TNM067 — Scientific Visualization
4. Tensor Field Visualization

September 27, 2021

1 Introduction

This lab will introduce you to basic tensor field visualization. We are going to derive a 2D tensor field from the 2D vector field we explored in lab 3 by finding the partial derivative of that field, also known as the Jacobian. This tensor field will then be visualized by using elliptic-glyphs. In the final part we will color the glyph to also visualize the orientation of the tensor.

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

3 Part 1 - The Jacobian - Derivative of a vector field.

From scalar field analysis we know that when we take the derivative of a scalar field in $\mathbb{R}^n$ we end up with a vector with n components for each input value, e.g. for a 2D image we get a 2D-vector. When we take the derivative of a vector field we have to treat each component separately, in other words, the derivative of a vector field in $\mathbb{R}^n$ is a a matrix with $n \times n$ components. This matrix is commonly refereed to as the Jacobian where each component is one of the partial derivatives. The Jacobian in $\mathbb{R}^2$ would look like:

$$J = \begin{bmatrix} \frac{\partial_{V_x}}{\partial_x} & \frac{\partial_{V_x}}{\partial_y} \\ \frac{\partial_{V_y}}{\partial_x} & \frac{\partial_{V_y}}{\partial_y} \end{bmatrix} \tag{1}$$

Since our vector field is not a continuous function but values on a discretized regular grid[1], we have to approximate the partial derivatives using the central difference theorem as

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

$$\frac{\partial_V}{\partial_y} \approx \frac{v(x, y + \Delta y) - v(x, y - \Delta y)}{2\Delta y} \tag{3}$$

where Δx and Δy are small stepping distances, usually the size of a pixel. Note that in equation 2 and 3 we write $\frac{\partial_V}{\partial_x}$ with out the subscript on V. This means we have a column vector that consists of the changes in both x and y in the vector field, e.g:

$$\frac{\partial_V}{\partial_x} = \begin{bmatrix} \frac{\partial_{V_x}}{\partial_x} \\ \frac{\partial_{V_y}}{\partial_x} \end{bmatrix} \tag{4}$$

For task 1, set the startup project in Visual Studio to “inviwo-unittests-tnm067lab4”.

Task 1:
Implement the function in `jacobian.cpp` based on the equations above. There are a bunch of unit tests written for this task that tries to capture common mistakes, see Table 1 for a description of each test.

4 Part 2 - Glyph-based visualization.

The following tasks are to be implemented in *tensor_glyphrenderer.geom*. The file is a geometry shader, which is a shader between the vertex shader and the fragment shader in the shader

[1] In this lab the data in the grid is derived from a continuous function and we could analytically find the derivative of our function, but the code we write should work for any vector field, hence we cannot use the known formula.

Test name	Test description
`FullTest`	Test all components, the Jacobian has to be exactly correct. If this test passes, all other tests will pass
`IgnoringScaling`	This test will pass when the Jacobian as a scaling factor is incorrect, but everything else is correct
`OnlyDiagonal`	This will only check the values on the diagonal of the Jacobian matrix
`OnlyNotDiagonal`	As previous test, but will test the upper right and lower left values of the Jacobian
`OnlyColNRowM`	Will only look at one component in the matrix, eg at row N and col M (4 tests in total)

Table 1: Description of the different unit tests for Task 1.

pipeline. The input to the geometry shader is defined by the primitive type used when rendering, for points we get a single point, for lines we get 2 points and for triangles we get 3 points[2]. The geometry shader can output either points, line_strips or triangle_strips. In our case our input is a single point (the origin of the ellipse) and the output is a triangle_strip (the ellipse).

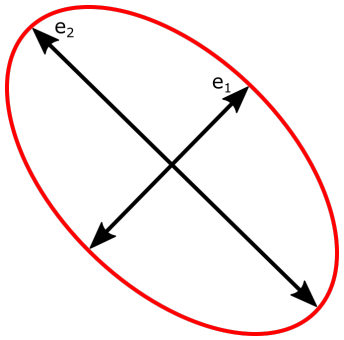

Figure 1: Eigen vectors e_1 and e_2 used as major and minor axis in an ellipse.

A common way to visualize a tensor field is to use glyphs. In 2D a commonly used glyph is the ellipse, while for 3D ellipsoids are the equivalent. Ellipses are often used because they have one major and one minor axis, and if we set these axes to be parallel to the eigen vectors and the length of the eigen values of the tensor we can decode some information on the orientation[3] and scaling of the tensor (see Figure 1).

We are going to create our ellipses by applying a transformation to each vertex in a circle.

[2]For lines and triangles one can also have adjacency information, then we would have 4 and 6 points instead of 2 and 3. If you are interested you can read more about geometry shaders at https://www.opengl.org/wiki/Geometry_Shader.

[3]By using a single color ellipse, we can only see rotation but will not detect flipping/mirroring.

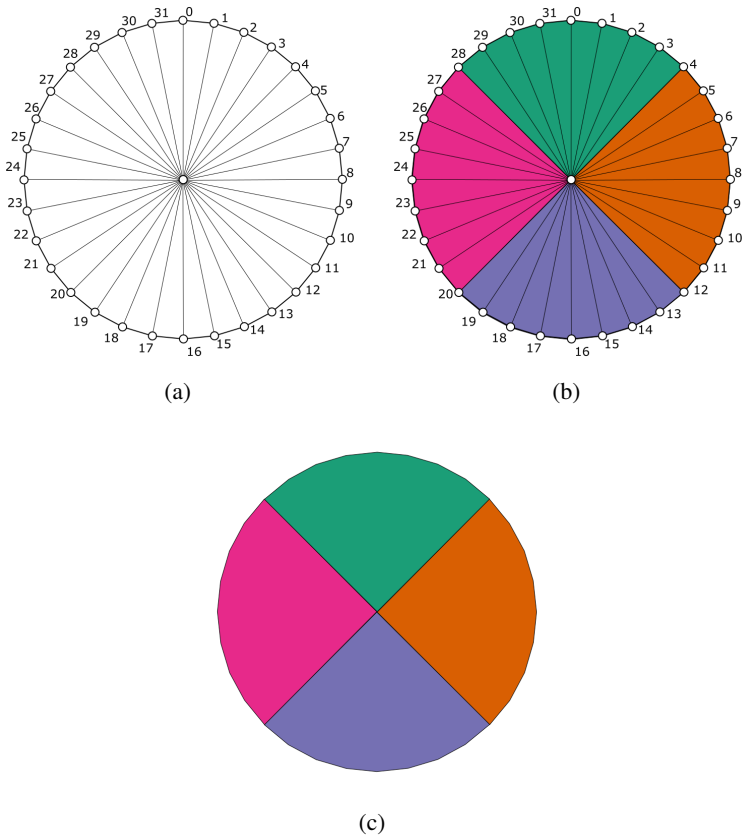

Figure 2: In a) an illustration of the triangles to be created in Task 2 is shown, b) shows the triangles colored by the segment it is within and c) shows the final glyph after Task 4, no transformation applied.

Task 2 — Create a circle:
Create a circle (not yet an ellipse) as the illustration in Figure 2(a).
In *tensor_glyphrenderer.geom*, modify the offset vectors o_1 and o_2 based on the angles a_1 and a_2 using *sin* and *cos* to create all 32 triangles for the circle.

The transformation we are going to apply to our glyphs is based on the Jacobian you calculated in Task 1. For each glyph we have called the Jacobian method from Task 1 and it is available in *tensor_glyphrenderer.geom* as the variable J. We do not want to apply J directly to our vertices, so create first a symmetric tensor from J as $J_{sym} = \frac{1}{2}(J + J^T)$.

Task 3 — Transform circle into an Ellipse:
Transform the offset vectors used to create the circle with the symmetric tensor J_{sym}.

So far we have a single colored glyph, this representation can only show the direction of the eigen vectors, not the sign of the direction, e.g. a flip of an eigen vector or a 180 degree rotation

will result in the same glyph. One way to improve the representation is to color the different segments of the ellipse, as can be seen in Figure 2(b) and 2(c).

Task 4 — Color the ellipse:
Use the value i from the for loop to determine where in the circle we are and set the color accordingly. Colors used in the figures are taken from the set "4-class Dark2" on the ColorBrewer webpage[a]:

- Green: `vec3(0.106,0.620,0.467)`
- Orange: `vec3(0.851,0.373,0.008)`
- Purple: `vec3(0.459,0.439,0.702)`
- Pink: `vec3(0.906,0.161,0.541)`

Hint: Set the Jacobian to the identity matrix while doing this task to see that the colors are at the correct place.

[a] http://colorbrewer2.org/#type=qualitative&scheme=Dark2&n=4