TNM067 — Scientific Visualization
3. Vector field visualization

September 21, 2021

1 Introduction

In this lab you will explore vector field visualization in both 2D and 3D. You will begin by deriving information from the vector field, such as vector magnitude, divergence and rotation. Then you will implement Line Integral Convolution to make a dense visualization of the vector field and enhancing it with the information from the first step. Finally you will explore a 3D vector field and compare different integration schemes.

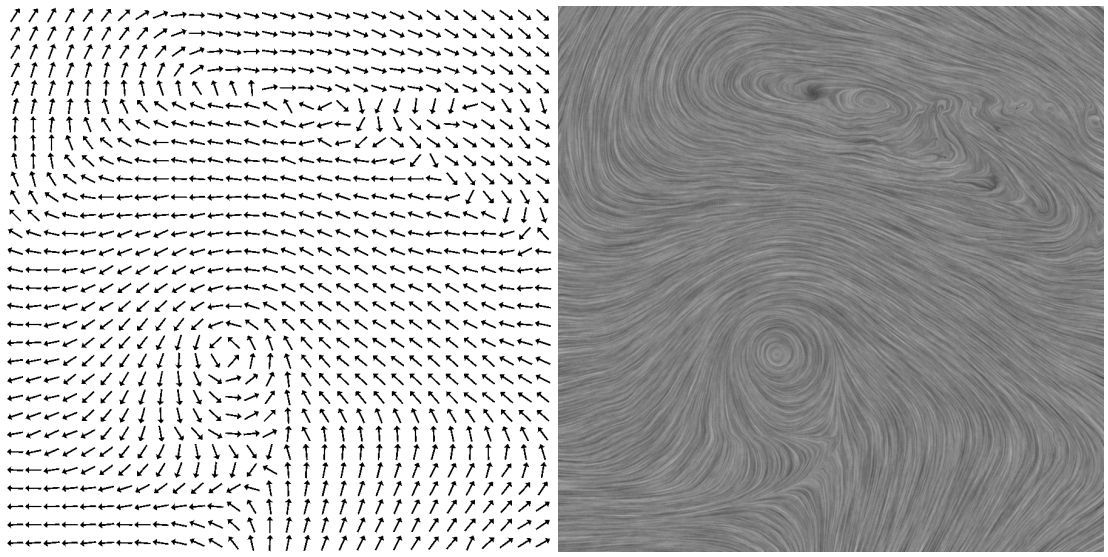
1.1 Change log

- 2017-09-01: Initial Version
- 2018-09-04: Typo fixes
- 2020-09-02: Update for 2020, Update for distance teaching.
- 2021-08-11: Typo fixes, clearer text

2 Documentation

On the following links you can read more about Inviwo and get help with questions related to Inviwo.

- Inviwo: (http://www.inviwo.org)
- Inviwo API Documentation: (https://inviwo.org/inviwo/doc/)
- Inviwo Issue tracker: (https://github.com/inviwo/inviwo/issues)



(a) (b)

Figure 1: Two visualizations of the same vector field, in a) sparse and b) dense.

3 Part 1 - 2D Vector Field Calculus

In this first part we will derive information from a 2D vector field. You are given three different 2D vector fields to explore, which you can change between in the *InputSelectorProcessor* in the supplied network. The first field is an analytic field defined as:

$$v(x, y) = \begin{bmatrix} \cos(x + 2y) - \frac{y}{2} \\ \sin(x - 2y) + \frac{x}{2} \end{bmatrix} \tag{1}$$

The second field is a randomly generated vector field that does not have the repetitive patterns from the first field. Finally we have a cross section of a real world hurricane data set. The data set consists of a simulation of Hurricane Isabel from 2003, in which we see the eye of the hurricane and the winds around it.

The vector fields in Inviwo are represented by an image with 2 channels. Each channel has a 32 bit float value, the first channel has the x component and the second channel has the y component. All code you write in this lab will be written in fragment shaders, as with all fragment shaders it will be called once per fragment. In our case, there is a one-to-one mapping between pixels and fragments. In the shader you can look up the vector field at position (x, y) with:

```
vec2 velo = texture2D(vfColor,texCoord_.xy).xy;
```

The information you are going to derive for each pixel is the magnitude, divergence and rotation as follows:

$$\text{magnitude:} \qquad |V| \quad = \sqrt{\left(v_x^2 + v_y^2\right)} \tag{2}$$

$$\text{divergence:} \quad \operatorname{div} V(x,y) = \frac{\partial_{V_x}}{\partial_x} + \frac{\partial_{V_y}}{\partial_y} \tag{3}$$

$$\text{rotation:} \quad \operatorname{rot} V(x,y) \; = \frac{\partial_{V_y}}{\partial_x} - \frac{\partial_{V_x}}{\partial_y} \tag{4}$$

Where $\frac{\partial_{V_a}}{\partial_b}$ means how the a-component of vector field V is changing when the b-coordinate is changing, e.g. $\frac{\partial_{V_x}}{\partial_y}$ means what happens to the x-component in the vector field when the y-coordinate changes. In analysis this is called the partial derivative and we are going to approximate them using the central difference theorem as

$$\frac{\partial_V}{\partial_x} \approx \frac{v(x+\Delta x, y) - v(x-\Delta x, y)}{2\Delta x} \tag{5}$$

$$\frac{\partial_V}{\partial_y} \approx \frac{v(x, y+\Delta y) - v(x, y-\Delta y)}{2\Delta y} \tag{6}$$

where Δx is a small stepping distance, usually the size of a pixel. Note that in equations 5 and 6 we write $\frac{\partial_V}{\partial_x}$ without the subscript on V. This means we have a column vector that consist of the changes in both x and y in the vector field, e.g.:

$$\frac{\partial_V}{\partial_x} = \begin{bmatrix} \frac{\partial_{V_x}}{\partial_x} \\ \frac{\partial_{V_y}}{\partial_x} \end{bmatrix} \tag{7}$$

The following three tasks are to be implemented in *vectorfieldinformation.frag* [1]. The *vectorfieldinformation.frag* is the fragment shader used in the processor *VectorFieldInformation*. You are given a workspace (*lab3part1.inv*) which is set up to get you started on these tasks.

Note: Since the code you write is only in fragment shaders you do not need to recompile Inviwo. Inviwo will automatically detect when the shader files have been modified and will recompile the shader for you automatically.

Task 1 — Vector Magnitude:
Implement the function *float magnitude(vec2 coord)* in *vectorfieldinformation.frag* based on equation 2.
How can vector magnitude be extended to 3D?

Task 2 — Divergence:
Implement the function *float divergence(vec2 coord)* in *vectorfieldinformation.frag* based on equation 2.
How can divergence be extended to 3D?
Some parts of the image are black, why?

[1] modules/tnm067lab3/glsl/vectorfieldinformation.frag

Task 3 — Rotation:
Implement the function *float rotation(vec2 coord)* in *vectorfieldinformation.frag* based on equation 2.
How can rotation be extended to 3D?
Some parts of the image are black, why?

Task 4 — Colorize the output:
For this task you will use existing processors in Inviwo.
The output from the three previous tasks is a gray scale image, use the Image Mapping Processor and the transfer function of that processor to convert the gray scale image to a color image. The image mapping assumes the input values to be between zero and one and our gray scale image can be in any range and can contain negative values, therefore, the gray scale image has to be normalized before it is connected to the Image Mapping processor. Design a colormap/transfer-function that is suitable for the different derived information, for example, we want to be able to easily see where the value is zero.

4 Part 2 - Line Integral Convolution

Line Integral Convolution is a dense visualization technique for vector fields (see Figure 1). By dense we mean that it uses every pixel in the rendering to encode information. In contrast we have for example Stream lines and arrow/hedgehog plots (see Figure 1) which are sparse visualizations, where information has to be interpolated in the mind between two or more glyphs.

LIC works by blurring a noise image using a convolution kernel derived from the underlying vector field. In practice this is done per output pixel by tracing a streamline from the the center of the pixel. For each step in the streamline tracing we sample the noise image and take the average of all samples.

Task 5 — Implement LIC:
Implement the function *traverse* in *lineintegralconvolution.frag* to traverse the vector field and sample the noise image. Extend the *main* function in the shader to call *traverse* twice, once for forward integration and once for backward integration.

Task 6 — Color coded LIC:
The LIC from the previous task can only show the direction of the field, but it fails to depict some other information, for example the velocity. Use processors in Inviwo to combine/mix the images from Task 4 and Task 5 such that we can also see the derived information in the LIC image.

5 Part 3 - Integration Schemes - Euler VS Runge-Kutta

When integrating vector fields there are different schemes that differ in performance and accuracy. The most simple one is the Euler method in which the vector field is sampled at the current location and the particle is moved in that direction. The Euler integration scheme is fast at the cost of accuracy, and is formulated as:

$$x_{n+1} = x_n + v(x_n)h \tag{8}$$

where x_n is current location at step n, $v(x)$ is the normalized flow direction of the vector field at x and h is the integration step length. This has an error on the order of $\mathcal{O}(h)$, e.g. a high error. To decrease the error it is common to sample the vector field at more than one location, as is done when using Runge-Kutta. Runge-Kutta is a family of integration schemes, often written as RK^N where N denotes the order in the family. Runge-Kutta of order one (RK^1) is the same as the Euler method. RK^4 is a quite common scheme and is already implemented within Inviwo. The RK^4 uses 4 samples in the vector field, k_1,k_2,k_3 and k_4 as

$$k_1 = v(x_n) \tag{9}$$
$$k_2 = v\left(x_n + \frac{h}{2}k_1\right) \tag{10}$$
$$k_3 = v\left(x_n + \frac{h}{2}k_2\right) \tag{11}$$
$$k_4 = v(x_n + k_3 h) \tag{12}$$

The next position is then found by adding a weighted average of the vectors to the starting points.

$$x_{n+1} = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{13}$$

This scheme has an error on the order of $\mathcal{O}(h^4)$, eg. a much lower error than the Euler method.

Task 7 — Euler VS Runge-Kutta:

1. Load the workspace lab3part3.inv. You will see a network visualizing a 3D rotational vector field (see Equation 14) using stream lines rendered as tubes. Explore the network and try to understand what each of the processors do.
2. The processor *Stream Lines* is the processor which creates stream lines by tracing the vector field. It has a property called *Integration Scheme* and another property called *Step size*. Experiment with those two properties to see the difference between Euler and Runge-Kutta. At what step sizes do the two methods "break"?

What does "break" mean in this context?

$$v(x, y, z) = \begin{bmatrix} -y \\ x \\ 0 \end{bmatrix} \tag{14}$$

Task 8 — Extend LIC to use Runge-Kutta: (optional)
When you implemented LIC, you most likely used the Euler integration scheme. Make the necessary updates in the shader to use Runge-Kutta instead of Euler.