I've completed the pseudocode for the streamline placement algorithm after our discussion last week. It is as follows:

```
generate initial seed point (random)
add seed point to queue

while queue is not empty:
    pop the first seed from the queue

    check if seed point is valid
    (in boundaries and not closer than d_sep to any other point)
    if this seed point is not valid:
        continue to next seed point in queue

    compute the entire streamline for this seed
    if this streamline has more than 5 vertices:
        add the streamline to the collection
    otherwise:
        continue to next seed point in queue

    loop through the points in the streamline:
        (every 5 points)
        calculate two new seed points orthogonal to streamline
        (one on each side)
        add the two seed points to the queue
```

I've been working on an implementation for streamline visualization so that I better understand the algorithm, and fill in any details I may have missed in the pseudocode. This implementation uses a basic queued seeding strategy and successfully traces random seeds or user-generated (by click) seeds. It does not yet successfully auto-seed to fill the field space. It also does not yet include a mesh loading solution, and as of now (for testing purposes) the vector fields are defined as continuous functions. Example output with user-generated seed points for vector field $v(x,y) = (1, \sin(x))$ can be seen to the right. I plan to complete the implementation by next meeting, and then work to extend the framework to include pathline tracing.
For our meeting today, we can go over the pseudocode for streamline placement, and I will demonstrate my implementation’s current (limited) capabilities. I would also like to further discuss integration schemes as well as the barycentric vector mesh interpolation scheme, as I am having some difficulty understanding how to implement them. Additional items to discuss are the use of cell-based sample distance controller and adaptive sample density control.

16 June 2016

This week I have implemented several new features into the streamline placement program. The vector field can now be defined as a combination of several “features” which themselves are vector-valued functions. This allows for greater flexibility in continuous vector field editing and serves as a stand-in while the vector mesh interpolator is developed. Additionally, the program now uses Runge-Kutta 4th-order integration for streamline advection. This greatly enhances the accuracy of streamlines and properly visualizes closed loops. See the two examples below.

I have also completely implemented the streamline placement algorithm as outlined from last week. Progress on this is shown below from left to right. To accomplish fast placement, the algorithm uses a binning structure for localized distance checking, which vastly reduces the amount of computation required. This structure is defined as a grid of bins or cells which fully cover the view. Each bin contains a reference to its neighbors (Moore Neighborhood). When a streamline is fully advected, its vertices are inserted into the bin structure such that each bin contains a reference to all of the vertices in its region. Then, when a seed candidate is checked
for separation distance to other points, the program only needs to check the points located in the current bin and its neighbors. Points outside these bins can be safely considered too far and therefore an expensive distance check can be avoided.

### First attempt at placement
allowing only one seed per bin (random placement).

### Second attempt at placement, seeds are separated by distance but still placed randomly.

### Third attempt, demonstrating successful termination of advection when within $d_{sep}$ of another streamline (manual placement).

Seed candidates are now placed evenly along the streamline according to a tuning parameter `candidate_spacing`. Progress can be seen below from left to right, with the final result on the right. To ensure equal spacing, the program employs linear interpolation to allow candidates to be placed in-between vertices. The program stores the partial arc length of the streamline at every vertex.

### Seed placement at every 5 vertices along the streamline.

### Seed placement when partial arc length exceeds `candidate_spacing`, still on vertices.

### Seed placement at interpolated locations along streamline, exactly fulfilling `candidate_spacing`.

The pseudocode for this algorithm is outlined below:
PlaceSeedCandidates(streamline) {
    orthoVec = new Vec2(0, 0)
    prevArcLen = 0
    t_len = 0

    // Loop through the points in the streamline
    for (each consecutive pair of vertices in streamline: v0, v1) {
        v0 = streamline.vertices[i-1]
        v1 = streamline.vertices[i]

        if (v1.partialArcLen - (prevArcLen + t_len) > candidate_spacing) {
            // Calculate percent along last segment
            lastSegLen = v1.partialArcLen - v0.partialArcLen
            t_len = candidate_spacing - v0.partialArcLen + prevArcLen + t_len
            t = t_len / lastSegLen

            // Calculate point using linear interpolation
            x = lerp(v0.x, v1.x, t)
            y = lerp(v0.y, v1.y, t)

            // Calculate orthogonal vector using vector field
            orthoVec = field.vec_at(x, y)
            temp = orthoVec.x
            k = this.d_sep / magnitude(orthoVec)
            orthoVec.x = -orthoVec.y * k
            orthoVec.y = temp * k

            // Create seed points
            s1 = new Point(x + orthoVec.x, y + orthoVec.y)
            s2 = new Point(x - orthoVec.x, y - orthoVec.y)

            // Add the two seed points to the queue
            seed_queue.push(s1, s2)
        }
        prevArcLen = v0.partialArcLen
    }
}
When the program was given time to run to completion under the new seeding algorithm, it failed to completely cover the region. Initially this looked to be a problem with seed candidates not being placed orthogonal to the initial seed. However, when this was implemented the program would place far more seed candidates than it could check, and thus create an infinite loop. The image above to the left demonstrates the partial coverage. To solve this, the program only needed to advect the streamlines in reverse in addition to forward advection. The central image above demonstrates full streamline advection (forward and reverse). Finally, the image above to the right demonstrates the full coverage after reverse advection was implemented. This solution is mostly satisfactory, with only small gaps due to the minimum streamline length requirement.

Future work lies in developing a fast strategy for detecting loops, as well as implementing the vector mesh interpolator.

23 June 2016

This week was mainly spent optimizing the file structure and build system for the project. I also fully implemented the loop detection algorithm. I also partially implemented the vector mesh field data structure.

The loop detection algorithm was taken directly from Z. Liu and R. Moorhead\(^1\) as it appears to be both robust and fast.

[Include loop detection algorithm]

[Include profiling metrics]

30 June 2016

This week was mainly spent successfully debugging the PLY file loading system. The PLY parser was written by Mikola Lysenko independently of this project, and is freely available under the BSD license\(^2\). Because the parser was written for Node.JS, there was some difficulty in getting it to work within the browser. Ultimately the solution I found was to use the JSPM package manager with SystemJS module/dependency loading.

[Describe PLY file loading]

---


\(^2\) https://github.com/mikolalysenko/parse-ply
The vector mesh field data structure was improved. When a vector mesh is loaded, a routine calculates the minimum bounding rectangle of each face. A uniform binning structure is created, and the face is added into the bins which overlap with its minimum bounding rectangle. This structure allows for fast point-in-triangle lookup by reducing the number of faces to check to the number of faces contained in the queried bin. This data structure is not completely robust, as it fails to handle meshes of variable triangulation.
density. Possible improvements could be made with a quadtree structure, however since most data sets are regular grids this feature is not a priority at this time.

There is currently a bug in which the point-in-triangle test fails when it should pass, and produces the result seen to the left.

7 July 2016

This week was spent on three main aspects: improving the user interface of the 2D vector field analysis tool, streamline tapering, and continuous field discretization. Additionally, metrics were captured for each data set at differing placement density.

The point-in-triangle bug from last week was solved. The issue arose from the generation of the face binning structure, where bins are ordered from top left to bottom right (down is positive) but the vector space is cartesian (down is negative). This means that vector mesh data are now fully supported. Files are loaded from the disk, and the seed point can be selected before generating streamlines.

The user interface of the 2D vector field analysis tool was improved by adding a more configurable parameter menu with the dat.GUI library. This open source library is licensed under the Apache 2.0 license. The updated GUI allows for manipulation of the algorithm parameters such as $d_{test}$, $d_{sep}$, and candidate spacing. It also provides a simple set of toggles to enable and disable visualization of various components such as the vector mesh, binning structures, seeds, and more. File loading is implemented as drag-and-drop.

The streamline tapering effect is a method of improving the visual quality by reducing artifacts caused by abrupt beginnings and ends of streamlines. It also helps to smooth out the apparent density by thinning out streamlines that are closer to others and thickening streamlines that are further away. The thickness coefficient is calculated during advection according to the following formula:
thickness = \frac{d - d_{test}}{d_{sep} - d_{test}} \text{ if } d \leq d_{sep}, \text{ else 1}

This formula is taken directly from B. Jobard and W. Lefer\textsuperscript{4}. The result of this can be seen in the figure to the left. It is worth noting that the first attempt at the tapering effect was unsuccessful. The original algorithm calculated streamline thickness as a post-processing step after all streamlines had been placed. This caused streamlines to appear unsmooth and often discontinuous.

Continuous field discretization refers to a method to transform a continuous field into a vector mesh field. This is a useful feature as it allows the creation of data sets within the user interface. The vector mesh data can then be exported in PLY format. As of this week’s report, the feature is not fully functional.

Algorithm run-time was evaluated on an AMD Athlon II x4 2.9 GHz processor. Rendering times are not included in these benchmarks.

[TODO: make a plot]

<table>
<thead>
<tr>
<th>D_sep = 0.1</th>
<th>D_sep = 0.05</th>
<th>D_sep = 0.01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streamlines</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithm runtime breakdown.

<table>
<thead>
<tr>
<th>Function</th>
<th>Self time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

14 July 2016

This week’s work focused mainly on completing continuous field discretization and reorganizing the codebase and data structures to better handle time-varying vector fields.

Continuous field discretization is now fully functional. A mesh resolution can be chosen from the user interface, and then the generated field mesh can be exported in PLY format. One potential improvement to the discretization algorithm would be to allow for alternate triangulation schemes such as Delaunay triangulation or variable density grids similar to quadtrees. This would allow for better approximation by increasing the level of detail where the field has small features and reducing the level of detail where features are large. Another possible enhancement would be to first run a low-pass filter over the continuous vector field before sampling at the mesh vertices. This would improve the vertex vector value by considering it as the average of its local surroundings rather than assuming the exact vector value at the vertex coordinate.

The following diagram demonstrates discretization of the continuous field from earlier reports at increasing levels of resolution from 2 to 64. Past a resolution of about 32, this vector field is sufficiently discretized to approximate the continuous function. This number varies depending on the level of detail present in the vector field.

21 July 2016

This week’s work was focused on fleshing out the unsteady flow capabilities. This required some drastic updates to the vector mesh data structure in order to store multiple vector values
for each vertex. The barycentric mesh interpolator was also updated so that vector values are interpolated in-between frames. A few other updates were made to the GUI to allow toggling and manipulation of more parameters.

For a continuous field, supporting unsteady flow simply requires incorporation of time into the field equation. The example field used in tests as shown is a double gyre with oscillating centers, defined by the following differential equation:

\[
\begin{bmatrix}
  \dot{u} \\
  \dot{v}
\end{bmatrix} = 
\begin{bmatrix}
  -A \pi \sin (\pi f_x(t)) \cos (\pi y) \\
  A \pi \sin (\pi f_x(t)) \cos (\pi y) \frac{df_x}{dx}
\end{bmatrix}
\]

Where

\[ f_x(t) = B \sin (Ct)x^2 + (1 - 2B \sin (Ct))x \]

The values \( A = 0.1, B = 0.25, \) and \( C = \pi \) were used to generate the field in the images below.

The top left image depicts the initial result of time-varying pathline advection, utilizing the same conditions for even spacing, loop detection, and tapering. The image to the top right removes these constraints to depict the entire flow through the chosen seeds. The two bottom images depict the flow at differing time steps with a limited pathline length.

The graphical user interface has been updated to allow the toggling of line tapering; seed candidate validations such as bounds checking, separation checking, and loop detection; and controls to manipulate the ending time step \( (t_{end}) \) and pathline length \( (t_{span}) \) for rendering. The limited length pathlines are drawn with an increasing opacity according to the following formula:

\[
\text{opacity} = \begin{cases} 
0 & t \leq t_{end} - t_{span} \\
\frac{t - (t_{end} - t_{span})}{t_{span}} & t_{end} - t_{span} \leq t \leq t_{end} \\
0 & t_{end} < t 
\end{cases}
\]
where t is the current timestep at the vertex.

The seeding strategy has not been modified from steady flow, and is clearly not capable of generating a representative pattern for the entirety of the field through its time domain. Finding a good seeding strategy will be the topic of this next week’s work. A possible solution is an interactive projected 3D visualization with the time domain as the third dimension.