Assignment #3: 2D Steady Vector Field Visualization

Due on October 19nd, before midnight

Goals:

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

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

Requirements of Submission:

You should submit your source code **in .py format** and a writing report **in a separate file (pdf/docx)** via Canvas before the deadline. Please use the following naming convention to name your zip file:

*Assignment3_**FirstName_LastName.py**, Assignment3_Report_**FirstName_LastName**.pdf (or .docx)*

Please DO NOT include the data sets in your submissions!!!!

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 (play with the scaling coefficient to produce a clear arrow representation with 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 parameter settings that are used to produce the captured results.

Important Notice on the Use of ChatGPT in Programming Assignments:

While the utilization of ChatGPT or similar large language models is permitted for programming assignments, it is not allowed during exams. We urge students to exhaust their understanding and attempt solutions independently before resorting to these tools. Remember: the true value of this course is derived from the challenges you overcome, and the knowledge you gain in the process.

Transparency is crucial: If you choose to use ChatGPT or any other external resources, it is mandatory to clearly indicate so in your assignment report. You will not be penalized if you use ChatGPT. However, if you failed to report it and were caught, it would be considered plagiarism and your score will be zero. Note that it was prohibited to borrow answers and codes from students who already took this course.

Tasks:

1. Writing questions (20 points)

1.1 Provide a **complete pseudo-code** for the LIC algorithm. You code should have sufficient details rather than only describing the process **(10 pts)**

1.2 Why placing streamlines is challenging, and what need to be considered when placing streamlines? **(10 pts)**

2. Generate arrow plots (25 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!

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 need to 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())
```

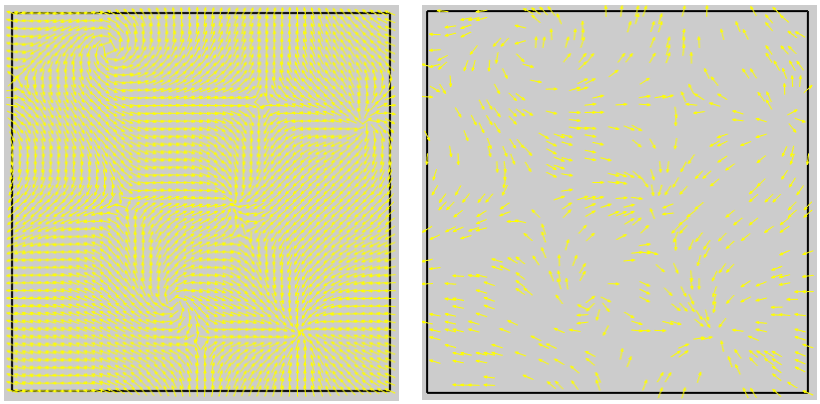


Figure 1. (left) Arrows placed at all grid points; (right) Randomly placed 500 arrows.

To get the full points for this section, you need to implement arrow placement for (1) all grid points, and (2) random sample with specified number of samples (see the above figure to the right)

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 result in float values in the range of [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: This is a crucial step for determining the location of each seed point. 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 (25 points)

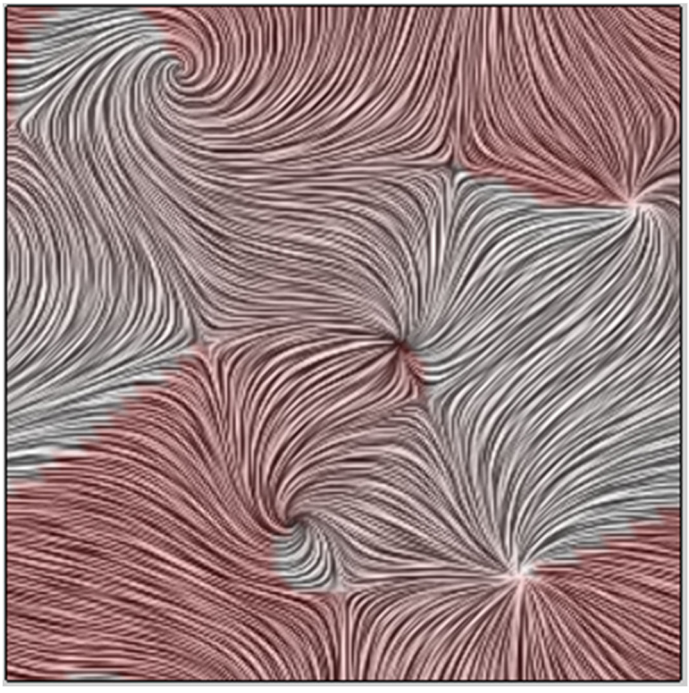
In this task, you will use VTK to produce LIC texture on the given data. It is recommended to use the `vtkImageDataLIC2D` filter. The rough steps include:

(1) Initialize the filter (like you did for the other filters in Assignments 1 and 2).
Note: The dataset you've been given is in a 2D mesh polydata format, incompatible with the `vtkImageDataLIC2D` filter. We've converted this data into the vtkImageData format for your convenience. Access the image data using `licFilter.SetInputData(self.imageData)`

(2) Specify the step size, number of steps, and other necessary LIC parameters to ensure a sharp and distinct LIC texture. Since different data sets have different sizes, the step size should be set based on a fraction of the data set's size (such as the square root of it's area). The number of steps can be adjusted to make the LIC texture more well-defined (but too much and the performance will suffer). You can use `licFilter.SetSteps(steps)` and `licFilter.SetStepSize(size)` to set these parameters.
Optionally: call `licFilter.SetContext(self.vtkWidget.GetRenderWindow())` to prevent the filter from opening a new unused window.

(3) Construct a lookup table with a grayscale color scheme to prevent unintended color mappings to LIC values. Refer to the `MakeLUT()` function on how to create a color lookup table and assignment 1 on how to create a grayscale color scheme.

(4) Establish an actor for the mapper. Adjust its opacity in relation to the LIC opacity slider, allowing the underlying color map to be discernible, and then return the actor.



A proper LIC image showing the color mapped data of the vector angles, where the LIC actor's opacity is set to 0.8. You need to receive this opacity value from the interface.

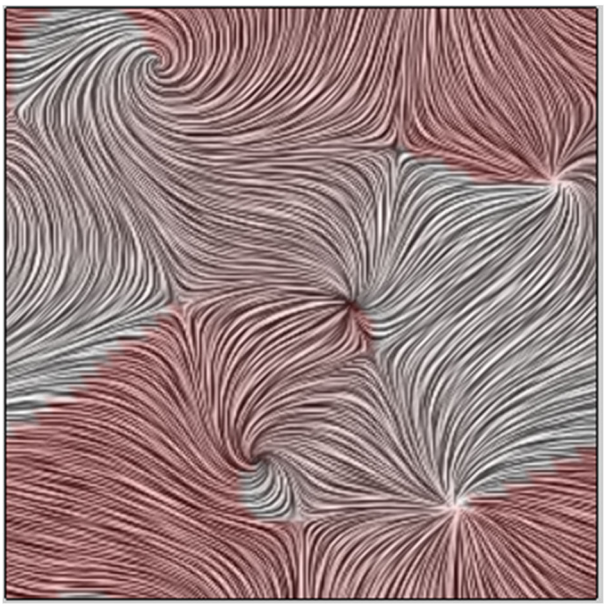
BONUS (5pts): The current LIC texture produced by the initial method might not meet our aesthetic expectations. This limitation arises from the simplistic color blending approach we've employed—lowering the opacity of the LIC image to reveal the underlying colors. This not only diminishes the clarity of the LIC image but also compromises the vibrancy of the colors beneath. To achieve a more appealing visual outcome, consider using the **`vtkImageBlend`** filter. This filter allows for a more sophisticated blending of two `vtkImageData` sources: one from `self.imageData` and the other from the LIC filter. By adjusting the blending opacities and blend mode, you can achieve a richer visual effect that enhances both the LIC texture and the colors underneath. Please **indicate in the report if you have implemented this to obtain bonus points.**

Additional Resources:

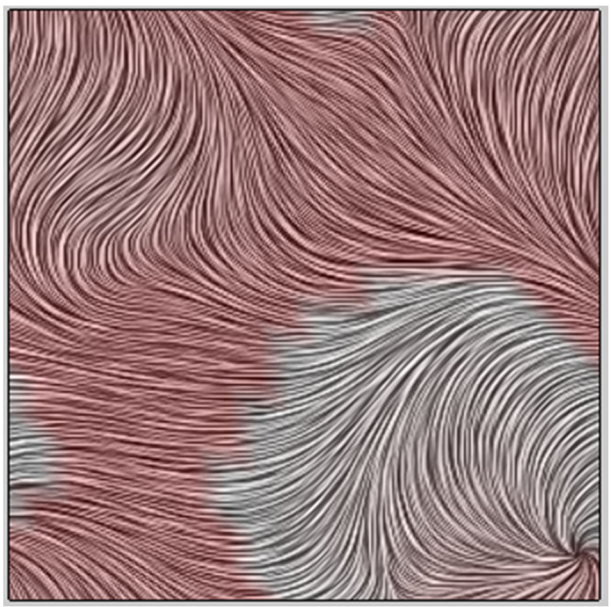
Please refer to the VTK tutorial (https://vtk.org/Wiki/VTK/Tutorials) and Python examples (https://examples.vtk.org/site/Python/) for more information and examples.

Have fun!

Example LIC textures for you to check the correctness of your implementation. Color plots are generated using the angle of the individual vectors. Feel free to use the color scheme you developed in Assignment 1!



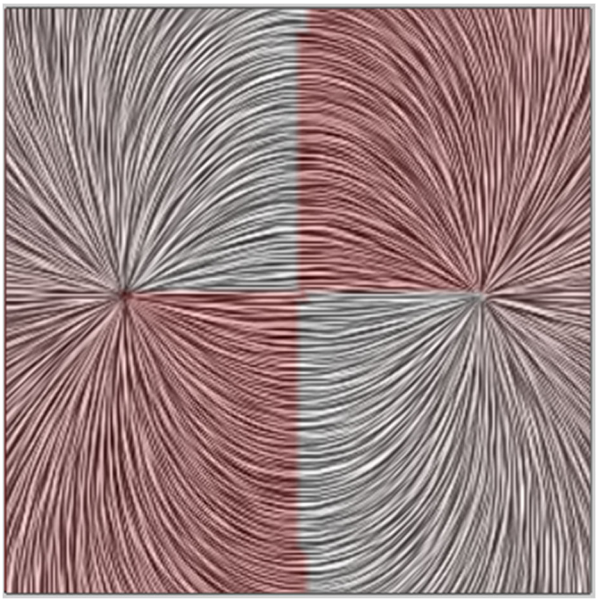
bnoise.vtk



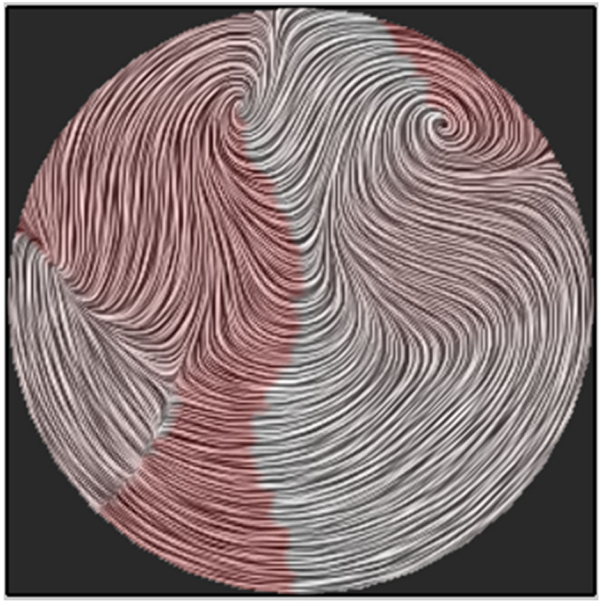
bruno3.vtk

cnoise.vtk

vnoise.vtk



dipole.vtk



diesel_field1.vtk