Assignment #2:

2D Vector Field Topological Analysis: Compute Differential Topology and Geometric-based Morse Decomposition

Due Mar 8th, before midnight

Goals:

By completing the first assignment, you should now be familiar with PLY format. In this assignment, you will be given a number of simple 2D vector field data sets. Your task is to analyze these vector fields using topological based methods. Two topology computations will be implemented. The first one is the differential topology and the other one is the discrete topology. The detailed tasks are described below.

You will be provided a skeleton program to start with. The skeleton program includes a basic visualization of 2D vector fields, i.e. the image-based flow visualization. You can find more information of this technique from this website (www.win.tue.nl/~vanwijk/ibfv/).

You need to submit the source code and the associated library to ensure it can be compiled and run. Otherwise, you will be asked to do a live demo. *Please also remember to submit a well-written report with the description of the implemented algorithms and data structures if you have developed your own one. Figures and tables showing the results should be included as well as the discussion to address the questions asked in the individual tasks*. Fail to do so will result in the loss of points for this project.

Tasks:

1. Extract Fixed Points (150 points)

According to the definition of fixed points, i.e. the places with zero vector values, you can solve for a linear system for each triangle to determine whether there is a fixed point within it. However, a more effective approach analysis is as follows (also described in the class).

a) Compute the winding number of each triangle. To do so, you need to compute the angle difference between two vectors defined at two adjacent vertices. Assume the vertices of each triangle are sorted in the counter clockwise orientation. You can use the following pseudo-code to compute the angle difference. (50 points)

```
Vertex *v1, *v2;
double angle1 = atan2(v1->vy, v1->vx);
double angle2 = atan2(v2->vy, v2->vx);
double ang_diff = angle2-angle1;
```

Next, you need to make sure this angle difference falls in the range of [-π, π].

```
if (ang_diff > π)  ang_diff -= (2*π);
else if (ang_diff < -π) ang_diff += (2*π);
```

After computing the three angle differences, you now need to determine whether their total sum is 2π or -2π. If it is, this triangle contains a fixed point. We proceed to b).

b) For the triangle that contains a fixed point, we need to locate its position and determine its type. Use the routine "`get_loc_Jacobian(…)`" to compute the linear form of the local vector field within a triangle that contains a fixed point. This should provide you also the position of the fixed point. After computing the Jacobian of the fixed point, you need to determine its type by solving the eigenvalues of the Jacobian matrix, which is a 2x2 matrix. The classification of a fixed point based on the eigenvalues of its Jacobian is discussed in the class. Use the routine "`get_fixedPt_type(…)`" to implement the classification. In particular, if the fixed point is a saddle, you also need to compute its eigen-vectors for the later separatrix computation. The eigen-vector corresponding to the smaller eigen value provides the incoming direction of the separatrix, while the other provides the outgoing direction (50 points).

c) Visualize the obtained fixed points (50 points).
After detecting all the fixed points, you will need to properly store them in an array for the later topology computation. I will recommend using the C++ template "std::vector<class T>". To visualize each fixed point as a circle, use the following function.

```
void draw_solid_circle(double cx, double cy)
{
        int i;
        double R = 0.008;  // the radius of the circle
        double theta = 0., deta ;
        deta = 2 * PI/49.;
        double x, y;

        glBegin(GL_POLYGON);
        for(i = 0; i < 50; i++, theta += deta)
        {
                x =  cx + R * cos(theta);
                y =  cy + R * sin(theta);

                glVertex2f(x, y);
        }
        glEnd();
}
```

You can modify the above function to draw a hollow circle around the solid circle to highlight its boundary. But this is not required.

(What to report) *How many fixed points do you find for each data set? What are their types, respectively? Report the number of sources (S+), sinks (S-), and saddles (Sa). Compute the value (S+)+(S-)-(Sa) (Note, S+, S-, and Sa are the numbers of sources, sinks, and saddles, respectively). Report what you get for each vector field.*

2. Compute ECG without periodic orbits (300 points)

2.1 Enhance your Corner Table (50 points)

Built on top of your current Corner table data structure, for each vertex, insert a list of Corners that are adjacent to this vertex. They correspond to the triangles that adjacent to this vertex as well. You need to sort these Corners counter clockwise.

As described in the class, these Corners can be sorted using the Corner operations “c.p.o.p” or “c.n.o.n” depending on the orientation of the mesh.

2.2 Compute a local frame for each triangle (**50 points**)

In order to trace streamline within a triangle locally, you need to set up a local coordinate system for each triangle, and map the vertices and the vector values defined on them onto this local coordinate system.

Modify the member function “`compute_local_frame()`” of `Triangle` class to construct the local coordinate system.

2.2 Implement streamline computation (150 points)

Given a starting position (x, y) and the triangle T that contains this starting point, perform a local tracing within each triangle the streamline passes through.

There is a class in the skeleton code that takes care of the computation of the streamline. Its member functions are not complete, which you need to implement in this assignment in order to have a fully functional streamline computation.

The main entrance of this streamline computation in the skeleton code is the function

```
void Trajectory::cal_one_traj(int face_id, double x, double y, int type);
```

This function computes a portion of the streamline within each triangle it passes at a time. The termination conditions of the streamline computation include: 1) reaches the boundary of the domain; 2) reach a fixed point; 3) too slow to get out of a triangle; 4) forms a loop (not included in the skeleton code yet).

The local tracing within a triangle is fulfilled in the function

```
int Trajectory::trace_in_triangle(int &face_id, double globalp[2], int type, int &flag);
```

Within each triangle, the tracing is comprised of a number of small steps in order to achieve accurate result. The value of the integration step size is left for you to play with. As part of the report, you will need to report the tracing result with different step sizes.

Let us assume we allow at most 100 small steps within a triangle, the following algorithm describes what you should do to accomplish the local tracing within a triangle.

```
for(i = 0; i < 100; i++)
    {
        ////1. calculate the barycentric coordinates for current point

        ////2. if current point is inside current triangle
        ////      store the current point to the temporary data structure,
        ////      then call the get_next_pt(...) routine to move on to the next point
        ////    you can use the following code to store it
```

```
                /*
                        temp_point_list[NumPoints].gpx = globalp[0];
                        temp_point_list[NumPoints].gpy = globalp[1];
                        temp_point_list[NumPoints].gpz = globalp[2];
                        temp_point_list[NumPoints].lpx = cur_point[0];
                        temp_point_list[NumPoints].lpy = cur_point[1];
                        temp_point_list[NumPoints].triangleid = face->index;
                        NumPoints++;
                */

                ////3. Otherwise, the current point is outside of current triangle
                ////   call the get_next_triangle(...) function to determine the next triangle
                ////   that the streamline enters
                ////   then break;

        }
```

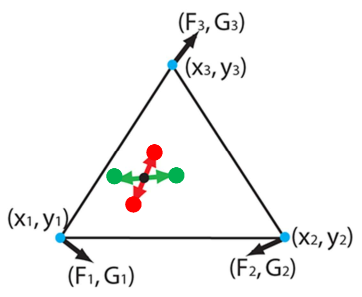
The functions you need to finish yourself for this task include:

```
void Trajectory::cal_one_traj(int face_id, double x, double y, int type);
int Trajectory::trace_in_triangle(int &face_id, double globalp[2], int type, int &flag);
void Trajectory::get_next_triangle(int &face_id, double pre[2], double cur[2], double param_t[2],
int type, int &PassVertornot, double alpha[3]);
void Trajectory::cross_a_vertex(int &face_id, double cur_p[2], double pre_p[2],int type, int
&passornot);
void  Trajectory::cross_boundary(double pre[2], double cur[2], int face_id, double alpha[3], int
&which_edge, double t[2]);
bool Trajectory::get_next_pt(double first[2], double second[2], int &face_id, double alpha[3
int type, unsigned char opt);
```

The descriptions of how you should implement these functions are provided in the skeleton code. Please let me know if you are not clear with any of these descriptions!

2.3 Compute separatrices starting from each saddle (50 points)

For each saddle type of fixed point, there are exactly four separatrices starting from or ending at it. Two of them are incoming streamlines and the other two are outgoing ones. To determine the incoming and outgoing directions, you need to rely on the eigen analysis of the Jacobian matrix of the saddle. Specifically, the incoming direction corresponds to the smaller eigen-value (negative) of the Jacobian matrix, while the outgoing direction corresponds to the larger eigen-value (positive). Each eigen-vector provides two directions for the two separatrices.



The function you need to finish in order to accomplish this task is

```
void  cal_seps();
```

Given a saddle and its two eigen-vectors obtained from the fixed point detection, compute a streamline from a point that is in one eigen-vector direction and is sufficiently close to the center of the saddle. That said, you need to compute the four seeding positions in the eigen-vector directions away from the saddle. The figure to the right illustrates this.

(What to report)

Describe your implementation of the streamline computation. Implement at least the Euler integration and RK2. Experiment with at least ***two different step sizes*** *for each integrator. Show the obtained streamlines starting from the exactly same location (say, a center of a triangle) using different step sizes. Use your streamline computation to compute the ECG for each vector field data. Take a snapshot of your result and include it in your report. Based on the analysis, please draw an abstract graph of the ECG for each vector field that you are provided and include them in your report. You can use the graph shown in the class lecture as a reference.*

Extra credits: Compute Discrete Topology (50 points)

You will implement a very simple Morse decomposition of vector fields, which will provide a very coarse decomposition result. However, it should serve a good starting point to understand how Morse decomposition works. In addition, based on this simple and fast decomposition, you can further achieve the effective detection of periodic orbits, which are difficult to extract using previous methods.

(What to report)

Based on the analysis, please draw an abstract graph of the MCG for each vector field you are provided and include them in your report.

Grades:

Tasks	*Total points*
1	150
2	300
extra	50

Submission of the assignment

You will need to submit the report and a link to the code (with a checksum) for this assignment via the blackboard learn system. You will also need to send me and TA an email about this submission.