2-Dimensional Vector Field Topology

Kenneth Rohde Christiansen, Aard Keimpema - 1.2 (no source listing)

Department of Mathematics and Computing Science
Rijksuniversiteit Groningen
Blauwborgje 3
NL-9747 AC Groningen, The Netherlands

{k.r.christiansen, k.keimpema}@student.rug.nl

Abstract
In this paper we discuss the use of Vector Field Topology as an intuitive way of visualizing vector fields. We introduce the reader to the basic concepts such as critical points, saddle points and integral curves. We then go into detail how one can construct a topology for a two-dimensional vector field. Finally we present an implementation we programmed using the visualization toolkit VTK from Kitware. The implementation is tested with a self generated dataset containing two critical points (saddle points). We will conclude with an evaluation of VTK.

Keywords: Vector Field Topology, Visualization, Computer Science, VTK.

1 Introduction

The first computer visualization techniques to emerge were methods based on existing experimental visualization techniques. For example, in the field of flow visualization, one would emulate the use of smoke trails. The reason for this is that these techniques are well-known to the researchers and are therefore intuitive for them to work with. Using computer visualization, however, gives us the ability to move beyond these traditional methods by including more information or manipulating the data in various ways. Also, at the same time, when too much information is shown simultaneously then the image often becomes too complicated to interpret. Using data selection we remove unimportant data and instead concentrate on the information which we are interested in.

In this paper we will deal with visualization of so-called flow fields. The concept definition of a flow field is defined as: The velocity of a fluid or gas as a function of position and time.

Flow visualization is an important subfield of scientific visualization. The reason for this is that flow visualization has always had to face the problem of dealing with large and complex datasets. Furthermore, flow fields are not directly visible to the naked eye. Because of these large datasets, feature extraction techniques show a lot of promise as they can dramatically reduce the complexity of a given flow field.

This paper will focus on one of the techniques that try viewing the global structure of the flow by looking at the topology of the vector field [Helman and Hesselink, 1990]. Notice that something like this is impossible to do with the use of experimental methods like smoke trails, but working with a computer makes this possible. This method has been investigated thoroughly since 1987 and there are various papers discussing developing a system to visualize this topology, see e.g. [Helman and Hesselink, 1991], [A. Globus and Lasinski, 1991] or [H. Hagen and Scheuermann, 2003]

2 2D Vector Field Topology

Flow topology is based on the theory of critical points. The reason behind this is simply that the tangent curves near a critical point determine the global structure of the flow. Images of vector field topology therefore display the topology characteristics of a vector field without displaying all kinds of redundant information. This makes it a very intuitive and useful method.

2.1 Critical points

Critical points are points where the vector vanishes (it is zero). There are 6 different types of these critical points; characterized according to the behavior of nearby tangent curves. The position of these critical points can be found by searching all cells in the flow field. Critical points only occur in cells where all components of the vector (2 in the case of 2-dimensional vector field topology) pass through zero. In order to find the exact location of the critical point we will have to do linear interpolation (or something similar if we don't have a rectangular grid).

We can then classify these by looking at the eigenvalues of the Jacobian matrix. The Jacobian matrix for a 2-dimensional vector (u, v) is given by

$$\left.\frac{\partial(u,v)}{\partial(x,y)}\right|_{x_0,y_0} = \left.\begin{pmatrix} \partial u/\partial x & \partial u/\partial y \\ \partial v/\partial x & \partial v/\partial y \end{pmatrix}\right|_{x_0,y_0} \quad (1)$$

As seen in figure 1 these 6 different types are classified by the sign of the real and imaginary part of the eigenvalues. The real part of the eigenvalue gives rise to an attraction (if $RE < 0$) or a repulsion (if $RE > 0$). The imaginary part of the eigenvalue give rise to a rotation of vector field around the critical point. When we have an imaginary eigenvalue we are left with an entirely rotational vector field around the critical point.

The most important critical point is the saddle point here we have a combination of attraction in one direction and repulsion in the other. The importance of saddle points as we shall see in the next section is that the tangent curves near a critical point determine the global structure of the flow.

Besides critical points there are also so-called attachment/detachment nodes. These are points on the wall of an object where tangent curves terminate or begin.

2.2 Creating the skeleton

A skeleton of a flow field is a figure containing the critical points of the flow field and the so-called integral curves connecting them. An integral curve can be thought of as the path a test particle takes when released infinitesimally close to a critical point By adding e.g. an arrow the direction of the integral curves can be displayed; giving a global image of the flow.

To draw the integral curves of a vector field one starts at the classified critical points and follow the vector field from there. The problem we face here is that we do not know the initial direction because at the vector field the critical point vanishes. The solution is that the eigenvectors of the Jacobian matrix at the critical point (see equation (1)) is the direction of the tangent curves at the critical point. As the Jacobian matrix has two eigenvectors there are four different tangent vector as for each eigenvector ν we have as tangent curves the vectors $\pm\nu$. Thus at the critical point we simply follow the vector field starting from the tangent curves. The magic of this, is that if you follow an integral curve from a critical point you will automatically end up in another critical point or an attachment or detachment node. Unfortunately we might miss a critical point due to numerical errors, but that can be solved by attaching the integral curve to a critical point when it comes very close.

The critical points known as saddle points are points where tangent curves arbitrary close to end up in different regions. These are very interesting, especially because the integral curves starting from saddle points are connected. If saddle points are used, flow topology can be thought of as curves dividing the flow into separate regions. The regions borders then consist of the integral curves connecting saddle points and so-called attachment or detachment nodes. Saddle points have the imaginary part of the two eigenvalues equal to 0, and the real part of the one positive and of the other negative.

3 The visualization toolkit VTK

In this project we have decided to use VTK [Kitware, 2004b] for implementing the discussed technique. The reasons for this is that VTK is powerful, freely available and is widely in use. VTK is an object oriented visualization library written in C++

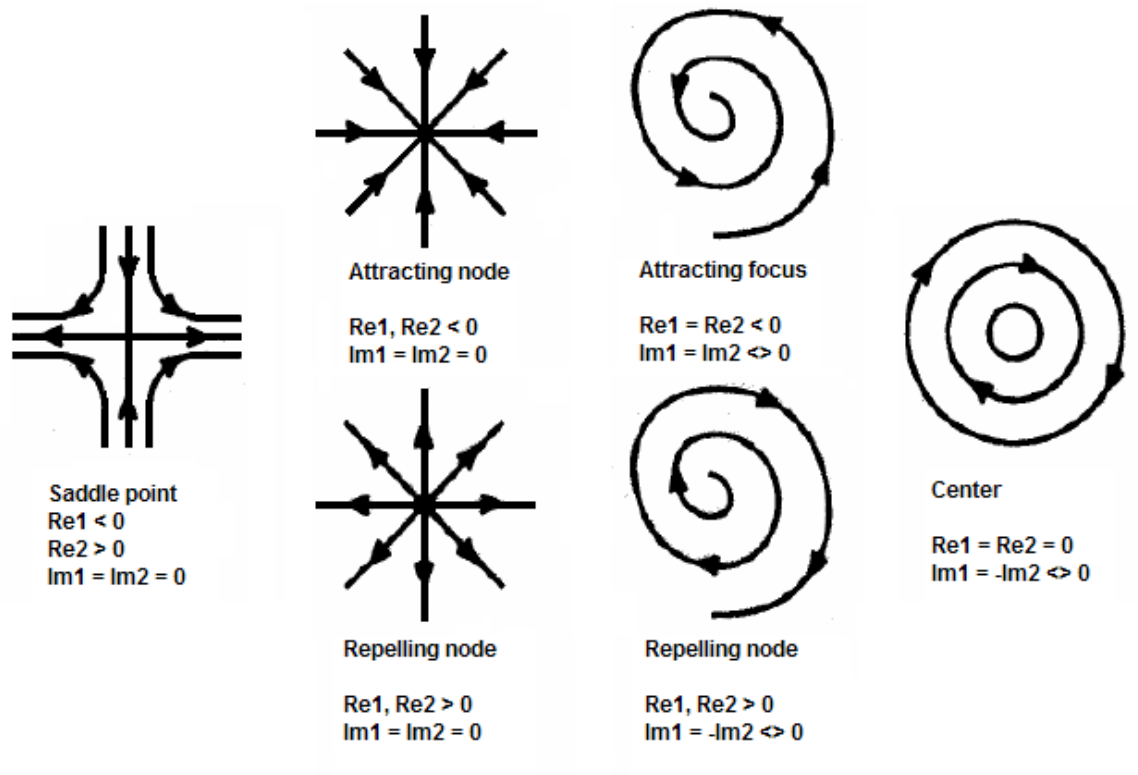

Figure 1: Classification of critical points, here Re1/Re2 is the real part of the first/second eigenvalue and Im1/Im2 is the imaginary part of the first/second eigenvalue.

having language bindings for TCL/TK, Java and Python.

The VTK library consist in total of 700 classes and is therefore somewhat overwhelming to the novice user. Because of it's size it is impossible to give a full account of VTK, therefore we will only give a global introduction here. In section 6 we will give an evaluation of VTK.

VTK knows 9 different types of objects

1. *Render Window*, manages a window into which a object is rendered.

2. *Render Master*, this object creates the actual rendering windows.

3. *Renderer*, does the actual rendering of lights, cameras, and actors.

4. *Light*, illuminates the actors in a scene.

5. *Camera*, controlls the view position and other properties of the camera such as e.g. the focal point.

6. *Actor*, an actual object that is drawn. Actors have associated with them mapper, property, and transform objects.

7. *Property*, represents the attributes of an actor. Examples of attributes are colour, lighting, shading, etc.

8. *Mapper*, represents the geometric definition of an actor. A mapper object may be share by multiple actor objects.

9. *Transform*, an object that specifies the position and orientation of actors, cameras and

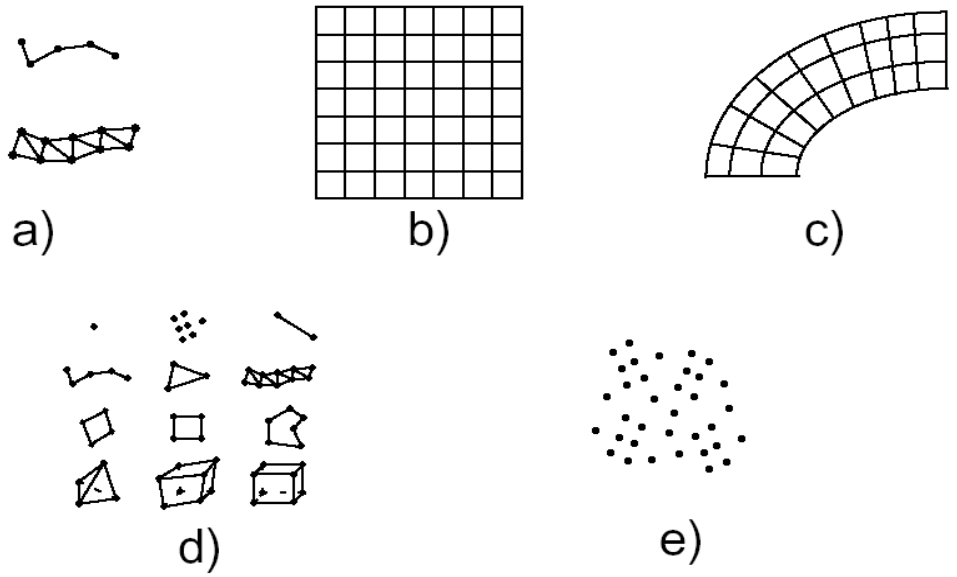

Figure 2: (a) Polygonal data, (b) Structured points (c) Structured grid (d) Unstructured grid (e) Object diagram using OMT notation.

lights.

A visualization in VTK is implemented by creating a network of these objects.

VTK knows 5 different types of datasets as shown in figure 2. In Kitware [1996] a full account of the various data types is given. We will not go into detail about all these data types here and only describe the StructuredPoints data type which we used in our project. The structure of the StructuredPoints data type is shown in figure 3. The first two lines identify the file as a VTK datafile and give a name for the dataset. Then we determine if the data is in ASCII or a binary format. Then we indicate that the dataset is of the Structured Points type. The `DIMENSIONS` attribute give the dimensions of the dataset in each direction. The `ORIGIN` gives the origin of the data. The `Spacing` attribute gives the distance between point in each dimension. With the `POINT_DATA` attribute we set the total number of points. Finally we need to set the type of the data, `VECTORS` for vector data and `SCALARS` for scalar data. We then set a name for the data and give the type of the data which is either integer or float.

```
# vtk DataFile Version 4.0
Title
ASCII|BINARY
DATASET STRUCTURED_POINTS
DIMENSIONS X Y Z
ORIGIN Ox Oy Oz
SPACING Sx Sy Sz
POINT_DATA NO_OF_POINTS
VECTORS|SCALARS  name data type
<<DATA>>
```

Figure 3: The Structured Points data type

3.1 The dataset

We were supposed to get a data set of vortex shedding behind a box, but unfortunatly this data set

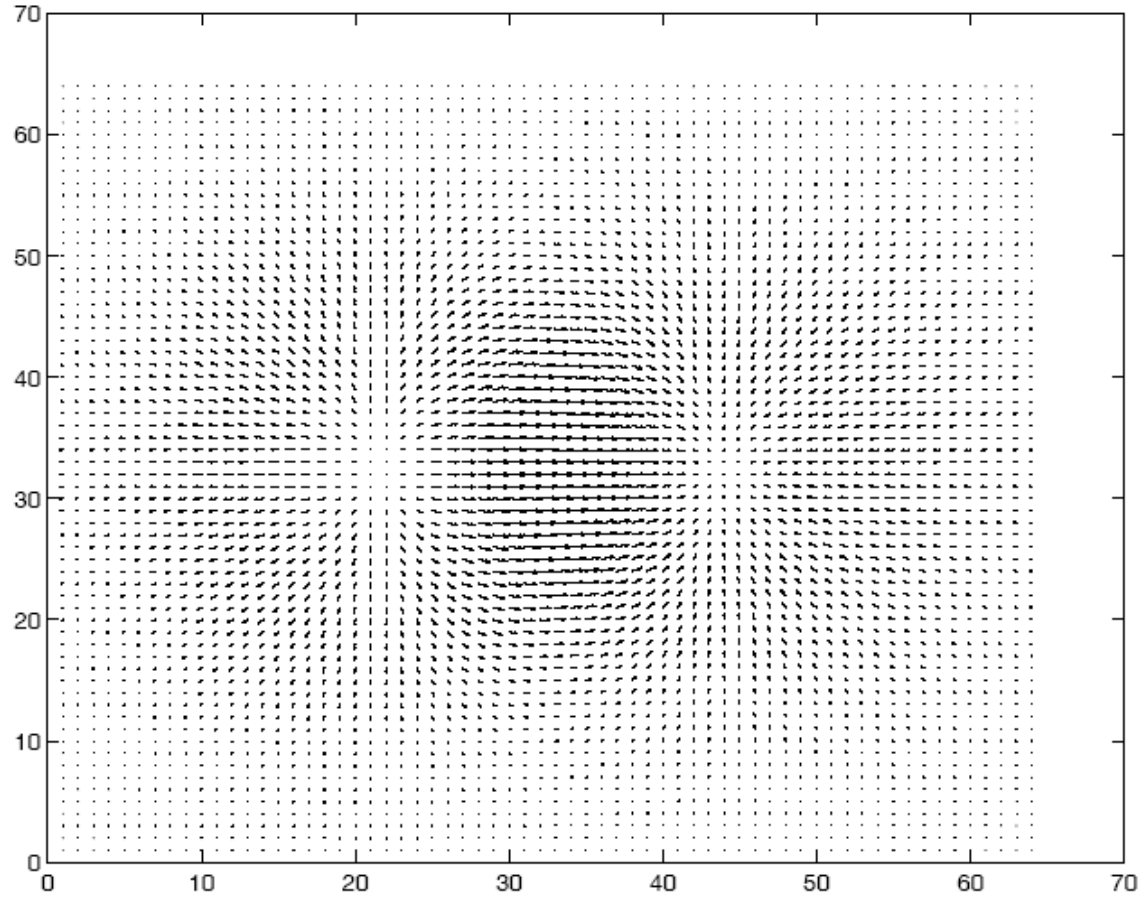

Figure 4: The above picture shows what the generated vector field looks like as visualized by Matlab.

isn't available anymore, so instead we got a 3D dataset of a tornado, from which we could use a slice, and discard one of the vector components. However, the dataset that was given to us was stored in AVS format. Using AVS was not an option as it is a commercial package and the university doesn't have a license. This meant that we had to spend quite some time reading about the file formats and writing a file converter. Eventually we wrote a file converter using information from these two resources: [Kitware, 1996], [Kraak, 1999]. Unfortunately we did not succeed in successfully converting the data. We followed the instruction in forementioned resources, but as the data is in binary form and VTK's error information is quite uninformative, we did not have enough information to solve the problem within reasonable time. The lab instructor provided us with some matlab code to generate an example flow field. This example field is shown in figure 4 and has two critical points (saddle points). We incorporated this example code in a matlab module that writes an appropriate header file and includes the vector field data in ASCII format. The matlab code is shown in figure 5.

```
N = 64;
[x,y]=meshgrid(-2:4/(N-1):2, -2:4/(N-1):2);
z = x .* exp(-x.^2 - y.^2);
[px,py] = gradient(z,4/(N-1),4/(N-1));
quiver(px, py);

fid = fopen('field.vtk', 'w');

fprintf(fid,'# vtk DataFile Version 4.0\n');
fprintf(fid,'Our dataset\n');
fprintf(fid,'ASCII\n');
fprintf(fid,'DATASET STRUCTURED_POINTS\n');
fprintf(fid,'DIMENSIONS %d %d 1\n',N,N);
fprintf(fid,'ORIGIN 0.000 0.000 0.00\n');
fprintf(fid,'SPACING 1 1 1\n');
fprintf(fid,'POINT_DATA %d\n', N*N);
fprintf(fid,'VECTORS vectors float\n');

for i=1:N*N
   fprintf(fid,'%f %f 0.000\n',px(i),py(i));
end

fclose(fid);
```

Figure 5: The dataset was created with the above Matlab code.

4 Implementation

4.1 Locating the critical points

The first step in creating a topological skeleton is locating the critical points. The most obvious method would be to search for zeros in the vector field. However there are practical obstacles in such an algorithm. For instance the zeros are almost never exactly on a grid point and thus we would need an interpolation technique to determine if there was a zero in a certain interval. For such a method to work accurately it needs a high sampling resolution of the flow field.

An alternative method for finding the critical points is using the direction of vectors around a candidate critical point. Because we are using the direction of these vectors to determine if we have a critical point as a by product we also classify the critical point. This is something that would have been necessary as an extra step when only the magnitude of the vector field is used.

In section 2.1 we discussed the properties of tangent vectors near critical points. We can use this information to locate the critical points, e.g. to search for a saddle point we look for a point in the vector field which has two incoming and two outgoing tangent curves.

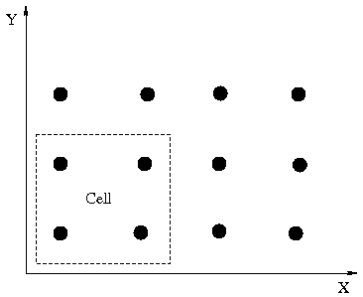

Figure 6: A cell is constructed from four grid points.

In detail the algorithm is as follows.

1. Divide the grid in cells, each cell consists of four points, see figure 6.

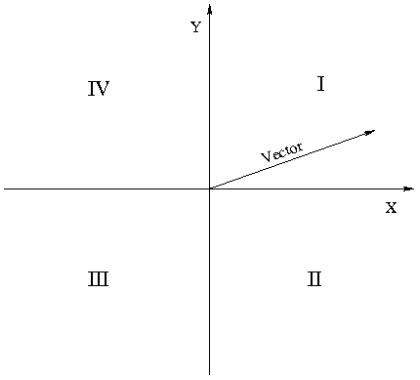

Figure 7: Definition of the four quadrants the direction of a vector can be in.

2. For each cell determine in which quadrant the direction of the vectors are, see figure 7.

3. Determine from the four vectors if we have a critical point. E.g. for a saddle point, we have two vectors entering and two vectors leaving the cell.

The actual C++ code is given in appendix 10.

4.2 Creating the skeleton

Now that we have found and classified the critical points we will create the topological skeleton. As discussed in section 2.2 the integral curves start at critical points and then follow the vector field.

When all integral curves are drawn the critical points will be connected in a topological skeleton. To draw the integral curves we need to start at the tangent vectors of the critical point. These tangent curves are simply the eigenvectors of the Jacobian matrix evaluated in that point. The computation of these tangent curves is not simple. The derivatives in the Jacobian matrix would have to be computed using numerical derivation and interpolation techniques. After this we would have to use a numerical library to compute the eigenvectors. When the tangent curves are found we follow the vector field. Because we in general are not starting from a grid point, we would have to interpolate the direction of the vector field at each step.

Because of these difficulties we have implemented a simpler method. Let ξ be a cell containing a critical point. Instead of the exact tangent curves we simply use the vectors in the corners. When the grid resolution is not too low, this is a good approximation. The algorithm is simply:

1. In each cell containing a critical point, select the vectors at the corners of the cell as a starting point.

2. From all four starting points, follow the vector field to the grid point that approximates the vector direction the best. If the starting vector is directed at the critical point then follow the vector field in opposite direction.

The crucial step is choosing the grid point which approximates the direction of the vector field the best. If we only consider neighboring points we can have 8 different direction (including the vertical neighbors). If we want to consider the next-neighboring points as well, we get 16 different directions (not 32, because then we would count certain directions twice).

Let's consider the case where we only consider neighboring points. How do we determine to which point to travel? To do this we divide the space in 16 regions and assign to each of the 8 neighboring points two of the regions. In figure 4.2 we show the division of space. This method can easily be extended to the case where we also look at next nearest neighbors.

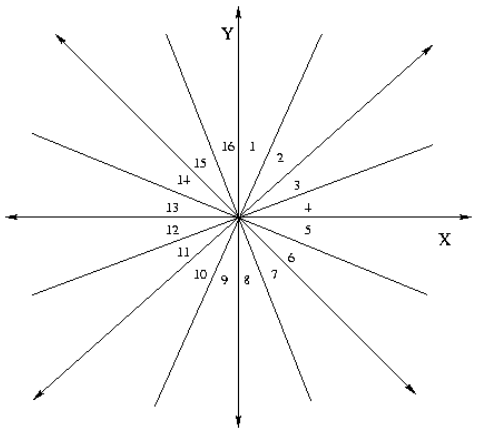

Figure 8: The 16 regions used to identify the direction of a vector

4.3 Drawing the integral curves

Drawing the integral curves is not done in a very sensible way. Supposedly there is a structure in VTK so that you can specify points and then draw the line (possibly interpolated to make it look smooth). Because of our limited knowledge of VTK plus time limit, we decided not to study the class library any further and instead simply drew separate lines between each sample point.

This proved a problem when we wanted to add direction vectors. First of all we wanted to draw the vector on the midpoint on the integral curve but since we never know the length of the line we draw, we also need to store all sample points so that we later can find the midpoint. Drawing the arrow is also not that easy as we need to store a vector for all the midpoints for all the curves and add that to a new StructuredPointsGrid which we then need to pass though a filter and then visualize.

We came up with an alternative - and frankly quite simple - method for visualizing the direction. The idea was to start with drawing a red line and then per step make the line less red and more green, so that it after some time would be fully green. Since we draw all lines starting from the critical points - also the ones ending there, we would sometimes have to start drawing red instead and change the color to green as we draw the curve pieces.

5 Visualizing the data using VTK

Our generated dataset is stored in StructuredPoints format. This means that we get structured points out of it and attached vectors. This is quite handy since we want to work with vectors, but visualizing these vectors directly from the format proved impossible. What we did instead was creating a StructuredPointsGrid and then added the vectors manually, as well as, adding the grid points manually. When this was done we could visualize them simply by using the Hedgehog filter, which created a line (polydata) for each vector.

The standard way of visualizing something in VTK is creating a mapper and an actor from the mapper than you then add to the scene. One of the most used mappers is the PolyDataMapper, which requires some kind of polydata. In order to get this polydata you need to use a filter. The hedgehog filter for instance creates a line (polydata) for each vector.

In our project we are visualizing 3 things: The outline of the dataset, the vectors in the dataset and then an approximation to the topological skeleton. Additionally we have one extra actor, which shows the legend.

6 Evaluation of VTK

The learning curve of VTK is steep, this is caused by two factors. First of all, there is very little documentation available. On the web there was no tutorial or user guide available, only a reference of the VTK classes.

Kitware, the maker of VTK, has decided to earn its money not by selling VTK licenses but by selling books, which is a welcomed business practice. Unfortunately, this means that most documentation is available in these books. There are two VTK related books available from Kitware; one is a general introduction to scientific visualization with some code examples in VTK and the other is a user guide.

The first book called “The Visualization Toolkit” [W. Schroeder and Lorensen, 1996] would better have been called “Scientific Visualization using VTK”. Apart from the class reference at the end (which is also available on the web) the book is mostly about scientific visualization with at the end of each chapter some information on how the information in that chapter can be used in VTK. It is very limited on VTK (despite the title).

The second book was impossible to be used as an auxiliary for this paper. It is not in print anymore and it is not available at our library. This is quite a unfortunate as we would have found it quite handy when learning VTK.

The second reason is VTK itself, which is very complex in its class structure. VTK has a huge class library, which in itself shouldn't be that much of a problem as Java and C# has that as well. The problem is more the relation between all the classes and the inconsistent naming. For instance most of the filters ends in Filter, but not all. This makes it very hard to grasp the underlaying concepts.

The biggest problem with VTK is that it contains a lot of different structures where in you can store your dataset. Getting to know these and the differences between them is not an easy task. The

way you read your data file and the format of the data file determine which structure that you will use, and converting to another structure is either not always possible or it is at least not obvious how to do it. The problem lies in the fact that the functions that you often want to use require your data to be in another format.

The data we generated was stored as structured points (as vectors) and this proved to be a problem. If our data had been of the format NASA PLOT3D, we could have generated velocity vectors with just a few functions calls. It would also be quite easy visualizing the vectors. How to do this with structured points was not that obvious to us.

VTK is supposely highly portable, which means that it should work on almost any platform. Unfortunately, it uses non-standard tools like Kitware's own build system CMake [Kitware, 2004a]. It proved to be quite difficult to compile and install VTK on a resent Linux system, and we never got it working at school. We ended up installing a separate Linux system and spent well over a day getting it installed. We find it quite a shame that Kitware don't provide installation packages (RPM, deb, etc.) for the most popular Linux systems; that would have made it a lot easier to use.

7 Results

Finally, we present the results of our program when applied to the generated example dataset. When locating the critical points, a slight problem of our method arises. On the boundaries of the dataset there are only three vectors available, thus, our algorithm is unable to locate critical points on the edges of the dataset. This is not a problem because a dataset that has a critical point at the border of its data set is arguably taken over the wrong domain. The information contained in the flow around critical points is essential to the understanding of the flow, a data set should therefore never have a critical point at its borders.

As discussed in section 4.2 the integral curve drawing algorithm we use, uses the vectors at the four corners of the cell containing a critical point as a starting point. We then follow the vector field from each of the four starting vectors. Because we end at a grid point after each step of the algorithm, which is an approximation, we do not follow the vector field a full accuracy. Essentially we divide the 360° of space in N 2-dimensional "cones" of $360/N$ degrees each. This means that if at each step of the algorithm we want to end on a neighboring point (including the diagonal neighbors) we divide space in 8 "cones" of 45° each.

In figure 9 we show the topological skeleton created using an algorithm where we only look at neighboring points. It is quite clear that this is to coarse an angular resolution. To remedy this we refined the implementation to the case where we also consider next neighboring points as ending points as shown in figure 10. This gives a much better representation of the vector field. It is still not perfect because the two integral curves should meet but as the meeting point is between two points this is an inherent limitation of the method. However, as to an interpreter of the diagram this fact is quite easy to seen very little information is lost. Additionally, we see that the curve is not perfectly smooth everywhere. This can solved by further refining the angular resolution.

8 Conclusion

In this assignment we constructed a program to create a topological skeleton of a 2-dimensional flow field. The program used VTK as its visualization backend. The learning curve of VTK was steep because of its lack of documentation and complex structure. Most of the time spent on this assignment was learning VTK, if only better documentation was available, this certainly could be avoided. This was doubly unfortunate because of the late availability of VTK on the university network this left very limited time for us to do our assignment in. Apart from this we found VTK to very powerful in that it will fit nearly everyones visualization needs.

The eventual tool we developed was able to nicely construct a skeleton of the example dataset and it is unfortunate that we were unable to test it on a real-life dataset. What is clear from the output from the tool is that the method of using topological skeletons give an intuitive picture of the flow field that makes the global structure clear of the flow clear to the interpreter.

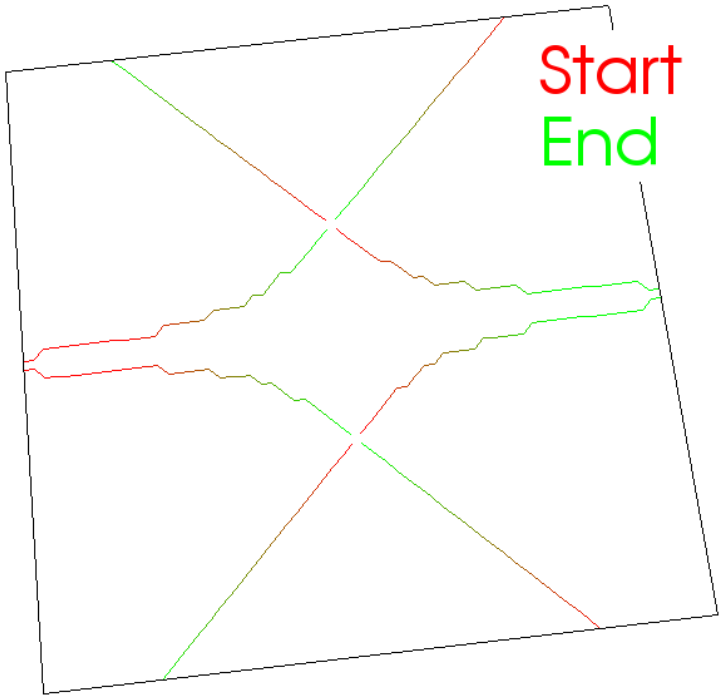

Figure 9: The Flow Field Topology skeleton created using only neighboring points. Red marks the start of an integral curve and green marks the end/direction.

9 Recommendations for future work

Future work will consist of improving the program and making sure it reads our generated tornado.vtk file. This mostly involved fixing the dataset converter, presumably by looking deeper into both AVS and VTK. It is possible to enhance our program to handle 3-dimensional cases as well, i.e. create topology skeletons for 3-dimensional flow fields.

Another refinement would be to use the exact location of the critical points and thus interpolate between the grid point. When drawing the integral curves we would then follow the vector by interpolating the vector field to the each position on the integral curve. This is a somewhat resource intensive method, a more simple method would be to simply increase the resolution of the grid. The vectors of the new data points are determined via interpolation. The major advantage of this method is that we can then our simple method at increased accuracy, but unfortunately it would also increase the memory consumption considerably.

10 Acknowledgements

We want to thank our teacher Mr. Roerdink for an interesting course, as well as our lab assistant Mr. Ogao for his help during this project period.

References

A. Globus, C. L. and Lasinski, T. (1991). A tool for visualizing the topology of three-dimensional vector fields. *Visualization '91*, 33 – 40.

H. Hagen, A. Ebert, R. v. L. and Scheuermann, G. (2003). Scientific visualization: methods and applications. *Proceedings of the 19th spring conference on Computer graphics*, 23 – 33.

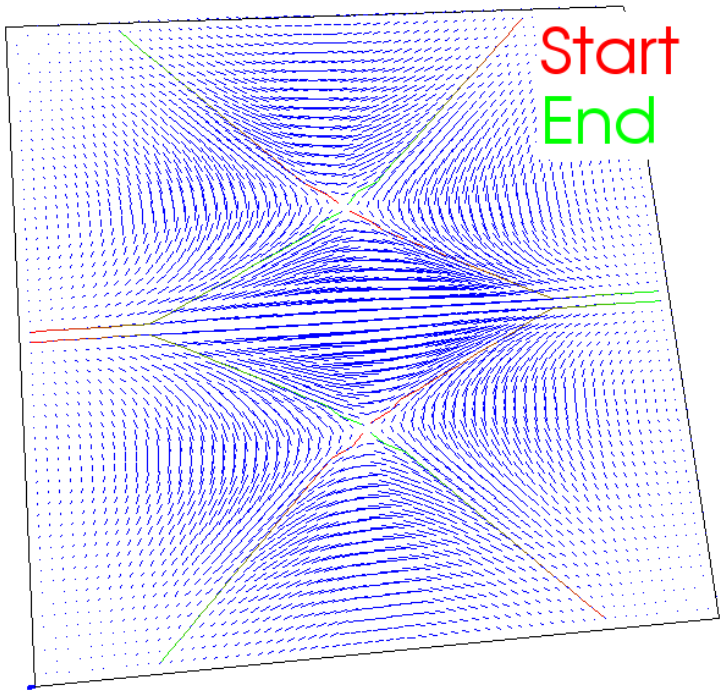

Figure 10: The Flow Field Topology skeleton created using neighboring and next-neighboring points. Red marks the start of an integral curve and green marks the end/direction. In order to demonstrate the accuracy of the method we also included the vector field in the visualization.

Helman, J. and Hesselink, L. (1990). Surface representation of two- and three-dimensional fluid flow topology. *Visualization '90*, 6–13.

Helman, J. and Hesselink, L. (1991). Analysis and representation of complex structures in separated flows. *SPIE Conference on Extracting Meaning From Complex Data*, **1459**, 88–96.

Kitware (1996). File formats for vtk version 4.2. *http://public.kitware.com/VTK/pdf/file-formats.pdf*, 1–19.

Kitware (2004a). Cmake (software). *http://www.cmake.org*.

Kitware (2004b). The visualization toolkit (software). *http://www.vtk.org*.

Kraak, J. (1999). Conversion of avs-files to vtk-files. *http://rc60.service.rug.nl/ oldhpcv/h-pc/VTK/vtk/avs2vtk/man.html*.

W. Schroeder, K. M. and Lorensen, B. (1996). *The Visualization Toolkit*. Prentice Hall, Upper Saddle River, NJ USA, 1st edn.