

for (unsigned int n = 0; n < \_cuttedPolygons.size(); n++)

{

of << "points" << n+1 << " = [" << endl;

const vector<Point>& points = \_cuttedPolygons[n].*getVertices*();

for (unsigned int i = 0; i < \_cuttedPolygons[n].*getNumberVertices*(); i++)

of << "\t" << points[i].getCoordinates()[0] << ", " << points[i].getCoordinates()[1] << endl;

of << "];" << endl;

of << "polygon" << n+1 << " = polyshape(points" << n+1 << ");" << endl

<< "plot(polygon" << n+1 << ", 'LineWidth', width);" << endl << endl;

}

of << "text(points(:,1), points(:,2), string(labs), 'VerticalAlignment', 'top', 'HorizontalAlignment', 'left');" << endl

<< "text(intersection(:,1), intersection(:,2), string(labsInt), 'VerticalAlignment', 'bottom', 'HorizontalAlignment', 'right', 'Color', 'r');" << endl

<< "hold off;" << endl;

of.close();

}

}

REFERENCE\_ELEMENT\_CLASS.HPP

#ifndef REFERENCEELEMENT\_H

#define REFERENCEELEMENT\_H

#include <Polygon.hpp>

#include <vector>

namespace **MeshNamespace** { class **Mesh**; }

namespace **ReferenceElementNamespace** {

class **IReferenceElement** {

public:

virtual void ***computeBoundingBox***(Polygon& polygon) = 0;

virtual void ***computeReferenceElement***(Polygon& polygon) = 0;

virtual const vector<Point>& ***getBoundingBoxVertices***() const = 0;

virtual const vector<Polygon>& ***getReferenceElementPolygons***() const = 0;

virtual const Point& ***getBoundingBoxVertexAtPosition***(unsigned int& i) = 0;

virtual void ***findBoundaryVertices***() = 0;

virtual const map<double, Point>& ***getMapSouthXCoordinates***() const = 0;

virtual const map<double, Point>& ***getMapNorthXCoordinates***() const = 0;

virtual const map<double, Point>& ***getMapEastYCoordinates***() const = 0;

virtual const map<double, Point>& ***getMapWestYCoordinates***() const = 0;

virtual void ***makeConforming***() = 0;

virtual const vector<Polygon>& ***getConformingReferenceElementPolygons***() const = 0;

};

class **ReferenceElement** : public IReferenceElement

{

private:

vector<Point> \_boundingBoxVertices;

vector<Polygon> \_referenceElementPolygons;

vector<Polygon> \_conformingReferenceElementPolygons;

Polygon \_boundingBox;

map<double, Point> \_mapSouthXCoordinates;

map<double, Point> \_mapNorthXCoordinates;

map<double, Point> \_mapWestYCoordinates;

map<double, Point> \_mapEastYCoordinates;

double \_minX;

double \_minY;

double \_maxX;

double \_maxY;

friend class MeshNamespace::Mesh;

public:

**ReferenceElement**();

void ***computeReferenceElement***(Polygon& polygon);

const vector<Polygon>& ***getReferenceElementPolygons***() const {return \_referenceElementPolygons;}

void **showReferenceElement**(const string filePath = "showReferenceElement.m");

void ***computeBoundingBox***(Polygon& polygon);

const vector<Point>& ***getBoundingBoxVertices***() const {return \_boundingBoxVertices;}

const Point& ***getBoundingBoxVertexAtPosition***(unsigned int& i) {return \_boundingBoxVertices[i];}

const Polygon& **getBoundingBox**(){return \_boundingBox;}

void ***findBoundaryVertices***();

const map<double, Point>& ***getMapSouthXCoordinates***() const {return \_mapSouthXCoordinates;}

const map<double, Point>& ***getMapNorthXCoordinates***() const {return \_mapNorthXCoordinates;}

const map<double, Point>& ***getMapEastYCoordinates***() const {return \_mapEastYCoordinates;}

const map<double, Point>& ***getMapWestYCoordinates***() const {return \_mapWestYCoordinates;}

void ***makeConforming***();

const vector<Polygon>& ***getConformingReferenceElementPolygons***() const {return \_conformingReferenceElementPolygons;}

};

}

#endif // REFERENCEELEMENT\_H

REFERENCE\_ELEMENT\_CLASS.CPP

#include "ReferenceElement.hpp"

#include "fstream"

namespace **ReferenceElementNamespace** {

ReferenceElement::**ReferenceElement**()

{

\_boundingBoxVertices.resize(4);

}

void ReferenceElement::***computeBoundingBox***(Polygon& polygon)

{

unsigned int n ;

unsigned int numberVertices;

double minX, minY, maxX, maxY;

//initializing variables to a firts value to avoid an iteration in the for cicle

minX = polygon.*getVertexAtPosition*(0).getCoordinates().x();

maxX = polygon.*getVertexAtPosition*(0).getCoordinates().x();

minY = polygon.*getVertexAtPosition*(0).getCoordinates().y();

maxY = polygon.*getVertexAtPosition*(0).getCoordinates().y();

numberVertices = polygon.*getNumberVertices*();

//looking for the extremes in x and y to compute the bounding box

for(n = 1; n < numberVertices; n++ )

{

if (polygon.*getVertexAtPosition*(n).getCoordinates().x() < minX)

minX = polygon.*getVertexAtPosition*(n).getCoordinates().x();

if (polygon.*getVertexAtPosition*(n).getCoordinates().x() > maxX)

maxX = polygon.*getVertexAtPosition*(n).getCoordinates().x();

if (polygon.*getVertexAtPosition*(n).getCoordinates().y() < minY)

minY = polygon.*getVertexAtPosition*(n).getCoordinates().y();

if (polygon.*getVertexAtPosition*(n).getCoordinates().y() > maxY)

maxY = polygon.*getVertexAtPosition*(n).getCoordinates().y();

}

//initializing attributes

\_boundingBoxVertices[0] = Point(Vector2d(minX, minY));

\_boundingBoxVertices[1] = Point(Vector2d(maxX, minY));

\_boundingBoxVertices[2] = Point(Vector2d(maxX, maxY));

\_boundingBoxVertices[3] = Point(Vector2d(minX, maxY));

\_minX = minX;

\_minY = minY;

\_maxX = maxX;

\_maxY = maxY;

\_boundingBox = Polygon(\_boundingBoxVertices);

}

void ReferenceElement::***computeReferenceElement***(Polygon& polygon)

{

unsigned int i, n, numberVertices;

vector<Polygon> referenceElement;

vector<Point> vertices;

bool temp = 0;

double minX, minY, maxX, maxY;

numberVertices = polygon.*getNumberVertices*();

minX = \_boundingBoxVertices[0].getCoordinates().x();

minY = \_boundingBoxVertices[0].getCoordinates().y();

maxX = \_boundingBoxVertices[2].getCoordinates().x();

maxY = \_boundingBoxVertices[2].getCoordinates().y();

//push\_back of the original polygon in the vector of polygons inside the bounding box

\_referenceElementPolygons.push\_back(polygon);

//looking for the first vertex of the initial polygon that lies on the lower edge of the bounding box and to start the iteration

for(n = 0; n < numberVertices && temp == 0; n++)

{

if(polygon.*getVertexAtPosition*(n).getCoordinates().y() == minY)

{

temp = 1;

vertices.push\_back(polygon.*getVertexAtPosition*(n));

}

}

//for cicle on all polygon vertices

for(i = 0; i < numberVertices; i++)

{

double x = polygon.*getVertexAtPosition*((n + i) % numberVertices).getCoordinates().x();

double y = polygon.*getVertexAtPosition*((n + i) % numberVertices).getCoordinates().y();

//controlling if vertices[0](it has intersection with one of the edges) and the next vertex of the polygon lie on the same edge to not form a new polygon that is a segment

if ((y == minY) && (polygon.*getVertexAtPosition*((n + i - 1) % numberVertices).getCoordinates().y() == minY))

vertices[0] = polygon.*getVertexAtPosition*((n +i) % numberVertices);

//controlling if vertices[0](it has intersection with one of the edges) and the next vertex of the polygon lie on the same edge to not form a new polygon that is a segment

else if ((x == maxX) && (polygon.*getVertexAtPosition*((n + i - 1) % numberVertices).getCoordinates().x() == maxX))

vertices[0] = polygon.*getVertexAtPosition*((n +i) % numberVertices);

//controlling if vertices[0](it has intersection with one of the edges) and the next vertex of the polygon lie on the same edge to not form a new polygon that is a segment

else if ((y == maxY) && (polygon.*getVertexAtPosition*((n + i - 1) % numberVertices).getCoordinates().y() == maxY))

vertices[0] = polygon.*getVertexAtPosition*((n +i) % numberVertices);

//controlling if vertices[0](it has intersection with one of the edges) and the next vertex of the polygon lie on the same edge to not form a new polygon that is a segment

else if ((x == minX) && (polygon.*getVertexAtPosition*((n + i - 1) % numberVertices).getCoordinates().x() == minX))

vertices[0] = polygon.*getVertexAtPosition*((n +i) % numberVertices);

//if the vertex analyzed doesn'lie on an edge of the bounding box, it will be a vertex of the new polygon

else if((x != maxX) && (x != minX) && (y != minY) && (y != maxY))

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

//if the vertex analyzed lies on the bottom edge of the bounding box

else if(y == minY)

{

//if vertices[0] lies on the lower edge we immediately form a new polygon

if (vertices[0].getCoordinates().y() == minY)

{

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

}

//if vertices[0] lies on the left edge of the bounding box we control if we must add the first bounding box vertex

else if (vertices[0].getCoordinates().x() == minX)

{

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

//controlling if the vertex analyzed is different from the first bounding box vertex in order to add the bounding box vertex in the new polygon

if (polygon.*getVertexAtPosition*((n + i) % numberVertices) != \_boundingBoxVertices[0])

vertices.push\_back(\_boundingBoxVertices[0]);

}

//if vertices[0] lies on the upper edge of the bounding box we control if we must add the first or the fourth bounding box vertex

else if (vertices[0].getCoordinates().y() == maxY)

{

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

//controlling if the analyzed vertex is different form the first bounding box vertex in order to add the bounding box vertex to the new polygon

// if(polygon.getVertexAtPosition((n + i) % numberVertices) != \_boundingBoxVertices[0]) // forse da togliere

// vertices.push\_back(\_boundingBoxVertices[0]);

//controlling if the analyzed vertex is different form the third bounding box vertex in order to add the bounding box vertex to the new polygon

// we add only the third vertex because the only possible case is that in which vertices[0] = boundingBoxVertices[0]

if(vertices[0] != \_boundingBoxVertices[3])

vertices.push\_back(\_boundingBoxVertices[3]);

}

reverse(vertices.begin(), vertices.end()); //reversing vector to have anticlockwise order in the position of the vertices vector, according to the indices

//adding the new polygon

\_referenceElementPolygons.push\_back(Polygon(vertices));

vertices.clear();

//adding the last vertex analyzed, in fact tha last vertex may be the vertex of the next polygon

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

}

//if the vertex analyzed lies on the right edge of the bounding box

else if (x == maxX)

{

//if vertices[0] lies on the same edge we immediately add a new polygon

if (vertices[0].getCoordinates().x() == maxX)

{

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

}

//if vertices[0] lies on the lower edge we control if we must add the second bounding box vertex

else if(vertices[0].getCoordinates().y() == minY)

{

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

//if the analyzed vertex is different form the first bounding box vertex

if (polygon.*getVertexAtPosition*((n + i) % numberVertices) != \_boundingBoxVertices[1])

vertices.push\_back(\_boundingBoxVertices[1]);

}

//reversing vector to have anticlockwise order in the position of the vertices vector, according to the indices

reverse(vertices.begin(), vertices.end());

\_referenceElementPolygons.push\_back(Polygon(vertices));

vertices.clear();

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

}

//if the analyzed vertex lies on the upper edge of the bounding box

else if (y == maxY)

{

//if vertices[0] lies on the same edge we immediately add a new polygon

if (vertices[0].getCoordinates().y() == maxY)

{

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

}

//if vertices[0] lies on the right edge of the bounding box

else if (vertices[0].getCoordinates().x() == maxX)

{

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

//if the analyzed vertex is different from the third bounding box vertex

if (polygon.*getVertexAtPosition*((n + i) % numberVertices) != \_boundingBoxVertices[2])

vertices.push\_back(\_boundingBoxVertices[2]);

}

//reversing vector to have anticlockwise order in the position of the vertices vector, according to the indices

reverse(vertices.begin(), vertices.end());

\_referenceElementPolygons.push\_back(Polygon(vertices));

vertices.clear();

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

}

//if the analyzed vertex lies on the left edge

else if (x == minX)

{

//if vertices[0] lies on the same edge we immediately add a new polygon

if ( vertices[0].getCoordinates().x() == minX)

{

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

}

//if vertices[0] lies on the upper edge of the bounding box

else if (vertices[0].getCoordinates().y() == maxY)

{

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

//if the analyzed vertex is different from the fourth bounding box

if (polygon.*getVertexAtPosition*((n + i) % numberVertices) != \_boundingBoxVertices[3])

vertices.push\_back(\_boundingBoxVertices[3]);

}

//reversing vector to have anticlockwise order in the position of the vertices vector, according to the indices

reverse(vertices.begin(), vertices.end());

\_referenceElementPolygons.push\_back(Polygon(vertices));

vertices.clear();

vertices.push\_back(polygon.*getVertexAtPosition*((n + i) % numberVertices));

}

}

}

void ReferenceElement::**showReferenceElement**(const string filePath)

{

ofstream of;

of.open (filePath);

of << "width = 1.0;" << endl

<< "figure;" << endl

<< "hold on;" << endl << endl;

for (unsigned int n = 0; n < \_referenceElementPolygons.size(); n++)

{

of << "points" << n+1 << " = [" << endl;

const vector<Point>& points = \_referenceElementPolygons[n].*getVertices*();

for (unsigned int i = 0; i < \_referenceElementPolygons[n].*getNumberVertices*(); i++)

of << "\t" << points[i].getCoordinates()[0] << ", " << points[i].getCoordinates()[1] << endl;

of << "];" << endl;

of << "polygon" << n+1 << " = polyshape(points" << n+1 << ");" << endl

<< "plot(polygon" << n+1 << ", 'LineWidth', width);" << endl << endl;

}

of << "hold off;" << endl;

of.close();

}

void ReferenceElement::***findBoundaryVertices***()

{

const vector<Point>& vertices = \_referenceElementPolygons[0].*getVertices*();

for(unsigned int i = 0; i < vertices.size(); i++)

{

if(vertices[i].getCoordinates().y() == \_minY )

{//If the point is on the south edge of the bounding box

if(vertices[i] != \_boundingBoxVertices[0] && vertices[i] != \_boundingBoxVertices[1])

\_mapSouthXCoordinates.insert(pair<double, Point>(vertices[i].getCoordinates().x(), vertices[i]));

}

else if(vertices[i].getCoordinates().y() == \_maxY)

{//If the point is on the north edge of the bounding box

if(vertices[i] != \_boundingBoxVertices[2] && vertices[i] != \_boundingBoxVertices[3])

\_mapNorthXCoordinates.insert(pair<double, Point>(vertices[i].getCoordinates().x(), vertices[i]));

}

else if(vertices[i].getCoordinates().x() == \_maxX )

{//If the point is on the east edge of the bounding box

if(vertices[i] != \_boundingBoxVertices[1] && vertices[i] != \_boundingBoxVertices[2])

\_mapEastYCoordinates.insert(pair<double, Point>(vertices[i].getCoordinates().y(), vertices[i]));

}

else if(vertices[i].getCoordinates().x() == \_minX )

{//If the point is on the west edge of the bounding box

if(vertices[i] != \_boundingBoxVertices[3] && vertices[i] != \_boundingBoxVertices[0])

\_mapWestYCoordinates.insert(pair<double, Point>(vertices[i].getCoordinates().y(), vertices[i]));

}

}

}

void ReferenceElement::***makeConforming***()

{

\_conformingReferenceElementPolygons = \_referenceElementPolygons;

unsigned int numberNewPolygons = \_referenceElementPolygons.size()-1;

for(unsigned int p = 0; p <= numberNewPolygons; p++)

{

const vector<Point>& vertices = \_referenceElementPolygons[p].*getVertices*();

unsigned int numberVertices = \_referenceElementPolygons[p].*getNumberVertices*();

for(unsigned int i = 0; i < numberVertices; i++)

{

map<double, Point>::iterator it;

//If the current polygons's edge is on the south edge of the bounding box(first if), I project onto it the non-corner Points

//from the north edge of the box, meaning I insert the projected point as a new vertex on the current polygon

if(vertices[i].getCoordinates().y() == \_minY && vertices[(i + 1) % numberVertices].getCoordinates().y() == \_minY)

{

for ( it = \_mapNorthXCoordinates.begin(); it != \_mapNorthXCoordinates.end(); it++)

{ //it->first is the x coordinate of the North x coordinates, so the projection is (it->first,\_minY)

//ordering by x, the correct segment to project to is selected by the next if

if (vertices[i].getCoordinates().x() < it->first

&& it->first < vertices[(i + 1) % numberVertices].getCoordinates().x())

\_conformingReferenceElementPolygons[p].*insertVertexAtPosition*(Point(it->first,\_minY), (i + 1)%numberVertices);

}

}

//If the edge is on the north edge of the bounding box, I project onto it the non-corner Points from the south edge of the box

else if(vertices[i].getCoordinates().y() == \_maxY && vertices[(i + 1) % numberVertices].getCoordinates().y() == \_maxY)

{

for ( it = \_mapSouthXCoordinates.begin(); it != \_mapSouthXCoordinates.end(); it++)

{

if (vertices[i].getCoordinates().x() > it->first

&& it->first > vertices[(i + 1) % numberVertices].getCoordinates().x())

\_conformingReferenceElementPolygons[p].*insertVertexAtPosition*(Point(it->first,\_maxY), (i + 1)%numberVertices);

}

}

//If the edge is on the east edge of the bounding box, I project onto it the non-corner Points from the west edge of the box

else if(vertices[i].getCoordinates().x() == \_maxX && vertices[(i + 1) % numberVertices].getCoordinates().x() == \_maxX)

{

for ( it = \_mapWestYCoordinates.begin(); it != \_mapWestYCoordinates.end(); it++)

{

if (vertices[i].getCoordinates().y() < it->first

&& it->first < vertices[(i + 1) % numberVertices].getCoordinates().y())

\_conformingReferenceElementPolygons[p].*insertVertexAtPosition*(Point(\_maxX, it->first), (i + 1)%numberVertices);

}

}

//If the edge is on the west edge of the bounding box, I project onto it the non-corner Points from the east edge of the box

else if(vertices[i].getCoordinates().x() == \_minX && vertices[(i + 1) % numberVertices].getCoordinates().x() == \_minX)

{

for ( it = \_mapEastYCoordinates.begin(); it != \_mapEastYCoordinates.end(); it++)

{

if (vertices[i].getCoordinates().y() > it->first

&& it->first > vertices[(i + 1) % numberVertices].getCoordinates().y())

\_conformingReferenceElementPolygons[p].*insertVertexAtPosition*(Point(\_minX, it->first), (i + 1)%numberVertices);

}

}

}

}

}

}

MESH\_CLASS.HPP

#ifndef MESHCLASS\_H

#define MESHCLASS\_H

#include<ReferenceElement.hpp>

using namespace ReferenceElementNamespace;

namespace **MeshNamespace** {

class **IMesh** {

public:

virtual void ***createMesh***() = 0;

virtual ReferenceElement ***TranslateConformingReferenceElement***(double xTranslation, double yTranslation ) = 0;

virtual Polygon ***TranslatePolygon***(const Polygon& polygon, double xTranslation, double yTranslation) = 0;

virtual void ***setRectangularDomain***(const double& length, const double& height) = 0;

virtual void ***setConvexDomain***(const Polygon & convexDomain) = 0;

virtual bool ***referenceElementInConvexDomain***(ReferenceElement& referenceElement ) = 0;

virtual void ***cutAndDiscardPolygonsOnRightSide***( ReferenceElement& referenceElement, Segment& line ) = 0;

virtual double &***getBoundingBoxLength***() = 0;

virtual double &***getBoundingBoxHeight***() = 0;

virtual unsigned int &***getMaxNumRefElementsPerRow***() = 0;

virtual unsigned int &***getMaxNumRefElementsPerColumn***() = 0;

virtual const Polygon &***getDomainBoundingBox***() = 0;

virtual const Polygon& ***getDomain***() const = 0;

virtual const vector<ReferenceElement>& ***getMeshCells***() const = 0;

virtual const Polygon& ***getReferenceElementBoundingBox***() = 0;

virtual double& ***getDomainBoundingBoxLength***() = 0;

virtual double& ***getDomainBoundingBoxHeight***() = 0;

virtual ReferenceElement& ***getReferenceElement***() = 0;

virtual void ***showMesh***(const string filePath = "showMesh.m") = 0;

};

class **Mesh** : public IMesh

{

private:

vector<ReferenceElement> \_meshCells; //output

ReferenceElement \_referenceElement; //it's conforming

vector<Polygon> \_conformingReferenceElementPolygons;

Polygon \_referenceElementBoundingBox;

double \_boundingBoxLength;

double \_boundingBoxHeight;

Polygon \_domain;

Polygon \_domainBoundingBox;

unsigned int \_maxNumRefElementsPerRow;

unsigned int \_maxNumRefElementsPerColumn;

double \_domainBoundingBoxLength;

double \_domainBoundingBoxHeight;

public:

**Mesh**();

**Mesh**(Polygon& referencePolygon);

void ***createMesh***();

ReferenceElement ***TranslateConformingReferenceElement***(double xTranslation, double yTranslation ) ;

Polygon ***TranslatePolygon***(const Polygon& polygon, double xTranslation, double yTranslation) ;

void ***setRectangularDomain***(const double& length, const double& height);

void ***setConvexDomain***(const Polygon & convexDomain);

bool ***referenceElementInConvexDomain***(ReferenceElement& referenceElement );

void ***cutAndDiscardPolygonsOnRightSide***( ReferenceElement& referenceElement, Segment& line );

double &***getBoundingBoxLength***(){return \_boundingBoxLength;}

double &***getBoundingBoxHeight***(){return \_boundingBoxHeight;}

unsigned int &***getMaxNumRefElementsPerRow***(){return \_maxNumRefElementsPerRow;}

unsigned int &***getMaxNumRefElementsPerColumn***(){return \_maxNumRefElementsPerColumn;}

const Polygon &***getDomainBoundingBox***(){return \_domainBoundingBox; }

const Polygon& ***getDomain***() const{return \_domain;}

const vector<ReferenceElement>& ***getMeshCells***() const {return \_meshCells;}

const Polygon& ***getReferenceElementBoundingBox***(){ return \_referenceElementBoundingBox;}

double& ***getDomainBoundingBoxLength***(){ return \_domainBoundingBoxLength;}

double& ***getDomainBoundingBoxHeight***(){ return \_domainBoundingBoxHeight;}

ReferenceElement& ***getReferenceElement***(){ return \_referenceElement;}

void ***showMesh***(const string filePath = "showMesh.m");

};

}

#endif // MESHCLASS\_H

MESH\_CLASS.CPP

#include "Mesh.hpp"

#include "Polygon.hpp"

#include "IntersectorPolygonLine.hpp"

#include <fstream>

using namespace ReferenceElementNamespace;

using namespace IntersectorPolygonLineNamespace;

namespace **MeshNamespace** {

Mesh::**Mesh**(){

\_boundingBoxLength = 0;

\_boundingBoxHeight = 0;

}

//for generic convex domain

Mesh::**Mesh**(Polygon& referencePolygon)

{

\_referenceElement.*computeBoundingBox*(*referencePolygon*);

\_referenceElement.*computeReferenceElement*(*referencePolygon*);

\_referenceElementBoundingBox = \_referenceElement.\_boundingBox;

\_referenceElement.*findBoundaryVertices*();

\_referenceElement.*makeConforming*();

\_conformingReferenceElementPolygons = \_referenceElement.*getConformingReferenceElementPolygons*();

\_boundingBoxLength = \_referenceElementBoundingBox.*getEdges*()[0].*getTangent*().norm();

\_boundingBoxHeight = \_referenceElementBoundingBox.*getEdges*()[1].*getTangent*().norm();

}

void Mesh::***createMesh***()

{

//Before calling this function a Mesh Object should be constructed and a domain should be set

//place referenceElement at (0,0)

double xTranslation = -\_referenceElementBoundingBox.*getVertices*()[0].getCoordinates().x();

double yTranslation = -\_referenceElementBoundingBox.*getVertices*()[0].getCoordinates().y();

\_referenceElement = *TranslateConformingReferenceElement*(xTranslation, yTranslation);

\_referenceElementBoundingBox = \_referenceElement.\_boundingBox;

for(unsigned int row = 0; row < \_maxNumRefElementsPerColumn; row++)

{

for(unsigned int column = 0; column < \_maxNumRefElementsPerRow; column++)//columns of the row

{

//translate the ConformingReferenceElement in multiples of the bounding box's dimensions

ReferenceElement translatedReferenceElement = *TranslateConformingReferenceElement*(\_boundingBoxLength\*column, \_boundingBoxHeight\*row);

if(*referenceElementInConvexDomain*(*translatedReferenceElement*)) // If the translated reference Element is inside the domain

{

\_meshCells.push\_back(translatedReferenceElement); //just push it

}

else

{

//check if any edge of the domain intersects the bounding box of the translated reference element. If there is no intersection

//the referenceElement is outside so it is not added to the matrix of ReferenceElements. If there is an intersection, the translated

//reference element is cutted and a new one is built with the polygons that are inside the domain

bool intersectionWithDomain = false;

for( unsigned int i = 0; i < \_domain.*getNumberVertices*(); i++) //fixes a domain edge

{

Segment line = \_domain.*getEdges*()[i];

Intersector1D1D intersector1d1d;

const vector<Segment> edges = translatedReferenceElement.\_boundingBox.*getEdges*();

intersector1d1d.*SetFirstSegment*(line.*getStart*().getCoordinates(), line.*getEnd*().getCoordinates());

for (unsigned int l = 0; l < 4; l++)//edges of ReferenceElement's Bounding Box

{

intersector1d1d.*SetSecondSegment*(edges[l].*getStart*().getCoordinates(), edges[l].*getEnd*().getCoordinates());

if(intersector1d1d.*ComputeIntersection*()) //If there is intersection

{

//if the intersection is OnSegment with rispect to the domain

if (intersector1d1d.*TypeIntersection*() == Intersector1D1D::IntersectionOnSegment)

{

intersectionWithDomain = true;

// We need this flag so that in the push back below

//the reference Element is not added if it falls outside the domain

*cutAndDiscardPolygonsOnRightSide*(*translatedReferenceElement*, *line*);//this modifies the input translatedReferenceElement

break;

}

}

}//end for //break

}//end for : loop over domain edges

if (intersectionWithDomain)

\_meshCells.push\_back(translatedReferenceElement); // add the element after it has been cutted one or more times

}//end else

}//end for: column loop

}//end for : row loop

}

ReferenceElement Mesh::***TranslateConformingReferenceElement***( double xTranslation, double yTranslation )

{

vector<Polygon> translatedConformingPolygons;

vector<Polygon> conformingPolygons = \_referenceElement.*getConformingReferenceElementPolygons*();

for (unsigned int i = 0; i < conformingPolygons.size(); i++)

{

Polygon translatedPolygon = *TranslatePolygon*(conformingPolygons[i], xTranslation, yTranslation);

translatedConformingPolygons.push\_back(translatedPolygon);

}

Polygon translatedBoundingBox = *TranslatePolygon*(\_referenceElementBoundingBox, xTranslation, yTranslation);

//now that we have a new vector of polygons and the bounding box in the new Position

//a new referenceElement is created to group all that data

ReferenceElement translatedReferenceElement;

translatedReferenceElement.\_conformingReferenceElementPolygons = translatedConformingPolygons;

translatedReferenceElement.\_boundingBox = translatedBoundingBox;

translatedReferenceElement.\_boundingBoxVertices = translatedBoundingBox.*getVertices*();

return translatedReferenceElement;

}

Polygon Mesh::***TranslatePolygon***(const Polygon& polygon, double xTranslation, double yTranslation)

{

Polygon translatedPolygon = Polygon();

const vector<Point>& initialVertices = polygon.*getVertices*();

for (vector<Point>::const\_iterator it = initialVertices.begin(); it != initialVertices.end(); it++)

{

Point initialVertex = Point(it->getCoordinates());

Point translation = Point(xTranslation, yTranslation);

Point TranslatedVertex = initialVertex + translation;

translatedPolygon.*appendVertex*(TranslatedVertex);

}

translatedPolygon.*buildUpPolygonEdges*(); // this fixes the edges, since the appendVertices only modifies the polygon's vertices

return translatedPolygon;

}

void Mesh::***setRectangularDomain***(const double& domainLength, const double& domainHeight)

{

vector<Point> rectangleVertices;

rectangleVertices.resize(4);

rectangleVertices[0] = Point(0, 0);

rectangleVertices[1] = Point(domainLength, 0);

rectangleVertices[2] = Point(domainLength, domainHeight);

rectangleVertices[3] = Point(0, domainHeight);

\_domain = Polygon(rectangleVertices);

\_domainBoundingBox = \_domain;

\_domainBoundingBoxLength = domainLength;

\_domainBoundingBoxHeight = domainHeight;

\_maxNumRefElementsPerRow = (unsigned int)ceil((domainLength/\_boundingBoxLength));

\_maxNumRefElementsPerColumn = (unsigned int)ceil((domainHeight/\_boundingBoxHeight));

\_meshCells.reserve(\_maxNumRefElementsPerColumn\*\_maxNumRefElementsPerRow);

}

void Mesh::***setConvexDomain***(const Polygon & convexDomain)

{ //create a Mesh object with arguments before setting the domain

if( \_boundingBoxLength == 0 || \_boundingBoxHeight == 0)

throw runtime\_error("first create a Mesh object with a polygon");

\_domain = convexDomain;

ReferenceElement ref;

ref.*computeBoundingBox*(*\_domain*);

\_domainBoundingBox = ref.getBoundingBox();

\_domainBoundingBoxLength = \_domainBoundingBox.*getEdges*()[0].*getTangent*().norm();

\_domainBoundingBoxHeight = \_domainBoundingBox.*getEdges*()[1].*getTangent*().norm();

//the next two attributes allow to know how many times the original referenceElement should be translated rightwards and upwards

\_maxNumRefElementsPerRow = (unsigned int)ceil((\_domainBoundingBoxLength/\_boundingBoxLength));

\_maxNumRefElementsPerColumn = (unsigned int)ceil((\_domainBoundingBoxHeight/\_boundingBoxHeight));

\_meshCells.reserve(\_maxNumRefElementsPerColumn\*\_maxNumRefElementsPerRow);

}

bool Mesh::***referenceElementInConvexDomain***(ReferenceElement& referenceElement)

{

unsigned int i;

bool flag = 1; // by default the referenceElemnt is in the domain

for(i = 0; i < 4 && flag == 1; i++) //loops over the four vertices of the polygon's bounding box

{

Point BBvertex = referenceElement.*getBoundingBoxVertexAtPosition*(*i*);

bool flag2 = \_domain.*pointInPolygon*(BBvertex);

if(flag2 == 0) // if one of the vertices of the polygons's bounding box is not in the domain

flag = 0; // it means the reference element is not in the domain, flag is set to 0 and the following for cycles are interrupted

}

return flag;

}

void Mesh::***cutAndDiscardPolygonsOnRightSide***( ReferenceElement& referenceElement, Segment& line)

{

//This method creates a new vector of polygons called newVectorOfPolygons in which to store each of the new Polygons obtained cutting

//the vector of polygons referenceElement with the segment line

//at the end the vector<Polygon> \_conformingReferenceElementPolygons is updated with the new Polygons.

vector<Polygon> newVectorOfPolygons;

for (unsigned int i = 0; i < referenceElement.*getConformingReferenceElementPolygons*().size(); i++)

{

Polygon &polygon = referenceElement.\_conformingReferenceElementPolygons[i];

//if polygon is on the rigth side, skip the polygon discarding it

if(polygon.*isPolygonOnTheRightSide*(line) == 1)

continue;

//If polygon is on the left side, just add the whole polygon, no need to cut

else if(polygon.*isPolygonOnTheRightSide*(line) == -1)

{

newVectorOfPolygons.push\_back(polygon);

}

// If polygon is not on the rigth nor on the left there is intersection

else

{

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, line);

intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

for (unsigned int j = 0; j < newPolygons.size(); j++)

{

//if the new polygon is on the left side, add it

if(newPolygons[j].*isPolygonOnTheRightSide*(line) == -1)

newVectorOfPolygons.push\_back(newPolygons[j]);

}

}

}

referenceElement.\_conformingReferenceElementPolygons.clear();

referenceElement.\_conformingReferenceElementPolygons = newVectorOfPolygons;//referenceElement is updated with the new polygons

}

void Mesh::***showMesh***(const string filePath)

{

if (\_meshCells.empty())

throw runtime\_error("First you need to create mesh");

ofstream of;

of.open (filePath);

of << "width = 1.0;" << endl

<< "figure;" << endl

<< "hold on;" << endl

<< "axis off;" << endl << endl;

for (unsigned int i = 0; i < \_meshCells.size(); i++)

{

const vector<Polygon> conformingReferenceElementPolygons = \_meshCells[i].*getConformingReferenceElementPolygons*();

int number = conformingReferenceElementPolygons.size();

of << "newColors = lines(" << number << ");" << endl;

of << "colororder(newColors)" << endl << endl;

for (int n = 0; n < number; n++)

{

of << "points" << number\*i+n+1 << " = [" << endl;

const vector<Point>& points = conformingReferenceElementPolygons[n].*getVertices*();

for (unsigned int i = 0; i < conformingReferenceElementPolygons[n].*getNumberVertices*(); i++)

of << "\t" << points[i].getCoordinates()[0] << ", " << points[i].getCoordinates()[1] << endl;

of << "];" << endl;

of << "polygon" << number\*i+n+1 << " = polyshape(points" << number\*i+n+1 << ");" << endl

<< "plot(polygon" << number\*i+n+1 << ", 'LineWidth', width);" << endl << endl;

}

}

of << "hold off;" << endl;

of.close();

}

}

TEST(GOOGLE\_TEST\_LIBRARY):

SEGMENT\_CLASS\_TEST.HPP

#ifndef \_\_TEST\_SEGMENTCLASS\_H

#define \_\_TEST\_SEGMENTCLASS\_H

#include <gtest/gtest.h>

#include <gmock/gmock.h>

#include <gmock/gmock-matchers.h>

#include "Segment.hpp"

using namespace testing;

using namespace std;

namespace **SegmentTesting** {

TEST(***TestSegmentClass***, ***SetterGetterMethods***)

{

Point start{1, 2};

Point end{2, 3};

Segment segment;

segment.*setStart*(start);

segment.*setEnd*(end);

try

{

EXPECT\_EQ(segment.*getStart*(), start);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(segment.*getEnd*(), end);

}

catch (const exception& exception)

{

FAIL();

}

try

{

segment.*setSegment*(start, end);

EXPECT\_EQ(segment.*getStart*(), start);

EXPECT\_EQ(segment.*getEnd*(), end);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestSegmentClass***, ***TangentMethods***)

{

Point start{1, 2};

Point end{2, 3};

Segment segment(start, end);

const Vector2d tangentVector(1, 1);

try

{

EXPECT\_EQ(segment.*getTangent*() , tangentVector);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(segment.*computeTangent*() , tangentVector);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestSegmentClass***, ***PointOnTheRightSide***)

{

Point start{1, 0};

Point end{2, 0};

Segment segment;

Point test1(0.0,-3.0);

Point test2(1.5,0.0);

Point test3(0.0,13.0);

segment = Segment(start, end);

segment.*computeTangent*();

try

{

EXPECT\_EQ(1, segment.*pointOnTheRightSide*(test1));

EXPECT\_EQ(0, segment.*pointOnTheRightSide*(test2));

EXPECT\_EQ(-1, segment.*pointOnTheRightSide*(test3));

}

catch (const exception& exception)

{

FAIL();

}

}

}

#endif

POLYGON\_CLASS\_TEST.HPP

#ifndef \_\_TEST\_POLYGONCLASS\_H

#define \_\_TEST\_POLYGONCLASS\_H

#include <gtest/gtest.h>

#include <gmock/gmock.h>

#include <gmock/gmock-matchers.h>

#include "Polygon.hpp"

using namespace testing;

using namespace std;

namespace **PolygonTesting** {

TEST(***TestPolygonClass***, ***SetterGetterMethods***)

{

Point p0(1, 2);

Point p1(3, 2);

Point p2(2, 6);

vector<Point> vertices;

vertices.resize(3);

vertices[0] = p0;

vertices[1] = p1;

vertices[2] = p2;

vector<Segment> edges;

edges.resize(3);

edges[0] = Segment(p0, p1);

edges[1] = Segment(p1, p2);

edges[2] = Segment(p2, p0);

Polygon poly1( vertices);

Polygon poly2;

try

{

EXPECT\_EQ(poly1.*getNumberVertices*(), 3);

EXPECT\_EQ(poly2.*getNumberVertices*(), 0);

}

catch (const exception& exception)

{

FAIL();

}

Point p00(1, 2);

Point p11(3, 2);

Point p22(2, 6);

poly2.*appendVertex*(p00);

poly2.*appendVertex*(p11);

poly2.*appendVertex*(p22);

try

{

EXPECT\_EQ(poly1.*getVertices*(), vertices);

}

catch (const exception& exception)

{

FAIL();

}

//Testing getVertices

try

{

EXPECT\_EQ(poly1.*getVertices*(), vertices);

}

catch (const exception& exception)

{

FAIL();

}

//Testing setVertexAtPosition

Point p3(3.5, 2.3);

Point p4(2.2, 3.4);

Point p5(2.5, 3.4);

poly1.*setVertexAtPosition*(p3, 2);

poly1.*setVertexAtPosition*(p4, 1);

poly1.*setVertexAtPosition*(p5, 0);

try

{

EXPECT\_EQ(poly1.*getVertices*()[2], p3);

EXPECT\_EQ(poly1.*getVertices*()[1], p4);

EXPECT\_EQ(poly1.*getVertices*()[0], p5);

}

catch (const exception& exception)

{

FAIL();

}

//Testing getVertexAtPosition, getLastVertex

try

{

EXPECT\_EQ(poly1.*getVertexAtPosition*(2), p3);

EXPECT\_EQ(poly1.*getVertexAtPosition*(1), p4);

EXPECT\_EQ(poly1.*getVertexAtPosition*(0), p5);

EXPECT\_EQ(poly1.*getLastVertex*(), p3);

}

catch (const exception& exception)

{

FAIL();

}

//Testing getEdges

try

{

EXPECT\_EQ(poly1.*getEdges*(), edges);

}

catch (const exception& exception)

{

FAIL();

}

//testing nextVertexId

int id = 3;

try

{

poly1.*nextVertexId*(id);

}

catch (const exception& exception)

{

EXPECT\_EQ("Error: Vertex id is not present", std::string(exception.*what*()));

}

id = 2;

try

{

EXPECT\_EQ(poly1.*nextVertexId*(id), 0);

}

catch (const exception& exception)

{

FAIL();

}

id = 0;

try

{

EXPECT\_EQ(poly1.*nextVertexId*(id), 1);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestPolygonClass***, ***InsertionMethods***)

{

//Testing appendVertex

vector<Point> vertices;

vertices.resize(3);

Point p0(1,2);

Point p1(3,2);

Point p2(2,6);

vertices[0] = p0;

vertices[1] = p1;

vertices[2] = p2;

Polygon poly1( vertices);

Point p3(1.5,3.8);

poly1.*appendVertex*(p3);

try

{

EXPECT\_EQ(poly1.*getVertices*()[3], p3);

EXPECT\_EQ(poly1.*getNumberVertices*(), 4);

}

catch (const exception& exception)

{

FAIL();

}

//Testing insertVertexAtPosition

Point p4(2.2,1.8);

poly1.*insertVertexAtPosition*(p4,1);

try

{

EXPECT\_EQ(poly1.*getVertices*()[0], p0);

EXPECT\_EQ(poly1.*getVertices*()[1], p4);

EXPECT\_EQ(poly1.*getVertices*()[2], p1);

EXPECT\_EQ(poly1.*getVertices*()[3], p2);

EXPECT\_EQ(poly1.*getVertices*()[4], p3);

EXPECT\_EQ(poly1.*getNumberVertices*(), 5);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestPolygonClass***, ***TestLabelMethods***)

{

Point p0(1, 2);

Point p1(3, 2);

Point p2(2, 6);

vector<Point> vertices;

vertices.resize(3);

vertices[0] = p0;

vertices[1] = p1;

vertices[2] = p2;

Polygon poly(vertices);

try

{

poly.*setDefaultVertexLabels*();

EXPECT\_EQ(0, poly.*getVertexAtPosition*(0).getPointLabel());

EXPECT\_EQ(1, poly.*getVertexAtPosition*(1).getPointLabel());

EXPECT\_EQ(2, poly.*getVertexAtPosition*(2).getPointLabel());

}

catch (const exception& exception)

{

FAIL();

}

try

{

poly.*setVertexLabelAtPosition*(0, 1);

EXPECT\_EQ(1, poly.*getVertexLabelAtPosition*(0));

}

catch (const exception& exception)

{

FAIL();

}

vector<unsigned int> vector;

vector.push\_back(3);

vector.push\_back(5);

vector.push\_back(7);

try

{

poly.*setVertexLabels*(vector);

EXPECT\_EQ(3, poly.*getVertexAtPosition*(0).getPointLabel());

EXPECT\_EQ(5, poly.*getVertexAtPosition*(1).getPointLabel());

EXPECT\_EQ(7, poly.*getVertexAtPosition*(2).getPointLabel());

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestPolygonClass***, ***AreaMethod***)

{

Point p0(0,0);

Point p1(4,0);

Point p2(4,4);

Point p3(0,4);

vector<Point> vertices;

vertices.resize(4);

vertices[0] = p0;

vertices[1] = p1;

vertices[2] = p2;

vertices[3] = p3;

Polygon polygon(vertices);

try

{

EXPECT\_FLOAT\_EQ(16, polygon.*ComputeArea*());

}

catch (const exception& exception)

{

FAIL();

}

Point p4(1,2);

Point p5(2,0);

polygon.*insertVertexAtPosition*(p4, 1);

polygon.*insertVertexAtPosition*(p5, 2);

try

{

EXPECT\_FLOAT\_EQ(14, polygon.*ComputeArea*());

}

catch (const exception& exception)

{

FAIL();

}

Point p6(3,7);

Point p7(2,4);

Point p8(1,9);

polygon.*insertVertexAtPosition*(p6, 5);

polygon.*insertVertexAtPosition*(p7, 6);

polygon.*insertVertexAtPosition*(p8, 7);

try

{

EXPECT\_FLOAT\_EQ(22, polygon.*ComputeArea*());

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestPolygonClass***, ***TestPointInPolygon***)

{

vector<Point> polygonVertices;

polygonVertices.resize(5);

polygonVertices[0] = Point(2.5, 1.0);

polygonVertices[1] = Point(4.0, 2.1);

polygonVertices[2] = Point(3.4, 4.2);

polygonVertices[3] = Point(1.6, 4.2);

polygonVertices[4] = Point(1.0, 2.1);

Polygon polygon(polygonVertices);

Point test1(2.0,2.0);

Point test2(2.5,6.0);

Point test3(5.0,2.0);

try {

EXPECT\_TRUE(polygon.*pointInPolygon*(test1));

EXPECT\_FALSE(polygon.*pointInPolygon*(test2));

EXPECT\_FALSE(polygon.*pointInPolygon*(test3));

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestPolygonClass***, ***TestIsPolygonOnTheRightSide***)

{

vector<Point> polygon1Vertices;

vector<Point> polygon2Vertices;

polygon1Vertices.resize(3);

polygon2Vertices.resize(3);

polygon1Vertices[0] = Point(1, 0);

polygon1Vertices[1] = Point(4, 0);

polygon1Vertices[2] = Point(3, 4);

polygon2Vertices[0] = Point(4, 0);

polygon2Vertices[1] = Point(6, 0);

polygon2Vertices[2] = Point(5, 4);

Polygon polygon1(polygon1Vertices);

Polygon polygon2(polygon2Vertices);

Point start1(3, -1);

Point end1(3, 4);

Point start2(4, -1);

Point end2(4, 5);

Point start4(0, 0);

Point end4(3, 0);

Point start5(0, 0);

Point end5(1.5, 2);

Segment line1(start1, end1);

Segment line2(start2, end2);

Segment line3(end1, start1);

Segment line4(start4, end4);

Segment line5(end4, start4);

Segment line6(start5, end5);

Segment line7(end5, start5);

try {

EXPECT\_EQ(polygon1.*isPolygonOnTheRightSide*(line1), 0);

EXPECT\_EQ(polygon1.*isPolygonOnTheRightSide*(line2), -1);

EXPECT\_EQ(polygon2.*isPolygonOnTheRightSide*(line1), 1);

EXPECT\_EQ(polygon2.*isPolygonOnTheRightSide*(line2), 1);

EXPECT\_EQ(polygon1.*isPolygonOnTheRightSide*(line3), 0);

EXPECT\_EQ(polygon2.*isPolygonOnTheRightSide*(line3), -1);

EXPECT\_EQ(polygon1.*isPolygonOnTheRightSide*(line4), -1);

EXPECT\_EQ(polygon2.*isPolygonOnTheRightSide*(line4), -1);

EXPECT\_EQ(polygon1.*isPolygonOnTheRightSide*(line5), 1);

EXPECT\_EQ(polygon2.*isPolygonOnTheRightSide*(line5), 1);

EXPECT\_EQ(polygon1.*isPolygonOnTheRightSide*(line6), 1);

EXPECT\_EQ(polygon2.*isPolygonOnTheRightSide*(line6), 1);

EXPECT\_EQ(polygon1.*isPolygonOnTheRightSide*(line7), -1);

EXPECT\_EQ(polygon2.*isPolygonOnTheRightSide*(line7), -1);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestPolygonClass***, ***TestBuildUpPolygon***)

{

Polygon poly;

Point p0(0,2);

Point p1(1,0);

Point p2(4,3);

Point p3(3,4);

poly.*appendVertex*(p0);

poly.*appendVertex*(p1);

poly.*appendVertex*(p2);

poly.*appendVertex*(p3);

try {

EXPECT\_EQ(poly.*getEdges*().size(), 0);

poly.*buildUpPolygonEdges*();

EXPECT\_EQ(poly.*getEdges*().size(), 4);

}

catch (const exception& ex)

{

FAIL();

}

}

}

#endif // \_\_TEST\_POLYGONCLASS\_H

INTERSECTOR1D1D\_CLASS\_TEST.HPP

#ifndef \_\_TEST\_INTERSECTOR1D1D\_H

#define \_\_TEST\_INTERSECTOR1D1D\_H

#include <gtest/gtest.h>

#include <gmock/gmock.h>

#include <gmock/gmock-matchers.h>

#include "Intersector1D1D.hpp"

using namespace testing;

namespace **Intersector1D1DTesting** {

TEST(***TestIntersector1D1D***, ***TestParallelIntersection***)

{

Vector2d a(0, 0);

Vector2d b(4, 0);

Vector2d c(1, 0);

Vector2d d(2, 0);

Intersector1D1D intersectorNew;

intersectorNew.*SetFirstSegment*(a, b);

intersectorNew.*SetSecondSegment*(c, d);

EXPECT\_TRUE(intersectorNew.*ComputeIntersection*());

EXPECT\_EQ(intersectorNew.*TypeIntersection*(), IIntersector1D1D::IntersectionParallelOnSegment);

EXPECT\_FLOAT\_EQ(0.25, intersectorNew.*FirstParametricCoordinate*());

EXPECT\_FLOAT\_EQ(0.5, intersectorNew.*SecondParametricCoordinate*());

}

TEST(***TestIntersector1D1D***, ***TestSegmentIntersection***)

{

Vector2d a(1, 0);

Vector2d b(5, 0);

Vector2d c(3, -6);

Vector2d d(3, 6);

Intersector1D1D intersector;

intersector.*SetFirstSegment*(a, b);

intersector.*SetSecondSegment*(c, d);

EXPECT\_TRUE(intersector.*ComputeIntersection*());

EXPECT\_EQ(intersector.*TypeIntersection*(), IIntersector1D1D::IntersectionOnSegment);

EXPECT\_FLOAT\_EQ(0.5, intersector.*SecondParametricCoordinate*());

EXPECT\_FLOAT\_EQ(0.5, intersector.*FirstParametricCoordinate*());

EXPECT\_LE(intersector.*FirstParametricCoordinate*(), 1.0);

EXPECT\_GE(intersector.*FirstParametricCoordinate*(), 0.0);

}

TEST(***TestIntersector1D1D***, ***TestOnLineIntersection***)

{

Vector2d a(3, 6);

Vector2d b(3, 2);

Vector2d c(5, 0);

Vector2d d(1, 0);

Intersector1D1D intersector;

intersector.*SetFirstSegment*(a, b);

intersector.*SetSecondSegment*(c, d);

EXPECT\_TRUE(intersector.*ComputeIntersection*());

EXPECT\_EQ(intersector.*TypeIntersection*(), IIntersector1D1D::IntersectionOnLine);

EXPECT\_FLOAT\_EQ(1.5, intersector.*FirstParametricCoordinate*());

EXPECT\_FLOAT\_EQ(0.5, intersector.*SecondParametricCoordinate*());

EXPECT\_GE(intersector.*FirstParametricCoordinate*(), 1.0);

}

}

#endif // \_\_TEST\_INTERSECTOR1D1D\_H

INTERSECTOR\_POLYGON\_LINE\_CLASS\_TEST.HPP

#ifndef \_\_TEST\_INTERSECTORPOLYGONLINECLASS\_H

#define \_\_TEST\_INTERSECTORPOLYGONLINECLASS\_H

#include <gtest/gtest.h>

#include <gmock/gmock.h>

#include <gmock/gmock-matchers.h>

#include<iostream>

#include "IntersectorPolygonLine.hpp"

#include "Eigen"

using namespace IntersectorPolygonLineNamespace;

using namespace testing;

using namespace std;

using namespace Eigen;

namespace **IntersectorPolygonLineTesting** {

class **MockSegment** : public ISegment {

public:

MOCK\_METHOD1(**setStart**, void(const Point& start));

MOCK\_CONST\_METHOD0(**getStart**, const Point&());

MOCK\_METHOD0(**getStart**, Point&());

MOCK\_METHOD1(**setEnd**, void(const Point& end));

MOCK\_CONST\_METHOD0(**getEnd**, const Point&());

MOCK\_METHOD0(**getEnd**, Point&());

MOCK\_METHOD2(**setSegment**, void(const Point& start, const Point& end));

MOCK\_METHOD0(**computeTangent**, const Vector2d&());

MOCK\_CONST\_METHOD0(**getTangent**, const Vector2d&());

MOCK\_CONST\_METHOD1(**pointOnTheRightSide**, int(const Point& point));

};

class **MockPolygon** : public IPolygon {

public:

MOCK\_CONST\_METHOD0(**getNumberVertices**, unsigned int());

MOCK\_CONST\_METHOD0(**getVertices**, const vector<Point>&());

MOCK\_METHOD2(**setVertexAtPosition**, void(const Point& vertex, const int& position));

MOCK\_METHOD1(**appendVertex**, void(const Point& vertex));

MOCK\_METHOD2(**insertVertexAtPosition**, void(const Point& vertex, const int& position));

MOCK\_CONST\_METHOD1(**getVertexAtPosition**, const Point&(const int& position));

MOCK\_CONST\_METHOD1(**nextVertexId**, int(unsigned int id));

MOCK\_CONST\_METHOD0(**getLastVertex**, const Point&());

MOCK\_CONST\_METHOD0(**ComputeArea**, double());

MOCK\_CONST\_METHOD0(**getEdges**, const vector<Segment>&());

MOCK\_METHOD2(**setVertexLabelAtPosition**, void(const int& position, int label));

MOCK\_CONST\_METHOD1(**getVertexLabelAtPosition**, int(const int& position));

MOCK\_METHOD1(**setVertexLabels**, void(const vector<unsigned int> &polygonVertices));

MOCK\_METHOD0(**setDefaultVertexLabels**, void());

MOCK\_METHOD1(**pointInPolygon**, bool(const Point& point));

MOCK\_CONST\_METHOD1(**isPolygonOnTheRightSide**, int(const Segment &line));

MOCK\_METHOD0(**buildUpPolygonEdges**, void());

};

class **MockIntersector1D1D** : public IIntersector1D1D {

public:

MOCK\_METHOD2(**SetFirstSegment**, void(const Vector2d& origin, const Vector2d& end));

MOCK\_METHOD1(**SetFirstSegment**, void(const Segment& s));

MOCK\_METHOD2(**SetSecondSegment**, void(const Vector2d& origin, const Vector2d& end));

MOCK\_METHOD1(**SetSecondSegment**, void(const Segment& s));

MOCK\_METHOD0(**ComputeIntersection**, bool());

MOCK\_METHOD0(**ParametricCoordinates**, const Vector2d&());

MOCK\_METHOD0(**FirstParametricCoordinate**, const double&());

MOCK\_METHOD0(**SecondParametricCoordinate**, const double&());

MOCK\_METHOD0(**PositionIntersectionInFirstEdge**, const Position&());

MOCK\_METHOD0(**PositionIntersectionInSecondEdge**, const Position&());

MOCK\_METHOD0(**TypeIntersection**, const Type&());

MOCK\_CONST\_METHOD2(**IntersectionFirstParametricCoordinate**, Vector2d(const Vector2d& origin,const Vector2d& end));

MOCK\_CONST\_METHOD1(**IntersectionFirstParametricCoordinate**, Vector2d(const Segment& s));

MOCK\_CONST\_METHOD2(**IntersectionSecondParametricCoordinate**, Vector2d(const Vector2d& origin,const Vector2d& end));

};

TEST(***TestIntersectorPolygonLine***, ***TestFindIntersectionVertices***)

{

vector<Point> polygonVertices;

polygonVertices.resize(5);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(4, 0);

polygonVertices[2] = Point(2, 2);

polygonVertices[3] = Point(4, 4);

polygonVertices[4] = Point(0, 4);

Point start(3, 0.5);

Point end(3, 3.5);

Vector2d tangent = Vector2d(0, 3);

vector<double> parametric;

parametric.resize(4);

parametric[0] = -0.1666666667;

parametric[1] = 0.16666666667;

parametric[2] = 0.83333333333;

parametric[3] = 1.16666666667;

vector<Segment> edges;

edges.resize(5);

edges[0] = Segment(Point(0, 0), Point(4, 0));

edges[1] = Segment(Point(4, 0), Point(2, 2));

edges[2] = Segment(Point(2, 2), Point(4, 4));

edges[3] = Segment(Point(4, 4), Point(0, 4));

edges[4] = Segment(Point(0, 4), Point(0, 0));

Point intp1(3, 0);

Point intp2(3, 1);

Point intp3(3, 3);

Point intp4(3, 4);

Intersector1D1D::Type type = Intersector1D1D::Type::IntersectionOnSegment;

MockSegment segment;

MockPolygon polygon;

MockIntersector1D1D intersector1d1d;

EXPECT\_CALL(segment, getStart())

.WillRepeatedly(ReturnRef(*start*));

EXPECT\_CALL(segment, getEnd())

.WillRepeatedly(ReturnRef(*end*));

EXPECT\_CALL(segment, computeTangent)

.WillRepeatedly(ReturnRef(*tangent*));

EXPECT\_CALL(polygon, getNumberVertices)

.WillRepeatedly(Return(5));

EXPECT\_CALL(polygon, getEdges)

.WillRepeatedly(ReturnRef(*edges*));

EXPECT\_CALL(intersector1d1d, TypeIntersection)

.WillRepeatedly(ReturnRef(*type*));

EXPECT\_CALL(intersector1d1d, ComputeIntersection)

.WillOnce(Return(true))

.WillOnce(Return(true))

.WillOnce(Return(true))

.WillOnce(Return(true))

.WillOnce(Return(false));

EXPECT\_CALL(intersector1d1d, IntersectionFirstParametricCoordinate(\_, \_))

.WillOnce(Return(Vector2d(3, 0)))

.WillOnce(Return(Vector2d(3, 1)))

.WillOnce(Return(Vector2d(3, 3)))

.WillOnce(Return(Vector2d(3, 4)));

EXPECT\_CALL(intersector1d1d, FirstParametricCoordinate)

.WillOnce(ReturnRef(*parametric[*0*]*))

.WillOnce(ReturnRef(*parametric[*1*]*))

.WillOnce(ReturnRef(*parametric[*2*]*))

.WillOnce(ReturnRef(*parametric[*3*]*));

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

try

{

EXPECT\_EQ(intersectionPoints[0].getPointLabel(), 7);

EXPECT\_EQ(intersectionPoints[1].getPointLabel(), 8);

EXPECT\_EQ(intersectionPoints[2].getPointLabel(), 9);

EXPECT\_EQ(intersectionPoints[3].getPointLabel(), 10);

EXPECT\_EQ(Point(intersectionPoints[0].getCoordinates()), intp1);

EXPECT\_EQ(Point(intersectionPoints[1].getCoordinates()), intp2);

EXPECT\_EQ(Point(intersectionPoints[2].getCoordinates()), intp3);

EXPECT\_EQ(Point(intersectionPoints[3].getCoordinates()), intp4);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***TestFindNewPoints***)

{

vector<Point> polygonVertices;

polygonVertices.resize(5);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(4, 0);

polygonVertices[2] = Point(2, 2);

polygonVertices[3] = Point(4, 4);

polygonVertices[4] = Point(0, 4);

Point start(3, 0.5);

Point end(3, 3.5);

Vector2d tangent = Vector2d(0, 3);

vector<double> parametric;

parametric.resize(4);

parametric[0] = -0.1666666667;

parametric[1] = 0.16666666667;

parametric[2] = 0.83333333333;

parametric[3] = 1.16666666667;

vector<Segment> edges;

edges.resize(5);

edges[0] = Segment(Point(0, 0), Point(4, 0));

edges[1] = Segment(Point(4, 0), Point(2, 2));

edges[2] = Segment(Point(2, 2), Point(4, 4));

edges[3] = Segment(Point(4, 4), Point(0, 4));

edges[4] = Segment(Point(0, 4), Point(0, 0));

Point intp1(3, 0);

Point intp2(3, 1);

Point intp3(3, 3);

Point intp4(3, 4);

Intersector1D1D::Type type = Intersector1D1D::Type::IntersectionOnSegment;

MockSegment segment;

MockIntersector1D1D intersector1d1d;

MockPolygon polygon;

EXPECT\_CALL(polygon, getNumberVertices)

.WillRepeatedly(Return(5));

EXPECT\_CALL(polygon, getEdges)

.WillRepeatedly(ReturnRef(*edges*));

EXPECT\_CALL(polygon, getVertices)

.WillRepeatedly(ReturnRef(*polygonVertices*));

EXPECT\_CALL(polygon, nextVertexId(0)).WillRepeatedly(Return(1));

EXPECT\_CALL(polygon, nextVertexId(1)).WillRepeatedly(Return(2));

EXPECT\_CALL(polygon, nextVertexId(2)).WillRepeatedly(Return(3));

EXPECT\_CALL(polygon, nextVertexId(3)).WillRepeatedly(Return(4));

EXPECT\_CALL(polygon, nextVertexId(4)).WillRepeatedly(Return(0));

EXPECT\_CALL(intersector1d1d, TypeIntersection)

.WillRepeatedly(ReturnRef(*type*));

try

{

EXPECT\_CALL(segment, getStart())

.WillRepeatedly(ReturnRef(*start*));

EXPECT\_CALL(segment, getEnd())

.WillRepeatedly(ReturnRef(*end*));

EXPECT\_CALL(segment, computeTangent)

.WillRepeatedly(ReturnRef(*tangent*));

EXPECT\_CALL(intersector1d1d, ComputeIntersection)

.WillOnce(Return(true))

.WillOnce(Return(true))

.WillOnce(Return(true))

.WillOnce(Return(true))

.WillOnce(Return(false));

EXPECT\_CALL(intersector1d1d, IntersectionFirstParametricCoordinate(\_, \_))

.WillOnce(Return(Vector2d(3, 0)))

.WillOnce(Return(Vector2d(3, 1)))

.WillOnce(Return(Vector2d(3, 3)))

.WillOnce(Return(Vector2d(3, 4)));

EXPECT\_CALL(intersector1d1d, FirstParametricCoordinate)

.WillOnce(ReturnRef(*parametric[*0*]*))

.WillOnce(ReturnRef(*parametric[*1*]*))

.WillOnce(ReturnRef(*parametric[*2*]*))

.WillOnce(ReturnRef(*parametric[*3*]*));

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

EXPECT\_EQ(newPoints.size(), 6);

EXPECT\_EQ(newPoints[0].getPointLabel(), 5);

EXPECT\_EQ(newPoints[1].getPointLabel(), 6);

EXPECT\_EQ(newPoints[2].getPointLabel(), 7);

EXPECT\_EQ(newPoints[3].getPointLabel(), 8);

EXPECT\_EQ(newPoints[4].getPointLabel(), 9);

EXPECT\_EQ(newPoints[5].getPointLabel(), 10);

EXPECT\_TRUE(intersector.*getStartFound*());

EXPECT\_TRUE(intersector.*getEndFound*());

EXPECT\_EQ(Point(newPoints[0].getCoordinates()), segment.*getStart*());

EXPECT\_EQ(Point(newPoints[1].getCoordinates()), segment.*getEnd*());

EXPECT\_EQ(Point(newPoints[2].getCoordinates()), intp1);

EXPECT\_EQ(Point(newPoints[3].getCoordinates()), intp2);

EXPECT\_EQ(Point(newPoints[4].getCoordinates()), intp3);

EXPECT\_EQ(Point(newPoints[5].getCoordinates()), intp4);

}

catch (const exception& exception)

{

FAIL();

}

Point start2(3,2);

Point end2(3, 3.5);

Vector2d tangent2 = Vector2d(0, 1.5);

vector<double> parametric2;

parametric2.resize(4);

parametric2[0] = -1.3333333333;

parametric2[1] = -0.6666666667;

parametric2[2] = 0.66666666667;

parametric2[3] = 1.33333333333;

try

{

EXPECT\_CALL(segment, getStart())

.WillRepeatedly(ReturnRef(*start2*));

EXPECT\_CALL(segment, getEnd())

.WillRepeatedly(ReturnRef(*end2*));

EXPECT\_CALL(segment, computeTangent)

.WillRepeatedly(ReturnRef(*tangent2*));

EXPECT\_CALL(intersector1d1d, ComputeIntersection)

.WillOnce(Return(true))

.WillOnce(Return(true))

.WillOnce(Return(true))

.WillOnce(Return(true))

.WillOnce(Return(false));

EXPECT\_CALL(intersector1d1d, IntersectionFirstParametricCoordinate(\_, \_))

.WillOnce(Return(Vector2d(3, 0)))

.WillOnce(Return(Vector2d(3, 1)))

.WillOnce(Return(Vector2d(3, 3)))

.WillOnce(Return(Vector2d(3, 4)));

EXPECT\_CALL(intersector1d1d, FirstParametricCoordinate)

.WillOnce(ReturnRef(*parametric2[*0*]*))

.WillOnce(ReturnRef(*parametric2[*1*]*))

.WillOnce(ReturnRef(*parametric2[*2*]*))

.WillOnce(ReturnRef(*parametric2[*3*]*));

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

EXPECT\_EQ(newPoints.size(), 5);

EXPECT\_EQ(newPoints[0].getPointLabel(), 6);

EXPECT\_EQ(newPoints[1].getPointLabel(), 7);

EXPECT\_EQ(newPoints[2].getPointLabel(), 8);

EXPECT\_EQ(newPoints[3].getPointLabel(), 9);

EXPECT\_EQ(newPoints[4].getPointLabel(), 10);

EXPECT\_FALSE(intersector.*getStartFound*());

EXPECT\_TRUE(intersector.*getEndFound*());

EXPECT\_EQ(Point(newPoints[0].getCoordinates()), segment.*getEnd*());

EXPECT\_EQ(Point(newPoints[1].getCoordinates()), intp1);

EXPECT\_EQ(Point(newPoints[2].getCoordinates()), intp2);

EXPECT\_EQ(Point(newPoints[3].getCoordinates()), intp3);

EXPECT\_EQ(Point(newPoints[4].getCoordinates()), intp4);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***TestExtendPolygonIfEndpointsOnEdge***)

{

vector<Point> polygonVertices;

polygonVertices.resize(5);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(4, 0);

polygonVertices[2] = Point(2, 2);

polygonVertices[3] = Point(4, 4);

polygonVertices[4] = Point(0, 4);

// setting labels

polygonVertices[0].setPointLabel(0);

polygonVertices[1].setPointLabel(1);

polygonVertices[2].setPointLabel(2);

polygonVertices[3].setPointLabel(3);

polygonVertices[4].setPointLabel(4);

Point start(3, 0.5);

Point end(3, 3.5);

Vector2d tangent = Vector2d(0, 3);

vector<double> parametric;

parametric.resize(4);

parametric[0] = -0.1666666667;

parametric[1] = 0.16666666667;

parametric[2] = 0.83333333333;

parametric[3] = 1.16666666667;

vector<Segment> edges;

edges.resize(5);

edges[0] = Segment(Point(0, 0), Point(4, 0));

edges[1] = Segment(Point(4, 0), Point(2, 2));

edges[2] = Segment(Point(2, 2), Point(4, 4));

edges[3] = Segment(Point(4, 4), Point(0, 4));

edges[4] = Segment(Point(0, 4), Point(0, 0));

Point intp1(3, 0);

Point intp2(3, 1);

Point intp3(3, 3);

Point intp4(3, 4);

Intersector1D1D::Type type = Intersector1D1D::Type::IntersectionOnSegment;

MockSegment segment;

MockIntersector1D1D intersector1d1d;

MockPolygon polygon;

EXPECT\_CALL(segment, getStart())

.WillRepeatedly(ReturnRef(*start*));

EXPECT\_CALL(segment, getEnd())

.WillRepeatedly(ReturnRef(*end*));

EXPECT\_CALL(segment, computeTangent)

.WillRepeatedly(ReturnRef(*tangent*));

EXPECT\_CALL(polygon, getNumberVertices)

.WillRepeatedly(Return(5));

EXPECT\_CALL(polygon, getEdges)

.WillRepeatedly(ReturnRef(*edges*));

EXPECT\_CALL(polygon, getVertices)

.WillRepeatedly(ReturnRef(*polygonVertices*));

EXPECT\_CALL(polygon, nextVertexId(0)).WillRepeatedly(Return(1));

EXPECT\_CALL(polygon, nextVertexId(1)).WillRepeatedly(Return(2));

EXPECT\_CALL(polygon, nextVertexId(2)).WillRepeatedly(Return(3));

EXPECT\_CALL(polygon, nextVertexId(3)).WillRepeatedly(Return(4));

EXPECT\_CALL(polygon, nextVertexId(4)).WillRepeatedly(Return(0));

EXPECT\_CALL(intersector1d1d, TypeIntersection)

.WillRepeatedly(ReturnRef(*type*));

EXPECT\_CALL(intersector1d1d, ComputeIntersection)

.WillOnce(Return(true))

.WillOnce(Return(true))

.WillOnce(Return(true))

.WillOnce(Return(true))

.WillOnce(Return(false));

EXPECT\_CALL(intersector1d1d, IntersectionFirstParametricCoordinate(\_, \_))

.WillOnce(Return(Vector2d(3, 0)))

.WillOnce(Return(Vector2d(3, 1)))

.WillOnce(Return(Vector2d(3, 3)))

.WillOnce(Return(Vector2d(3, 4)));

EXPECT\_CALL(intersector1d1d, FirstParametricCoordinate)

.WillOnce(ReturnRef(*parametric[*0*]*))

.WillOnce(ReturnRef(*parametric[*1*]*))

.WillOnce(ReturnRef(*parametric[*2*]*))

.WillOnce(ReturnRef(*parametric[*3*]*));

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 7);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 3);

EXPECT\_EQ(newPolygons[2].*getNumberVertices*(), 3);

EXPECT\_EQ(newExtendedPolygons[0].*getNumberVertices*(), 9);

EXPECT\_EQ(newExtendedPolygons[1].*getNumberVertices*(), 4);

EXPECT\_EQ(newExtendedPolygons[2].*getNumberVertices*(), 4);

EXPECT\_TRUE(intersector.*getStartFound*());

EXPECT\_TRUE(intersector.*getEndFound*());

//Polygon 0-7--8-2-9--10-4

EXPECT\_EQ(newPolygons[0].*getVertexLabelAtPosition*(1), 7);

EXPECT\_EQ(newPolygons[0].*getVertexLabelAtPosition*(2), 8);

EXPECT\_EQ(newPolygons[0].*getVertexLabelAtPosition*(4), 9);

EXPECT\_EQ(newPolygons[0].*getVertexLabelAtPosition*(5), 10);

//Polygon 0-7-5-8-2-9-6-10-4

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(1), 7);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(2), 5);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(3), 8);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(5), 9);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(6), 6);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(7), 10);

//Polygon 1-8--7

EXPECT\_EQ(newPolygons[1].*getVertexLabelAtPosition*(0), 1);

EXPECT\_EQ(newPolygons[1].*getVertexLabelAtPosition*(1), 8);

EXPECT\_EQ(newPolygons[1].*getVertexLabelAtPosition*(2), 7);

//Polygon 1-8-5-7

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(0), 1);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(1), 8);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(2), 5);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(3), 7);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons1a***) //concave hexagon

{

vector<Point> polygonVertices;

polygonVertices.resize(6);

polygonVertices[0] = Point(1.5, 1.0);

polygonVertices[1] = Point(5.6, 1.5);

polygonVertices[2] = Point(5.5, 4.8);

polygonVertices[3] = Point(4.0, 6.2);

polygonVertices[4] = Point(3.2, 4.2);

polygonVertices[5] = Point(1.0, 4.0);

Polygon polygon(polygonVertices);

Point start(2.0, 3.7);

Point end(4.1, 5.9);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 3);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 8);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 3);

EXPECT\_EQ(newPolygons[2].*getNumberVertices*(), 3);

EXPECT\_EQ(newExtendedPolygons[0].*getNumberVertices*(), 10);

EXPECT\_EQ(newExtendedPolygons[1].*getNumberVertices*(), 4);

EXPECT\_EQ(newExtendedPolygons[2].*getNumberVertices*(), 4);

}

catch (const exception& exception)

{

FAIL();

}

try

{

//Polygon 0-1-2-11-7-10-4-9-6-8-0

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(3), 11);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(4), 7);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(5), 10);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(6), 4);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(7), 9);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(8), 6);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(9), 8);

//Polygon 3-10-7-11

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(0), 3);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(1), 10);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(2), 7);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(3), 11);

//Polygon 5-8-6-9

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(0), 5);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(1), 8);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(2), 6);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(3), 9);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons1b***) //concave hexagon swapped segment endpoints

{

vector<Point> polygonVertices;

polygonVertices.resize(6);

polygonVertices[0] = Point(1.5, 1.0);

polygonVertices[1] = Point(5.6, 1.5);

polygonVertices[2] = Point(5.5, 4.8);

polygonVertices[3] = Point(4.0, 6.2);

polygonVertices[4] = Point(3.2, 4.2);

polygonVertices[5] = Point(1.0, 4.0);

Polygon polygon(polygonVertices);

Point end(2.0, 3.7);

Point start(4.1, 5.9);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 3);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 8);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 3);

EXPECT\_EQ(newPolygons[2].*getNumberVertices*(), 3);

EXPECT\_EQ(newExtendedPolygons[0].*getNumberVertices*(), 10);

EXPECT\_EQ(newExtendedPolygons[1].*getNumberVertices*(), 4);

EXPECT\_EQ(newExtendedPolygons[2].*getNumberVertices*(), 4);

}

catch (const exception& exception)

{

FAIL();

}

try

{

//Polygon 0-1-2-8-6-9-4-10-7-11

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(3), 8);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(4), 6);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(5), 9);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(6), 4);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(7), 10);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(8), 7);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(9), 11);

//Polygon 3-9-6-8

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(0), 3);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(1), 9);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(2), 6);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(3), 8);

//Polygon 5-11-7-10

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(0), 5);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(1), 11);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(2), 7);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(3), 10);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons1c***) //concave hexagon , start/end consecutive

{

vector<Point> polygonVertices;

polygonVertices.resize(6);

polygonVertices[0] = Point(1.5, 1.0);

polygonVertices[1] = Point(5.6, 1.5);

polygonVertices[2] = Point(5.5, 4.8);

polygonVertices[3] = Point(4.0, 6.2);

polygonVertices[4] = Point(3.2, 4.2);

polygonVertices[5] = Point(1.0, 4.0);

Polygon polygon(polygonVertices);

Point start(2.0, 3.7);

Vector2d p1(2.0, 3.7);

Vector2d p2(4.1, 5.9);

Vector2d p3 = 0.9 \* p1 + 0.1 \* p2;

Point end(p3);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 3);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 8);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 3);

EXPECT\_EQ(newPolygons[2].*getNumberVertices*(), 3);

EXPECT\_EQ(newExtendedPolygons[0].*getNumberVertices*(), 10);

EXPECT\_EQ(newExtendedPolygons[1].*getNumberVertices*(), 3);

EXPECT\_EQ(newExtendedPolygons[2].*getNumberVertices*(), 5);

}

catch (const exception& exception)

{

FAIL();

}

try

{

//Polygon 0-1-2-11-7-10-4-9-7-6-8

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(3), 11);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(4), 10);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(5), 4);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(6), 9);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(7), 7);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(8), 6);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(9), 8);

//Polygon 3-10-11

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(0), 3);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(1), 10);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(2), 11);

//Polygon 5-8-6-7-9

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(0), 5);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(1), 8);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(2), 6);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(3), 7);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(4), 9);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons1d***) //concave hexagon, start in/ end out

{

vector<Point> polygonVertices;

polygonVertices.resize(6);

polygonVertices[0] = Point(1.5, 1.0);

polygonVertices[1] = Point(5.6, 1.5);

polygonVertices[2] = Point(5.5, 4.8);

polygonVertices[3] = Point(4.0, 6.2);

polygonVertices[4] = Point(3.2, 4.2);

polygonVertices[5] = Point(1.0, 4.0);

Polygon polygon(polygonVertices);

Point start(2.0, 3.7);

Point end(3.06, 4.81);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 3);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 8);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 3);

EXPECT\_EQ(newPolygons[2].*getNumberVertices*(), 3);

EXPECT\_EQ(newExtendedPolygons[0].*getNumberVertices*(), 9);

EXPECT\_EQ(newExtendedPolygons[1].*getNumberVertices*(), 3);

EXPECT\_EQ(newExtendedPolygons[2].*getNumberVertices*(), 4);

}

catch (const exception& exception)

{

FAIL();

}

try

{

//Polygon 0-1-2-11-10-4-9-6-8

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(3), 11);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(4), 10);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(5), 4);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(6), 9);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(7), 6);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(8), 8);

//Polygon 3-10-11

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(0), 3);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(1), 10);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(2), 11);

//Polygon 5-8-6-9

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(0), 5);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(1), 8);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(2), 6);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(3), 9);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons1e***) //concave hexagon, start out/ end out

{

vector<Point> polygonVertices;

polygonVertices.resize(6);

polygonVertices[0] = Point(1.5, 1.0);

polygonVertices[1] = Point(5.6, 1.5);

polygonVertices[2] = Point(5.5, 4.8);

polygonVertices[3] = Point(4.0, 6.2);

polygonVertices[4] = Point(3.2, 4.2);

polygonVertices[5] = Point(1.0, 4.0);

Polygon polygon(polygonVertices);

Point start(3.06, 4.81);

Point end(4.8, 6.1);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 3);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 8);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 3);

EXPECT\_EQ(newPolygons[2].*getNumberVertices*(), 3);

EXPECT\_EQ(newExtendedPolygons[0].*getNumberVertices*(), 8);

EXPECT\_EQ(newExtendedPolygons[1].*getNumberVertices*(), 3);

EXPECT\_EQ(newExtendedPolygons[2].*getNumberVertices*(), 3);

}

catch (const exception& exception)

{

FAIL();

}

try

{

//Polygon 0-1-2-11-10-4-9-8

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(3), 11);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(4), 10);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(5), 4);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(6), 9);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(7), 8);

//Polygon 3-10-11

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(0), 3);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(1), 10);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(2), 11);

//Polygon 5-8-9

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(0), 5);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(1), 8);

EXPECT\_EQ(newExtendedPolygons[2].*getVertexLabelAtPosition*(2), 9);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons2a***)//convex pentagon with segment aligned with a vertex

{

vector<Point> polygonVertices;

polygonVertices.resize(5);

polygonVertices[0] = Point(2.5, 1.0);

polygonVertices[1] = Point(4.0, 2.1);

polygonVertices[2] = Point(3.4, 4.2);

polygonVertices[3] = Point(1.6, 4.2);

polygonVertices[4] = Point(1.0, 2.1);

Polygon polygon(polygonVertices);

Point start(1.4, 2.75);

Point end(3.6, 2.2);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 2);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*() , 4);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 4);

EXPECT\_EQ(newExtendedPolygons[0].*getNumberVertices*() , 6);

EXPECT\_EQ(newExtendedPolygons[1].*getNumberVertices*(), 6);

}

catch (const exception& exception)

{

FAIL();

}

try

{

//Polygon 0-1-6-5-7-4

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(0), 0);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(1), 1);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(2), 6);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(3), 5);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(4), 7);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(5), 4);

//Polygon 2-3-7-5-6-1

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(0), 1);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(1), 2);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(2), 3);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(3), 7);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(4), 5);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(5), 6);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons2b***)//convex pentagon with segment passing through 2 vertices

{

vector<Point> polygonVertices;

polygonVertices.resize(5);

polygonVertices[0] = Point(2.5, 1.0);

polygonVertices[1] = Point(4.0, 2.1);

polygonVertices[2] = Point(3.4, 4.2);

polygonVertices[3] = Point(1.6, 4.2);

polygonVertices[4] = Point(1.0, 2.1);

Polygon polygon(polygonVertices);

Point start(0.0, 2.1);

Point end(5, 2.1);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 2);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*() , 3);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 4);

EXPECT\_EQ(newExtendedPolygons[0].*getNumberVertices*() , 3);

EXPECT\_EQ(newExtendedPolygons[1].*getNumberVertices*(), 4);

}

catch (const exception& exception)

{

FAIL();

}

try

{

//Polygon 0-1-4

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(0), 0);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(1), 1);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(2), 4);

//Polygon 1-2-3-4

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(0), 1);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(1), 2);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(2), 3);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(3), 4);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons3***)//poligono a piacere 1

{

vector<Point> polygonVertices;

polygonVertices.resize(8);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(8, 0);

polygonVertices[2] = Point(4.64, 1.5);

polygonVertices[3] = Point(8.93, 4.48);

polygonVertices[4] = Point(2.18, 5.36);

polygonVertices[5] = Point(5.52, 9.22);

polygonVertices[6] = Point(0, 5);

polygonVertices[7] = Point(6, 4);

Polygon polygon(polygonVertices);

Point start(7.28, -0.72);

Point end(1.2, 9.14);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 5);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 6);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 3);

EXPECT\_EQ(newPolygons[2].*getNumberVertices*(), 6);

EXPECT\_EQ(newPolygons[3].*getNumberVertices*(), 6);

EXPECT\_EQ(newPolygons[4].*getNumberVertices*(), 3);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons4***)//poligono a piacere 1 -segmento finisce in un vertice

{

vector<Point> polygonVertices;

polygonVertices.resize(8);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(8, 0);

polygonVertices[2] = Point(4.64, 1.5);

polygonVertices[3] = Point(8.93, 4.48);

polygonVertices[4] = Point(2.18, 5.36);

polygonVertices[5] = Point(5.52, 9.22);

polygonVertices[6] = Point(0, 5);

polygonVertices[7] = Point(6, 4);

Polygon polygon(polygonVertices);

Point start(7.28, -0.72);

Point end(5.52, 9.22); //=polygonVertices[5]

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 3);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 10);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 3);

EXPECT\_EQ(newPolygons[2].*getNumberVertices*(), 3);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons5***) //TEST FACILE1- square, endpoints outside

{

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(4, 0);

polygonVertices[2] = Point(4, 4);

polygonVertices[3] = Point(0, 4);

Polygon polygon(polygonVertices);

Point end(5, 1);

Point start(2, 5);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 2);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*() , 5);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 3);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons5b***) //TEST FACILE2 - square, endpoints inside

{

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(4, 0);

polygonVertices[2] = Point(4, 4);

polygonVertices[3] = Point(0, 4);

Polygon polygon(polygonVertices);

Point end(3.03, 3.63);

Point start(3.48, 3.03);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 2);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*() , 5);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 3);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons5c***) //TEST FACILE3 - square, one endpoint inside, one outside

{

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(4, 0);

polygonVertices[2] = Point(4, 4);

polygonVertices[3] = Point(0, 4);

Polygon polygon(polygonVertices);

Point start(3.03, 3.63);

Point end(2, 5);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 2);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*() , 5);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 3);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons5d***) //TEST FACILE4 - square, one endpoint inside, one outside

{

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(4, 0);

polygonVertices[2] = Point(4, 4);

polygonVertices[3] = Point(0, 4);

Polygon polygon(polygonVertices);

Point start(3.48, 3.03);

Point end(5, 1);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 2);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*() , 5);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 3);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons6***) // rettangolo

{

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(1, 1);

polygonVertices[1] = Point(5, 1);

polygonVertices[2] = Point(5, 3.1);

polygonVertices[3] = Point(1, 3.1);

Polygon polygon(polygonVertices);

Point start(2, 1.2);

Point end(4, 3);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 2);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*() , 4);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 4);

EXPECT\_EQ(newExtendedPolygons[0].*getNumberVertices*(), 6);

EXPECT\_EQ(newExtendedPolygons[1].*getNumberVertices*(), 6);

}

catch (const exception& exception)

{

FAIL();

}

try

{

//Polygon 0-6-4-5-7-3

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(0), 0);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(1), 6);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(2), 4);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(3), 5);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(4), 7);

EXPECT\_EQ(newExtendedPolygons[0].*getVertexLabelAtPosition*(5), 3);

//Polygon 1-2-7-5-4-6

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(0), 1);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(1), 2);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(2), 7);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(3), 5);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(4), 4);

EXPECT\_EQ(newExtendedPolygons[1].*getVertexLabelAtPosition*(5), 6);

}

catch (const exception& exception)

{

FAIL();

}

}

//test con il poligono di Dauria ma cambiando la posizione dei capi del segmento

TEST(***TestIntersectorPolygonLine***, ***FindPolygons7***) //poligono a piacere 2 D' Auria

{

vector<Point> polygonVertices;

polygonVertices.resize(10);

polygonVertices[0] = Point(2, -2);

polygonVertices[1] = Point(0, -1);

polygonVertices[2] = Point(3, 1);

polygonVertices[3] = Point(0, 2);

polygonVertices[4] = Point(3, 2);

polygonVertices[5] = Point(3, 3);

polygonVertices[6] = Point(-1, 3);

polygonVertices[7] = Point(-3, 1);

polygonVertices[8] = Point(0, 0);

polygonVertices[9] = Point(-3, -2);

Polygon polygon(polygonVertices);

Point start(-4, -4);

Point end(4, 4);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 4);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 6);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 7);

EXPECT\_EQ(newPolygons[2].*getNumberVertices*(), 3);

EXPECT\_EQ(newPolygons[3].*getNumberVertices*(), 3);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***FindPolygons7b***) //poligono a piacere 2 D' Auria ( a forma di sigma)

{

vector<Point> polygonVertices;

polygonVertices.resize(10);

polygonVertices[0] = Point(2, -2);

polygonVertices[1] = Point(0, -1);

polygonVertices[2] = Point(3, 1);

polygonVertices[3] = Point(0, 2);

polygonVertices[4] = Point(3, 2);

polygonVertices[5] = Point(3, 3);

polygonVertices[6] = Point(-1, 3);

polygonVertices[7] = Point(-3, 1);

polygonVertices[8] = Point(0, 0);

polygonVertices[9] = Point(-3, -2);

Polygon polygon(polygonVertices);

Point start(0, -3);

Point end(0, 4);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 5);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 3);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 4);

EXPECT\_EQ(newPolygons[2].*getNumberVertices*(), 4);

EXPECT\_EQ(newPolygons[3].*getNumberVertices*(), 5);

EXPECT\_EQ(newPolygons[4].*getNumberVertices*(), 4);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***CutPolygonOnEdge***) //for Project 2

{

vector<Point> polygonVertices;

polygonVertices.resize(10);

polygonVertices[0] = Point(0, 2);

polygonVertices[1] = Point(1, 1);

polygonVertices[2] = Point(2, 2);

polygonVertices[3] = Point(3, 0);

polygonVertices[4] = Point(4, 0);

polygonVertices[5] = Point(5, 2);

polygonVertices[6] = Point(6, 2);

polygonVertices[7] = Point(6, 3);

polygonVertices[8] = Point(3, 4);

polygonVertices[9] = Point(0, 3);

Polygon polygon(polygonVertices);

Point start(0, 0);

Point end(6, 0);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 1);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 10);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***CutPolygonOnEdge2***) //for Project 2

{

vector<Point> polygonVertices;

polygonVertices.resize(10);

polygonVertices[0] = Point(0, 2);

polygonVertices[1] = Point(1, 1);

polygonVertices[2] = Point(2, 2);

polygonVertices[3] = Point(3, 0);

polygonVertices[4] = Point(4, 0);

polygonVertices[5] = Point(5, 2);

polygonVertices[6] = Point(6, 2);

polygonVertices[7] = Point(6, 3);

polygonVertices[8] = Point(3, 4);

polygonVertices[9] = Point(0, 3);

Polygon polygon(polygonVertices);

Point start(3.5, 0);

Point end(3.8, 0);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 1);

EXPECT\_EQ(newExtendedPolygons.size(), 1);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 10);

EXPECT\_EQ(newExtendedPolygons[0].*getNumberVertices*(), 12);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***CutPolygonOnEdge3***) //for Project 2

{

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(1, 0);

polygonVertices[2] = Point(1, 1);

polygonVertices[3] = Point(0, 1);

Polygon polygon(polygonVertices);

Point start(-1, 0);

Point end(0.5, 0);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 1);

EXPECT\_EQ(newExtendedPolygons.size(), 1);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 4);

EXPECT\_EQ(newExtendedPolygons[0].*getNumberVertices*(), 5);

}

catch (const exception& exception)

{

FAIL();

}

}

TEST(***TestIntersectorPolygonLine***, ***CutPolygonOnEdge4***) //for Project 2

{

vector<Point> polygonVertices;

polygonVertices.resize(8);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(1, 0);

polygonVertices[2] = Point(1, 1);

polygonVertices[3] = Point(2, 1);

polygonVertices[4] = Point(2, 0);

polygonVertices[5] = Point(3, 0);

polygonVertices[6] = Point(3, 2);

polygonVertices[7] = Point(0, 2);

Polygon polygon(polygonVertices);

Point start(1.5, 0);

Point end(2.5, 0);

Segment segment(start, end);

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

vector<Point> intersectionPoints = intersector.*findIntersectionVertices*();

vector<Polygon> newPolygons = intersector.*findPolygons*();

vector<Point> newPoints = intersector.*findNewPoints*();

vector<Polygon> newExtendedPolygons = intersector.*getExtendedPolygons*();

try

{

EXPECT\_EQ(newPolygons.size(), 1);

EXPECT\_EQ(newExtendedPolygons.size(), 1);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 8);

EXPECT\_EQ(newExtendedPolygons[0].*getNumberVertices*(), 9);

}

catch (const exception& exception)

{

FAIL();

}

}

}

namespace **IntersectorPolygonLineNamespace**{

TEST(***TestIntersectorPolygonLine***, ***TestCheckInConicCombination***)

{

vector<Point> polygonVertices;

polygonVertices.resize(5);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(4, 0);

polygonVertices[2] = Point(2, 2);

polygonVertices[3] = Point(4, 4);

polygonVertices[4] = Point(0, 4);

Polygon polygon(polygonVertices);

Point start(1, 0);

Point end(3, 4);

Segment segment(start, end);

Vector2d vect = segment.*getTangent*();

Vector2d v1(-2,2);

Vector2d v2(2,2);

int flag = -2;

Intersector1D1D intersector1d1d;

IntersectorPolygonLine intersector(intersector1d1d, polygon, segment);

try

{

EXPECT\_EQ(intersector.*checkInConicCombination*(vect, v1, v2, *flag*), true);

EXPECT\_EQ(flag, 1);

}

catch (const exception& exception)

{

FAIL();

}

Point start2(3, 4);

Point end2(1, 0);

Segment segment2(start2, end2);

Vector2d vect2 = segment2.*getTangent*();

int flag2 = -2;

try

{

EXPECT\_EQ(intersector.*checkInConicCombination*(vect2, v1, v2, *flag2*), true);

EXPECT\_EQ(flag2, -1);

}

catch (const exception& exception)

{

FAIL();

}

Point start3(0, 1);

Point end3(4, 3);

Segment segment3(start3, end3);

Vector2d vect3 = segment3.*getTangent*();

int flag3 = -2;

try

{

EXPECT\_EQ(intersector.*checkInConicCombination*(vect3, v1, v2, *flag3*), false);

EXPECT\_EQ(flag3, 0);

}

catch (const exception& exception)

{

FAIL();

}

}

}

#endif

POLYGON\_CUTTER\_CLASS\_TEST.HPP

#ifndef \_\_TEST\_POLYGONCUTTER\_H

#define \_\_TEST\_POLYGONCUTTER\_H

#include <gtest/gtest.h>

#include <gmock/gmock.h>

#include <gmock/gmock-matchers.h>

#include "PolygonCutter.hpp"

using namespace PolygonCutterNamespace;

using namespace testing;

using namespace std;

namespace **PolygonCutterTesting** {

class **MockIntersectorPolygonLine** : public IIntersectorPolygonLine {

private:

MOCK\_CONST\_METHOD4(**checkInConicCombination**, bool(const Vector2d &line, const Vector2d &v1, const Vector2d &v2, int &flag));

MOCK\_METHOD4(**extendPolygonIfEndpointsOnEdge**, void(int &numPoly, const int &firstIntersectionId, const int &secondIntersectionId, const bool &forwardDirection));

public:

MOCK\_METHOD0(**findIntersectionVertices**, const vector<Point>&());

MOCK\_METHOD0(**findPolygons**, const vector<Polygon>&());

MOCK\_METHOD0(**getExtendedPolygons**, const vector<Polygon>&());

MOCK\_METHOD0(**findNewPoints**, const vector<Point>&());

MOCK\_CONST\_METHOD0(**getStartFound**, bool());

MOCK\_CONST\_METHOD0(**getEndFound**, bool());

};

class **MockPolygon** : public IPolygon {

public:

MOCK\_CONST\_METHOD0(**getNumberVertices**, unsigned int());

MOCK\_CONST\_METHOD0(**getVertices**, const vector<Point>&());

MOCK\_METHOD2(**setVertexAtPosition**, void(const Point& vertex, const int& position));

MOCK\_METHOD1(**appendVertex**, void(const Point& vertex));

MOCK\_METHOD2(**insertVertexAtPosition**, void(const Point& vertex, const int& position));

MOCK\_CONST\_METHOD1(**getVertexAtPosition**, const Point&(const int& position));

MOCK\_CONST\_METHOD1(**nextVertexId**, int(unsigned int id));

MOCK\_CONST\_METHOD0(**getLastVertex**, const Point&());

MOCK\_CONST\_METHOD0(**ComputeArea**, double());

MOCK\_CONST\_METHOD0(**getEdges**, const vector<Segment>&());

MOCK\_METHOD2(**setVertexLabelAtPosition**, void(const int& position, int label));

MOCK\_CONST\_METHOD1(**getVertexLabelAtPosition**, int(const int& position));

MOCK\_METHOD1(**setVertexLabels**, void(const vector<unsigned int> &polygonVertices));

MOCK\_METHOD0(**setDefaultVertexLabels**, void());

MOCK\_METHOD1(**pointInPolygon**, bool(const Point& point));

MOCK\_CONST\_METHOD1(**isPolygonOnTheRightSide**, int(const Segment &line));

MOCK\_METHOD0(**buildUpPolygonEdges**, void());

};

TEST(***TestPolygonCutter***, ***CutPolygon7***) //poligono a Piacere 2 D' Auria

{

vector<Point> polygonVertices;

polygonVertices.resize(10);

polygonVertices[0] = Point(2, -2);

polygonVertices[1] = Point(0, -1);

polygonVertices[2] = Point(3, 1);

polygonVertices[3] = Point(0, 2);

polygonVertices[4] = Point(3, 2);

polygonVertices[5] = Point(3, 3);

polygonVertices[6] = Point(-1, 3);

polygonVertices[7] = Point(-3, 1);

polygonVertices[8] = Point(0, 0);

polygonVertices[9] = Point(-3, -2);

Point start(-4, -4);

Point end(4, 4);

Segment segment(start, end);

vector<Point> intersectionPoints;

intersectionPoints.resize(6);

intersectionPoints[0] = Point(-2, -2);

intersectionPoints[1] = Point(0, 0);

intersectionPoints[2] = Point(0, 0);

intersectionPoints[3] = Point(1.5, 1.5);

intersectionPoints[4] = Point(2, 2);

intersectionPoints[5] = Point(3, 3);

vector<Polygon> cuttedPolygons;

cuttedPolygons.resize(4);

cuttedPolygons[0].*appendVertex*(Point(2, -2));

cuttedPolygons[0].*appendVertex*(Point(0, -1));

cuttedPolygons[0].*appendVertex*(Point(3, 1));

cuttedPolygons[0].*appendVertex*(Point(1.5, 1.5));

cuttedPolygons[0].*appendVertex*(Point(0, 0));

cuttedPolygons[0].*appendVertex*(Point(-2, -2));

cuttedPolygons[1].*appendVertex*(Point(0, 2));

cuttedPolygons[1].*appendVertex*(Point(2, 2));

cuttedPolygons[1].*appendVertex*(Point(3, 3));

cuttedPolygons[1].*appendVertex*(Point(-1, 3));

cuttedPolygons[1].*appendVertex*(Point(-3, 1));

cuttedPolygons[1].*appendVertex*(Point(0, 0));

cuttedPolygons[1].*appendVertex*(Point(1.5, 1.5));

cuttedPolygons[2].*appendVertex*(Point(3, 2));

cuttedPolygons[2].*appendVertex*(Point(3, 3));

cuttedPolygons[2].*appendVertex*(Point(2, 2));

cuttedPolygons[3].*appendVertex*(Point(-3, -2));

cuttedPolygons[3].*appendVertex*(Point(-2, -2));

cuttedPolygons[3].*appendVertex*(Point(0, 0));

vector<Point> newPoints;

MockIntersectorPolygonLine intersector;

MockPolygon polygon;

PolygonCutter polygonCutter(intersector, polygon, segment);

EXPECT\_CALL(intersector, findIntersectionVertices)

.WillOnce(ReturnRef(*intersectionPoints*));

// on\_call perchè non ci interessa il suo test

ON\_CALL(intersector, findNewPoints)

.WillByDefault(ReturnRef(*newPoints*));

EXPECT\_CALL(intersector, findPolygons)

.WillOnce(ReturnRef(*cuttedPolygons*));

EXPECT\_CALL(polygon, getVertices)

.WillOnce(ReturnRef(*polygonVertices*));

polygonCutter.*cutPolygon*();

const vector<Polygon>& newPolygons = polygonCutter.*getCuttedPolygons*();

polygonCutter.*showPolygon*("showPolygon7.m");

try

{

EXPECT\_EQ(newPolygons.size(), 4);

}

catch (const exception& exception)

{

FAIL();

}

try

{

EXPECT\_EQ(newPolygons[0].*getNumberVertices*(), 6);

EXPECT\_EQ(newPolygons[1].*getNumberVertices*(), 7);

EXPECT\_EQ(newPolygons[2].*getNumberVertices*(), 3);

EXPECT\_EQ(newPolygons[3].*getNumberVertices*(), 3);

}

catch (const exception& exception)

{

FAIL();

}

}

}

#endif

REFERENCE\_ELEMENT\_CLASS\_TEST.HPP

#ifndef \_\_TEST\_REFERENCEELEMENTCLASS\_H

#define \_\_TEST\_REFERENCEELEMENTCLASS\_H

#include <gtest/gtest.h>

#include <gmock/gmock.h>

#include <gmock/gmock-matchers.h>

#include "iostream"

#include "IntersectorPolygonLine.hpp"

#include "ReferenceElement.hpp"

using namespace IntersectorPolygonLineNamespace;

using namespace testing;

using namespace std;

using namespace ReferenceElementNamespace;

namespace **ReferenceElementTesting** {

TEST(***TestReferenceElementClass***, ***ComputeBoundingBox***)

{

vector<Point> polygonVertices;

polygonVertices.resize(10);

polygonVertices[0] = Point(2, -2);

polygonVertices[1] = Point(0, -1);

polygonVertices[2] = Point(3, 1);

polygonVertices[3] = Point(0, 2);

polygonVertices[4] = Point(3, 2);

polygonVertices[5] = Point(3, 3);

polygonVertices[6] = Point(-1, 3);

polygonVertices[7] = Point(-3, 1);

polygonVertices[8] = Point(0, 0);

polygonVertices[9] = Point(-3, -2);

Point firstReferenceElementPoint = Point(Vector2d(-3, -2));

Point secondReferenceElementPoint =Point(Vector2d(3, -2));

Point thirdReferenceElementPoint = Point(Vector2d(3, 3));

Point fourthReferenceElementPoint = Point(Vector2d(-3, 3));

Polygon polygon = Polygon(polygonVertices);

ReferenceElement element;

element.*computeBoundingBox*(*polygon*);

try {

EXPECT\_EQ(firstReferenceElementPoint, element.*getBoundingBoxVertices*()[0]);

EXPECT\_EQ(secondReferenceElementPoint, element.*getBoundingBoxVertices*()[1]);

EXPECT\_EQ(thirdReferenceElementPoint,element.*getBoundingBoxVertices*()[2]);

EXPECT\_EQ(fourthReferenceElementPoint, element.*getBoundingBoxVertices*()[3]);

}

catch (const exception&)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***ComputeReferenceElement1***) //Con il Poligono di D'Auria

{

vector<Point> polygonVertices, vector1, vector2, vector3;

Point p0, p1, p2, p3, p4, p5, p6, p7, p8, p9;

polygonVertices.resize(10);

ReferenceElement referenceElement;

p0 = Point(2, -2);

p1 = Point(0, -1);

p2 = Point(3, 1);

p3 = Point(0, 2);

p4 = Point(3, 2);

p5 = Point(3, 3);

p6 = Point(-1, 3);

p7 = Point(-3, 1);

p8 = Point(0, 0);

p9 = Point(-3, -2);

polygonVertices[0] = p0;

polygonVertices[1] = p1;

polygonVertices[2] = p2;

polygonVertices[3] = p3;

polygonVertices[4] = p4;

polygonVertices[5] = p5;

polygonVertices[6] = p6;

polygonVertices[7] = p7;

polygonVertices[8] = p8;

polygonVertices[9] = p9;

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

referenceElement.showReferenceElement("showReferenceElement7.m");

try {

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[1], referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(0));

EXPECT\_EQ(p2, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(1));

EXPECT\_EQ(p1, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(2));

EXPECT\_EQ(p0, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(3));

EXPECT\_EQ(p4, referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(0));

EXPECT\_EQ(p3, referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(1));

EXPECT\_EQ(p2, referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(2));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[3], referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(0));

EXPECT\_EQ(p7, referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(1));

EXPECT\_EQ(p6, referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(2));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[0], referenceElement.*getReferenceElementPolygons*()[4].*getVertexAtPosition*(0));

EXPECT\_EQ(p8, referenceElement.*getReferenceElementPolygons*()[4].*getVertexAtPosition*(1));

EXPECT\_EQ(p7, referenceElement.*getReferenceElementPolygons*()[4].*getVertexAtPosition*(2));

EXPECT\_EQ(5, referenceElement.*getReferenceElementPolygons*().size());

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***ComputeReferenceElement2***)

{

vector<Point> polygonVertices;

Point p0, p1, p2, p3, p4, p5;

polygonVertices.resize(6);

ReferenceElement referenceElement;

p0 = Point(1.5, 1.0);

p1 = Point(5.6, 1.5);

p2 = Point(5.5, 4.8);

p3 = Point(4.0, 6.2);

p4 = Point(3.2, 4.2);

p5 = Point(1.0, 4.0);

polygonVertices[0] = p0;

polygonVertices[1] = p1;

polygonVertices[2] = p2;

polygonVertices[3] = p3;

polygonVertices[4] = p4;

polygonVertices[5] = p5;

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

try {

EXPECT\_EQ(5, referenceElement.*getReferenceElementPolygons*().size());

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[1], referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(0));

EXPECT\_EQ(p1, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(1));

EXPECT\_EQ(p0, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(2));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[2], referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(0));

EXPECT\_EQ(p3, referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(1));

EXPECT\_EQ(p2, referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(2));

EXPECT\_EQ(p1, referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(3));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[3], referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(0));

EXPECT\_EQ(p5, referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(1));

EXPECT\_EQ(p4, referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(2));

EXPECT\_EQ(p3, referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(3));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[0], referenceElement.*getReferenceElementPolygons*()[4].*getVertexAtPosition*(0));

EXPECT\_EQ(p0, referenceElement.*getReferenceElementPolygons*()[4].*getVertexAtPosition*(1));

EXPECT\_EQ(p5, referenceElement.*getReferenceElementPolygons*()[4].*getVertexAtPosition*(2));

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***ComputeReferenceElement3***)

{

vector<Point> polygonVertices;

Point p0, p1, p2, p3;

polygonVertices.resize(4);

ReferenceElement referenceElement;

p0 = Point(0, 0);

p1 = Point(4, 0);

p2 = Point(4, 4);

p3 = Point(0, 4);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(4, 0);

polygonVertices[2] = Point(4, 4);

polygonVertices[3] = Point(0, 4);

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

try {

EXPECT\_EQ(1, referenceElement.*getReferenceElementPolygons*().size());

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***ComputeReferenceElement4***)

{

vector<Point> polygonVertices;

Point p0, p1, p2, p3, p4;

polygonVertices.resize(5);

ReferenceElement referenceElement;

p0 = Point(2.5, 1.0);

p1 = Point(4.0, 2.1);

p2 = Point(3.4, 4.2);

p3 = Point(1.6, 4.2);

p4 = Point(1.0, 2.1);

polygonVertices[0] = p0;

polygonVertices[1] = p1;

polygonVertices[2] = p2;

polygonVertices[3] = p3;

polygonVertices[4] = p4;

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

try {

EXPECT\_EQ(5, referenceElement.*getReferenceElementPolygons*().size());

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[1], referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(0));

EXPECT\_EQ(p1, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(1));

EXPECT\_EQ(p0, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(2));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[2], referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(0));

EXPECT\_EQ(p2, referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(1));

EXPECT\_EQ(p1, referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(2));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[3], referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(0));

EXPECT\_EQ(p4, referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(1));

EXPECT\_EQ(p3, referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(2));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[0], referenceElement.*getReferenceElementPolygons*()[4].*getVertexAtPosition*(0));

EXPECT\_EQ(p0, referenceElement.*getReferenceElementPolygons*()[4].*getVertexAtPosition*(1));

EXPECT\_EQ(p4, referenceElement.*getReferenceElementPolygons*()[4].*getVertexAtPosition*(2));

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***ComputeReferenceElement5***)

{

vector<Point> polygonVertices;

Point p0, p1, p2, p3, p4, p5, p6, p7;

polygonVertices.resize(8);

ReferenceElement referenceElement;

p0 = Point(0, 0);

p1 = Point(8, 0);

p2 = Point(4.64, 1.5);

p3 = Point(8.93, 4.48);

p4 = Point(2.18, 5.36);

p5 = Point(5.52, 9.22);

p6 = Point(0, 5);

p7 = Point(6, 4);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(8, 0);

polygonVertices[2] = Point(4.64, 1.5);

polygonVertices[3] = Point(8.93, 4.48);

polygonVertices[4] = Point(2.18, 5.36);

polygonVertices[5] = Point(5.52, 9.22);

polygonVertices[6] = Point(0, 5);

polygonVertices[7] = Point(6, 4);

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

try {

EXPECT\_EQ(5, referenceElement.*getReferenceElementPolygons*().size());

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[1], referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(0));

EXPECT\_EQ(p3, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(1));

EXPECT\_EQ(p2, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(2));

EXPECT\_EQ(p1, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(3));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[2], referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(0));

EXPECT\_EQ(p5, referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(1));

EXPECT\_EQ(p4, referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(2));

EXPECT\_EQ(p3, referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(3));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[3], referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(0));

EXPECT\_EQ(p6, referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(1));

EXPECT\_EQ(p5, referenceElement.*getReferenceElementPolygons*()[3].*getVertexAtPosition*(2));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[0], referenceElement.*getReferenceElementPolygons*()[4].*getVertexAtPosition*(0));

EXPECT\_EQ(p7, referenceElement.*getReferenceElementPolygons*()[4].*getVertexAtPosition*(1));

EXPECT\_EQ(p6, referenceElement.*getReferenceElementPolygons*()[4].*getVertexAtPosition*(2));

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***ComputeReferenceElement6***)

{

vector<Point> polygonVertices;

Point p0, p1, p2;

polygonVertices.resize(3);

ReferenceElement referenceElement;

p0 = Point(0, 0);

p1 = Point(1, 0);

p2 = Point(1,1);

polygonVertices[0] = p0;

polygonVertices[1] = p1;

polygonVertices[2] = p2;

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

try {

EXPECT\_EQ(2, referenceElement.*getReferenceElementPolygons*().size());

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[3], referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(0));

EXPECT\_EQ(p0, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(1));

EXPECT\_EQ(p2, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(2));

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***ComputeReferenceElement8***)

{

vector<Point> polygonVertices;

Point p0, p1, p2;

polygonVertices.resize(3);

ReferenceElement referenceElement;

p0 = Point(0, 0);

p1 = Point(1, 0);

p2 = Point(0, 1);

polygonVertices[0] = p0;

polygonVertices[1] = p1;

polygonVertices[2] = p2;

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

try {

EXPECT\_EQ(2, referenceElement.*getReferenceElementPolygons*().size());

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[2], referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(0));

EXPECT\_EQ(p2, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(1));

EXPECT\_EQ(p1, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(2));

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***ComputeReferenceElement9***)

{

vector<Point> polygonVertices;

Point p0, p1, p2;

polygonVertices.resize(3);

ReferenceElement referenceElement;

p0 = Point(1, 1);

p1 = Point(0, 1);

p2 = Point(0.5, 0);

polygonVertices[0] = p0;

polygonVertices[1] = p1;

polygonVertices[2] = p2;

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

try {

EXPECT\_EQ(3, referenceElement.*getReferenceElementPolygons*().size());

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[1], referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(0));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[2], referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(1));

EXPECT\_EQ(p2, referenceElement.*getReferenceElementPolygons*()[1].*getVertexAtPosition*(2));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[0], referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(0));

EXPECT\_EQ(p2, referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(1));

EXPECT\_EQ(referenceElement.*getBoundingBoxVertices*()[3], referenceElement.*getReferenceElementPolygons*()[2].*getVertexAtPosition*(2));

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***findBoundaryPoints***)

{

vector<Point> polygonVertices;

Point p0, p1, p2, p3;

polygonVertices.resize(4);

ReferenceElement referenceElement;

p0 = Point(0, 2);

p1 = Point(1, 0);

p2 = Point(4, 3);

p3 = Point(3, 4);

polygonVertices[0] = p0;

polygonVertices[1] = p1;

polygonVertices[2] = p2;

polygonVertices[3] = p3;

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

referenceElement.*findBoundaryVertices*();

try {

EXPECT\_EQ(5, referenceElement.*getReferenceElementPolygons*().size());

EXPECT\_EQ(1, referenceElement.*getMapSouthXCoordinates*().size());

EXPECT\_EQ(1, referenceElement.*getMapNorthXCoordinates*().size());

EXPECT\_EQ(1, referenceElement.*getMapEastYCoordinates*().size());

EXPECT\_EQ(1, referenceElement.*getMapWestYCoordinates*().size());

map<double, Point>::const\_iterator it ;

it = referenceElement.*getMapSouthXCoordinates*().begin();

EXPECT\_EQ(p1, it->second);

EXPECT\_EQ(p1.getCoordinates().x(), it->first);

it = referenceElement.*getMapNorthXCoordinates*().begin();

EXPECT\_EQ(p3, it->second);

EXPECT\_EQ(p3.getCoordinates().x(), it->first);

it = referenceElement.*getMapEastYCoordinates*().begin();

EXPECT\_EQ(p2, it->second);

EXPECT\_EQ(p2.getCoordinates().y(), it->first);

it = referenceElement.*getMapWestYCoordinates*().begin();

EXPECT\_EQ(p0, it->second);

EXPECT\_EQ(p0.getCoordinates().y(), it->first);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***findBoundaryPoints2***)

{

vector<Point> polygonVertices;

Point p0, p1, p2, p3, p4, p5, p6, p7, p8;

polygonVertices.resize(9);

ReferenceElement referenceElement;

p0 = Point(0, 0);

p1 = Point(1, 2);

p2 = Point(2, 0);

p3 = Point(3, 2);

p4 = Point(4, 0);

p5 = Point(5, 0);

p6 = Point(5, 3);

p7 = Point(2.5, 4);

p8 = Point(0, 3);

polygonVertices[0] = p0;

polygonVertices[1] = p1;

polygonVertices[2] = p2;

polygonVertices[3] = p3;

polygonVertices[4] = p4;

polygonVertices[5] = p5;

polygonVertices[6] = p6;

polygonVertices[7] = p7;

polygonVertices[8] = p8;

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

referenceElement.*findBoundaryVertices*();

try {

EXPECT\_EQ(5, referenceElement.*getReferenceElementPolygons*().size());

EXPECT\_EQ(2, referenceElement.*getMapSouthXCoordinates*().size());

EXPECT\_EQ(1, referenceElement.*getMapNorthXCoordinates*().size());

EXPECT\_EQ(1, referenceElement.*getMapEastYCoordinates*().size());

EXPECT\_EQ(1, referenceElement.*getMapWestYCoordinates*().size());

map<double, Point>::const\_iterator it ;

it = referenceElement.*getMapSouthXCoordinates*().begin();

EXPECT\_EQ(p2, it->second);

EXPECT\_EQ(2, it->first);

it++;

EXPECT\_EQ(p4, it->second);

EXPECT\_EQ(4, it->first);

it = referenceElement.*getMapNorthXCoordinates*().begin();

EXPECT\_EQ(p7, it->second);

EXPECT\_EQ(2.5, it->first);

it = referenceElement.*getMapEastYCoordinates*().begin();

EXPECT\_EQ(p6, it->second);

EXPECT\_EQ(3, it->first);

it = referenceElement.*getMapWestYCoordinates*().begin();

EXPECT\_EQ(p8, it->second);

EXPECT\_EQ(3, it->first);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***MakeConforming***)//quadrilatero

{

vector<Point> polygonVertices;

Point p0, p1, p2, p3;

polygonVertices.resize(4);

ReferenceElement referenceElement;

p0 = Point(0, 2);

p1 = Point(1, 0);

p2 = Point(4, 3);

p3 = Point(3, 4);

polygonVertices[0] = p0;

polygonVertices[1] = p1;

polygonVertices[2] = p2;

polygonVertices[3] = p3;

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

referenceElement.*findBoundaryVertices*();

referenceElement.*makeConforming*();

try {

EXPECT\_EQ(5, referenceElement.*getConformingReferenceElementPolygons*().size());

EXPECT\_EQ(4, referenceElement.*getConformingReferenceElementPolygons*()[0].*getNumberVertices*());

EXPECT\_EQ(5, referenceElement.*getConformingReferenceElementPolygons*()[1].*getNumberVertices*());

EXPECT\_EQ(3, referenceElement.*getConformingReferenceElementPolygons*()[2].*getNumberVertices*());

EXPECT\_EQ(5, referenceElement.*getConformingReferenceElementPolygons*()[3].*getNumberVertices*());

EXPECT\_EQ(3, referenceElement.*getConformingReferenceElementPolygons*()[4].*getNumberVertices*());

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***MakeConforming2a***)

{

vector<Point> polygonVertices;

Point p0, p1, p2, p3, p4, p5, p6, p7, p8;

polygonVertices.resize(9);

ReferenceElement referenceElement;

p0 = Point(0, 0);

p1 = Point(1, 2);

p2 = Point(2, 0);

p3 = Point(3, 2);

p4 = Point(4, 0);

p5 = Point(5, 0);

p6 = Point(5, 3);

p7 = Point(2.5, 4);

p8 = Point(0, 3);

polygonVertices[0] = p0;

polygonVertices[1] = p1;

polygonVertices[2] = p2;

polygonVertices[3] = p3;

polygonVertices[4] = p4;

polygonVertices[5] = p5;

polygonVertices[6] = p6;

polygonVertices[7] = p7;

polygonVertices[8] = p8;

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

referenceElement.*findBoundaryVertices*();

referenceElement.*makeConforming*();

try {

EXPECT\_EQ(5, referenceElement.*getConformingReferenceElementPolygons*().size());

EXPECT\_EQ(9, referenceElement.*getConformingReferenceElementPolygons*()[0].*getNumberVertices*());

EXPECT\_EQ(3, referenceElement.*getConformingReferenceElementPolygons*()[1].*getNumberVertices*());

EXPECT\_EQ(4, referenceElement.*getConformingReferenceElementPolygons*()[2].*getNumberVertices*());

EXPECT\_EQ(4, referenceElement.*getConformingReferenceElementPolygons*()[3].*getNumberVertices*());

EXPECT\_EQ(4, referenceElement.*getConformingReferenceElementPolygons*()[4].*getNumberVertices*());

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***MakeConforming2b***)

{

vector<Point> polygonVertices;

Point p0, p1, p2, p3, p4, p5, p6, p7, p8;

polygonVertices.resize(9);

ReferenceElement referenceElement;

p0 = Point(0, 0);

p1 = Point(1, 2);

p2 = Point(2, 0);

p3 = Point(3, 2);

p4 = Point(4, 0);

p5 = Point(5, 0);

p6 = Point(5, 3);

p7 = Point(2, 4);

p8 = Point(0, 3);

polygonVertices[0] = p0;

polygonVertices[1] = p1;

polygonVertices[2] = p2;

polygonVertices[3] = p3;

polygonVertices[4] = p4;

polygonVertices[5] = p5;

polygonVertices[6] = p6;

polygonVertices[7] = p7;

polygonVertices[8] = p8;

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

referenceElement.*findBoundaryVertices*();

referenceElement.*makeConforming*();

try {

EXPECT\_EQ(5, referenceElement.*getConformingReferenceElementPolygons*().size());

EXPECT\_EQ(9, referenceElement.*getConformingReferenceElementPolygons*()[0].*getNumberVertices*());

EXPECT\_EQ(3, referenceElement.*getConformingReferenceElementPolygons*()[1].*getNumberVertices*());

EXPECT\_EQ(3, referenceElement.*getConformingReferenceElementPolygons*()[2].*getNumberVertices*());

EXPECT\_EQ(4, referenceElement.*getConformingReferenceElementPolygons*()[3].*getNumberVertices*());

EXPECT\_EQ(3, referenceElement.*getConformingReferenceElementPolygons*()[4].*getNumberVertices*());

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestReferenceElementClass***, ***MakeConforming3***)//con il Poligono di D'Auria

{

vector<Point> polygonVertices, vector1, vector2, vector3;

Point p0, p1, p2, p3, p4, p5, p6, p7, p8, p9;

polygonVertices.resize(10);

ReferenceElement referenceElement;

p0 = Point(2, -2);

p1 = Point(0, -1);

p2 = Point(3, 1);

p3 = Point(0, 2);

p4 = Point(3, 2);

p5 = Point(3, 3);

p6 = Point(-1, 3);

p7 = Point(-3, 1);

p8 = Point(0, 0);

p9 = Point(-3, -2);

polygonVertices[0] = p0;

polygonVertices[1] = p1;

polygonVertices[2] = p2;

polygonVertices[3] = p3;

polygonVertices[4] = p4;

polygonVertices[5] = p5;

polygonVertices[6] = p6;

polygonVertices[7] = p7;

polygonVertices[8] = p8;

polygonVertices[9] = p9;

Polygon polygon = Polygon(polygonVertices);

referenceElement.*computeBoundingBox*(*polygon*);

referenceElement.*computeReferenceElement*(*polygon*);

referenceElement.*findBoundaryVertices*();

referenceElement.*makeConforming*();

try {

EXPECT\_EQ(5, referenceElement.*getConformingReferenceElementPolygons*().size());

EXPECT\_EQ(12, referenceElement.*getConformingReferenceElementPolygons*()[0].*getNumberVertices*());

EXPECT\_EQ(4, referenceElement.*getConformingReferenceElementPolygons*()[1].*getNumberVertices*());

EXPECT\_EQ(3, referenceElement.*getConformingReferenceElementPolygons*()[2].*getNumberVertices*());

EXPECT\_EQ(4, referenceElement.*getConformingReferenceElementPolygons*()[3].*getNumberVertices*());

EXPECT\_EQ(3, referenceElement.*getConformingReferenceElementPolygons*()[4].*getNumberVertices*());

}

catch (const exception& ex)

{

FAIL();

}

}

}

#endif

MESH\_CLASS\_TEST.HPP

#ifndef \_\_TEST\_MESHCLASS\_H

#define \_\_TEST\_MESHCLASS\_H

#include <gtest/gtest.h>

#include <gmock/gmock.h>

#include <gmock/gmock-matchers.h>

#include "iostream"

#include "IntersectorPolygonLine.hpp"

#include "ReferenceElement.hpp" //eliminare quello che non serva

#include "Mesh.hpp"

using namespace testing;

using namespace std;

using namespace ReferenceElementNamespace;

using namespace MeshNamespace;

using namespace IntersectorPolygonLineNamespace;//eliminare quello che non serva

namespace **MeshTesting** {

TEST(***TestMeshClass***, ***TestMeshConstructor***)

{

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 2);

polygonVertices[1] = Point(1, 0);

polygonVertices[2] = Point(4, 3);

polygonVertices[3] = Point(3, 4);

Polygon basePolygon(polygonVertices);

Mesh myMesh(*basePolygon*);

try

{

double BoundingBoxLength = myMesh.*getBoundingBoxLength*();

double BoundingBoxHeight = myMesh.*getBoundingBoxHeight*();

EXPECT\_EQ(myMesh.*getBoundingBoxLength*(), 4);

EXPECT\_EQ(myMesh.*getBoundingBoxHeight*(), 4);

myMesh.*setRectangularDomain*(4\*BoundingBoxLength + 1, 3\*BoundingBoxHeight+1);

EXPECT\_EQ(myMesh.*getDomainBoundingBoxLength*(), 17);

EXPECT\_EQ(myMesh.*getDomainBoundingBoxHeight*(), 13);

EXPECT\_EQ(myMesh.*getDomain*().*getEdges*()[0].*getTangent*().norm(), 17);

EXPECT\_EQ(myMesh.*getDomain*().*getEdges*()[1].*getTangent*().norm(), 13);

EXPECT\_EQ(myMesh.*getDomain*().*getEdges*()[2].*getTangent*().norm(), 17);

EXPECT\_EQ(myMesh.*getDomain*().*getEdges*()[3].*getTangent*().norm(), 13);

EXPECT\_EQ(myMesh.*getDomainBoundingBoxHeight*(), 13);

EXPECT\_EQ(myMesh.*getMaxNumRefElementsPerRow*(), 5);

EXPECT\_EQ(myMesh.*getMaxNumRefElementsPerColumn*(), 4);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestReferenceElementInConvexDomain***)

{

ReferenceElement referenceElement;

Mesh mesh = Mesh();

vector<Point> polygonVertices;

polygonVertices.resize(5);

polygonVertices[0] = Point(2.5, 1.0);

polygonVertices[1] = Point(4.0, 2.1);

polygonVertices[2] = Point(3.4, 4.2);

polygonVertices[3] = Point(1.6, 4.2);

polygonVertices[4] = Point(1.0, 2.1);

Polygon polygon(polygonVertices);

referenceElement.*computeReferenceElement*(*polygon*);

try {

EXPECT\_TRUE(mesh.*referenceElementInConvexDomain*(*referenceElement*));

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestReferenceElementInConvexDomain2***)

{

ReferenceElement referenceElement;

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 0);

polygonVertices[1] = Point(4, 0);

polygonVertices[2] = Point(4, 4);

polygonVertices[3] = Point(0, 4);

Polygon polygon(polygonVertices);

Mesh mesh = Mesh(*polygon*);

mesh.*setRectangularDomain*(8,8);

try {

EXPECT\_TRUE(mesh.*referenceElementInConvexDomain*(*mesh.getReferenceElement()*));

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestTranslatePolygon***)

{

Mesh myMesh;

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 2);

polygonVertices[1] = Point(1, 0);

polygonVertices[2] = Point(4, 3);

polygonVertices[3] = Point(3, 4);

Polygon polygon(polygonVertices);

Point translatedPoint0= Point(7, 7);

Point translatedPoint1 = Point(8, 5);

Point translatedPoint2 = Point(11, 8);

Point translatedPoint3 = Point(10, 9);

try {

Polygon translatedPolygonByUsingTranslatePolygon = myMesh.*TranslatePolygon*(polygon, 7 , 5);

EXPECT\_EQ( translatedPoint0, translatedPolygonByUsingTranslatePolygon.*getVertexAtPosition*(0));

EXPECT\_EQ( translatedPoint1, translatedPolygonByUsingTranslatePolygon.*getVertexAtPosition*(1));

EXPECT\_EQ( translatedPoint2, translatedPolygonByUsingTranslatePolygon.*getVertexAtPosition*(2));

EXPECT\_EQ( translatedPoint3, translatedPolygonByUsingTranslatePolygon.*getVertexAtPosition*(3));

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestTranslateReferenceElement***)

{

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 2);

polygonVertices[1] = Point(1, 0);

polygonVertices[2] = Point(4, 3);

polygonVertices[3] = Point(3, 4);

Polygon polygon(polygonVertices);

Mesh myMesh(*polygon*);

Point translatedPoint0= Point(7, 7);

Point translatedPoint1 = Point(8, 5);

Point translatedPoint2 = Point(11, 8);

Point translatedPoint3 = Point(10, 9);

Point translatedPointA= Point(7, 5);

Point translatedPointB= Point(11, 5);

Point translatedPointC= Point(11, 9);

Point translatedPointD= Point(7, 9);

Point translatedPointW= Point(7, 8);

Point translatedPointS= Point(10, 5);

Point translatedPointE= Point(11, 7);

Point translatedPointN= Point(8, 9);

try {

ReferenceElement translatedReferenceElement = myMesh.*TranslateConformingReferenceElement*(7,5);

EXPECT\_EQ( translatedReferenceElement.*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ( translatedReferenceElement.*getConformingReferenceElementPolygons*()[0].*getEdges*().size(), 4);

EXPECT\_EQ( translatedReferenceElement.*getConformingReferenceElementPolygons*()[1].*getEdges*().size(), 5);

EXPECT\_EQ( translatedReferenceElement.*getConformingReferenceElementPolygons*()[2].*getEdges*().size(), 3);

EXPECT\_EQ( translatedReferenceElement.*getConformingReferenceElementPolygons*()[3].*getEdges*().size(), 5);

EXPECT\_EQ( translatedReferenceElement.*getConformingReferenceElementPolygons*()[4].*getEdges*().size(), 3);

EXPECT\_EQ( translatedPoint0, translatedReferenceElement.*getConformingReferenceElementPolygons*()[0].*getVertexAtPosition*(0));

EXPECT\_EQ( translatedPoint1, translatedReferenceElement.*getConformingReferenceElementPolygons*()[0].*getVertexAtPosition*(1));

EXPECT\_EQ( translatedPoint2, translatedReferenceElement.*getConformingReferenceElementPolygons*()[0].*getVertexAtPosition*(2));

EXPECT\_EQ( translatedPoint3, translatedReferenceElement.*getConformingReferenceElementPolygons*()[0].*getVertexAtPosition*(3));

EXPECT\_EQ( translatedPointS, translatedReferenceElement.*getConformingReferenceElementPolygons*()[1].*getVertexAtPosition*(0));

EXPECT\_EQ( translatedPointB, translatedReferenceElement.*getConformingReferenceElementPolygons*()[1].*getVertexAtPosition*(1));

EXPECT\_EQ( translatedPointE, translatedReferenceElement.*getConformingReferenceElementPolygons*()[1].*getVertexAtPosition*(2));

EXPECT\_EQ( translatedPoint2, translatedReferenceElement.*getConformingReferenceElementPolygons*()[1].*getVertexAtPosition*(3));

EXPECT\_EQ( translatedPoint1, translatedReferenceElement.*getConformingReferenceElementPolygons*()[1].*getVertexAtPosition*(4));

EXPECT\_EQ( translatedPointC, translatedReferenceElement.*getConformingReferenceElementPolygons*()[2].*getVertexAtPosition*(0));

EXPECT\_EQ( translatedPoint3, translatedReferenceElement.*getConformingReferenceElementPolygons*()[2].*getVertexAtPosition*(1));

EXPECT\_EQ( translatedPoint2, translatedReferenceElement.*getConformingReferenceElementPolygons*()[2].*getVertexAtPosition*(2));

EXPECT\_EQ( translatedPointN, translatedReferenceElement.*getConformingReferenceElementPolygons*()[3].*getVertexAtPosition*(0));

EXPECT\_EQ( translatedPointD, translatedReferenceElement.*getConformingReferenceElementPolygons*()[3].*getVertexAtPosition*(1));

EXPECT\_EQ( translatedPointW, translatedReferenceElement.*getConformingReferenceElementPolygons*()[3].*getVertexAtPosition*(2));

EXPECT\_EQ( translatedPoint0, translatedReferenceElement.*getConformingReferenceElementPolygons*()[3].*getVertexAtPosition*(3));

EXPECT\_EQ( translatedPoint3, translatedReferenceElement.*getConformingReferenceElementPolygons*()[3].*getVertexAtPosition*(4));

EXPECT\_EQ( translatedPointA, translatedReferenceElement.*getConformingReferenceElementPolygons*()[4].*getVertexAtPosition*(0));

EXPECT\_EQ( translatedPoint1, translatedReferenceElement.*getConformingReferenceElementPolygons*()[4].*getVertexAtPosition*(1));

EXPECT\_EQ( translatedPoint0, translatedReferenceElement.*getConformingReferenceElementPolygons*()[4].*getVertexAtPosition*(2));

EXPECT\_EQ( translatedPointA, translatedReferenceElement.getBoundingBox().*getVertexAtPosition*(0));

EXPECT\_EQ( translatedPointB, translatedReferenceElement.getBoundingBox().*getVertexAtPosition*(1));

EXPECT\_EQ( translatedPointC, translatedReferenceElement.getBoundingBox().*getVertexAtPosition*(2));

EXPECT\_EQ( translatedPointD, translatedReferenceElement.getBoundingBox().*getVertexAtPosition*(3));

EXPECT\_EQ( translatedReferenceElement.getBoundingBox().*getEdges*().size(), 4);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestCutAndDiscardPolygonsOnRighSide***)

{

Mesh myMesh;

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 2);

polygonVertices[1] = Point(1, 0);

polygonVertices[2] = Point(4, 3);

polygonVertices[3] = Point(3, 4);

Point start1(1, -1);

Point end1(1, 5);

Segment line1(start1, end1);

Point start2(0, 1);

Point end2(5, 1);

Segment line2(start2, end2);

Point start3(0, 4);

Point end3(1, 1);

Segment line3(start3, end3);

Polygon polygon(polygonVertices);

try {

ReferenceElement refElem;

refElem.*computeBoundingBox*(*polygon*);

refElem.*computeReferenceElement*( *polygon*);

refElem.*findBoundaryVertices*();

refElem.*makeConforming*();

vector<Polygon> vectorPolygons;

vectorPolygons = refElem.*getConformingReferenceElementPolygons*();

EXPECT\_EQ( vectorPolygons.size(), 5);

//vertical cut

vector<Polygon> vectorPolygonsAfterCutAndDiscard;

myMesh.*cutAndDiscardPolygonsOnRightSide*(*refElem*, *line1*);

vectorPolygonsAfterCutAndDiscard = refElem.*getConformingReferenceElementPolygons*();

EXPECT\_EQ( vectorPolygonsAfterCutAndDiscard.size(), 3);

EXPECT\_EQ( vectorPolygonsAfterCutAndDiscard[0].*getNumberVertices*(), 3);

EXPECT\_EQ( vectorPolygonsAfterCutAndDiscard[1].*getNumberVertices*(), 5);

EXPECT\_EQ( vectorPolygonsAfterCutAndDiscard[2].*getNumberVertices*(), 3);

EXPECT\_EQ(vectorPolygonsAfterCutAndDiscard[0].*ComputeArea*() + vectorPolygonsAfterCutAndDiscard[1].*ComputeArea*()

+ vectorPolygonsAfterCutAndDiscard[2].*ComputeArea*(), 4 );

//horizontal cut after the previous cut

vector<Polygon> vectorPolygonsAfterCutAndDiscard2;

myMesh.*cutAndDiscardPolygonsOnRightSide*(*refElem*, *line2*);

vectorPolygonsAfterCutAndDiscard2 = refElem.*getConformingReferenceElementPolygons*();

EXPECT\_EQ( vectorPolygonsAfterCutAndDiscard2.size(), 3);

EXPECT\_EQ( vectorPolygonsAfterCutAndDiscard2[0].*getNumberVertices*(), 4);

EXPECT\_EQ( vectorPolygonsAfterCutAndDiscard2[1].*getNumberVertices*(), 5);

EXPECT\_EQ( vectorPolygonsAfterCutAndDiscard2[2].*getNumberVertices*(), 3);

EXPECT\_EQ(vectorPolygonsAfterCutAndDiscard2[0].*ComputeArea*() + vectorPolygonsAfterCutAndDiscard2[1].*ComputeArea*()

+ vectorPolygonsAfterCutAndDiscard2[2].*ComputeArea*(), 3 );

//slanted cut after previous cut

vector<Polygon> vectorPolygonsAfterCutAndDiscard3;

myMesh.*cutAndDiscardPolygonsOnRightSide*(*refElem*, *line3*);

vectorPolygonsAfterCutAndDiscard3 = refElem.*getConformingReferenceElementPolygons*();

EXPECT\_EQ( vectorPolygonsAfterCutAndDiscard3.size(), 2);

EXPECT\_EQ( vectorPolygonsAfterCutAndDiscard3[0].*getNumberVertices*(), 3);

EXPECT\_EQ( vectorPolygonsAfterCutAndDiscard3[1].*getNumberVertices*(), 4);

EXPECT\_EQ(vectorPolygonsAfterCutAndDiscard3[0].*ComputeArea*() + vectorPolygonsAfterCutAndDiscard3[1].*ComputeArea*(), 1.5 );

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestCreateMesh\_RectangularDomain***)

{

//for rectangular mesh of (4+1)x(3+1) reference elements

// the base polygon is a quadrilateral

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 2);

polygonVertices[1] = Point(1, 0);

polygonVertices[2] = Point(4, 3);

polygonVertices[3] = Point(3, 4);

Polygon basePolygon(polygonVertices);

Mesh myMesh(*basePolygon*);

double BoundingBoxLength = myMesh.*getBoundingBoxLength*();

double BoundingBoxHeight = myMesh.*getBoundingBoxHeight*();

myMesh.*setRectangularDomain*(4\*BoundingBoxLength + 1, 3\*BoundingBoxHeight+1);

try

{

myMesh.*createMesh*();

vector<ReferenceElement> myMeshCells = myMesh.*getMeshCells*();

myMesh.*showMesh*();

EXPECT\_EQ(myMeshCells.size(),20);

EXPECT\_EQ(myMeshCells[0].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[1].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[2].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[3].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[4].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[5].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[6].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[7].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[8].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[9].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[10].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[11].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[12].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[13].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[14].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[15].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[16].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[17].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[18].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[19].*getConformingReferenceElementPolygons*().size(), 2);

double totalArea = 0;

double domainArea = myMesh.*getDomain*().*ComputeArea*();

for(unsigned int i = 0; i < myMeshCells.size(); i++)

for (unsigned int j = 0; j < myMeshCells[i].*getConformingReferenceElementPolygons*().size(); j++)

totalArea += myMeshCells[i].*getConformingReferenceElementPolygons*()[j].*ComputeArea*();

EXPECT\_DOUBLE\_EQ(totalArea, domainArea);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestCreateMesh\_RectangularDomain2***)

{

//for rectangular mesh of (4+1)x(3+1) reference elements

// the base polygon is the concave hexagon from project 1

vector<Point> polygonVertices;

polygonVertices.resize(6);

polygonVertices[0] = Point(1.5, 1.0);

polygonVertices[1] = Point(5.6, 1.5);

polygonVertices[2] = Point(5.5, 4.8);

polygonVertices[3] = Point(4.0, 6.2);

polygonVertices[4] = Point(3.2, 4.2);

polygonVertices[5] = Point(1.0, 4.0);

Polygon basePolygon(polygonVertices);

Mesh myMesh(*basePolygon*);

double BoundingBoxLength = myMesh.*getBoundingBoxLength*();

double BoundingBoxHeight = myMesh.*getBoundingBoxHeight*();

myMesh.*setRectangularDomain*(4\*BoundingBoxLength + 1, 3\*BoundingBoxHeight+1);

try

{

myMesh.*createMesh*();

vector<ReferenceElement> myMeshCells = myMesh.*getMeshCells*();

myMesh.*showMesh*("showMesh2.m");

EXPECT\_EQ(myMeshCells.size(),20);

EXPECT\_EQ(myMeshCells[0].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[1].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[2].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[3].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[4].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[5].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[6].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[7].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[8].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[9].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[10].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[11].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[12].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[13].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[14].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[15].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[16].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[17].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[18].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[19].*getConformingReferenceElementPolygons*().size(), 3);

double totalArea = 0;

double domainArea = myMesh.*getDomain*().*ComputeArea*();

for(unsigned int i = 0; i < myMeshCells.size(); i++)

for (unsigned int j = 0; j < myMeshCells[i].*getConformingReferenceElementPolygons*().size(); j++)

totalArea += myMeshCells[i].*getConformingReferenceElementPolygons*()[j].*ComputeArea*();

EXPECT\_DOUBLE\_EQ(totalArea, domainArea);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestCreateMesh\_RectangularDomain3***)

{

//for rectangular mesh of (4+1)x(3+1) reference elements

//the base polygon is D'Auria's polygon with the shape of a sigma

vector<Point> polygonVertices;

polygonVertices.resize(10);

polygonVertices[0] = Point(2, -2);

polygonVertices[1] = Point(0, -1);

polygonVertices[2] = Point(3, 1);

polygonVertices[3] = Point(0, 2);

polygonVertices[4] = Point(3, 2);

polygonVertices[5] = Point(3, 3);

polygonVertices[6] = Point(-1, 3);

polygonVertices[7] = Point(-3, 1);

polygonVertices[8] = Point(0, 0);

polygonVertices[9] = Point(-3, -2);

Polygon basePolygon(polygonVertices);

Mesh myMesh(*basePolygon*);

double BoundingBoxLength = myMesh.*getBoundingBoxLength*();

double BoundingBoxHeight = myMesh.*getBoundingBoxHeight*();

myMesh.*setRectangularDomain*(4\*BoundingBoxLength + 1, 3\*BoundingBoxHeight+1);

try

{

myMesh.*createMesh*();

vector<ReferenceElement> myMeshCells = myMesh.*getMeshCells*();

myMesh.*showMesh*("showMesh3.m");

EXPECT\_EQ(myMeshCells.size(),20);

EXPECT\_EQ(myMeshCells[0].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[1].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[2].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[3].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[4].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[5].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[6].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[7].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[8].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[9].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[10].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[11].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[12].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[13].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[14].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[15].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[16].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[17].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[18].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[19].*getConformingReferenceElementPolygons*().size(), 2);

double totalArea = 0;

double domainArea = myMesh.*getDomain*().*ComputeArea*();

for(unsigned int i = 0; i < myMeshCells.size(); i++)

for (unsigned int j = 0; j < myMeshCells[i].*getConformingReferenceElementPolygons*().size(); j++)

totalArea += myMeshCells[i].*getConformingReferenceElementPolygons*()[j].*ComputeArea*();

EXPECT\_DOUBLE\_EQ(totalArea, domainArea);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestCreateMesh\_RectangularDomain4***)

{

//for rectangular mesh of (4+1)x(3+1) reference elements

//the base polygon is the convex pentagon from project 1

vector<Point> polygonVertices;

polygonVertices.resize(5);

polygonVertices[0] = Point(2.5, 1.0);

polygonVertices[1] = Point(4.0, 2.1);

polygonVertices[2] = Point(3.4, 4.2);

polygonVertices[3] = Point(1.6, 4.2);

polygonVertices[4] = Point(1.0, 2.1);

Polygon basePolygon(polygonVertices);

Mesh myMesh(*basePolygon*);

double BoundingBoxLength = myMesh.*getBoundingBoxLength*();

double BoundingBoxHeight = myMesh.*getBoundingBoxHeight*();

myMesh.*setRectangularDomain*(4\*BoundingBoxLength + 1, 3\*BoundingBoxHeight+1);

try

{

myMesh.*createMesh*();

vector<ReferenceElement> myMeshCells = myMesh.*getMeshCells*();

myMesh.*showMesh*("showMesh4.m");

EXPECT\_EQ(myMeshCells.size(),20);

EXPECT\_EQ(myMeshCells[0].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[1].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[2].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[3].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[4].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[5].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[6].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[7].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[8].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[9].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[10].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[11].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[12].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[13].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[14].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[15].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[16].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[17].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[18].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[19].*getConformingReferenceElementPolygons*().size(), 2);

double totalArea = 0;

double domainArea = myMesh.*getDomain*().*ComputeArea*();

for(unsigned int i = 0; i < myMeshCells.size(); i++)

for (unsigned int j = 0; j < myMeshCells[i].*getConformingReferenceElementPolygons*().size(); j++)

totalArea += myMeshCells[i].*getConformingReferenceElementPolygons*()[j].*ComputeArea*();

EXPECT\_DOUBLE\_EQ(totalArea, domainArea);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestSetConvexDomain***)

{

vector<Point> domainVertices;

domainVertices.resize(4);

domainVertices[0] = Point(0, 0);

domainVertices[1] = Point(16, 0);

domainVertices[2] = Point(20,14);

domainVertices[3] = Point(6,16);

Polygon domain(domainVertices);

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 2);

polygonVertices[1] = Point(1, 0);

polygonVertices[2] = Point(4, 3);

polygonVertices[3] = Point(3, 4);

Polygon basePolygon(polygonVertices);

Mesh myMesh(*basePolygon*);

myMesh.*setConvexDomain*(domain);

try

{

EXPECT\_EQ(myMesh.*getDomainBoundingBoxHeight*(), 16);

EXPECT\_EQ(myMesh.*getDomainBoundingBoxLength*(), 20);

EXPECT\_EQ(myMesh.*getMaxNumRefElementsPerColumn*(), 4);

EXPECT\_EQ(myMesh.*getMaxNumRefElementsPerRow*(), 5);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestCreateMesh\_ConvexDomain1a***)

{ //the test polygon is a quadrilateral

//the domain is a quadrilateral

vector<Point> domainVertices;

domainVertices.resize(4);

domainVertices[0] = Point(0, 0);

domainVertices[1] = Point(16, 0);

domainVertices[2] = Point(20,14);

domainVertices[3] = Point(6,16);

Polygon domain(domainVertices);

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 2);

polygonVertices[1] = Point(1, 0);

polygonVertices[2] = Point(4, 3);

polygonVertices[3] = Point(3, 4);

Polygon basePolygon(polygonVertices);

Mesh myMesh(*basePolygon*);

myMesh.*setConvexDomain*(domain);

try

{

myMesh.*createMesh*();

vector<ReferenceElement> myMeshCells = myMesh.*getMeshCells*();

myMesh.*showMesh*("showMeshConvexDomain1a.m");

EXPECT\_EQ(myMeshCells.size(),19);

EXPECT\_EQ(myMeshCells[0].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[1].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[2].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[3].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[4].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[5].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[6].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[7].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[8].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[9].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[10].*getConformingReferenceElementPolygons*().size(), 1);

EXPECT\_EQ(myMeshCells[11].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[12].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[13].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[14].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[15].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[16].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[17].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[18].*getConformingReferenceElementPolygons*().size(), 4);

double totalArea = 0;

double domainArea = myMesh.*getDomain*().*ComputeArea*();

for(unsigned int i = 0; i < myMeshCells.size(); i++)

for (unsigned int j = 0; j < myMeshCells[i].*getConformingReferenceElementPolygons*().size(); j++)

totalArea += myMeshCells[i].*getConformingReferenceElementPolygons*()[j].*ComputeArea*();

EXPECT\_DOUBLE\_EQ(totalArea, domainArea);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestCreateMesh\_ConvexDomain1b***)

{

//the test polygon is the convex hexagon from project 1

//the domain is a quadrilateral

vector<Point> domainVertices;

domainVertices.resize(4);

domainVertices[0] = Point(0, 0);

domainVertices[1] = Point(16, 0);

domainVertices[2] = Point(20,14);

domainVertices[3] = Point(6,16);

Polygon domain(domainVertices);

vector<Point> polygonVertices;

polygonVertices.resize(6);

polygonVertices[0] = Point(1.5, 1.0);

polygonVertices[1] = Point(5.6, 1.5);

polygonVertices[2] = Point(5.5, 4.8);

polygonVertices[3] = Point(4.0, 6.2);

polygonVertices[4] = Point(3.2, 4.2);

polygonVertices[5] = Point(1.0, 4.0);

Polygon basePolygon(polygonVertices);

Mesh myMesh(*basePolygon*);

myMesh.*setConvexDomain*(domain);

try

{

myMesh.*createMesh*();

vector<ReferenceElement> myMeshCells = myMesh.*getMeshCells*();

myMesh.*showMesh*("showMeshConvexDomain1b.m");

EXPECT\_EQ(myMeshCells.size(),15);

EXPECT\_EQ(myMeshCells[0].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[1].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[2].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[3].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[4].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[5].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[6].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[7].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[8].*getConformingReferenceElementPolygons*().size(), 1);

EXPECT\_EQ(myMeshCells[9].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[10].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[11].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[12].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[13].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[14].*getConformingReferenceElementPolygons*().size(), 2);

double totalArea = 0;

double domainArea = myMesh.*getDomain*().*ComputeArea*();

for(unsigned int i = 0; i < myMeshCells.size(); i++)

for (unsigned int j = 0; j < myMeshCells[i].*getConformingReferenceElementPolygons*().size(); j++)

totalArea += myMeshCells[i].*getConformingReferenceElementPolygons*()[j].*ComputeArea*();

EXPECT\_DOUBLE\_EQ(totalArea, domainArea);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestCreateMesh\_ConvexDomain1c***)

{

//the domain is a quadrilateral

//the base polygon is D'Auria's polygon with the shape of a sigma

vector<Point> domainVertices;

domainVertices.resize(4);

domainVertices[0] = Point(0, 0);

domainVertices[1] = Point(16, 0);

domainVertices[2] = Point(20,14);

domainVertices[3] = Point(6,16);

Polygon domain(domainVertices);

vector<Point> polygonVertices;

polygonVertices.resize(10);

polygonVertices[0] = Point(2, -2);

polygonVertices[1] = Point(0, -1);

polygonVertices[2] = Point(3, 1);

polygonVertices[3] = Point(0, 2);

polygonVertices[4] = Point(3, 2);

polygonVertices[5] = Point(3, 3);

polygonVertices[6] = Point(-1, 3);

polygonVertices[7] = Point(-3, 1);

polygonVertices[8] = Point(0, 0);

polygonVertices[9] = Point(-3, -2);

Polygon basePolygon(polygonVertices);

Mesh myMesh(*basePolygon*);

myMesh.*setConvexDomain*(domain);

try

{

myMesh.*createMesh*();

vector<ReferenceElement> myMeshCells = myMesh.*getMeshCells*();

myMesh.*showMesh*("showMeshConvexDomain1c.m");

EXPECT\_EQ(myMeshCells.size(),14);

EXPECT\_EQ(myMeshCells[0].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[1].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[2].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[3].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[4].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[5].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[6].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[7].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[8].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[9].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[10].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[11].*getConformingReferenceElementPolygons*().size(), 1);

EXPECT\_EQ(myMeshCells[12].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[13].*getConformingReferenceElementPolygons*().size(), 2);

double totalArea = 0;

double domainArea = myMesh.*getDomain*().*ComputeArea*();

for(unsigned int i = 0; i < myMeshCells.size(); i++)

for (unsigned int j = 0; j < myMeshCells[i].*getConformingReferenceElementPolygons*().size(); j++)

totalArea += myMeshCells[i].*getConformingReferenceElementPolygons*()[j].*ComputeArea*();

EXPECT\_DOUBLE\_EQ(totalArea, domainArea);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestCreateMesh\_ConvexDomain2a***)

{ //the test polygon is a quadrilateral

//the domain is an hexagon

vector<Point> domainVertices;

domainVertices.resize(6);

domainVertices[0] = Point(6, 0);

domainVertices[1] = Point(14.2, 2.2);

domainVertices[2] = Point(16.39,10.39);

domainVertices[3] = Point(10.39, 16.39);

domainVertices[4] = Point(2.2,14.2);

domainVertices[5] = Point(0,6);

Polygon domain(domainVertices);

vector<Point> polygonVertices;

polygonVertices.resize(4);

polygonVertices[0] = Point(0, 2);

polygonVertices[1] = Point(1, 0);

polygonVertices[2] = Point(4, 3);

polygonVertices[3] = Point(3, 4);

Polygon basePolygon(polygonVertices);

Mesh myMesh(*basePolygon*);

myMesh.*setConvexDomain*(domain);

try

{

myMesh.*createMesh*();

vector<ReferenceElement> myMeshCells = myMesh.*getMeshCells*();

myMesh.*showMesh*("showMeshConvexDomain2a.m");

EXPECT\_EQ(myMeshCells.size(),18);

EXPECT\_EQ(myMeshCells[0].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[1].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[2].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[3].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[4].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[5].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[6].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[7].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[8].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[9].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[10].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[11].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[12].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[13].*getConformingReferenceElementPolygons*().size(), 2);

EXPECT\_EQ(myMeshCells[14].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[15].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[16].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[17].*getConformingReferenceElementPolygons*().size(), 3);

double totalArea = 0;

double domainArea = myMesh.*getDomain*().*ComputeArea*();

for(unsigned int i = 0; i < myMeshCells.size(); i++)

for (unsigned int j = 0; j < myMeshCells[i].*getConformingReferenceElementPolygons*().size(); j++)

totalArea += myMeshCells[i].*getConformingReferenceElementPolygons*()[j].*ComputeArea*();

EXPECT\_DOUBLE\_EQ(totalArea, domainArea);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestCreateMesh\_ConvexDomain2b***)

{ //the test polygon is the convex hexagon from project 1

//the domain is an hexagon

vector<Point> domainVertices;

domainVertices.resize(6);

domainVertices[0] = Point(6, 0);

domainVertices[1] = Point(14.2, 2.2);

domainVertices[2] = Point(16.39,10.39);

domainVertices[3] = Point(10.39, 16.39);

domainVertices[4] = Point(2.2,14.2);

domainVertices[5] = Point(0,6);

Polygon domain(domainVertices);

vector<Point> polygonVertices;

polygonVertices.resize(6);

polygonVertices[0] = Point(1.5, 1.0);

polygonVertices[1] = Point(5.6, 1.5);

polygonVertices[2] = Point(5.5, 4.8);

polygonVertices[3] = Point(4.0, 6.2);

polygonVertices[4] = Point(3.2, 4.2);

polygonVertices[5] = Point(1.0, 4.0);

Polygon basePolygon(polygonVertices);

Mesh myMesh(*basePolygon*);

myMesh.*setConvexDomain*(domain);

try

{

myMesh.*createMesh*();

vector<ReferenceElement> myMeshCells = myMesh.*getMeshCells*();

myMesh.*showMesh*("showMeshConvexDomain2b.m");

EXPECT\_EQ(myMeshCells.size(),14);

EXPECT\_EQ(myMeshCells[0].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[1].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[2].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[3].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[4].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[5].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[6].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[7].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[8].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[9].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[10].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[11].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[12].*getConformingReferenceElementPolygons*().size(), 1);

EXPECT\_EQ(myMeshCells[13].*getConformingReferenceElementPolygons*().size(), 3);

double totalArea = 0;

double domainArea = myMesh.*getDomain*().*ComputeArea*();

for(unsigned int i = 0; i < myMeshCells.size(); i++)

for (unsigned int j = 0; j < myMeshCells[i].*getConformingReferenceElementPolygons*().size(); j++)

totalArea += myMeshCells[i].*getConformingReferenceElementPolygons*()[j].*ComputeArea*();

EXPECT\_DOUBLE\_EQ(totalArea, domainArea);

}

catch (const exception& ex)

{

FAIL();

}

}

TEST(***TestMeshClass***, ***TestCreateMesh\_ConvexDomain2c***)

{ //the base polygon is D'Auria's polygon with the shape of a sigma

//the domain is an hexagon

vector<Point> domainVertices;

domainVertices.resize(6);

domainVertices[0] = Point(6, 0);

domainVertices[1] = Point(14.2, 2.2);

domainVertices[2] = Point(16.39,10.39);

domainVertices[3] = Point(10.39, 16.39);

domainVertices[4] = Point(2.2,14.2);

domainVertices[5] = Point(0,6);

Polygon domain(domainVertices);

vector<Point> polygonVertices;

polygonVertices.resize(10);

polygonVertices[0] = Point(2, -2);

polygonVertices[1] = Point(0, -1);

polygonVertices[2] = Point(3, 1);

polygonVertices[3] = Point(0, 2);

polygonVertices[4] = Point(3, 2);

polygonVertices[5] = Point(3, 3);

polygonVertices[6] = Point(-1, 3);

polygonVertices[7] = Point(-3, 1);

polygonVertices[8] = Point(0, 0);

polygonVertices[9] = Point(-3, -2);

Polygon basePolygon(polygonVertices);

Mesh myMesh(*basePolygon*);

myMesh.*setConvexDomain*(domain);

try

{

myMesh.*createMesh*();

vector<ReferenceElement> myMeshCells = myMesh.*getMeshCells*();

myMesh.*showMesh*("showMeshConvexDomain2c.m");

EXPECT\_EQ(myMeshCells.size(),11);

EXPECT\_EQ(myMeshCells[0].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[1].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[2].*getConformingReferenceElementPolygons*().size(), 3);

EXPECT\_EQ(myMeshCells[3].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[4].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[5].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[6].*getConformingReferenceElementPolygons*().size(), 4);

EXPECT\_EQ(myMeshCells[7].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[8].*getConformingReferenceElementPolygons*().size(), 5);

EXPECT\_EQ(myMeshCells[9].*getConformingReferenceElementPolygons*().size(), 1);

EXPECT\_EQ(myMeshCells[10].*getConformingReferenceElementPolygons*().size(),3);

double totalArea = 0;

double domainArea = myMesh.*getDomain*().*ComputeArea*();

for(unsigned int i = 0; i < myMeshCells.size(); i++)

for (unsigned int j = 0; j < myMeshCells[i].*getConformingReferenceElementPolygons*().size(); j++)

totalArea += myMeshCells[i].*getConformingReferenceElementPolygons*()[j].*ComputeArea*();

EXPECT\_DOUBLE\_EQ(totalArea, domainArea);

}

catch (const exception& ex)

{

FAIL();

}

}

}

#endif

PYTHON:

POINT\_CLASS.PY

**import** numpy **as** np  
**from** src **import** CONSTANT  
  
  
**class** Point:  
 **def** \_\_init\_\_(self, coordinates):  
 self.\_\_coordinates = coordinates  
 self.\_\_label = -2  
  
 **def** set\_coordinates(self, x: float, y: float):  
 self.\_\_coordinates = np.array([x, y])  
  
 **def** get\_coordinates(self):  
 **return** self.\_\_coordinates  
  
 **def** set\_point\_label(self, label):  
 self.\_\_label = label  
  
 **def** get\_point\_label(self):  
 **return** self.\_\_label  
  
 **def** \_\_add\_\_(self, other):  
 **return** Point(self.\_\_coordinates + other.\_\_coordinates)  
  
 **def** \_\_sub\_\_(self, other):  
 **return** Point(self.\_\_coordinates - other.\_\_coordinates)  
  
 **def** \_\_eq\_\_(self, other):  
 **return** np.linalg.norm(self.\_\_coordinates - other.\_\_coordinates) < CONSTANT.POINT\_TOLERANCE  
  
 **def** \_\_ne\_\_(self, other):  
 **return not** (self == other)  
  
 **def** \_\_lt\_\_(self, other):  
 **return** self.\_\_label < other.\_\_label

SEGMENT\_CLASS.PY

**from** src **import** PointClass **as** PC  
  
  
**class** ISegment:  
  
 **def** set\_start(self, start: PC.Point):  
 **pass  
  
 def** get\_start(self):  
 **pass  
  
 def** set\_end(self, end: PC.Point):  
 **pass  
  
 def** get\_end(self):  
 **pass  
  
 def** set\_segment(self, start: PC.Point, end: PC.Point):  
 **pass  
  
 def** compute\_tangent(self):  
 **pass  
  
 def** get\_tangent(self):  
 **pass  
  
  
class** Segment(ISegment):  
  
 **def** \_\_init\_\_(self, start: PC.Point, end: PC.Point):  
 self.\_\_start = start  
 self.\_\_end = end  
 self.\_\_tangent = end.get\_coordinates() - start.get\_coordinates()  
  
 **def** set\_start(self, start: PC.Point):  
 self.\_\_start = start  
  
 **def** get\_start(self):  
 **return** self.\_\_start  
  
 **def** set\_end(self, end: PC.Point):  
 self.\_\_end = end  
  
 **def** get\_end(self):  
 **return** self.\_\_end  
  
 **def** set\_segment(self, start: PC.Point, end: PC.Point):  
 self.\_\_start = start  
 self.\_\_end = end  
  
 **def** compute\_tangent(self):  
 self.\_\_tangent = (self.\_\_end - self.\_\_start).get\_coordinates()  
 **return** self.\_\_tangent  
  
 **def** get\_tangent(self):  
 **return** self.\_\_tangent  
  
 **def** \_\_eq\_\_(self, other):  
 **return** (self.get\_start() == other.get\_start()) **and** (self.get\_end() == other.get\_end())

POLYGON\_CLASS.PY

**import** src.SegmentClass **as** SG  
  
  
**class** IPolygon:  
 **def** get\_number\_vertices(self):  
 **pass  
  
 def** get\_vertices(self):  
 **pass  
  
 def** set\_vertex\_at\_position(self, vertex, position):  
 **pass  
  
 def** append\_vertex(self, vertex):  
 **pass  
  
 def** insert\_vertex\_at\_position(self, vertex, position):  
 **pass  
  
 def** get\_vertex\_at\_position(self, position):  
 **pass  
  
 def** next\_vertex\_id(self, id):  
 **pass  
  
 def** get\_last\_vertex(self):  
 **pass  
  
 def** get\_edges(self):  
 **pass  
  
 def** set\_vertex\_label\_at\_position(self, position, label):  
 **pass  
  
 def** get\_point\_label\_at\_position(self, position):  
 **pass  
  
 def** set\_vertices\_label(self, polygon\_vertices):  
 **pass  
  
 def** set\_default\_vertices\_label(self):  
 **pass  
  
  
class** Polygon(IPolygon):  
 **def** \_\_init\_\_(self, vertices):  
 self.\_\_number\_vertices = len(vertices)  
 self.\_\_vertices = vertices  
 self.\_\_edges = []  
  
 **for** i **in** range(0, self.\_\_number\_vertices):  
 **if** i == (self.\_\_number\_vertices - 1):  
 self.\_\_edges.append(SG.Segment(self.\_\_vertices[i], self.\_\_vertices[0]))  
 **else**:  
 self.\_\_edges.append(SG.Segment(self.\_\_vertices[i], self.\_\_vertices[i + 1]))  
  
 **def** get\_number\_vertices(self):  
 **return** self.\_\_number\_vertices  
  
 **def** get\_vertices(self):  
 **return** self.\_\_vertices  
  
 **def** set\_vertex\_at\_position(self, vertex, position):  
 self.\_\_vertices[position] = vertex  
  
 **def** append\_vertex(self, vertex):  
 self.\_\_vertices.append(vertex)  
 self.\_\_number\_vertices = len(self.\_\_vertices)  
  
 **def** insert\_vertex\_at\_position(self, vertex, position):  
 self.\_\_vertices.insert(position, vertex)  
 self.\_\_number\_vertices = len(self.\_\_vertices)  
  
 **def** get\_vertex\_at\_position(self, position):  
 **return** self.\_\_vertices[position]  
  
 **def** next\_vertex\_id(self, id):  
 **if** id < 0 **or** id >= self.\_\_number\_vertices:  
 **raise** ValueError(**"Vertex id is not present"**)  
  
 **if** id == self.\_\_number\_vertices - 1:  
 **return** 0  
  
 **else**:  
 **return** id + 1  
  
 **def** get\_last\_vertex(self):  
 **return** self.\_\_vertices[self.\_\_number\_vertices - 1]  
  
 **def** get\_edges(self):  
 **return** self.\_\_edges  
  
 **def** set\_vertex\_label\_at\_position(self, position, label):  
 self.\_\_vertices[position].set\_point\_label(label)  
  
 **def** get\_point\_label\_at\_position(self, position):  
 **return** self.\_\_vertices[position].get\_point\_label()  
  
 **def** set\_vertices\_label(self, polygon\_vertices):  
 **for** i **in** range(0, self.\_\_number\_vertices):  
 label = polygon\_vertices[i]  
 self.\_\_vertices[i].set\_point\_label(label)  
  
 **def** set\_default\_vertices\_label(self):  
 **for** i **in** range(0, self.\_\_number\_vertices):  
 self.\_\_vertices[i].set\_point\_label(i)

INTERSECTOR1D1D\_CLASS.PY

**from** src **import** CONSTANT  
**import** numpy **as** np  
**import** numpy.linalg **as** nalg  
**import** math  
**from** enum **import** Enum  
  
  
**class** Type(Enum):  
 NoIntersection = 0  
 IntersectionOnLine = 1  
 IntersectionOnSegment = 2  
 IntersectionParallelOnLine = 3  
 IntersectionParallelOnSegment = 4  
  
  
**class** Position(Enum):  
 Begin = 0  
 Inner = 1  
 End = 2  
 Outer = 3  
  
  
**class** IIntersector1D1D:  
 **def** set\_first\_segment(self, origin, end):  
 **pass  
  
 def** set\_second\_segment(self, origin, end):  
 **pass  
  
 def** compute\_intersection(self):  
 **pass  
  
 def** parametric\_coordinates(self):  
 **pass  
  
 def** first\_parametric\_coordinate(self):  
 **pass  
  
 def** second\_parametric\_coordinate(self):  
 **pass  
  
 def** position\_intersection\_in\_first\_segment(self):  
 **pass  
  
 def** position\_intersection\_in\_second\_segment(self):  
 **pass  
  
 def** type\_intersection(self):  
 **pass  
  
 def** intersection\_first\_parametric\_coordinate(self, origin, end):  
 **pass  
  
 def** intersection\_second\_parametric\_coordinate(self, origin, end):  
 **pass  
  
  
class** Intersector1D1D(IIntersector1D1D):  
 **def** \_\_init\_\_(self):  
 self.\_\_type = Type.NoIntersection  
 self.\_\_positionIntersectionFirstEdge = Position.Begin  
 self.\_\_positionIntersectionSecondEdge = Position.Begin  
 self.\_\_resultParametricCoordinates = np.array([0, 0])  
 self.\_\_originFirstSegment = np.array([0, 0])  
 self.\_\_rightHandSide = np.array([0, 0])  
 self.\_\_matrixTangentVector = np.zeros((2, 2))  
  
 **def** set\_first\_segment(self, origin, end):  
 self.\_\_matrixTangentVector[:, 0] = end - origin  
 self.\_\_originFirstSegment = origin  
  
 **def** set\_second\_segment(self, origin, end):  
 self.\_\_matrixTangentVector[:, 1] = origin - end  
 self.\_\_rightHandSide = origin - self.\_\_originFirstSegment  
  
 **def** parametric\_coordinates(self):  
 **return** self.\_\_resultParametricCoordinates  
  
 **def** first\_parametric\_coordinate(self):  
 **return** self.\_\_resultParametricCoordinates[0]  
  
 **def** second\_parametric\_coordinate(self):  
 **return** self.\_\_resultParametricCoordinates[1]  
  
 **def** position\_intersection\_in\_first\_segment(self):  
 **return** self.\_\_positionIntersectionFirstEdge  
  
 **def** position\_intersection\_in\_second\_segment(self):  
 **return** self.\_\_positionIntersectionSecondEdge  
  
 **def** type\_intersection(self):  
 **return** self.\_\_type  
  
 **def** intersection\_first\_parametric\_coordinate(self, origin, end):  
 **return** (1 - self.\_\_resultParametricCoordinates[0]) \* origin + self.\_\_resultParametricCoordinates[0] \* end  
  
 **def** intersection\_second\_parametric\_coordinate(self, origin, end):  
 **return** (1 - self.\_\_resultParametricCoordinates[1]) \* origin + self.\_\_resultParametricCoordinates[1] \* end  
  
 **def** compute\_intersection(self):  
 parallelism = nalg.det(self.\_\_matrixTangentVector)  
 self.\_\_type = Type.NoIntersection  
 intersection = **False** check = CONSTANT.INTERSECTION\_TOLERANCE \* CONSTANT.INTERSECTION\_TOLERANCE \* nalg.norm(  
 self.\_\_matrixTangentVector[:, 0]) \* nalg.norm(self.\_\_matrixTangentVector[:, ])  
  
 **if** parallelism \* parallelism >= check:  
 solverMatrix = np.array([[self.\_\_matrixTangentVector[1, 1], -self.\_\_matrixTangentVector[0, 1]],  
 [-self.\_\_matrixTangentVector[1, 0], self.\_\_matrixTangentVector[0, 0]]])  
  
 self.\_\_resultParametricCoordinates = np.dot(solverMatrix, self.\_\_rightHandSide)  
 self.\_\_resultParametricCoordinates /= parallelism  
  
 **if** (self.\_\_resultParametricCoordinates[1] > -CONSTANT.INTERSECTION\_TOLERANCE **and** self.\_\_resultParametricCoordinates[1] - 1.0 < CONSTANT.INTERSECTION\_TOLERANCE):  
 self.\_\_type = Type.IntersectionOnLine  
 intersection = **True  
  
 if** (self.\_\_resultParametricCoordinates[0] > -CONSTANT.INTERSECTION\_TOLERANCE **and** self.\_\_resultParametricCoordinates[0] - 1.0 < CONSTANT.INTERSECTION\_TOLERANCE):  
 self.\_\_type = Type.IntersectionOnSegment  
  
 **else**:  
 parallelism2 = math.fabs(  
 self.\_\_matrixTangentVector[0, 0] \* self.\_\_rightHandSide[1] - self.\_\_rightHandSide[0] \*  
 self.\_\_matrixTangentVector[1, 0])  
 squaredNormFirstEdge = np.square(nalg.norm(self.\_\_matrixTangentVector[:, 0]))  
 check2 = CONSTANT.INTERSECTION\_TOLERANCE \* CONSTANT.INTERSECTION\_TOLERANCE \* squaredNormFirstEdge \* nalg.norm(  
 self.\_\_rightHandSide)  
 **if** (parallelism2 \* parallelism2 <= check2):  
 tempNorm = 1.0 / squaredNormFirstEdge  
 self.\_\_resultParametricCoordinates[0] = np.dot(self.\_\_matrixTangentVector[:, 0], self.\_\_rightHandSide)  
 self.\_\_resultParametricCoordinates[1] = self.\_\_resultParametricCoordinates[0] - np.dot(  
 self.\_\_matrixTangentVector[:, 0], self.\_\_matrixTangentVector[:, 1])  
 self.\_\_resultParametricCoordinates = tempNorm \* self.\_\_resultParametricCoordinates  
 intersection = **True** self.\_\_type = Type.IntersectionParallelOnLine  
  
 **if** (self.\_\_resultParametricCoordinates[1] < self.\_\_resultParametricCoordinates[0]):  
 tmp = self.\_\_resultParametricCoordinates[0]  
 self.\_\_resultParametricCoordinates[0] = self.\_\_resultParametricCoordinates[1]  
 self.\_\_resultParametricCoordinates[1] = tmp  
  
 **if** ((self.\_\_resultParametricCoordinates[0] > -CONSTANT.INTERSECTION\_TOLERANCE **and** self.\_\_resultParametricCoordinates[0] - 1.0 < CONSTANT.INTERSECTION\_TOLERANCE) **or** (  
 self.\_\_resultParametricCoordinates[1] > -CONSTANT.INTERSECTION\_TOLERANCE **and** self.\_\_resultParametricCoordinates[1] - 1.0 < CONSTANT.INTERSECTION\_TOLERANCE)):  
 self.\_\_type = Type.IntersectionParallelOnSegment  
  
 **else**:  
 **if** (self.\_\_resultParametricCoordinates[0] < CONSTANT.INTERSECTION\_TOLERANCE **and** self.\_\_resultParametricCoordinates[  
 1] - 1.0 > -CONSTANT.INTERSECTION\_TOLERANCE):  
 self.\_\_type = Type.IntersectionParallelOnSegment  
  
 **if** (self.\_\_resultParametricCoordinates[0] < -CONSTANT.INTERSECTION\_TOLERANCE **or** self.\_\_resultParametricCoordinates[0] - 1.0 > CONSTANT.INTERSECTION\_TOLERANCE):  
 self.\_\_positionIntersectionFirstEdge = Position.Outer  
  
 **elif** ((self.\_\_resultParametricCoordinates[0] > -CONSTANT.INTERSECTION\_TOLERANCE) **and** (  
 self.\_\_resultParametricCoordinates[0] < CONSTANT.INTERSECTION\_TOLERANCE)):  
 self.\_\_resultParametricCoordinates[0] = 0.0  
 self.\_\_positionIntersectionFirstEdge = Position.Begin  
  
 **elif** ((self.\_\_resultParametricCoordinates[0] > 1.0 - CONSTANT.INTERSECTION\_TOLERANCE) **and** (  
 self.\_\_resultParametricCoordinates[0] < 1.0 + CONSTANT.INTERSECTION\_TOLERANCE)):  
 self.\_\_resultParametricCoordinates[0] = 1.0  
 self.\_\_positionIntersectionFirstEdge = Position.End  
  
 **else**:  
 self.\_\_positionIntersectionFirstEdge = Position.Inner  
  
 **if** (self.\_\_resultParametricCoordinates[1] < -CONSTANT.INTERSECTION\_TOLERANCE **or** self.\_\_resultParametricCoordinates[1] > 1.0 + CONSTANT.INTERSECTION\_TOLERANCE):  
 self.\_\_positionIntersectionSecondEdge = Position.Outer  
  
 **elif** ((self.\_\_resultParametricCoordinates[1] > -CONSTANT.INTERSECTION\_TOLERANCE) **and** (  
 self.\_\_resultParametricCoordinates[1] < CONSTANT.INTERSECTION\_TOLERANCE)):  
 self.\_\_resultParametricCoordinates[1] = 0.0  
 self.\_\_positionIntersectionSecondEdge = Position.Begin  
  
 **elif** ((self.\_\_resultParametricCoordinates[1] > 1.0 - CONSTANT.INTERSECTION\_TOLERANCE) **and** (  
 self.\_\_resultParametricCoordinates[1] < 1.0 + CONSTANT.INTERSECTION\_TOLERANCE)):  
 self.\_\_resultParametricCoordinates[1] = 1.0  
 self.\_\_positionIntersectionSecondEdge = Position.End  
  
 **else**:  
 self.\_\_positionIntersectionSecondEdge = Position.Inner  
  
 **return** intersection

INTERSECTOR\_POLYGON\_LINE\_CLASS.PY

**import** src.PolygonClass **as** PLG  
**import** src.Intersector1D1D **as** INTSEC  
**import** collections **as** co  
**import** src.PointClass **as** PC  
**import** src.CONSTANT **as** CONST  
  
  
**class** IIntersectorPolygonLine:  
 **def** find\_intersection\_vertices(self):  
 **pass  
  
 def** find\_polygons(self):  
 **pass  
  
 def** get\_extended\_polygons(self):  
 **pass  
  
 def** find\_new\_points(self):  
 **pass  
  
 def** extended\_polygon\_if\_end\_points\_on\_edge(self, num\_poly, first\_intersection\_id, second\_intersection\_id, forward\_direction):  
 **pass  
  
 def** check\_in\_conic\_combination(self, line: PLG.SG.PC.np.array, v1: PLG.SG.PC.np.array, v2: PLG.SG.PC.np.array,  
 flag: int):  
 **pass  
  
 def** search\_key\_index(self, multimap, key):  
 **pass  
  
  
class** intersector\_polygon\_line(IIntersectorPolygonLine):  
 **def** \_\_init\_\_(self, intersector: INTSEC.IIntersector1D1D, polygon: PLG.IPolygon, segment: PLG.SG.ISegment):  
 self.\_\_intersector = intersector  
 self.\_\_polygon = polygon  
 self.\_\_line = segment  
 self.\_\_intersection\_vertices = []  
 self.\_\_new\_points = []  
 self.\_\_extended\_new\_polygons = []  
 self.\_\_new\_polygons = []  
 self.\_\_intersection\_id\_to\_segment\_id = []  
 self.\_\_segment\_id\_to\_intersection\_id = []  
 self.\_\_found\_vertex = []  
 self.\_\_start\_found = **False** self.\_\_end\_found = **False** self.\_\_flag = []  
 self.\_\_mmap\_parametric\_coordinate\_to\_intersection\_point = []  
 self.\_\_mmap\_parametric\_coordinate\_to\_segment\_id = []  
 self.\_\_map\_segment\_id\_to\_parametric\_coordinate = co.OrderedDict()  
 self.\_\_map\_intersection\_id\_to\_parametric\_coordinate = co.OrderedDict()  
  
 **for** i **in** range(0, polygon.get\_number\_vertices()):  
 self.\_\_segment\_id\_to\_intersection\_id.append(-1)  
 self.\_\_found\_vertex.append(**False**)  
 self.\_\_flag.append(0)  
  
 self.\_\_intersector.set\_first\_segment(segment.get\_start().get\_coordinates(), segment.get\_end().get\_coordinates())  
  
 **def** find\_intersection\_vertices(self):  
 edges = self.\_\_polygon.get\_edges()  
 self.\_\_polygon.set\_default\_vertices\_label()  
 self.\_\_line.get\_start().set\_point\_label(self.\_\_polygon.get\_number\_vertices())  
 self.\_\_line.get\_end().set\_point\_label(self.\_\_polygon.get\_number\_vertices() + 1)  
  
 **for** i **in** range(0, self.\_\_polygon.get\_number\_vertices()):  
 self.\_\_intersector.set\_second\_segment(edges[i].get\_start().get\_coordinates(),  
 edges[i].get\_end().get\_coordinates())  
 **if** (self.\_\_intersector.compute\_intersection() == **True**):  
 intersection = self.\_\_intersector.intersection\_first\_parametric\_coordinate(  
 self.\_\_line.get\_start().get\_coordinates(), self.\_\_line.get\_end().get\_coordinates())  
 s = self.\_\_intersector.first\_parametric\_coordinate()  
 **if** (self.search\_key\_index(self.\_\_mmap\_parametric\_coordinate\_to\_intersection\_point, s) == len(self.\_\_mmap\_parametric\_coordinate\_to\_intersection\_point)):  
 self.\_\_mmap\_parametric\_coordinate\_to\_intersection\_point.append((s, PC.Point(intersection)))  
 self.\_\_mmap\_parametric\_coordinate\_to\_intersection\_point = sorted(self.\_\_mmap\_parametric\_coordinate\_to\_intersection\_point)  
 self.\_\_mmap\_parametric\_coordinate\_to\_segment\_id.append((s, i))  
 self.\_\_mmap\_parametric\_coordinate\_to\_segment\_id = sorted(self.\_\_mmap\_parametric\_coordinate\_to\_segment\_id)  
 self.\_\_map\_segment\_id\_to\_parametric\_coordinate[i] = s  
 self.\_\_map\_segment\_id\_to\_parametric\_coordinate = co.OrderedDict(sorted(self.\_\_map\_segment\_id\_to\_parametric\_coordinate.items()))  
 **else**:  
 is\_in\_conic = self.check\_in\_conic\_combination(self.\_\_line.compute\_tangent(), edges[i - 1].get\_tangent(), edges[i].get\_tangent(), i)  
  
 **if**(is\_in\_conic == **True**):  
 self.\_\_mmap\_parametric\_coordinate\_to\_intersection\_point.append((s, PC.Point(intersection)))  
 self.\_\_mmap\_parametric\_coordinate\_to\_intersection\_point = sorted(self.\_\_mmap\_parametric\_coordinate\_to\_intersection\_point)  
 self.\_\_mmap\_parametric\_coordinate\_to\_segment\_id.append((s, i))  
 self.\_\_mmap\_parametric\_coordinate\_to\_segment\_id = sorted(self.\_\_mmap\_parametric\_coordinate\_to\_segment\_id)  
 self.\_\_map\_segment\_id\_to\_parametric\_coordinate[i] = s  
 self.\_\_map\_segment\_id\_to\_parametric\_coordinate = co.OrderedDict(sorted(self.\_\_map\_segment\_id\_to\_parametric\_coordinate.items()))  
  
 **for** i **in** range(0, len(self.\_\_mmap\_parametric\_coordinate\_to\_intersection\_point)):  
 self.\_\_intersection\_vertices.append(self.\_\_mmap\_parametric\_coordinate\_to\_intersection\_point[i][1])  
 self.\_\_intersection\_vertices[i].set\_point\_label(self.\_\_polygon.get\_number\_vertices() + 2 + i)  
  
 **for** i **in** range(0, len(self.\_\_mmap\_parametric\_coordinate\_to\_segment\_id)):  
 self.\_\_intersection\_id\_to\_segment\_id.append(self.\_\_mmap\_parametric\_coordinate\_to\_segment\_id[i][1])  
 self.\_\_segment\_id\_to\_intersection\_id[self.\_\_mmap\_parametric\_coordinate\_to\_segment\_id[i][1]] = i  
  
 **for** i **in** range(0, len(self.\_\_intersection\_id\_to\_segment\_id)):  
 segment\_position = self.\_\_intersection\_id\_to\_segment\_id[i]  
 segment\_start\_point: PC.Point = edges[segment\_position].get\_start()  
 **if**(self.\_\_intersection\_vertices[i] == segment\_start\_point):  
 self.\_\_intersection\_vertices[i].set\_point\_label(segment\_start\_point.get\_point\_label())  
  
 segment\_end\_point = edges[segment\_position].get\_end()  
 **if**(self.\_\_intersection\_vertices[i] == segment\_end\_point):  
 self.\_\_intersection\_vertices[i].set\_point\_label(segment\_end\_point.get\_point\_label())  
  
 **for** i **in** range(0, len(self.\_\_intersection\_id\_to\_segment\_id)):  
 parametric\_coordinate = self.\_\_map\_segment\_id\_to\_parametric\_coordinate[self.\_\_intersection\_id\_to\_segment\_id[i]]  
 self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[i] = parametric\_coordinate  
 self.\_\_map\_intersection\_id\_to\_parametric\_coordinate = co.OrderedDict(sorted(self.\_\_map\_intersection\_id\_to\_parametric\_coordinate.items()))  
  
 **if** (len(self.\_\_intersection\_vertices) % 2 == 1):  
 **raise** ValueError (**"errore nel numero dei vertici: vertici dispari"**)  
  
 **return** self.\_\_intersection\_vertices  
  
  
  
  
 **def** find\_polygons(self):  
 **if**(len(self.\_\_intersection\_vertices) == 0):  
 **raise** ValueError (**"first you need to find intersection vertices"**)  
  
 new\_polygons = self.\_\_new\_polygons  
 extended\_new\_polygons = self.\_\_extended\_new\_polygons  
 vertices = self.\_\_polygon.get\_vertices()  
 num\_poly = 0  
  
 **for** i **in** range(0, len(self.\_\_found\_vertex)):  
  
 **if**(self.\_\_found\_vertex[i] == **False**):  
 new\_polygons.append(PLG.Polygon([]))  
 extended\_new\_polygons.append(PLG.Polygon([]))  
 id = i  
  
 **while True**:  
 **if**(self.\_\_segment\_id\_to\_intersection\_id[id] == -1):  
 new\_polygons[num\_poly].append\_vertex(vertices[id])  
 extended\_new\_polygons[num\_poly].append\_vertex(vertices[id])  
  
 self.\_\_found\_vertex[id] = **True** id = self.\_\_polygon.next\_vertex\_id(id)  
  
 **else**:  
 new\_polygons[num\_poly].append\_vertex(vertices[id])  
 extended\_new\_polygons[num\_poly].append\_vertex(vertices[id])  
  
 self.\_\_found\_vertex[id] = **True** intersection\_id = self.\_\_segment\_id\_to\_intersection\_id[id]  
  
 **if**(self.\_\_intersection\_vertices[intersection\_id] != vertices[id]):  
 new\_polygons[num\_poly].append\_vertex(self.\_\_intersection\_vertices[intersection\_id])  
 extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_intersection\_vertices[intersection\_id])  
  
 **if**(intersection\_id % 2 == 0):  
 forward\_direction = **True** first\_intersection\_id = intersection\_id  
 second\_intersection\_id = intersection\_id + 1  
 self.extended\_polygon\_if\_end\_points\_on\_edge(num\_poly, first\_intersection\_id, second\_intersection\_id, forward\_direction)  
 intersection\_id += 1  
  
 **else**:  
 forward\_direction = **False** first\_intersection\_id = intersection\_id - 1  
 second\_intersection\_id = intersection\_id  
 self.extended\_polygon\_if\_end\_points\_on\_edge(num\_poly, first\_intersection\_id, second\_intersection\_id, forward\_direction)  
 intersection\_id -= 1  
  
 id = self.\_\_intersection\_id\_to\_segment\_id[intersection\_id]  
 **if**(self.\_\_intersection\_vertices[intersection\_id] != vertices[self.\_\_polygon.next\_vertex\_id(id)]):  
  
 **if**((forward\_direction **and** self.\_\_intersection\_vertices[intersection\_id] == self.\_\_intersection\_vertices[intersection\_id - 1])  
 **or** ((**not** forward\_direction) **and** self.\_\_intersection\_vertices[intersection\_id] == self.\_\_intersection\_vertices[intersection\_id + 1])):  
 self.\_\_found\_vertex[id] = **True  
  
 else**:  
 new\_polygons[num\_poly].append\_vertex(self.\_\_intersection\_vertices[intersection\_id])  
 extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_intersection\_vertices[intersection\_id])  
  
 id = self.\_\_polygon.next\_vertex\_id(id)  
 **if**(id == i):  
 **break** num\_poly = num\_poly + 1  
 **return** new\_polygons  
  
  
 **def** get\_intersection\_vertices(self):  
 **return** self.\_\_intersection\_vertices  
  
 **def** get\_new\_point(self):  
 **return** self.\_\_new\_points  
  
 **def** get\_extended\_polygons(self):  
 **return** self.\_\_extended\_new\_polygons  
  
 **def** find\_new\_points(self):  
 **if** (self.\_\_start\_found == **True**):  
 self.\_\_new\_points.append(self.\_\_line.get\_start())  
 **if** (self.\_\_end\_found == **True**):  
 self.\_\_new\_points.append(self.\_\_line.get\_end())  
  
 **for** i **in** range (0, len(self.\_\_intersection\_vertices)):  
 segment\_position = self.\_\_intersection\_id\_to\_segment\_id[i]  
 segment\_start\_point = self.\_\_polygon.get\_edges()[segment\_position].get\_start()  
 segment\_end\_point = self.\_\_polygon.get\_edges()[segment\_position].get\_end()  
  
 **if** (self.\_\_intersection\_vertices[i] != segment\_start\_point **and** self.\_\_intersection\_vertices[i] != segment\_end\_point):  
 self.\_\_new\_points.append(self.\_\_intersection\_vertices[i])  
  
 **return** self.\_\_new\_points  
  
 **def** extended\_polygon\_if\_end\_points\_on\_edge(self, num\_poly, first\_intersection\_id, second\_intersection\_id, forward\_direction):  
  
 **if**(self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[first\_intersection\_id] < 0 **and** self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[second\_intersection\_id] > 0 **and** self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[second\_intersection\_id] < 1):  
 self.\_\_extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_line.get\_start())  
 self.\_\_start\_found = **True  
  
 elif**(self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[first\_intersection\_id] < 0 **and** self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[second\_intersection\_id] > 1):  
 **if**(forward\_direction == **True**):  
 self.\_\_extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_line.get\_start())  
 self.\_\_start\_found = **True** self.\_\_extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_line.get\_end())  
 self.\_\_end\_found = **True  
  
 else**:  
 self.\_\_extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_line.get\_end())  
 self.\_\_end\_found = **True** self.\_\_extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_line.get\_start())  
 self.\_\_start\_found = **True  
  
 elif**(self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[first\_intersection\_id] > 0 **and** self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[first\_intersection\_id] < 1 **and** self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[second\_intersection\_id] > 1):  
 self.\_\_extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_line.get\_end())  
 self.\_\_end\_found = **True  
  
 elif**(self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[first\_intersection\_id] > 0 **and** self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[first\_intersection\_id] < 1 **and** self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[second\_intersection\_id] < 0):  
 self.\_\_extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_line.get\_start())  
 self.\_\_start\_found = **True  
  
 elif**(self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[first\_intersection\_id] > 1 **and** self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[second\_intersection\_id] < 0):  
 **if**(forward\_direction == **True**):  
 self.\_\_extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_line.get\_end())  
 self.\_\_end\_found =**True** self.\_\_extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_line.get\_start())  
 self.\_\_start\_found = **True  
  
 else**:  
 self.\_\_extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_line.get\_start())  
 self.\_\_start\_found = **True** self.\_\_extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_line.get\_end())  
 self.\_\_end\_found = **True  
  
 elif**(self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[first\_intersection\_id] > 1 **and** self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[second\_intersection\_id] < 1 **and** self.\_\_map\_intersection\_id\_to\_parametric\_coordinate[second\_intersection\_id] > 0):  
 self.\_\_extended\_new\_polygons[num\_poly].append\_vertex(self.\_\_line.get\_end())  
 self.\_\_end\_found = **True  
  
  
 def** check\_in\_conic\_combination(self, line, v1, v2, i):  
 nline = line / PLG.SG.PC.np.linalg.norm(line)  
 nv1 = v1 / PLG.SG.PC.np.linalg.norm(v1)  
 nv2 = v2 / PLG.SG.PC.np.linalg.norm(v2)  
 bisector = nv1 + nv2  
 bisector = bisector / PLG.SG.PC.np.linalg.norm(bisector)  
 theta = INTSEC.math.acos(PLG.SG.PC.np.dot(bisector, nv1))  
 alpha = INTSEC.math.acos(PLG.SG.PC.np.dot(bisector, nline))  
  
 **if** (alpha < theta):  
 self.\_\_flag[i] = 1  
 **return True  
  
 elif** ( alpha > INTSEC.math.pi/2 **and** (INTSEC.math.pi-alpha) < theta):  
 self.\_\_flag[i] = -1  
 **return True** self.\_\_flag[i] = 0  
 **return False  
  
 def** search\_key\_index(self, multimap, key):  
 temp1 = **False** temp2 = 0  
 **for** i **in** range (0, len(multimap)):  
 **if** abs(multimap[i][0] - key) < CONST.POINT\_TOLERANCE:  
 temp1 = **True** temp2 = i  
 **break  
  
 if**(temp1 == **True**):  
 **return** temp2  
  
 **else**:  
 **return** len(multimap)

POLYGON\_CUTTER\_CLASS.PY

**import** src.IntersectorPolygonLine **as** IPL  
  
  
**class** IPolygonCutter:  
 **def** cut\_polygon(self):  
 **pass  
  
 def** get\_cutted\_polygon(self):  
 **pass  
  
 def** get\_intersection\_vertices(self):  
 **pass  
  
 def** get\_new\_points(self):  
 **pass  
  
  
class** PolygonCutter(IPolygonCutter):  
 **def** \_\_init\_\_(self, intersector: IPL.IIntersectorPolygonLine, polygon: IPL.PLG.IPolygon, segment: IPL.PLG.SG.ISegment):  
 self.\_\_intersector = intersector  
 self.\_\_polygon = polygon  
 self.\_\_cutter = segment  
  
 self.\_\_cutted\_polygons = []  
 self.\_\_intersectionVertices = []  
 self.\_\_new\_points = []  
  
 **def** cut\_polygon(self):  
 self.\_\_intersectionVertices = self.\_\_intersector.find\_intersection\_vertices()  
 self.\_\_new\_points = self.\_\_intersector.find\_new\_points()  
 self.\_\_cutted\_polygons = self.\_\_intersector.find\_polygons()  
  
 **return** self.\_\_cutted\_polygons  
  
 **def** get\_cutted\_polygon(self):  
 **return** self.\_\_cutted\_polygons  
  
 **def** get\_new\_points(self):  
 **return** self.\_\_new\_points  
  
 **def** get\_intersection\_vertices(self):  
 **return** self.\_\_intersectionVertices

TEST:

SEGMENT\_TEST.PY

**from** unittest **import** TestCase  
**import** src.SegmentClass **as** SG  
  
  
**class** TestSegmentClass(TestCase):  
 **def** test\_setter\_getter\_methods(self):  
 start = SG.PC.Point(SG.PC.np.array([1, 2]))  
 end = SG.PC.Point(SG.PC.np.array([2, 3]))  
 segment = SG.Segment(SG.PC.Point(SG.PC.np.array([0, 0])), SG.PC.Point(SG.PC.np.array([0, 0])))  
  
 **try**:  
 segment.set\_start(start)  
 segment.set\_end(end)  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(start, segment.get\_start())  
 self.assertEqual(end, segment.get\_end())  
 **except** Exception **as** ex:  
 self.fail()  
  
 segment = SG.Segment(SG.PC.Point(SG.PC.np.array([0, 0])), SG.PC.Point(SG.PC.np.array([0, 0])))  
 **try**:  
 segment.set\_segment(start, end)  
 self.assertEqual(start, segment.get\_start())  
 self.assertEqual(end, segment.get\_end())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_tangent\_method(self):  
 start = SG.PC.Point(SG.PC.np.array([1, 2]))  
 end = SG.PC.Point(SG.PC.np.array([2, 3]))  
 segment = SG.Segment(start, end)  
 tangent = SG.PC.np.array([1,1])  
  
 **try**:  
 SG.PC.np.testing.assert\_array\_equal(tangent, segment.compute\_tangent())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 SG.PC.np.testing.assert\_array\_equal(tangent, segment.get\_tangent())  
 **except** Exception **as** ex:  
 self.fail()

POLYGON\_TEST.PY

**from** unittest **import** TestCase  
**import** src.PolygonClass **as** PLG  
**import** src.PointClass **as** PC  
**import** src.SegmentClass **as** SG  
**import** numpy **as** np  
  
  
**class** TestPolygonClass(TestCase):  
 **def** test\_setter\_getter\_methods(self):  
  
 p0 = PC.Point(np.array([1, 2]))  
 p1 = PC.Point(np.array([3, 2]))  
 p2 = PC.Point(np.array([2, 6]))  
  
 vertices = [p0, p1, p2]  
  
 edges = [SG.Segment(p0, p1), SG.Segment(p1, p2), SG.Segment(p2, p0)]  
  
 **try**:  
 polygon = PLG.Polygon(vertices)  
 self.assertEqual(3, polygon.get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
 **try**:  
 self.assertEqual(polygon.get\_vertices(), vertices)  
 **except** Exception **as** ex:  
 self.fail()  
  
 p3 = PC.Point(np.array([3.5, 2.3]))  
 p4 = PC.Point(np.array([2.2, 3.4]))  
 p5 = PC.Point(np.array([2.5, 3.4]))  
  
 **try**:  
 polygon.set\_vertex\_at\_position(p3, 2)  
 polygon.set\_vertex\_at\_position(p4, 1)  
 polygon.set\_vertex\_at\_position(p5, 0)  
 self.assertEqual(p3, polygon.get\_vertices()[2])  
 self.assertEqual(p4, polygon.get\_vertices()[1])  
 self.assertEqual(p5, polygon.get\_vertices()[0])  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(p3, polygon.get\_vertex\_at\_position(2))  
 self.assertEqual(p4, polygon.get\_vertex\_at\_position(1))  
 self.assertEqual(p5, polygon.get\_vertex\_at\_position(0))  
 self.assertEqual(p3, polygon.get\_last\_vertex())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(edges, polygon.get\_edges())  
 **except** Exception **as** ex:  
 self.fail()  
 id = 3  
 **try**:  
 polygon.next\_vertex\_id(id)  
 self.fail()  
 **except** Exception **as** ex:  
 self.assertEqual(**"Vertex id is not present"**, str(ex))  
  
 id = 2  
  
 **try**:  
 self.assertEqual(0, polygon.next\_vertex\_id(id))  
 **except** Exception **as** ex:  
 self.fail()  
  
 id = 0  
  
 **try**:  
 self.assertEqual(1, polygon.next\_vertex\_id(id))  
 **except** Exception **as** ex:  
 self.fail()  
  
  
  
 **def** test\_insertion\_method(self):  
 p0 = PC.Point(np.array([1, 2]))  
 p1 = PC.Point(np.array([3, 2]))  
 p2 = PC.Point(np.array([2, 6]))  
  
 vertices = [p0, p1, p2]  
  
 polygon = PLG.Polygon(vertices)  
 p3 = PC.Point(np.array([1.5,3.8]))  
  
 **try**:  
 polygon.append\_vertex(p3)  
 self.assertEqual(p3, polygon.get\_vertices()[3])  
 self.assertEqual(4, polygon.get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 p4 = PC.Point(np.array([2.2,1.8]))  
  
 **try**:  
 polygon.insert\_vertex\_at\_position(p4, 1)  
 self.assertEqual(p0, polygon.get\_vertex\_at\_position(0))  
 self.assertEqual(p4, polygon.get\_vertex\_at\_position(1))  
 self.assertEqual(p1, polygon.get\_vertex\_at\_position(2))  
 self.assertEqual(p2, polygon.get\_vertex\_at\_position(3))  
 self.assertEqual(p3, polygon.get\_vertex\_at\_position(4))  
 self.assertEqual(5, polygon.get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_labels\_method(self):  
 p0 = PC.Point(np.array([1, 2]))  
 p1 = PC.Point(np.array([3, 2]))  
 p2 = PC.Point(np.array([2, 6]))  
  
 vertices = [p0, p1, p2]  
 polygon = PLG.Polygon(vertices)  
  
 **try**:  
 polygon.set\_default\_vertices\_label()  
 self.assertEqual(0, polygon.get\_vertex\_at\_position(0).get\_point\_label())  
 self.assertEqual(1, polygon.get\_vertex\_at\_position(1).get\_point\_label())  
 self.assertEqual(2, polygon.get\_vertex\_at\_position(2).get\_point\_label())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 polygon.set\_vertex\_label\_at\_position(0, 1)  
 self.assertEqual(1, polygon.get\_point\_label\_at\_position(0))  
 **except** Exception **as** ex:  
 self.fail()  
  
 vector = [3, 5, 7]  
  
 **try**:  
 polygon.set\_vertices\_label(vector)  
 self.assertEqual(3, polygon.get\_vertex\_at\_position(0).get\_point\_label())  
 self.assertEqual(5, polygon.get\_vertex\_at\_position(1).get\_point\_label())  
 self.assertEqual(7, polygon.get\_vertex\_at\_position(2).get\_point\_label())  
 **except** Exception **as** ex:  
 self.fail()

INTERSECTOR1D1D\_TEST.PY

**from** unittest **import** TestCase  
**from** src **import** Intersector1D1D  
  
  
**class** TestIntersector1D1D(TestCase):  
 **def** test\_parallel\_intersection(self):  
 a = Intersector1D1D.np.array([0, 0])  
 b = Intersector1D1D.np.array([4, 0])  
 c = Intersector1D1D.np.array([1, 0])  
 d = Intersector1D1D.np.array([2, 0])  
  
 intersector = Intersector1D1D.Intersector1D1D()  
  
 **try**:  
 intersector.set\_first\_segment(a, b)  
 intersector.set\_second\_segment(c, d)  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertTrue(intersector.compute\_intersection())  
 self.assertEqual(intersector.type\_intersection(), Intersector1D1D.Type.IntersectionParallelOnSegment)  
 self.assertAlmostEqual(0.5, intersector.second\_parametric\_coordinate())  
 self.assertAlmostEqual(0.25, intersector.first\_parametric\_coordinate())  
 **except** Exception **as** ex:  
 self.fail()  
  
  
 **def** test\_segment\_intersection(self):  
 a = Intersector1D1D.np.array([1, 0])  
 b = Intersector1D1D.np.array([5, 0])  
 c = Intersector1D1D.np.array([3, -6])  
 d = Intersector1D1D.np.array([3, 6])  
  
 intersector = Intersector1D1D.Intersector1D1D()  
  
 **try**:  
 intersector.set\_first\_segment(a, b)  
 intersector.set\_second\_segment(c, d)  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertTrue(intersector.compute\_intersection())  
 self.assertEqual(intersector.type\_intersection(), Intersector1D1D.Type.IntersectionOnSegment)  
 self.assertAlmostEqual(0.5, intersector.first\_parametric\_coordinate())  
 self.assertAlmostEqual(0.5, intersector.second\_parametric\_coordinate())  
 self.assertLessEqual(intersector.first\_parametric\_coordinate(), 1.0)  
 self.assertGreaterEqual(intersector.first\_parametric\_coordinate(), 0.0)  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_on\_line\_intersection(self):  
 a = Intersector1D1D.np.array([3, 6])  
 b = Intersector1D1D.np.array([3, 2])  
 c = Intersector1D1D.np.array([5, 0])  
 d = Intersector1D1D.np.array([1, 0])  
  
 intersector = Intersector1D1D.Intersector1D1D()  
  
 **try**:  
 intersector.set\_first\_segment(a, b)  
 intersector.set\_second\_segment(c, d)  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertTrue(intersector.compute\_intersection())  
 self.assertEqual(intersector.type\_intersection(), Intersector1D1D.Type.IntersectionOnLine)  
 self.assertAlmostEqual(1.5, intersector.first\_parametric\_coordinate())  
 self.assertAlmostEqual(0.5, intersector.second\_parametric\_coordinate())  
 self.assertGreaterEqual(intersector.first\_parametric\_coordinate(), 1.0)  
 **except** Exception **as** ex:  
 self.fail()

INTERSECTOR\_POLYGON\_LINE\_TEST.PY

**from** unittest **import** TestCase  
**import** src.IntersectorPolygonLine **as** IPL  
**import** src.Intersector1D1D **as** INTSEC  
  
  
**class** TestIntersectorPolygonLine(TestCase):  
 **def** test\_find\_intersection\_vertices(self):  
  
 p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.5, 1.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.6, 1.5]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.5, 4.8]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 6.2]))  
 p4 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.2, 4.2]))  
 p5 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.0, 4.0]))  
  
 polygonvertices = [p0, p1, p2, p3, p4, p5]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.0, 3.7]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.1, 5.9]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = IPL.INTSEC.Intersector1D1D()  
  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
  
 **try**:  
 intersection\_points = intersector.find\_intersection\_vertices()  
 self.assertEqual(4, len(intersection\_points))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_1a(self):  
 p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.5, 1.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.6, 1.5]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.5, 4.8]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 6.2]))  
 p4 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.2, 4.2]))  
 p5 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.0, 4.0]))  
  
 polygonvertices = [p0, p1, p2, p3, p4, p5]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.0, 3.7]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.1, 5.9]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
 **try**:  
 self.assertEqual(3, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
 **try**:  
 self.assertEqual(8, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[1].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[2].get\_number\_vertices())  
 self.assertEqual(10, new\_extended\_polygons[0].get\_number\_vertices())  
 self.assertEqual(4, new\_extended\_polygons[1].get\_number\_vertices())  
 self.assertEqual(4, new\_extended\_polygons[2].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
 **try**:  
 self.assertEqual(11, new\_extended\_polygons[0].get\_point\_label\_at\_position(3))  
 self.assertEqual(7, new\_extended\_polygons[0].get\_point\_label\_at\_position(4))  
 self.assertEqual(10, new\_extended\_polygons[0].get\_point\_label\_at\_position(5))  
 self.assertEqual(4, new\_extended\_polygons[0].get\_point\_label\_at\_position(6))  
 self.assertEqual(9, new\_extended\_polygons[0].get\_point\_label\_at\_position(7))  
 self.assertEqual(6, new\_extended\_polygons[0].get\_point\_label\_at\_position(8))  
 self.assertEqual(8, new\_extended\_polygons[0].get\_point\_label\_at\_position(9))  
  
 self.assertEqual(3, new\_extended\_polygons[1].get\_point\_label\_at\_position(0))  
 self.assertEqual(10, new\_extended\_polygons[1].get\_point\_label\_at\_position(1))  
 self.assertEqual(7, new\_extended\_polygons[1].get\_point\_label\_at\_position(2))  
 self.assertEqual(11, new\_extended\_polygons[1].get\_point\_label\_at\_position(3))  
  
 self.assertEqual(5, new\_extended\_polygons[2].get\_point\_label\_at\_position(0))  
 self.assertEqual(8, new\_extended\_polygons[2].get\_point\_label\_at\_position(1))  
 self.assertEqual(6, new\_extended\_polygons[2].get\_point\_label\_at\_position(2))  
 self.assertEqual(9, new\_extended\_polygons[2].get\_point\_label\_at\_position(3))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygon\_1b(self): *#same polygon with swapped point in the segment* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.5, 1.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.6, 1.5]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.5, 4.8]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 6.2]))  
 p4 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.2, 4.2]))  
 p5 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.0, 4.0]))  
  
 polygonvertices = [p0, p1, p2, p3, p4, p5]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.1, 5.9]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.0, 3.7]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(3, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
 **try**:  
 self.assertEqual(8, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[1].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[2].get\_number\_vertices())  
 self.assertEqual(10, new\_extended\_polygons[0].get\_number\_vertices())  
 self.assertEqual(4, new\_extended\_polygons[1].get\_number\_vertices())  
 self.assertEqual(4, new\_extended\_polygons[2].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(8, new\_extended\_polygons[0].get\_point\_label\_at\_position(3))  
 self.assertEqual(9, new\_extended\_polygons[0].get\_point\_label\_at\_position(5))  
 self.assertEqual(4, new\_extended\_polygons[0].get\_point\_label\_at\_position(6))  
 self.assertEqual(6, new\_extended\_polygons[0].get\_point\_label\_at\_position(4))  
 self.assertEqual(10, new\_extended\_polygons[0].get\_point\_label\_at\_position(7))  
 self.assertEqual(7, new\_extended\_polygons[0].get\_point\_label\_at\_position(8))  
 self.assertEqual(11, new\_extended\_polygons[0].get\_point\_label\_at\_position(9))  
  
 self.assertEqual(3, new\_extended\_polygons[1].get\_point\_label\_at\_position(0))  
 self.assertEqual(9, new\_extended\_polygons[1].get\_point\_label\_at\_position(1))  
 self.assertEqual(6, new\_extended\_polygons[1].get\_point\_label\_at\_position(2))  
 self.assertEqual(8, new\_extended\_polygons[1].get\_point\_label\_at\_position(3))  
  
 self.assertEqual(5, new\_extended\_polygons[2].get\_point\_label\_at\_position(0))  
 self.assertEqual(11, new\_extended\_polygons[2].get\_point\_label\_at\_position(1))  
 self.assertEqual(7, new\_extended\_polygons[2].get\_point\_label\_at\_position(2))  
 self.assertEqual(10, new\_extended\_polygons[2].get\_point\_label\_at\_position(3))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_1c(self): *# start/end point consecutive* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.5, 1.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.6, 1.5]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.5, 4.8]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 6.2]))  
 p4 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.2, 4.2]))  
 p5 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.0, 4.0]))  
  
 polygonvertices = [p0, p1, p2, p3, p4, p5]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.0, 3.7]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.0, 3.7]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.1, 5.9]))  
 p3 = p1.get\_coordinates() \* 0.9 + p2.get\_coordinates() \* 0.1  
 end = IPL.PLG.SG.PC.Point(p3)  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(3, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(8, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[1].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[2].get\_number\_vertices())  
 self.assertEqual(10, new\_extended\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_extended\_polygons[1].get\_number\_vertices())  
 self.assertEqual(5, new\_extended\_polygons[2].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(11, new\_extended\_polygons[0].get\_point\_label\_at\_position(3))  
 self.assertEqual(10, new\_extended\_polygons[0].get\_point\_label\_at\_position(4))  
 self.assertEqual(4, new\_extended\_polygons[0].get\_point\_label\_at\_position(5))  
 self.assertEqual(9, new\_extended\_polygons[0].get\_point\_label\_at\_position(6))  
 self.assertEqual(7, new\_extended\_polygons[0].get\_point\_label\_at\_position(7))  
 self.assertEqual(6, new\_extended\_polygons[0].get\_point\_label\_at\_position(8))  
 self.assertEqual(8, new\_extended\_polygons[0].get\_point\_label\_at\_position(9))  
  
 self.assertEqual(3, new\_extended\_polygons[1].get\_point\_label\_at\_position(0))  
 self.assertEqual(10, new\_extended\_polygons[1].get\_point\_label\_at\_position(1))  
 self.assertEqual(11, new\_extended\_polygons[1].get\_point\_label\_at\_position(2))  
  
 self.assertEqual(5, new\_extended\_polygons[2].get\_point\_label\_at\_position(0))  
 self.assertEqual(8, new\_extended\_polygons[2].get\_point\_label\_at\_position(1))  
 self.assertEqual(6, new\_extended\_polygons[2].get\_point\_label\_at\_position(2))  
 self.assertEqual(7, new\_extended\_polygons[2].get\_point\_label\_at\_position(3))  
 self.assertEqual(9, new\_extended\_polygons[2].get\_point\_label\_at\_position(4))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_1d(self): *#start in, end out* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.5, 1.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.6, 1.5]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.5, 4.8]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 6.2]))  
 p4 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.2, 4.2]))  
 p5 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.0, 4.0]))  
  
 polygonvertices = [p0, p1, p2, p3, p4, p5]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.0, 3.7]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.06, 4.81]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(3, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(8, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[1].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[2].get\_number\_vertices())  
 self.assertEqual(9, new\_extended\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_extended\_polygons[1].get\_number\_vertices())  
 self.assertEqual(4, new\_extended\_polygons[2].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(11, new\_extended\_polygons[0].get\_point\_label\_at\_position(3))  
 self.assertEqual(10, new\_extended\_polygons[0].get\_point\_label\_at\_position(4))  
 self.assertEqual(4, new\_extended\_polygons[0].get\_point\_label\_at\_position(5))  
 self.assertEqual(9, new\_extended\_polygons[0].get\_point\_label\_at\_position(6))  
 self.assertEqual(6, new\_extended\_polygons[0].get\_point\_label\_at\_position(7))  
 self.assertEqual(8, new\_extended\_polygons[0].get\_point\_label\_at\_position(8))  
  
 self.assertEqual(3, new\_extended\_polygons[1].get\_point\_label\_at\_position(0))  
 self.assertEqual(10, new\_extended\_polygons[1].get\_point\_label\_at\_position(1))  
 self.assertEqual(11, new\_extended\_polygons[1].get\_point\_label\_at\_position(2))  
  
 self.assertEqual(5, new\_extended\_polygons[2].get\_point\_label\_at\_position(0))  
 self.assertEqual(8, new\_extended\_polygons[2].get\_point\_label\_at\_position(1))  
 self.assertEqual(6, new\_extended\_polygons[2].get\_point\_label\_at\_position(2))  
 self.assertEqual(9, new\_extended\_polygons[2].get\_point\_label\_at\_position(3))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_1e(self): *#start out, end out* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.5, 1.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.6, 1.5]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.5, 4.8]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 6.2]))  
 p4 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.2, 4.2]))  
 p5 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.0, 4.0]))  
  
 polygonvertices = [p0, p1, p2, p3, p4, p5]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.06, 4.81]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.8, 6.1]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(3, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(8, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[1].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[2].get\_number\_vertices())  
 self.assertEqual(8, new\_extended\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_extended\_polygons[1].get\_number\_vertices())  
 self.assertEqual(3, new\_extended\_polygons[2].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(11, new\_extended\_polygons[0].get\_point\_label\_at\_position(3))  
 self.assertEqual(10, new\_extended\_polygons[0].get\_point\_label\_at\_position(4))  
 self.assertEqual(4, new\_extended\_polygons[0].get\_point\_label\_at\_position(5))  
 self.assertEqual(9, new\_extended\_polygons[0].get\_point\_label\_at\_position(6))  
 self.assertEqual(8, new\_extended\_polygons[0].get\_point\_label\_at\_position(7))  
  
 self.assertEqual(3, new\_extended\_polygons[1].get\_point\_label\_at\_position(0))  
 self.assertEqual(10, new\_extended\_polygons[1].get\_point\_label\_at\_position(1))  
 self.assertEqual(11, new\_extended\_polygons[1].get\_point\_label\_at\_position(2))  
  
 self.assertEqual(5, new\_extended\_polygons[2].get\_point\_label\_at\_position(0))  
 self.assertEqual(8, new\_extended\_polygons[2].get\_point\_label\_at\_position(1))  
 self.assertEqual(9, new\_extended\_polygons[2].get\_point\_label\_at\_position(2))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_2a(self): *#pentagon* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.5, 1.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 2.1]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.4, 4.2]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.6, 4.2]))  
 p4 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.0, 2.1]))  
  
 polygonvertices = [p0, p1, p2, p3, p4]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.4, 2.75]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.6, 2.2]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(2, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(4, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(4, new\_polygons[1].get\_number\_vertices())  
 self.assertEqual(6, new\_extended\_polygons[0].get\_number\_vertices())  
 self.assertEqual(6, new\_extended\_polygons[1].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(0, new\_extended\_polygons[0].get\_point\_label\_at\_position(0))  
 self.assertEqual(1, new\_extended\_polygons[0].get\_point\_label\_at\_position(1))  
 self.assertEqual(6, new\_extended\_polygons[0].get\_point\_label\_at\_position(2))  
 self.assertEqual(5, new\_extended\_polygons[0].get\_point\_label\_at\_position(3))  
 self.assertEqual(7, new\_extended\_polygons[0].get\_point\_label\_at\_position(4))  
 self.assertEqual(4, new\_extended\_polygons[0].get\_point\_label\_at\_position(5))  
  
 self.assertEqual(1, new\_extended\_polygons[1].get\_point\_label\_at\_position(0))  
 self.assertEqual(2, new\_extended\_polygons[1].get\_point\_label\_at\_position(1))  
 self.assertEqual(3, new\_extended\_polygons[1].get\_point\_label\_at\_position(2))  
 self.assertEqual(7, new\_extended\_polygons[1].get\_point\_label\_at\_position(3))  
 self.assertEqual(5, new\_extended\_polygons[1].get\_point\_label\_at\_position(4))  
 self.assertEqual(6, new\_extended\_polygons[1].get\_point\_label\_at\_position(5))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_2b(self): *#pentagon with segment passing through 2 vertices* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.5, 1.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 2.1]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.4, 4.2]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.6, 4.2]))  
 p4 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.0, 2.1]))  
  
 polygonvertices = [p0, p1, p2, p3, p4]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 2.1]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.0, 2.1]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(2, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(3, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(4, new\_polygons[1].get\_number\_vertices())  
 self.assertEqual(3, new\_extended\_polygons[0].get\_number\_vertices())  
 self.assertEqual(4, new\_extended\_polygons[1].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(0, new\_extended\_polygons[0].get\_point\_label\_at\_position(0))  
 self.assertEqual(1, new\_extended\_polygons[0].get\_point\_label\_at\_position(1))  
 self.assertEqual(4, new\_extended\_polygons[0].get\_point\_label\_at\_position(2))  
  
 self.assertEqual(1, new\_extended\_polygons[1].get\_point\_label\_at\_position(0))  
 self.assertEqual(2, new\_extended\_polygons[1].get\_point\_label\_at\_position(1))  
 self.assertEqual(3, new\_extended\_polygons[1].get\_point\_label\_at\_position(2))  
 self.assertEqual(4, new\_extended\_polygons[1].get\_point\_label\_at\_position(3))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_3(self): *# professore D'Auria's test polygon* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 0.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([8.0, 0.0]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.64, 1.5]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([8.93, 4.48]))  
 p4 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.18, 5.36]))  
 p5 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.52, 9.22]))  
 p6 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 5.0]))  
 p7 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([6.0, 4.0]))  
  
 polygonvertices = [p0, p1, p2, p3, p4, p5, p6, p7]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([7.28, -0.72]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.2, 9.14]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(5, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(6, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[1].get\_number\_vertices())  
 self.assertEqual(6, new\_polygons[2].get\_number\_vertices())  
 self.assertEqual(6, new\_polygons[3].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[4].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_4(self): *# professore D'Auria's test polygon with segment that ends on a vertex* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 0.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([8.0, 0.0]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.64, 1.5]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([8.93, 4.48]))  
 p4 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.18, 5.36]))  
 p5 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.52, 9.22]))  
 p6 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 5.0]))  
 p7 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([6.0, 4.0]))  
  
 polygonvertices = [p0, p1, p2, p3, p4, p5, p6, p7]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([7.28, -0.72]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.52, 9.22])) *#corresponding to polygon\_vertices[5]* segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(3, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(10, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[1].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[2].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_5(self): *#square test with segment.end outside* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 0.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 0.0]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 4.0]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 4.0]))  
  
 polygonvertices = [p0, p1, p2, p3]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.0, 5.0]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.0, 1.0]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(2, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(5, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[1].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_5b(self): *#square test with segment.end inside* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 0.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 0.0]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 4.0]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 4.0]))  
  
 polygonvertices = [p0, p1, p2, p3]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.48, 3.03]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.03, 3.63]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(2, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(5, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[1].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_5c(self): *#square test with segment.start inside and segment.end outside* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 0.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 0.0]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 4.0]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 4.0]))  
  
 polygonvertices = [p0, p1, p2, p3]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.03, 3.63]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.0, 5.0]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(2, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(5, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[1].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_5d(self): *#square test with segment.start inside and segment.end outside* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 0.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 0.0]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 4.0]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 4.0]))  
  
 polygonvertices = [p0, p1, p2, p3]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.48, 3.03]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.0, 1.0]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(2, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(5, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[1].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **def** test\_find\_polygons\_6(self): *#rectangle* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.0, 1.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.0, 1.0]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([5.0, 3.1]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([1.0, 3.1]))  
  
 polygonvertices = [p0, p1, p2, p3]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.0, 1.2]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 3.0]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(2, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(4, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(4, new\_polygons[1].get\_number\_vertices())  
 self.assertEqual(6, new\_extended\_polygons[0].get\_number\_vertices())  
 self.assertEqual(6, new\_extended\_polygons[1].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()  
  
 **try**:  
 self.assertEqual(0, new\_extended\_polygons[0].get\_point\_label\_at\_position(0))  
 self.assertEqual(6, new\_extended\_polygons[0].get\_point\_label\_at\_position(1))  
 self.assertEqual(4, new\_extended\_polygons[0].get\_point\_label\_at\_position(2))  
 self.assertEqual(5, new\_extended\_polygons[0].get\_point\_label\_at\_position(3))  
 self.assertEqual(7, new\_extended\_polygons[0].get\_point\_label\_at\_position(4))  
 self.assertEqual(3, new\_extended\_polygons[0].get\_point\_label\_at\_position(5))  
  
 self.assertEqual(1, new\_extended\_polygons[1].get\_point\_label\_at\_position(0))  
 self.assertEqual(2, new\_extended\_polygons[1].get\_point\_label\_at\_position(1))  
 self.assertEqual(7, new\_extended\_polygons[1].get\_point\_label\_at\_position(2))  
 self.assertEqual(5, new\_extended\_polygons[1].get\_point\_label\_at\_position(3))  
 self.assertEqual(4, new\_extended\_polygons[1].get\_point\_label\_at\_position(4))  
 self.assertEqual(6, new\_extended\_polygons[1].get\_point\_label\_at\_position(5))  
 **except** Exception **as** ex:  
 self.fail()  
  
 *# professor D'Auria's polygon test* **def** test\_find\_polygons\_7(self): *#professor D'Auria's polygon test* p0 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([2.0, -2.0]))  
 p1 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, -1.0]))  
 p2 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.0, 1.0]))  
 p3 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 2.0]))  
 p4 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.0, 2.0]))  
 p5 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([3.0, 3.0]))  
 p6 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([-1.0, 3.0]))  
 p7 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([-3.0, 1.0]))  
 p8 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([0.0, 0.0]))  
 p9 = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([-3.0, -2.0]))  
  
 polygonvertices = [p0, p1, p2, p3, p4, p5, p6, p7, p8, p9]  
 polygon = IPL.PLG.Polygon(polygonvertices)  
 start = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([-4.0, -4.0]))  
 end = IPL.PLG.SG.PC.Point(IPL.INTSEC.np.array([4.0, 4.0]))  
 segment = IPL.PLG.SG.Segment(start, end)  
  
 intersector1d1d = INTSEC.Intersector1D1D()  
 intersector = IPL.intersector\_polygon\_line(intersector1d1d, polygon, segment)  
 intersection\_points = intersector.find\_intersection\_vertices()  
 new\_polygons = intersector.find\_polygons()  
 new\_points = intersector.find\_new\_points()  
 new\_extended\_polygons = intersector.get\_extended\_polygons()  
  
 **try**:  
 self.assertEqual(4, len(new\_polygons))  
 **except** Exception **as** ex:  
 self.fail()  
 **try**:  
 self.assertEqual(6, new\_polygons[0].get\_number\_vertices())  
 self.assertEqual(7, new\_polygons[1].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[2].get\_number\_vertices())  
 self.assertEqual(3, new\_polygons[3].get\_number\_vertices())  
 **except** Exception **as** ex:  
 self.fail()