Assignment #4: 2D Steady Vector Field Visualization

Due on October 22nd, before midnight

Goals and Requirements:

The goals of this assignment include (1) understanding the characteristics of 2D steady vector fields, (2) implementing a number of classic visualization techniques for 2D steady vector fields, including arrow plots, streamline integration, and LIC.

A skeleton code in Python will be provided to you to help you get started.

You should submit your source code and report **in a single .zip file** via Blackboard before the deadline.

Your report should include your answers to the writing questions **AND** high-quality images captured from your implemented program. In particular, **for each given 2D vector field**, please include the following in your report

(1) An arrow plot (with clear arrow representation and little or no overlapping)

(2) A streamline placement result. Please provide the number of streamlines and the selected seeding strategy (uniform or random) for each result.

(3) LIC texture result. Please provide the integration length used, i.e., zz% of the image resolution in X or Y dimension. This (zz%) is a user controllable parameter.

Tasks:

1. Writing questions (20 points)

1.1 Why visualizing vector fields is more challenging than visualizing scalar fields? **(5 pts)**

1.2 Provide a complete pseudo-code for the LIC algorithm. **(10 pts)**

1.3 What are the **features** that people care about in vector fields? **(5pts)**

2. Generate arrow plots (20 points)

To show the arrow plot of a loaded vector field, do the following (when arrow plot checkbox is toggled on)

First, Choose the arrow as the glyph type.

```
glyphSource = vtk.vtkGlyphSource2D()
glyphSource.SetGlyphTypeToArrow()
glyphSource.FilledOff()
```

Second, setup a vtkGlyph2D object.

```
glyph2D = vtk.vtkGlyph2D()
glyph2D.SetSourceConnection(glyphSource.GetOutputPort())
glyph2D.SetInputData(self.reader.GetOutput())
glyph2D.OrientOn()
glyph2D.SetScaleModeToScaleByVector()
glyph2D.SetScaleFactor(0.03) # adjust the length of the arrows accordingly
glyph2D.Update()
```

Third, create a mapper and add an actor to show the arrows.

```
arrows_mapper = vtk.vtkPolyDataMapper()
arrows_mapper.SetInputConnection(glyph2D.GetOutputPort())
arrows_mapper.Update()

self.arrow_actor = vtk.vtkActor()
self.arrow_actor.SetMapper(arrows_mapper)
self.arrow_actor.GetProperty().SetColor(0,0,1) # set the color you want
```

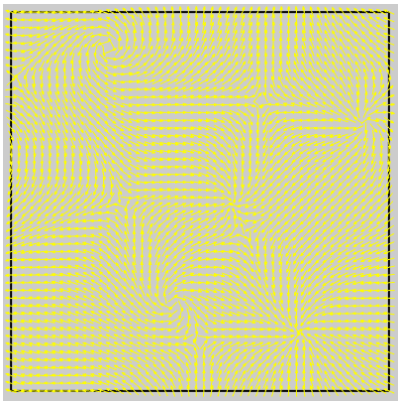
Do not forget to add your arrow actor to the renderer for rendering!

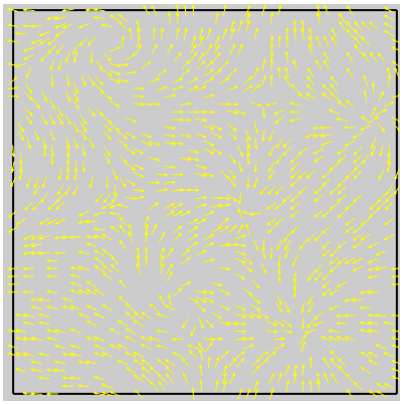
BONUS (5pts): The above will place an arrow at each grid point of the mesh/grid where the vector field is defined (see the left image below). For some grid that has dense grid points, the generated arrow plot can be too cluttering (i.e., the arrows may overlap each other or too small to see). To address this, you can utilize the `vtkMaskPoints` object. This is a density filter that you can add **BEFORE** the `vtkGlyph2D` object. The following provides an example of such a filter

```
densityFilter = vtk.vtkMaskPoints()
densityFilter.SetInputData(self.reader.GetOutput())
densityFilter.RandomModeOn() # enable the random sampling mechanism
densityFilter.SetRandomModeType(3) #specify the sampling mode
densityFilter.Update()
```

You then can use the `SetMaximumNumberOfPoints()` functions provided in the `vtkMaskPoints` to achieve down sampling or change to the random sampling method of the arrow plot with the `SetRandomModeType()` function. You also need to set the input to your `vtkGlyph2D` object as

```
glyph2D.SetInputData(densityFilter.GetOutput())
```


Arrow placed at every grid point


Uniform sample with 900 arrows


Random sample with 500 arrows

3. Compute and visualize streamlines (30 points)

You can compute a streamline from a given starting position (or a seed point) in VTK using `vtkStreamTracer()`. Each seed with coordinate (x, y) can be represented as a point in a `vtkPoints` object like below.

```
seeds = vtk.vtkPoints()
seeds.InsertNextPoint(x, y, 0) # the third value is for z coordinate. For our 2D planar data, we set it as 0
```

Note that if each time only one streamline is computed from a seed, you can also use `SetStartPosition(x, y, z)` to specify the starting position of the streamline. However, since you are going to do multiple streamline tracing next, we will recommend using `vtkPoints` to store the seeds.

We then convert the `vtkPoints()` object (that may contain one or more seed points) into a `vtkPolyData` object to be used as the source for the `vtkStreamTracer`.

```
seedPolyData = vtk.vtkPolyData()
seedPolyData.setPoints(seeds) # 'seeds' is the vtkPoints object above
```

After getting the seeds, you can generate a `vtkStreamTracer` object to compute streamlines as follows

```
stream_tracer = vtk.vtkStreamTracer()
stream_tracer.SetInputData(self.reader.GetPolyDataOutput()) # set vector field
stream_tracer.SetSourceData(seedPolyData) # pass in the seeds
```

Then, you need to choose your integrator to use for the streamline calculation. The following uses RK 45.

```
stream_tracer.SetIntegratorTypeToRungeKutta45()
stream_tracer.SetIntegrationDirectionToBoth()
```

You can play with other parameters to adjust the accuracy of the integration, such as the integration step size. You can find more information about these parameters in the following link https://vtk.org/doc/nightly/html/classvtkStreamTracer.html

Next, you need to generate a number of seeds to compute a few streamlines for visualization. In this assignment, you are required to implement the following two seeding strategies.

(1) Uniform seed generation

Given a user-specified number of seeds (usually a square of certain integer, say 100), then the uniform seeding strategy will generate N * N seeds in the domain with uniform distance between neighboring seeds. Here N is the square root of the input integer (e.g., N=10 if the input is 100), which indicates the number of seeds along each axis direction. That said, an intuitive way to achieve uniform seeding is to subdivide the regular domain (say a unit square [0, 1] x [0, 1])) into a N * N uniform grid. The grid points are the seeding positions. In this way, the coordinate of a grid point (xi, yj) can be computed as follows

```
xi=i*(1.0/(N-1))
yj=j*(1.0/(N-1))
```

(2) Random seed generation

Generating N random seeds is relatively easy. You can call the random number generation function to generate two random integers, one for x and other for y. Then, you should normalize these two integers so that they lead to float values in [0, 1]. In VTK, you can use the "random" library as follows

```
x = (random.randint(0,32768) / 32768.0
y = (random.randint(0,32768) / 32768.0
```

Perform the above (x, y) random generation N times to get N seeds.

For both seeding strategies, after generating a new seed, add it to the list of the `vtkPoints()` object (e.g., seeds above using the `InsertNextPoint()` function).

Mapping the seeds to the original domain: You can first generate the coordinate of a point (x, y) in a 2D regular domain of [0, 1] x [0,1] (i.e., a square with size 1 and its lower left corner is at (0, 0)), then convert this coordinate into a point in the original domain where the vector field is defined. The original domain of the data is obtained as follows.

use `bound=self.reader.GetPolyDataOutput().GetBounds()` to get the domain of the data. This function will return 4 values for the 2D planar domain, `bound[0]` records the smallest x coordinate of the domain, `bound[1]` records the largest x coordinate, while `bound[2]` stores the smallest y coordinate and `bound[3]` stores the largest y coordinate. `bound[4]` and `bound[5]` store the information of the z coordinate, which is not needed for this assignment.

Now, using the above information of the domain, you can convert (x, y) obtained above into the corresponding point in the original domain as follows

```
x′=bound[0] + x*(bound[1]-bound[0])
y′=bound[2] + y*(bound[3]-bound[2])
```

4. Compute and visualize LIC texture (30 points)

In this task, you will implement the LIC framework as you sketched out when answering question 1.2. There are a number of critical steps that you need to pay attention to.

First, you need to create a white noise texture for the LIC calculation. **The `create_noise_texture(self)` function in the skeleton code does this for you**. The white noise texture is stored in the variable `self.noise_tex`, which is a 3D array with dimension of IMG_RES*IMG_RES*3. Since this is a gray white noise texture, all three color channels have the same value. To access the color value for a pixel with index (r, c), use `self.noise_tex[r][c][0].`

Second, you need to determine the coordinate of the center of each pixel in the original space where the vector field is defined. This is **similar to** the seed generation described in **Task 3**. Specifically, for each pixel with index (i, j) (with i the row index and j and column index), the coordinate of its center is computed as

```
x=(j+0.5)/IMG_RES # remember j is the column index that corresponds to x direction!
y=(i+0.5)/IMG_RES # i is the row index corresponding to the y direction!
```

To map this (x,y) coordinate to the coordinate (x', y') in the original domain, you can use the transformation expressed above (i.e., the two lines of code right above Task 4).

Third, when you pass the converted coordinate (x', y') as the starting point to a `vtkStreamTracer()` object, use its `SetStartPosition(x, y, bound[4])` function directly. The reason we use `bound[4]` instead of 0 here is because the 2D plane may not be sitting at z=0. Also, you should call the following two functions to set the tracing parameters properly!

```
# Set the maximum integration length of the streamline. This parameter will determine the smoothness of your LIC texture and the computation time. It is the parameter that you should play with (zz%)!
SetMaximumPropagation (double )
# Specify a constant step size that is equivalent to the size of a pixel (what will that be?)
SetInitialIntegrationStep(double )
```

Once you complete the streamline tracing above, use the following to obtain the streamline geometry as a `vtkPolyData`

```
streamline = streamTracer.GetOutput()
```

From this vtkPolyData, you can further extract its points as follows

```
integrationPts = streamline.GetPoints()
nPts = integrationPts.GetNumberOfPoints()
```

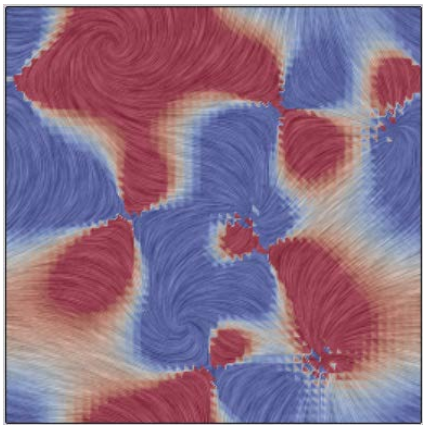
For each of these points, you then figure out which pixel (and pixel index) it corresponds to so that you can obtain the color of this pixel from the white noise texture to perform the color convolution. To simplify this process, you can simply accumulate (or add) the colors of all the pixels this streamline pass through and take the average color as the color to assign to the pixel.
That is, in the given skeleton code, you should try to determine the following:

```
self.LIC_tex[i][j][0] = self.LIC_tex[i][j][1] = self.LIC_tex[i][j][2] = ?
```

Finally, the LIC texture will be converted to a `vtkImageData` for rendering. **This is already provided to you in the skeleton code.**

Note that since LIC involves a large amount of streamline integration with additional calculations in object space (including locating triangles and performing interpolation), the performance of the LIC can be slow (e.g., 2 minutes or longer for an image with resolution 256x256). So, do not worry if your program is not responsive after you enable the LIC computation. Also, during the development and debugging of your LIC computation, I will recommend to use a smaller image with resolution of 128x128 or even smaller.

BONUS (5pts): The above LIC texture can be combined with the color plot that shows certain physical property of the flow (e.g., velocity magnitude, curl, divergence, etc.). To combine the color plot with the LIC texture, you need to first generate a color plot with the exact same resolution as the LIC texture. Then, you need to determine the colors of the individual pixels of this color plot based on the physical attribute (scalar) values at those individual pixels. Once you determine the color for a given pixel based on the attribute scalar value, you can use the following formula to combine this color with the LIC texture color at the same pixel.



*final_color = LIC_color * weight + attribute_color * (1- weight)*

"weight" is a positive number between 0 and 1 (e.g., a default value can be 0.5).
The image to the right shows an example of the blending of the LIC texture with the curl field (using blue-white-red color scheme) for the **bnoise.vtk** data. In the skeleton code, a `generate_uniform_grid()` function is provided to you so you can generate a uniform grid with the exact same resolution as the LIC texture image. You can use it as the starting point to complete this bonus task.

Grading rubric:

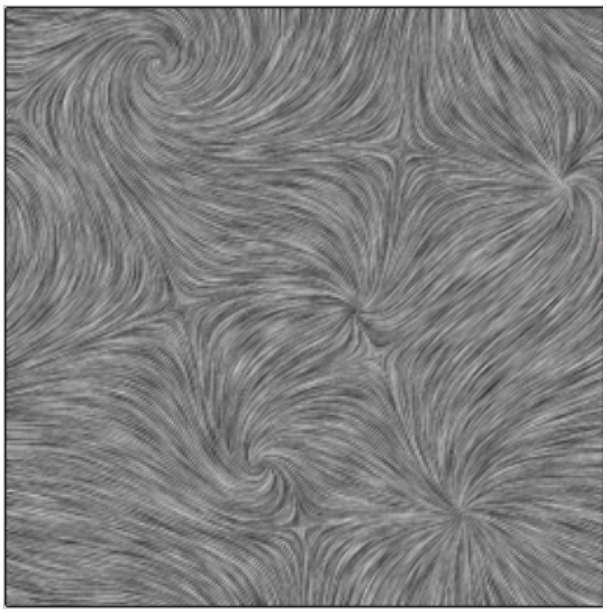
Tasks	*Total points*
1	20
2	20
3	30
4	30
Arrow density adjustment (bonus)	*5*
LIC+color plot blending (bonus)	*5*

Suggestions:

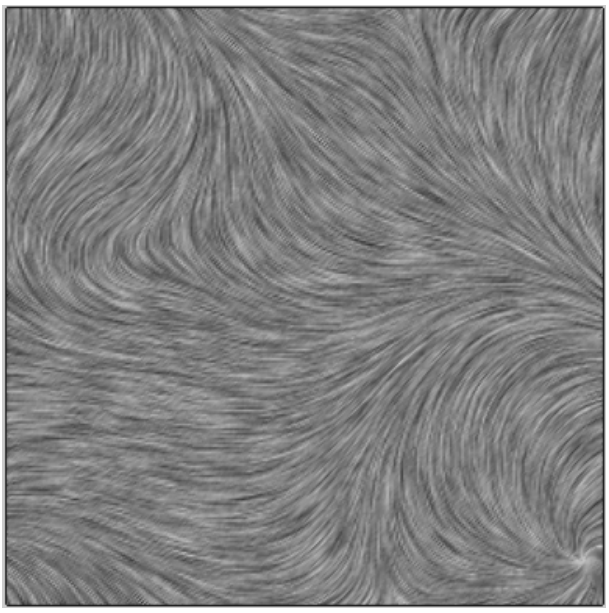
Please refer to the VTK tutorial (https://vtk.org/Wiki/VTK/Tutorials) and Python examples (https://lorensen.github.io/VTKExamples/site/Python/) for more information and examples.

Have fun!

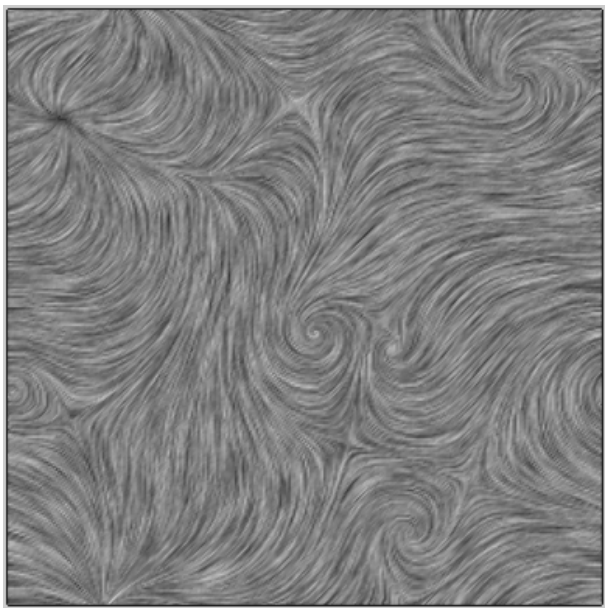
Example LIC textures for you to check the correctness of your implementation.



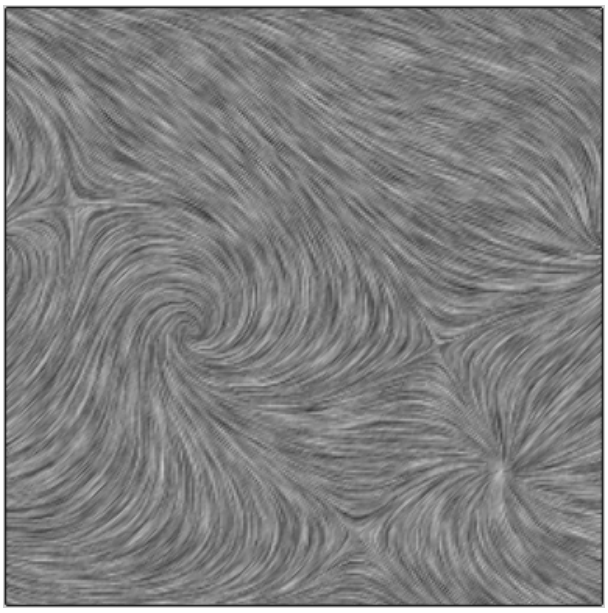
bnoise.vtk



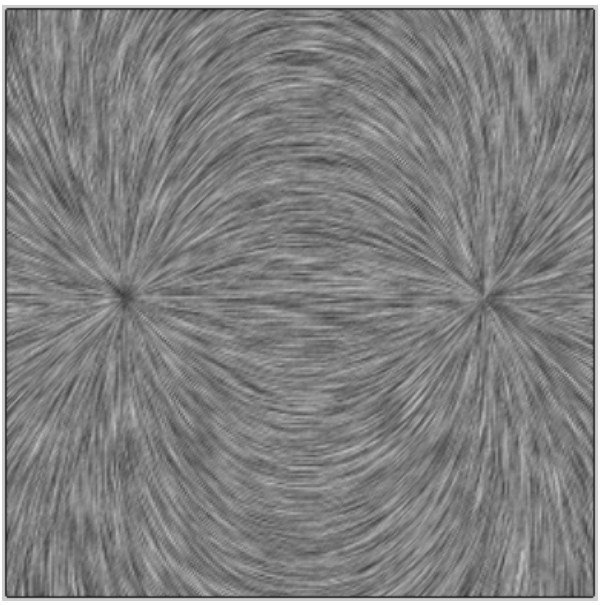
bruno3.vtk



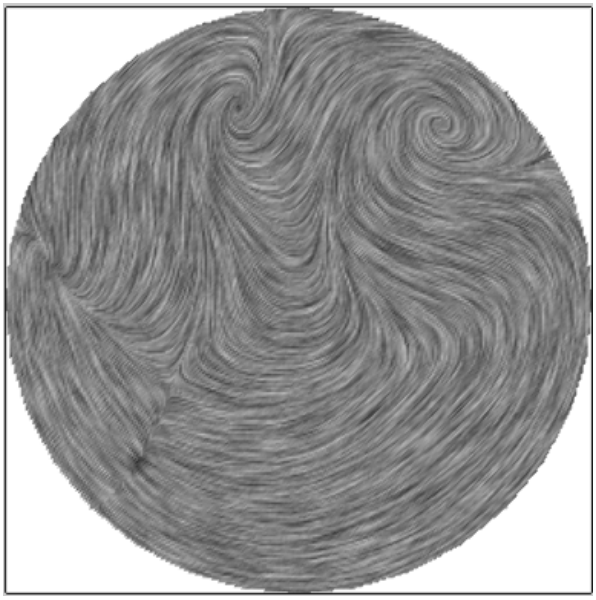
cnoise.vtk



vnoise.vtk



dipole.vtk



diesel_field1.vtk