**Slope Stability**

Problem: **Find intersection of two polygons (using Sutherland – Hodgman algorithm)**

Outcome: ***Ordered* sequence of nodes of polygon that is result of intersection of big polygon with smaller polygons (soil layers).**

1. Problem was narrowed down to finding intersection of big **convex** polygon and arbitrary (concave or convex) polygons (soil layers)

The Algorithm I choose to implement this task is Sutherland – Hodgman algorithm which is straightforward, small and fits our task:

<https://en.wikipedia.org/wiki/Sutherland%E2%80%93Hodgman_algorithm>

The pseudocode for algorithm:

List outputList = subjectPolygon;

  for (Edge clipEdge in clipPolygon) do

     List inputList = outputList;

     outputList.clear();

     Point S = inputList.last;

     for (Point E in inputList) do

        if (E inside clipEdge) then

           if (S not inside clipEdge) then

              outputList.add(ComputeIntersection(S,E,clipEdge));

           end if

           outputList.add(E);

        else if (S inside clipEdge) then

           outputList.add(ComputeIntersection(S,E,clipEdge));

        end if

        S = E;

     done

  done

2) To implement this pseudo code we need some auxiliary functions.

**i) function “***computeIntersection”*  **that computes intersection between SE segment and line that contains “clipEdge” of convex polygon**

Intersection is calculated based on formula of intersection (P1,P2) between two lines represented by two points on each line:

<https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection>

Case where determinant is equal to zero (lines are parallel) is almost impossible, because we use this function only when points S and E are on different half planes formed from line AB, and distance from points S and E is more than 0.00001. So if case with det==0 can be probably omitted.

**C++ code: of function:**

*//function computeIntersection returns node with coordinates of intersection*

*//between lines formed by side of convex polygon AB and line segment SE*

*node computeIntersection (node A, node B, node S, node E)*

*{*

*node intersect;*

*double det = (A.x-B.x)\*(S.y-E.y) - (A.y-B.y)\*(S.x-E.x);*

*std::cout << "Determinat "<<det<<"\n";*

*double a = (A.x\*B.y-A.y\*B.x);*

*double b = (S.x\*E.y-S.y\*E.x);*

*if (det==0.0) //case of determinant equal 0 (line are almost parallel - very rear)*

*{*

*std::cout << "Determinat-zero case: "<<det<<"\n";*

*double d1 = distLinePoint(A,B,S);*

*double d2 = distLinePoint(A,B,E);*

*intersect.x = (d1\*E.x + d2\*S.x)/(d1+d2); //intersect.x = (S.x+E.x)/2; simly takes half distanse not very accurate*

*intersect.y = (d1\*E.y + d2\*S.y)/(d1+d2); //intersect.y = (S.y+E.y)/2;*

*}*

*else {*

*intersect.x = (a\*(S.x-E.x)- b\*(A.x-B.x))/det;*

*intersect.y = (a\*(S.y-E.y)- b\*(A.y-B.y))/det;*

*std::cout << "Determinat - non-zero: "<<det<<"\n";*

*}*

*return intersect;*

*}*

**ii) Function “*printNode*” prints coordinate of Node passed to it**

*//function "printNode" prins out node*

*void printNode (node A)*

*{*

*std::cout<<" ("<<A.x<<", "<<A.y<<") ";*

*}*

**iii) Function “printPolygon” prints coordinates of nodes of polygon poly**

*//Function "printPolygon" prints nodes using "printNode function"*

*void printPolygon (std::list<node> poly)*

*{*

*for\_each (poly.begin(), poly.end(), printNode);*

*std::cout<<"\n";*

*}*

**iiii) Function “distLinePoint” that computes shortest distance between point Z and line AB**

Formula which is used is from Wikipedia:

<https://en.wikipedia.org/wiki/Distance_from_a_point_to_a_line>

formula used: distance between point and line defined by two points.

In case if AB (distance between AB) is 0 we use lowest possible float number: *std::numeric\_limits<float>::min()*

**C++ code:**

*//function distLinePoint calculates distance between*

*//line AB and point Z*

*double distLinePoint (node A, node B, node Z )*

*{*

*double S = abs((B.y-A.y)\*Z.x - (B.x-A.x)\*Z.y + B.x\*A.y - B.y\*A.x);*

*double AB = sqrt( (B.y-A.y)\*(B.y-A.y) + (B.x-A.x)\*(B.x-A.x) );*

*if (AB==0) AB = std::numeric\_limits<float>::min();*

*return S/AB;*

*}*

**iiiii) Function “*halfPlaneOrient*” that computes if point X is on the right side of directed edge AB, on the left side of directed edge AB or X belongs to line AB.**

Function returns -1 if X lies on right side of from directed edge (line) AB, returns 0 if x belongs line AB, and returns 1 if X is on the left side from edge AB

Logic of function is based on calculating determinant of two vectors AB and AX.

<http://mathworld.wolfram.com/VectorOrientation.html>

If determinant of is less than 0: then AB and AX are negatively oriented and angle between AB and AX is between 180 and 360 and X is on left side of edge AB.

If determinant of is greater than 0: then AB and AX are positively oriented and angle between AB and AX is between 0 and 180 and X is on left side of edge AB.

If determinant of is equal 0: then X lies on line AB (because angle between AB and AX is 0 or 180).

Determinant is calculated through coordinates of points A,B and X

D=(X.x - A.x)\*(B.y-A.y) - (X.y-A.y)\*(B.x-A.x)

**C++ code for this function:**

*/function halfPlaneOrient returns -1 if point X is on left side of line AB*

*// returns 0 if X belongs line AB and returns -1 if x is on right side of plane AB*

*int halfPlaneOrient (node A, node B, node X)*

*{*

*double orient = (X.x - A.x)\*(B.y-A.y) - (X.y-A.y)\*(B.x-A.x);*

*if (orient >0.000001) return 1;*

*else if (orient < -0.000001) return -1;*

*else return 0;*

*}*

iiiiii) **Function “PolygonWinding” check if polygon is not convex or has collinear sides, if yes prints error message and exits. If polygon is OK (is convex) then function returns: -1 if polygon is winded counterclockwise; and returns +1 if polygon is winded clockwise.**

For determining if polygon is convex or not I used following logic:

For each side AB and next point in polygon C, I determined orientation of Vector AB and AC. For clockwise winded polygon **all** those pair of vectors will have angle between them from 0 degree to 180 degree. And halfPlaneOrient function for points A,B,C should always return:+1.

For counterclockwise winded polygon **all** those pair of vectors will have angle between then from 180 degree to 360 degree. And halfPlaneOrient function for points A,B,C should always return: -1.

<http://mathworld.wolfram.com/ConvexPolygon.html>

<http://mathworld.wolfram.com/VectorOrientation.html>

**C++ code for function:**

*//function "poligonWinding" checkes polygon for concavity*

*//if polygon is not convex it exits with error message "not convex"*

*//if polygon has collinear sides it exits with eror message*

*//if poligon is convex it returns +1 if poligon is winded CounterClockwise*

*// returns -1 if polygon is winded Clockwise*

*int polygonWinding(std::list<node> poly)*

*{*

*int halfPlaneSign = 0, halfPlaneSignNew = 0; //initial value of winding numbers*

*node startNode = poly.back(); //stastNode is beginning of current side of polygon - last node in polygon list*

*node endNode, nextNode; //endNode is end of current side of polygon, nextNode - next node after current side*

*int i=0;//for testin only*

*for (std::list<node>::iterator it=poly.begin(); it != poly.end(); ++it)*

*{*

*endNode = \*it;*

*i++; //for testing only*

*if ( std::next(it,1) == poly.end() ) nextNode = poly.front();*

*else nextNode = \*(std::next(it));*

*halfPlaneSignNew = halfPlaneOrient( startNode, endNode, nextNode); //calculating sign of current winding*

*std::cout<<"\n "<<i <<") start is: ";*

*printNode(startNode);*

*std::cout<<" end is ";*

*printNode(endNode);*

*std::cout<<"next node is ";*

*printNode(nextNode);*

*std::cout<<"\n HalfPlaneSign = "<<halfPlaneSign<<". HalfPlaneSignNew ="<<halfPlaneSignNew<<". \n";*

*if (halfPlaneSignNew == 0 )*

*{std::cout<<"\n colinear nodes detecter - ERROR\n"; exit (EXIT\_FAILURE); } // Collinear nodes detected*

*else if (halfPlaneSign\*halfPlaneSignNew < 0)*

*{std::cout<<"\n Polygon is not convex - ERROR\n"; exit (EXIT\_FAILURE); } // Non-convex polygon detected, product of two*

*heighboring windings less than 0*

*//winding (angles between neighbour sides) are same sign - switch to next side*

*halfPlaneSign = halfPlaneSignNew;*

*startNode = endNode;*

*}*

*return halfPlaneSign;*

*}*

iiiiiii)**Function PolygonClip implement main part of Sutherland – Hodgman algorithm**

<https://en.wikipedia.org/wiki/Sutherland%E2%80%93Hodgman_algorithm>

**C++ code for function:**

*//Function "PolygonClip" takes 3 arguments.*

*//convexPoly - is big convex polygon.*

*//soilPoly - polygon representins soil levels*

*//outoutPoly - emty polygon which will be modified by and returned with sequens of Nodes representing intersection of two polygons*

*void polygonClip (std::list<node> & convexPoly, std::list<node> & soilPoly, std::list<node> & outputPoly)*

*{*

*outputPoly.assign(soilPoly.begin(), soilPoly.end()); //assigning soil poligon to output*

*node S, E; // declaring S and E are start and end points of each side of clipping "soilPoly"*

*node startOfEdge = convexPoly.back(); //start point of edge of "convexPoly" polygon*

*node endOfEdge; //end point of edge of "convexPoly" polygon*

*std::list<node> inputList; //input list for each step of algorithm*

*int sOrient, eOrient; // here orientations (winding) of points S and E according to edge of "convexPoly" will be stored*

*int winding = polygonWinding(convexPoly); //winding of big "convexPoly"*

*//this for loop goes through edges of big "convexPoly" polygon*

*for (std::list<node>::iterator it1=convexPoly.begin(); it1 != convexPoly.end(); ++it1)*

*{*

*endOfEdge = \*it1; //ending point of edge (for looping)*

*inputList = outputPoly;*

*outputPoly.clear(); //clearing output polygon list*

*S=inputList.back(); // last element of inputList is assigned to S - starting point*

*//this for loop goes through nodes of inputList*

*for (std::list<node>::iterator it2=inputList.begin(); it2 != inputList.end(); ++it2)*

*{*

*E = \*it2; //end of tracing node*

*sOrient = halfPlaneOrient(startOfEdge, endOfEdge, S); //orientation of point S*

*eOrient = halfPlaneOrient(startOfEdge, endOfEdge, E); //orientation of point E*

*if (eOrient == winding) //if point E is inside convexPolygon halfplane*

*{*

*if (sOrient != winding) //if point S is outside convexPolygon halfplane*

*{ outputPoly.push\_back( computeIntersection(startOfEdge, endOfEdge, S, E));*

*} //push intersection in output*

*outputPoly.push\_back(E); // push E point in output*

*}*

*else if (sOrient == winding) // if point S inside convexPolygon halfplane*

*{outputPoly.push\_back( computeIntersection(startOfEdge, endOfEdge, S, E)); //push intersection in output*

*}*

*S=E;*

*}*

*startOfEdge = endOfEdge;*

*}*

*return;*

*}*

**Copmplete code of finding intersection of two polygons (Satherland-Hodgman Algorithm):**

*http://rextester.com/ITRWA36115*

*//g++ 5.4.0*

*#include <iostream>*

*#include <limits>*

*#include <math.h>*

*#include <stdlib.h>*

*#include <list>*

*#include <algorithm>*

*//Node structure*

*struct node {*

*double x;*

*double y;};*

*//function "printNode" prins out node*

*void printNode (node A)*

*{*

*std::cout<<" ("<<A.x<<", "<<A.y<<") ";*

*}*

*//Function "printPolygon" prints nodes using "printNode function"*

*void printPolygon (std::list<node> poly)*

*{*

*for\_each (poly.begin(), poly.end(), printNode);*

*std::cout<<"\n";*

*}*

*//function distTwoPoints returns distanse between two nodes*

*double distTwoPoints (node A, node B)*

*{*

*double dist = sqrt( (B.y-A.y)\*(B.y-A.y) + (B.x-A.x)\*(B.x-A.x) );*

*return dist;*

*}*

*//Function "RemoveDuplicateNodes" removes identical nodes in list next to each other*

*void removeDuplicateNodes (std::list<node> & outputPoly)*

*{*

*node start, end; //start and end are two consequetive node to check for equality*

*start = outputPoly.back();*

*std::list<node>::iterator it = outputPoly.begin();*

*while (it != outputPoly.end())*

*{*

*end = \*it;*

*if( distTwoPoints(start,end) < 1.0e-10 )*

*{it = outputPoly.erase (it); //remove current element if previous was same*

*}*

*else*

*{*

*start = \*it; //swich start element and increment itarator*

*++it;*

*}*

*}*

*}*

*//function distLinePoint calculates distance between*

*//line AB and point Z*

*double distLinePoint (node A, node B, node Z )*

*{*

*double S = fabs((B.y-A.y)\*Z.x - (B.x-A.x)\*Z.y + B.x\*A.y - B.y\*A.x);*

*double AB = sqrt( (B.y-A.y)\*(B.y-A.y) + (B.x-A.x)\*(B.x-A.x) );*

*if (AB==0) AB = std::numeric\_limits<float>::min();*

*return S/AB;*

*}*

*//function halfPlaneOrient returns -1 if point X is on left side of line AB*

*//returns 0 if X belongs line AB and returns -1 if x is on right side of plane AB*

*int halfPlaneOrient (node A, node B, node X)*

*{*

*double orient = (X.x - A.x)\*(B.y-A.y) - (X.y-A.y)\*(B.x-A.x);*

*if (orient >0.0000001) return 1;*

*else if (orient < -0.0000001) return -1;*

*else return 0;*

*}*

*//Function "poligonWinding" returns polygon winding and checkes polygon for concavity*

*//if polygon is not convex it exits with error message "not convex"*

*//if polygon has collinear sides it exits with eror message*

*//if poligon is convex it returns +1 if poligon is winded CounterClockwise*

*// returns -1 if polygon is winded Clockwise*

*int polygonWinding(std::list<node> poly)*

*{*

*int halfPlaneSign = 0, halfPlaneSignNew = 0; //initial value of winding numbers*

*node startNode = poly.back(); //stastNode is beginning of current side of polygon - last node in polygon list*

*node endNode, nextNode; //endNode is end of current side of polygon, nextNode - next node after current side*

*for (std::list<node>::iterator it=poly.begin(); it != poly.end(); ++it)*

*{*

*endNode = \*it;*

*if ( std::next(it,1) == poly.end() ) nextNode = poly.front(); //if nextNode(node after current) is beyond last element in list - then nextNode is first element*

*else nextNode = \*(std::next(it));*

*halfPlaneSignNew = halfPlaneOrient( startNode, endNode, nextNode); //calculating sign of current winding*

*if (halfPlaneSignNew == 0 )*

*{std::cout<<"\n colinear nodes detecter - ERROR\n"; exit (EXIT\_FAILURE); } // Collinear nodes detected*

*else if (halfPlaneSign\*halfPlaneSignNew < 0)*

*{std::cout<<"\n Polygon is not convex - ERROR\n"; exit (EXIT\_FAILURE); } // Non-convex polygon detected, product of two heighboring windings less than 0*

*//winding (angles between neighbour sides) are same sign - switch to next side*

*halfPlaneSign = halfPlaneSignNew;*

*startNode = endNode;*

*}*

*return halfPlaneSign;*

*}*

*//function computeIntersection returns node with coordinates of intersection*

*//between lines formed by side of convex polygon AB and line segment SE*

*node computeIntersection (node A, node B, node S, node E)*

*{*

*node intersect;*

*double det = (A.x-B.x)\*(S.y-E.y) - (A.y-B.y)\*(S.x-E.x);*

*double a = (A.x\*B.y-A.y\*B.x);*

*double b = (S.x\*E.y-S.y\*E.x);*

*if (det==0.0) //case of determinant equal 0 (line are almost parallel - very rear)*

*{*

*std::cout << "Determinat-zero case: "<<det<<"\n";*

*double d1 = distLinePoint(A,B,S);*

*double d2 = distLinePoint(A,B,E);*

*intersect.x = (d1\*E.x + d2\*S.x)/(d1+d2); //intersect.x = (S.x+E.x)/2; simly takes half distanse not very accurate*

*intersect.y = (d1\*E.y + d2\*S.y)/(d1+d2); //intersect.y = (S.y+E.y)/2;*

*}*

*else {*

*intersect.x = (a\*(S.x-E.x)- b\*(A.x-B.x))/det;*

*intersect.y = (a\*(S.y-E.y)- b\*(A.y-B.y))/det;*

*std::cout << "Determinat - non-zero: "<<det<<"\n";*

*}*

*return intersect;*

*}*

*//Function "PolygonClip" takes 3 arguments.*

*//convexPoly - is big convex polygon.*

*//soilPoly - polygon representins soil levels*

*//outoutPoly - emty polygon which will be modified by and returned with sequens of Nodes representing intersection of two polygons*

*void polygonClip (std::list<node> & convexPoly, std::list<node> & soilPoly, std::list<node> & outputPoly)*

*{*

*outputPoly.assign(soilPoly.begin(), soilPoly.end()); //assigning soil poligon to output*

*node S, E; // declaring S and E are start and end points of each side of clipping "soilPoly"*

*node startOfEdge = convexPoly.back(); //start point of edge of "convexPoly" polygon*

*node endOfEdge; //end point of edge of "convexPoly" polygon*

*std::list<node> inputList; //input list for each step of algorithm*

*int sOrient, eOrient; // here orientations (winding) of points S and E according to edge of "convexPoly" will be stored*

*int winding = polygonWinding(convexPoly); //winding of big "convexPoly"*

*//this for loop goes through edges of big "convexPoly" polygon*

*for (std::list<node>::iterator it1=convexPoly.begin(); it1 != convexPoly.end(); ++it1)*

*{*

*endOfEdge = \*it1; //ending point of edge (for looping)*

*inputList = outputPoly;*

*outputPoly.clear(); //clearing output polygon list*

*S=inputList.back(); // last element of inputList is assigned to S - starting point*

*//this for loop goes through nodes of inputList*

*for (std::list<node>::iterator it2=inputList.begin(); it2 != inputList.end(); ++it2)*

*{*

*E = \*it2; //end of tracing node*

*sOrient = halfPlaneOrient(startOfEdge, endOfEdge, S); //orientation of point S*

*eOrient = halfPlaneOrient(startOfEdge, endOfEdge, E); //orientation of point E*

*if (eOrient == winding) //if point E is inside convexPolygon halfplane*

*{*

*if (sOrient != winding) //if point S is outside convexPolygon halfplane*

*{ outputPoly.push\_back( computeIntersection(startOfEdge, endOfEdge, S, E));} //push intersection in output*

*outputPoly.push\_back(E); // push E point in output*

*}*

*else if (sOrient == winding) // if point S inside convexPolygon halfplane*

*{*

*outputPoly.push\_back( computeIntersection(startOfEdge, endOfEdge, S, E)); //push intersection in output*

*}*

*S=E;*

*}*

*startOfEdge = endOfEdge;*

*}*

*//test print*

*std::cout<<"\n with possible duplicate nodes";*

*printPolygon(outputPoly);*

*removeDuplicateNodes(outputPoly);*

*return;*

*}*

*int main()*

*{*

*std::cout << "Hello, world!\n";*

*//input of vertices of "convexPoly", and "soilPoly" polygons in an arrays*

*node convexArray [] = { {-4,0}, {6,-6}, {28,16}, {-4,19} };*

*//node convexArray [] = { {6,-6}, {28,16}, {-4,19}, {-4,0}};*

*node soilArray [] = { {0,0}, {6.3,8}, {24,8}, {20,-2}, {3,-5}, {-2,0} }; //P1*

*//node soilArray [] = { {6.3,8}, {7,9}, {11,10}, {14,16}, {20,16}, {17,8} }; //P2*

*//node soilArray [] = { {20,16}, {29,16}, {24,14}, {24,8}, {17,8} }; //P3*

*//node soilArray [] = { {24,8}, {29,8}, {29,16}, {24,14} }; //P4*

*//Assigning arrays values to corresponding lists "convexPoly" and "soilPoly"*

*std::list<node> convexPoly (convexArray, convexArray + sizeof(convexArray)/sizeof(\*convexArray));*

*std::list<node> soilPoly (soilArray, soilArray + sizeof(soilArray)/sizeof(\*soilArray));*

*std::list<node> outputPoly; //list for intersection of two polygons*

*//printout of convex polygon nodes*

*std::cout << "\nThe convex polygon is: ";*

*printPolygon(convexPoly);*

*//printout of soil polygon*

*std::cout << "\nThe soil polygon is: ";*

*printPolygon(soilPoly);*

*//main function call*

*polygonClip(convexPoly, soilPoly, outputPoly);*

*//printout of intersection polygon nodes*

*std::cout << "\n\n The intersection of two polygonsis: ";*

*printPolygon(outputPoly);*

*}*

For calculation of polygon intersection Area we use “Shoelace Formula”

<https://en.wikipedia.org/wiki/Shoelace_formula>