Time-Coherent Streamline Placement

Bachelor's Thesis

Alexander Baucke

Supervisor
Prof. Dr. Filip Sadlo

Heidelberg, Germany, July 25, 2024

Faculty of Mathematics and Computer Science
Heidelberg University

Declaration of Authorship

I hereby certify that I have written the work myself and that I have not used any sources or aids other than those specified and that I have marked what has been taken over from other people's works, either verbatim or in terms of content, as foreign. I also certify that the electronic version of my thesis transmitted completely corresponds in content and wording to the printed version. I agree that this electronic version is being checked for plagiarism at the university using plagiarism software.

first and last name	city, date and signature

Abstract

A common element used to represent steady vector fields are integral lines capable of showing flow dynamics, called streamlines. There are many possible criteria that can be used to define the quality of a set of streamlines when visualizing a vector field. Commonly used placement properties include the uniformity and length of the streamlines, with even spacing and maximum length being the preferred choice for high visual appeal.

We have noticed that an important criterion for creating visualizations of *unsteady* fields, such as animations, is largely unaccounted for. Currently, algorithms can only animate a vector field frame-by-frame without actually being time-aware. The result is many visually pleasing images, however there is no coherence between each image, as lines are only placed to fulfill the frame-limited optimum.

In this work, an algorithm capable of time coherent streamline placement—that is lines that move as little as possible between frames— is introduced. An image-guided approach based on the work by Turk and Banks will be implemented to generate initial streamline placements according to the above criteria. We then introduce modifications to several components. The first key change revolves around the energy measure, a function used to define the uniformity of line placement by comparing it with a target brightness. Another change is the kernel Turk and Banks use in order to blur the image, which we also adapt for time coherence in combination with a new seeding strategy. We show how the individual changes affect the streamline placement locally and globally, and apply the algorithm to various datasets.

Zusammenfassung

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Contents

1 Introduction

The visualization of vector fields is a heavily used component in many fields like aerospace engineering, geology, fluid dynamics, material science, or the biomedical sector. Common examples include weather systems, rotor design, analysis of marker movement inside of cells, or the magnetic field of a star. Displaying such fields in a human-focused way is invaluable when analyzing the underlying systems' dynamics, and allows for faster and more accurate data analysis.

Contemporary research is mostly focused on continuous and steady vector fields. Lines are usually optimized with respect to a single frame, which in turn has a very pleasing appearance. If we introduce small changes over time, i.e. use an unsteady vector field, it is unlikely those algorithms will prefer lines similar to the ones in the last frame. This causes the streamlines to move about and shift between frames a lot more than the time-induced changes to the field would warrant. The sporadic movement makes it very difficult to perceive the flow change of the field. Therefore, creating an animation of an unsteady vector field's behavior is very difficult, time-consuming, and requires a lot of human intervention in order to achieve good results. This is extremely limiting, as most (if not all) vector fields represent a dynamic system. Therefore, the focus of this thesis will be threefold:

- How a lack of coherence of streamlines in different time steps of a continuous, unsteady field affects the visual quality of a field. A criterion to measure time coherence will also be presented.
- The implementation and underlying ideas of an image-based algorithm generating evenly-spaced long streamlines.
- Adaptations of the algorithm to allow a controlled bias between similarity to a previous time step, and optimality w.r.t. the current step.

We have chosen an image-based approach (opposed to a feature-guided one, see Section 2 for a disambiguation) for our work, because the movement of lines in-between time frames is better suited for an appearance-focused algorithm. Another

reason is that feature-guided algorithms usually act locally, whereas the movement of a line from one step to another is a global constraint. The succeeding sections are structured as follows:

Chapter 2 presents some recent developments and concepts.

Chapter 3 contains fundamentals about vector fields and streamlines.

Chapter 4 is about the base algorithm, the definition of a criterion for time coherence, and changes made to account for time coherence.

Chapter 5 contains the implementation and some design ideas.

Chapter 6 examines the individual changes' effects on different datasets, and we discuss issues, performance, and other measurements.

Chapter 7 concludes the thesis with some ideas for future work and improvements.

2 Related Work

Most of the works published in the field of streamline placement algorithms can be divided into two categories:

2.0.1 Image-Guided

The goal of this approach is to achieve a very uniform spacing of lines, giving an almost "hand-drawn" appearance. Generated images will usually have high visual quality, at the risk of potentially missing or misrepresenting important features.

One of the first and most prominent examples in this category was written by Greg Turk and David Banks [96], which we have adopted as the base for our algorithm as well. In their research paper, a function (called the "Energy Function") is defined such that it maps an input image containing the potential streamlines to a scalar. The scalar represents the quality of the image, roughly defined as the uniformity of the gray scales it contains. Adding/moving/removing/resizing streamlines is done randomly, at every step the energy function is used to determine whether a change gets accepted or discarded. The algorithm is finished when the energy function reaches a lower energy bound, or a maximum number of consecutive rejected steps is exceeded.

2.0.2 Feature-Guided

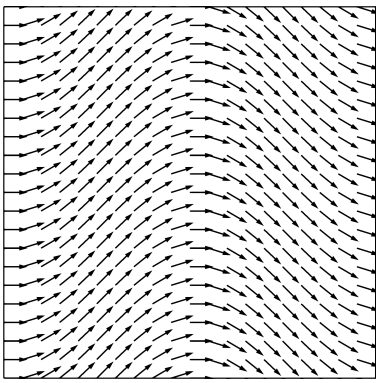
Feature-Guided algorithms examine the underlying field structure before placing seeds. They search for critical points or patterns in the field and then seed around them, capturing them in much higher detail. The resulting images inherently represent the critical points much better, at the cost of some visual appeal compared to the aforementioned group.

2.0.3 Other Relevant Works

- Time Coherence in 2D
- Approach I've used so far due to simplicity

Do not forget to use references [Han+19] like done here [HS19] to enable the bibliography [Jun+17; SJMS19; SRS18; ZRLS19].

3 Fundamentals



(a) A vector field defined by $u(x, y) = (1, \sin(x))^T$, visualized using arrows placed on a regular grid.

In this section, we describe the elementary components used in the remainder of this thesis. Since this work is about the placement of streamlines in vector fields, we start with the fundamentals of the field *vector field visualization* in Section 3.1. Our algorithm uses an image-guided base, hence we will also include some topics from the area of *image processing* in Section 3.2. We conclude this chapter with a brief overview of the roots of unity in Section 3.3, because they are used in the development of an initial prototype in Section 4.1.2.

3.1 Vector Field Visualization

Vector Field A vector field represents how vectorized elements act over a spatial domain. Concerning vector fields representing flow, this means that for every point in a domain, we can obtain the force acting at that point.

More formally, we can define a vector field as a map from n-dimensional scalars to m-dimensional scalars. We can write it as an n-m-valued function, and in this thesis, we will only care about cases of $n = m$ in two and three dimensions. There are several ways to obtain such fields, one is via an algebraic definition such as

$u(x, y) = (1, \sin(x))^T$, giving us a field like the one seen in Figure 3. If we want our force to not only depend on spatial input but also on another scalar like a time component, we write this as $u(x, y, t)$ for the 2D case. We call vector fields *steady* if they are not time-dependent; otherwise, we refer to them as *unsteady*. Another distinction is *continuity*, which is analogous to the algebraic definition of other functions. The fields in this work are all going to be continuous.

Critical Points A vector field can have points with distinct characteristics, called critical points. In the 2D case, only four commonly used critical points exist, which we briefly describe here.

Source Given a field such as $u(x, y) = (x, y)$, at every point applies a force away from the origin. If we think about this as a non-compressible flow, this is equivalent to matter being created at the point $(0, 0)$ and flowing outward. We refer to such a point as a *source*.

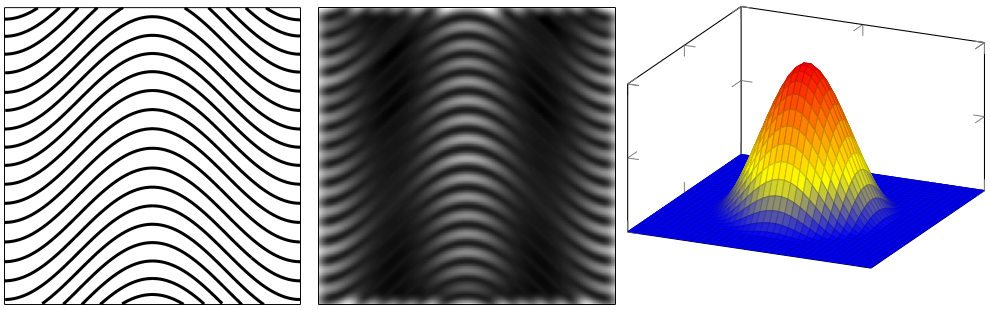
Sink Similarly, $u(x, y) = (-x, -y)$ would give us a *sink* at $(0, 0)$, equivalent to destroying liquid flowing inward.

Saddle A saddle is an area where matter is pinched in one direction and stretched in another, e.g. in a field defined by $u(x, y) = (-x, y)$.

Periodic Orbit $u(x, y) = (-y, x)$ creates circular paths around the origin where, after traveling a certain distance, a particle arrives at the point it started at. These critical points are called *periodic orbits*.

Streamlines Given a vector field u and a point P, we can trace the movement of this point through u by integrating over the field. Intuitively, we can step through the field by choosing the next point $P_n = P_{n-1} + c \cdot u(P_{n-1})$, with c being a step size scale. If we do this an infinite number of times with $\pm c$ close to zero, we end up with a set of points S we have passed through, which defines the streamline. S has two notable properties:

- Every point $P \in S$ inside this set has a velocity equal to $u(P)$. Hence, a streamline is tangent to the vector field at every point.
- No matter which point inside S we use as P_0, we will always obtain the same set S as its streamline. Therefore, any point inside S is a potential *seed* yielding the streamline S.



(a) (b) (c)

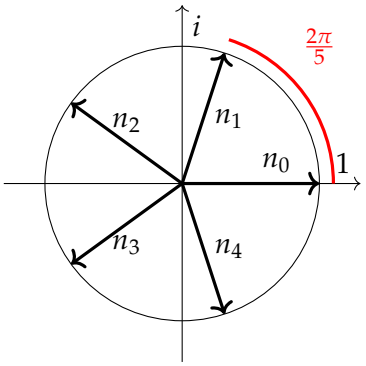
Figure 3.2: A set of streamlines (a) generated for the field in Figure 3(a). (b) Low-pass version of the image after convolving it with the kernel shown in (c).

Spatial Coherence If we want to visualize a vector field, we want its features to be easily identifiable. At the same time, we do not want to introduce distractions or artifacts due to the visualization technique. The deciding factors of uniformity in streamline visualization are streamline length and density. Longer streamlines make for a smoother appearance, whereas many short lines tend to obfuscate and hinder the recognition of important features like the aforementioned critical points. Strong spatial coherence is shown in Figure 3.2(a).

3.2 Image processing

Convolution A process often found in image- or signal processing. A kernel (Figure 3(c)) is applied to every pixel in an image, affecting it and other surrounding pixels by adding or subtracting its value at that position.

Blurring A special type of convolution, making edges in an image less sharp. Note the difference between black and white contrast for Figure 3.2(a) and (b).



(a)

The 5th roots of unity $n_0 ... n_4$ partition the unit circle equally

3.3 Roots of Unity

With i being the complex number, we can use Euler's equation

$$n_j = e^{ji2\pi/k}, j = 0, 1, ..., k-1$$

to obtain k numbers lying on the complex unit circle, which are called the k-th roots of unity. Notable properties are their length of exactly one, and that they divide the unit circle equally. We can convert them to vectors in $\mathbb{R}^2$ using

$$\vec{v}_j = (\mathrm{Re}(n_j), \mathrm{Im}(n_j))^T$$

4 Method

In this chapter, we describe the motivation of and concepts used for developing an algorithm supporting time coherence. We start with an implementation that was ultimately deemed infeasible in Section 4.1, including some mathematical concepts and the reasoning behind foregoing it in favor of a more global approach. Section 4.2 is about the algorithm we decided on using, with motivation, a concept, and finally some implementation ideas for time coherence in Section 4.3, Section 4.4, and Section 4.5 respectively.

4.1 Initial Greedy Algorithm

> **Motivation:** Since time coherence only exists as a connection from one set of streamlines to another, we started by creating an algorithm capable of generating such streamlines. We could then use this algorithm to generate sets of streamlines for time frames without time coherence as a negative example and to illustrate the effects of its absence. Initially, the algorithm was developed for the 2D case, the extension to 3D is shown in subsection 4.1.2.

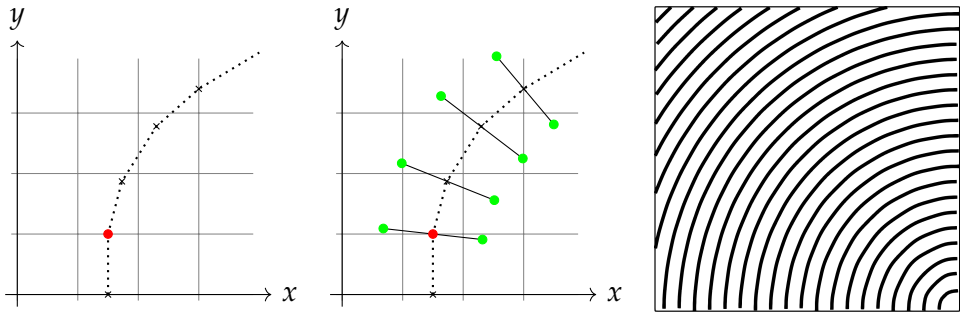
The algorithm uses two operations, streamline traversal and seed filtering, which are executed round-robin until there is no more room for new streamlines. The result of the algorithm is a space-filling set of streamlines with even spacing in 3D fields.

We use 2 essential parameters to control how the images are generated. The first one, d_s, is the neighbor search distance. It controls the distance from the current streamline where new streamlines start at, allowing us to fill the whole space. In order to add longer liens while guaranteeing even spacing, we introduce a second parameter d_c, which is the neighbor cutoff distance. As soon as a streamline comes within this distance to another streamline, it is terminated.

The typical range for d_c is $0.5\,d_s \leq d_c \leq 0.75\,d_s$. Making d_c smaller than $0.5\,d_s$ allows streamlines to get very close, introducing visual clutter and a crowded appearance.

At values $\geq 0.75\, d_s$, a lot of streamlines are very short in forward or backward time due to the proximity of d_s and d_c, causing most of them to be removed immediately.

4.1.1 Two-Dimensional Implementation



(a) (b) (c)

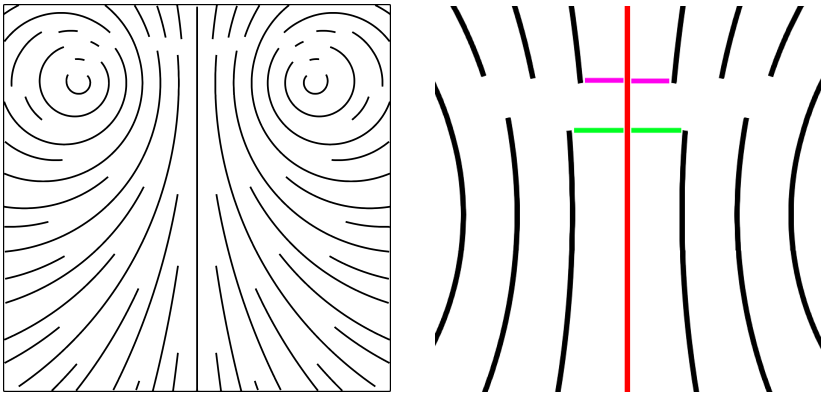
Figure 4.1: (a): Forward and backward streamline integration through the field starting at the seed (red). The chosen sample points are drawn as a cross, and include the seed. (b): Equidistant seed candidates (green) obtained from the normals at the sample points are chosen for the next iteration. (c): Lines with equal distance generated by our algorithm in a field defined by $u(x, y) = (y, -x)$.

Streamline Traversal

First, we choose a seed point from a list of candidates (initially an arbitrary point from the dataset) and remove it from the list. Then we integrate forward and backward (fig. 4.1 (a)) to obtain the other points on the streamline, until a number of steps is reached, we cross the bounding box, or we get too close to another streamline. If the total length of the streamline is too small, we remove it.
To obtain the seed candidates, we compute the normals of the field at these points, and add points a distance d_s away from them to the list (fig. 4.1 (b)).

Seed Filtering

The number of neighbor seed candidates is roughly $2n$, where n is the number of samples of the streamline we are integrating. To quickly remove seeds that cannot produce a good streamline, we use two filtering conditions. The first condition arises from the fact that roughly half the seeds generated by a streamline L_1 during the traversal process will lie close to - if not exactly on - the preceding streamline, L_0. This happens because the seeds are placed a distance d_s away, while the streamlines are mostly that same distance apart themselves. The second condition is the distance to other seeds, as starting a streamline from a seed that is too close to another will immediately end it. Therefore, we filter two sets of seeds: The ones generated during the traversal process and those present in the entire field.



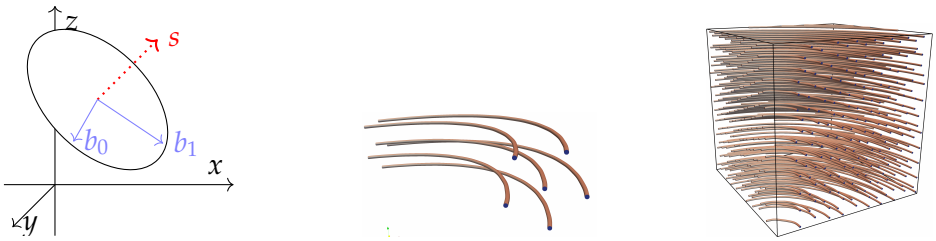
(a) (b)

Figure 4.2: (a): The streamlines created in a double gyro dataset. (b): Top center view of (a). New streamlines created from the center streamline (red) are cut off due to reaching d_c (magenta). New streamlines are only drawn at distance grater than d_c, producing the gap between the upper and lower three streamlines (black). The lower streamlines continue at a distance of d_S (green).

Line Cutoff and Optimization

In order to quickly filter the $2n$-pairs for points on the parent streamline, we store the parent's points and remove any of the $2n$ seeds from the current step if they are too close. When removing other seeds we come across, we employ a grid with a spacing of d_c and use a dictionary to obtain lists of points inside the grid cell. If we add a point during integration, we simply look up the coordinate and its 8 neighboring cells as a key to points in the area. If a seed closer than d_c is found in those cells, it is subsequently removed. This allows for an easy way to end the integration when getting too close to another streamline as well: We need only keep track of whether we came close to another streamline's points during integration (provided the integration step size is less than d_c, which is usually the case).

4.1.2 Steady Field Streamline Placement in 3D



(a) (b) (c)

Figure 4.3: (a): The normal plane of a streamline segment s with two orthonormal basis vectors b_0, b_1. (b): A clear view of the center streamline with five neighbor streamlines evenly distributed at distance d_S. (c): A filled cube containing 254 streamlines. Notice the resemblance to fig. 4.1 (c).

The most important part to generate streamlines in 3D is obtaining the seed locations. In three dimensions, a vector has infinitely many normals, which all lie on a plane it is a normal of. Therefore, we define a number of points to evenly distribute around the streamline. The process of obtaining these points is as follows. Instead of the two trivial normals in 2D, we now construct a normal plane around the streamline trajectory at the streamline's points. For this, we find two vectors which are linearly independent of each other and the trajectory, and then orthonormalize them to receive two orthonormal basis vectors b_0, b_1 (see fig. 4.3 (a)). Having found b_0 and b_1, we can use the roots of unity (fundamentals, **??**) to find evenly spaced points on the unit circle. With i being the complex number, we obtain k roots of unity via

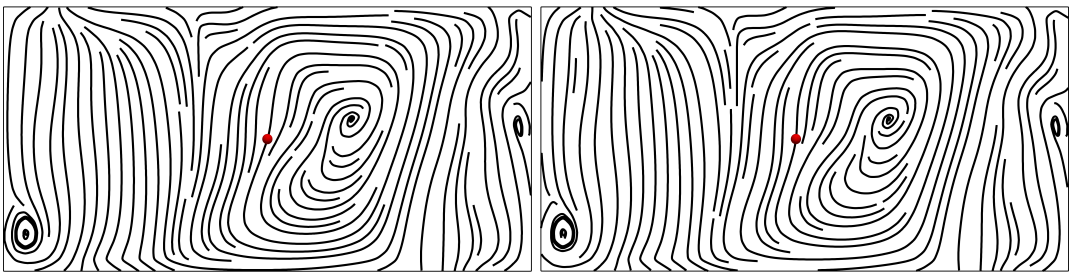
$$n_j = e^{ji2\pi/k}, j = 0, 1, ..., k-1$$

Since the magnitude of n_j is always one, we do a simple basis transformation into the 3D frame of reference. We obtain the j-th 3D-vector v_j from the j-th root using our basis vectors b_0 and b_1:

$$v_j = re(n_j) * b_0 + im(n_j) * b_1$$

This gives us k uniformly placed vectors in the normal plane around the current streamline segment. The rest of the algorithm stays largely the same as in 2D, the grid used for seed filtering is extended into the 3rd dimension accordingly. See fig. 4.3 (b, c) for an example using a 3D vector field.

Shortcomings

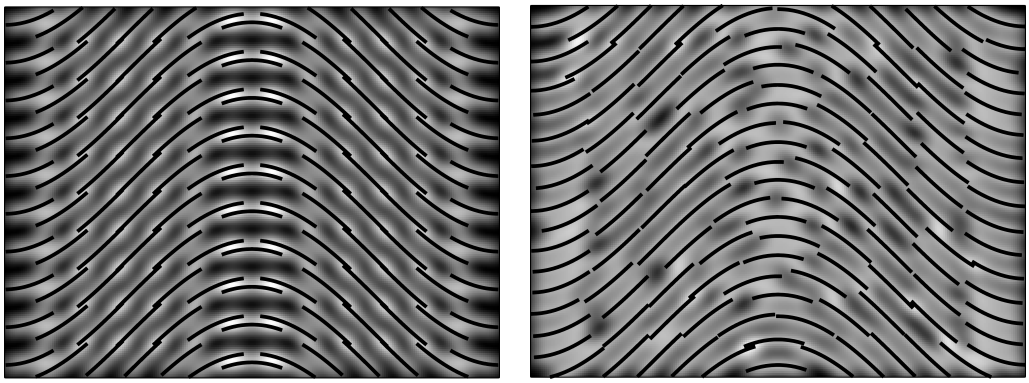


(a) (b)

Figure 4.4: Two different line placements due to a seed choice difference of only $1e-4\%$ of image width.

While this algorithm can quickly fill a space with decent streamlines, the main problem is the inability to cope with small changes to streamlines. This is mainly a result of the strong hierarchical nature: Since every streamline after the first comes from its predecessor, a change at the "root" can have drastic consequences for the succeeding streamlines (see fig. 4.4). In the context of time coherence, this makes the algorithm unsustainable, as an unsteady field is practically guaranteed to change at least slightly in the whole domain. To make the streamlines time coherent, we need to be able to move streamlines around without affecting the global scope too much. With this approach, changing the position of a streamline after the generation of its neighbors causes a re-evaluation of every streamline it is a predecessor of. Due to this problem, we do not see an effective way to implement time coherence, and supersede this algorithm in favor of an image-guided one.

4.2 The Image-Guided Algorithm by Turk and Banks



(a) (b)

Figure 4.5: (a): Lines placed with seeds on a regular grid. (b): The same lines after optimization through seed shifting. Notice the more even grayscale image in the center and on the sides. Both images contain 130 lines, with a length of 10% of the screen width each.

In this chapter, we introduce the image-guided streamline placement algorithm developed by Greg Turk and David Banks. The algorithm forms a more suitable basis for time coherence due to how it reacts to changes to specific lines. More specifically, it retains a good amount of control regarding the line placement and length, while still allowing the lines to move and relax the spacing between them.

4.2.1 Overview

The following section will be about the three core components that make up the image-guided sections of the algorithm: the lines themselves, the low-pass filter, and the energy measure. The filter is used to create a blurred image of the streamlines, where the lines themselves are brighter, and empty spaces darker (fig. 4.5(a)). An optimal line placement w.r.t. density (neither too crowded, nor too sparse) is reached when the low-pass image has roughly the same gray coloring everywhere. We can intuitively define the energy measure as the squared differences of the blurred image from a uniform grayscale background, also referred to as the *target brightness*. Simply knowing the energy however is not enough, we need to be able to manipulate the line placement effectively in order to minimize it. This is achieved by using various randomized changes to the position (fig. 4.5(b)), length, or number of streamlines, and is described in greater detail in section 4.2.3. Whenever a change leads to a decrease in energy, the change is accepted and, if not, reverted.

4.2.2 Energy Measure

The method used by Turk and Banks defines three important components to measure image quality as the sum of deviations of a low-pass image from a uniform greyscale target.

1. The first component is a collection of (straight) line segments from each line, each of which can be converted to a line formula of the form $Ax + By - C = 0$. They refer to the image defined by these purely analytical lines as I, however this image is never actually created and exists purely implicitly.

$$I(x,y) = \begin{cases} 1, & \text{pixel lies on line} \\ 0, & \text{else} \end{cases}$$

2. The second component is the low-pass filter L. It uses a kernel to generate the filtered image of a line. Given a falloff distance R and $r = \sqrt{x^2 + y^2}/R$, the kernel is defined as:

$$K(x,y) = \begin{cases} 2r^3 - 3r^2 + 1, & r < 1 \\ 0, & r >= 1 \end{cases}$$

   Every pixel within the bounding box of a segment + filter diameter has its brightness altered according to this kernel. Due to the implementation of the convolution and the scaling of the filter brightness, it is guaranteed that every pixel reaches a maximum brightness of 1. This is invariant w.r.t. the number or length of segments as long as only a single, straight line section is considered. Conversely, the only way to get a brightness grater than one is either via a curve in the line or two lines being next to each other. This is immensely useful, as it allows two lines to get close to each other from the ends, but not from the sides (fig. 4.5(b)).

3. In order to determine the energy of the image generated by the kernel application, the following expression (called the *energy function*) is used:

$$E(I) = \int_x \int_y [(L * I)(x,y) - t]^2 \, dx \, dy$$

   With t referring to the *target brightness*, in their source code the number one is used.

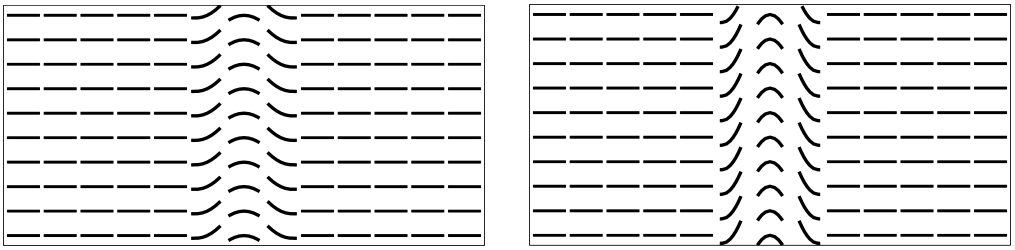
It is pivotal for this energy measure to react to minimal changes for the algorithm to successfully converge. Therefore, this algorithm does *not* use a simple rasterization and blur technique, as the initial discretization/rasterization of the line would remove too much information, even when using anti-aliasing or other more sophisticated rasterization techniques like Bresenham's line algorithm. While we discuss the actual implementation in chapter 5, the takeaway here is that the line remains in its analytical form as segments between points. Then, every point inside the bounding box around every line segment calculates its brightness based on the line equation, not by simply blurring pixels on the line, leading to much higher sensitivity.

4.2.3 Randomized Optimizations

Turk and Banks define six actions:

- **Insert, Delete:** Add or remove a line from the image.
- **Lengthen, Shorten:** In-/Decrease the length of a line on one or both ends.
- **Combine:** Join two lines head-to-tail.
- **Move:** Translate the seed of a line by a small distance.

These actions are selected randomly with random parameters, then applied to a random line. The algorithm terminates after an energy range was reached, or accepted changes become rare enough to not introduce changes anymore.
If the change was deemed beneficial according to a decrease in energy, it is accepted, otherwise the changes are reverted. This causes a "drift" of the lines toward a more uniform energy level. Naturally, this depends heavily on the choice of t. If t were to be chosen closer to 2, the image would become very crowded to reach the increased target gray level.



(a) (b)

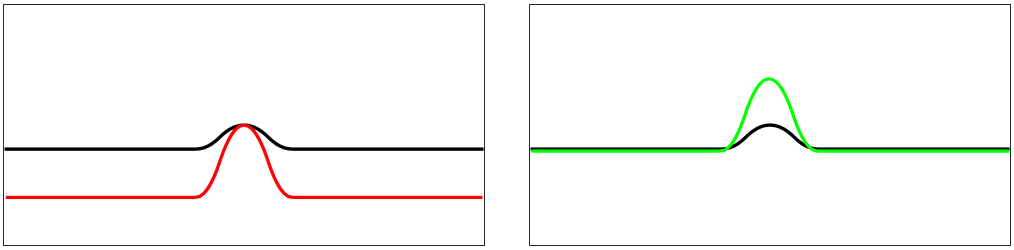
Figure 4.6: (a), (b): A vector field undergoing a change over time, increasing the amplitude at its center. It is described by $u(x, y, t) = (1, \sin(5t\pi x)$ if $0.4 \leq x \leq 0.6$ else $0)^T$ for $t = 1$ and $t = 3$ in (a) and (b) respectively.

4.3 Why Time Coherence is Necessary

When visualizing a vector field using streamlines, it is desirable to capture the features of the field as accurately as possible. For image-based solutions, this means that we are interested in a line placement that poses little to no visual distractions from the general field flow. Common distractions include artifacts like shapes arising from the position of lines, but not being created by the actual field. Other forms of visual clutter include strong variations in density, or empty spaces, which make it very hard to judge a field's behavior in these regions. Usually, there is a trade-off between the accuracy of capturing a field and visual clarity, with image-guided approaches favoring the latter.

A new type of visual artifact is introduced when we add a time axis to our data. Where before, this approach could solely focus on optimizing a single image w.r.t. visual criteria, we now have to take into account the movement of lines from one time step to another. What we are looking for is a line placement that allows large portions of the image to stay the same between two time steps, i.e. the largest possible "overlap" between the two images. This new criterion may also counteract the spacial distribution of lines, since creating or removing lines will severely impact how the image looks from one time step to the next.

We can illustrate this behavior using a field like the one defined by Figure 4.6. We take a closer look at $u(x, y, t)$ at two time steps, namely $t = 1$ and $t = 3$. The notable change of the field's trajectory from one time step to the next is a strong ridge forming in the center column.



(a) (b)

Figure 4.7: (a): A single streamline seeded exactly at the center of the image at time $t = 1$ (black) and $t = 3$ (red). (b): The same initial streamline at $t = 1$, however the streamline kept the same height in areas of little movement on the left and right at time $t = 3$ (green) because the algorithm shifted the seed.

We now look at an intuitive display of a temporal artifact. In Figure 4.7 (a), we see how the increase in ridge height leads to movement of the whole line, which is visually irritating as it distracts from the actual feature undergoing the change. In fact, it suggests that instead of the center moving upward, the outer regions move downward, giving room for misconception of the field's behavior. What we want instead, is as little movement as possible, as can be seen in the transition shown in Figure 4.7 (b). Here, large portions remain stationary, with only segments close to the center rising upwards. An important factor for the generation of such images is the seed choice. We have chosen the same seed in both cases in order to compare the placement with and without our algorithm optimizing the seed position to respect what we define in the next chapter as *time coherence*.

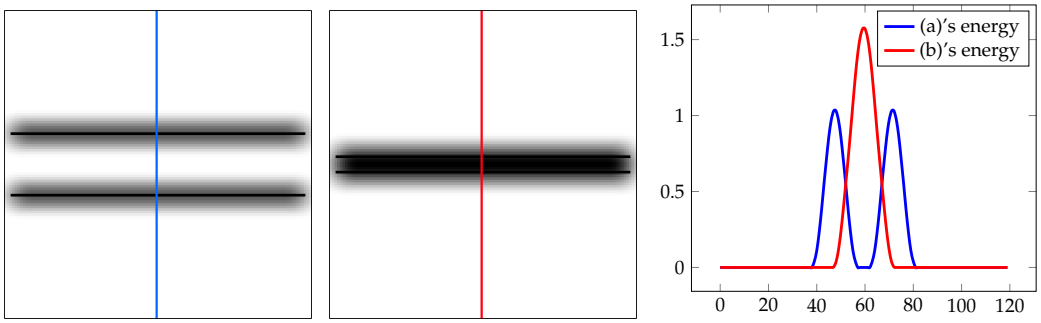


Figure 4.8: How the proximity of lines changes the brightness of their respective footprints in a 120x120px image. (a): Two lines approx. three times the blur radius apart. (b): Two lines only 3/4 the blur radius apart. (c): The brightness of the more distant lines (fig. a, blue) and the closer ones (fig. b, red). On the x-axis we see the height of the pixels taken along the red and blue strips in (a) and (b) respectively, counted from the top.

4.4 Time Coherence - Definition

Based on the desired image overlap and concepts from computing the energy measure, we can infer a measure for time coherence. In fact, we can use a definition analogous to E already defined by Turk and Banks. To avoid confusion, we will from now on write the spacial components (former E and L) as E_s, L_s to better separate them from their temporal counterparts E_t and L_t. Instead of the comparison between a low-pass image of lines and a constant target brightness, the temporal energy E_t now depends on two sets of lines (I_0, I_1) and uses a different low-pass filter (L_t):

$$E_t(I, I') = \int_x \int_y [(L_t * I_0)(x, y) - (L_t * I_1)(x, y)]^2 \, dx \, dy$$

This measure would give us the squared difference between the brightnesses of two different sets of lines. Allowing for a new kernel gives us more freedom to change *how* we measure the temporal energy, as we do not want it to behave the same way as the spatial energy does. For example, the measure E_s of two neighboring streamlines is the strongest in the center of them, not at the actual line positions, which can be seen in fig. 4.8. This means that by using E_s instead of E_t, we would change the number or position of lines by drawing them into the center of two former lines, thereby intently worsening time coherence. Conversely, this would also prohibit streamline creation at darker spots, giving rise to holes in our

line image.

The optimum for time coherence would of course be two identical sets of streamlines, which effectively minimize E_t to zero.

4.5 Adding Time Coherence

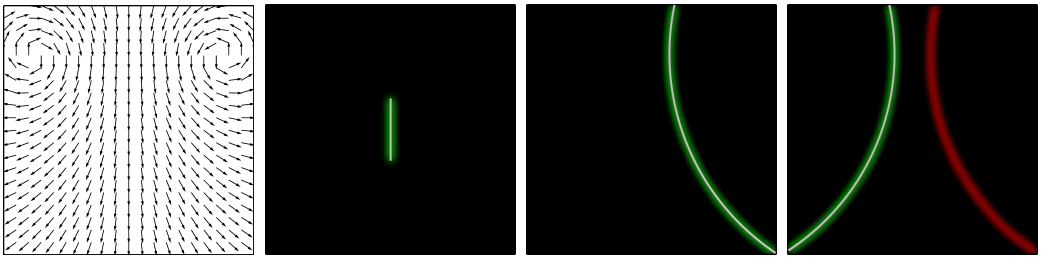
This section will be about how we translated the definition from section 4.4 into a functional component for our algorithm. We use the aforementioned energy measure to induce a process we refer to as *coaxing*, as we are applying continued pressure to lines to move them in the direction of their former footprints.
A second addition called *shattering* will be introduced as well, where lines are split into smaller segments (*fragments*) to be used as the starting layout in the next timestep. This can increase coherence as well by seeding lines at the same place, and works especially well in combination with coaxing.

4.5.1 Coaxing

Since most of the algorithm's optimization is centered around the comparison with an energy level before and after an action was taken, modifying the energy function provides a lot of leverage regarding how lines are placed. In order to make the algorithm favor previous line positions, we therefore rewrite the energy function as the linear interpolation between E_s and E_t. This gives us good control over how much time coherence we apply, as choosing too much will cause a degradation in image quality. Given the previous frame's low-pass image with the time kernel $(L_t * I_0)$ we use:

$$
\begin{aligned}
E(I_0, I_1) &= \alpha E_s(I) + (1-\alpha) E_t(I_0, I_1) \\
E_s(I_1) &= \int_x \int_y [(L_s * I_1)(x,y) - t]^2 \, dx \, dy \\
E_t(I_0, I_1) &= \int_x \int_y [(L_t * I_0)(x,y) - (L_t * I_1)(x,y)]^2 \, dx \, dy
\end{aligned}
$$

We have found values for α in the range $[0.4, 0.8]$ to be effective. Choosing a higher value causes very few lines to be drawn, and just yields some sections due to the inhibitory effect on the lengthen and join operations when leaving the previous line's footprint. This gets exacerbated by the gaps between the fragments being cemented in the new $L_t * I$, not allowing them to reconnect in subsequent frames.



(a) (b) (c) (d)

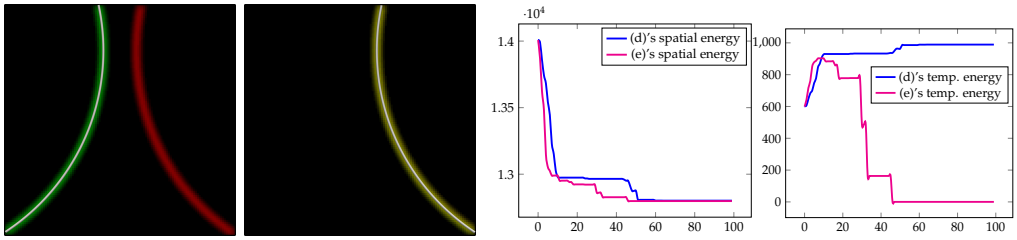
Figure 4.9: Lines may diverge from the same origin when using $\alpha = 0$. The field in this figure is *steady*. (a): We use the double gyre to show divergent behavior in (d), visualized with an arrow plot. (b): The initial seed and starting line length. (c): Random move and lengthen steps reach a local minimum energy by increasing streamline length, choosing a side at random. (d): A different result may occur with the same starting conditions.

Since we compare many images and footprints from this subsection onward, it is useful to include a distinction using different color channels. To save caption space, we use a consistent coloring to show streamline movement between time frames. **Streamlines** are drawn in white, and we only draw them for the current time step. **Footprints** from the current frame's streamlines are drawn in green, those from the last step in red. The higher the intensity of a pixel, the stronger the energy in that region. High time coherence therefore leads to most of the image being yellow, with few red or green areas. We only draw the footprints obtained using the filter L_t, as those from L_s can be easily inferred while L_t provides more visual clarity due to the reduced blur radius.

> **Note**: The algorithm may perform a combination of move and lengthen operations at once. Even with a constant field, it is possible for streamlines to move around or change their length slightly due to this inherent randomness.

We now take a closer look at the energy development for different line positions seen in fig. 4.9 (a-d). We start with a simple, steady field shown in (a), and a constant starting position for every execution at the center (b). After 100 optimization steps, the streamline has grown to the maximum length possible (c), thereby reaching a minimum in spatial energy. (d): The two likely outcomes of how the starting line develops under the specified starting conditions.

We can see that due to the randomness of the algorithm, there is a $\approx 50\%$ chance of ending up on either side of the center ridge (d).



(d) (e) (f) (g)

Figure 4.10: (d): A different result may occur with the same starting conditions, the lengths are invariant in this case. (e): An example for an outcome in favor of temporal energy. (f): Spacial energy vs optimization steps. (g): Temporal energy vs optimization steps.

On fig. 4.10 (f), we see a plot of the spatial energy vs the current optimization step. The maximum spatial energy equals the image resolution at 120x120. Initially, we can see a stark decline in energy for both curves. This is caused by to a fast lengthening process, with an average of about 2.5% image size per step, per side. The first plateau phase is reached after ≈ 15 steps, as the streamlines reach the edge of the domain and cannot lengthen further. Due to the random movements, the line ends drift along the upper and lower domain borders, slowly lengthening due to the curvature. A final stronger decline happens at 30 steps for (d), and at 50 steps for (e), as the curvature is strong enough do allow larger regions of spaces to be filled by lengthening once again. The delay between (d) and (e) is caused by the random nature of the algorithm, and is equal if both runs use the same seed. Our streamline reaches the minimum possible energy of about 12500, with a difference of 1300. Since the line length lies at roughly 140px and the filter is about 14px wide (note: L_t as shown is half the size of L_s), these numbers aren't surprising.

The temporal energy is shown on plot (g), where we can see a stark difference in the development of (d) and (e). Many features regarding local rate of change are recognizable in both plots, e.g. the plateau of (e) centered around the 40th step's mark. The final temporal difference lies at 1000 vs the spatial difference at 1300, again a result of the reduced filter radius.

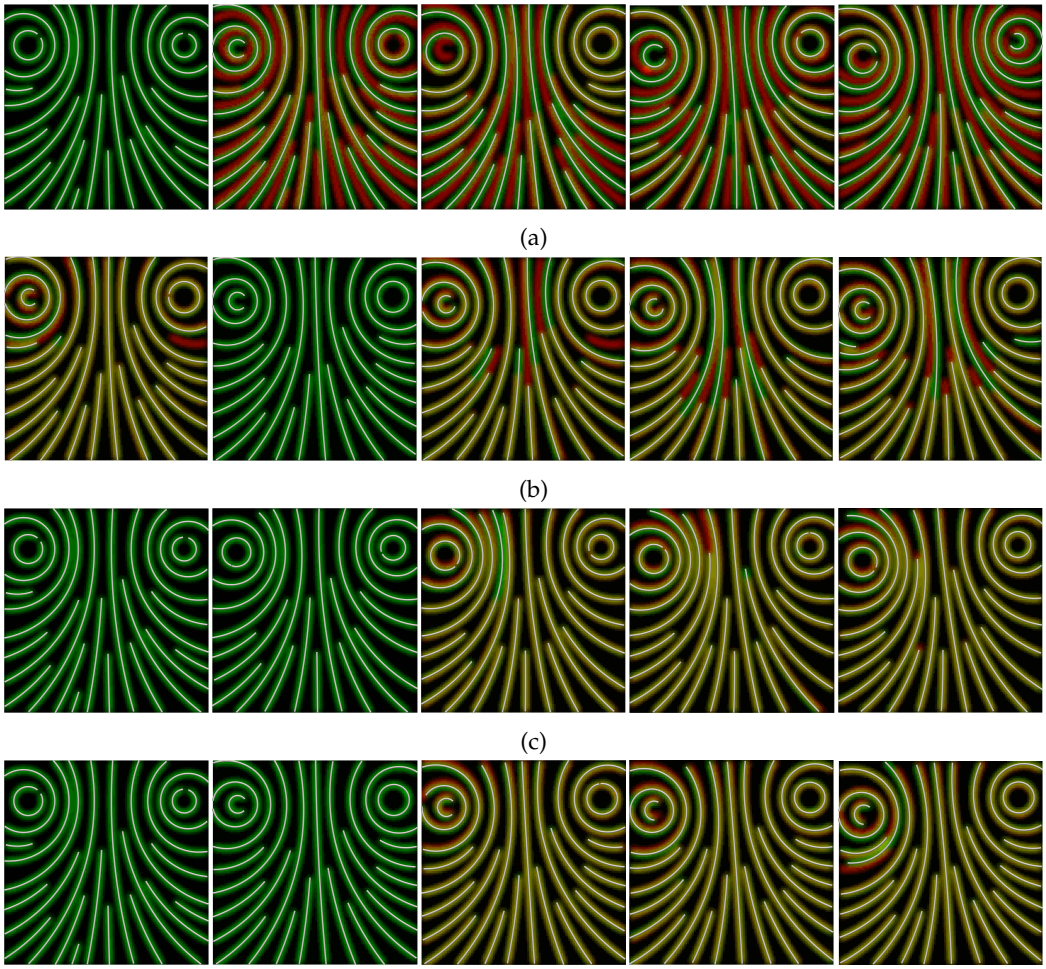
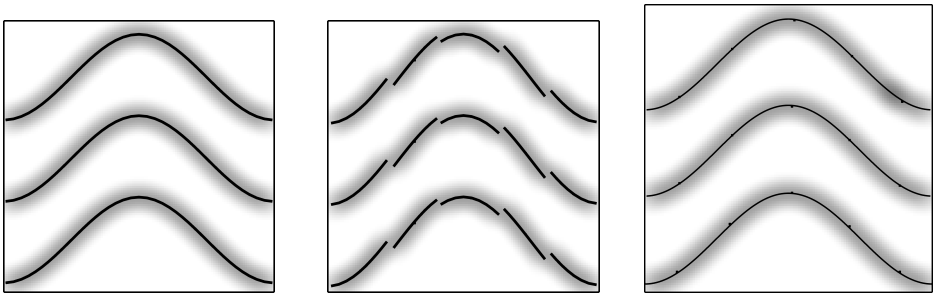


Figure 4.11: Comparison of different α values between the first five steps for an unsteady version of the double gyro field from fig. 4.9. Each row represents five time steps and for each step, the left vortex moves down by 3% of the image height. (a): With $\alpha = 0$ most lines have low coherence and field movement is hard to notice between frames. (b): $\alpha = 1/3$, TBD (c): $\alpha = 2/3$, Good coherence in most regions, some strong changes remain (d): $\alpha = 1$, Even better coherence with only slight changes, even in areas of field movement.

4.5.2 Parameter Study for α

Since α drives how similar we want our images to be vs. how much we care about spatial placement and uniformity, we briefly look at some examples for extreme cases and more moderate choices, examining some notable differences. We start with a double vortex field, where the left vortex undergoes a downward motion. The distance travelled between each time step is exactly 3% of the image height.

In figures (a - d), we set $\alpha = 0$, and the streamlines move sporadically with time coherence only being achieved by chance and in areas almost completely unaffected by the change. For (e - h), $\alpha = 1/3$. We can immediately see some improvements, as the center is becoming more stationary. The footprints overall gain a more compact appearance, as green and yellow parts aren't as far apart as they were previously. The optimal range is reached somewhere around $\alpha = 2/3$ (i - l), most of the footprints are now yellow with only slight divergence in areas not affected by the y-shift of the vortex. At the same time, we preserve an even appearance of line spacing without particularly bunched or sparse areas. For high values of α (m-p), we can tell that the image has become a lot worse compared to before. We see large empty spaces, where lines cannot be drawn anymore due to the strong energy punishment inflicted by E_t having such high weight. These spaces can only increase in size as lines cannot grow past the previous footprint. If the field changes further, the line length decreases in order to still fit the footprint left behind, causing less overall line presence every step. We conclude that a good starting value for α would be in the $[0.5, 0.75]$ range. This avoids the sharp decline in spatial quality, while still retaining good control over the placement to make the streamlines remain time coherent.

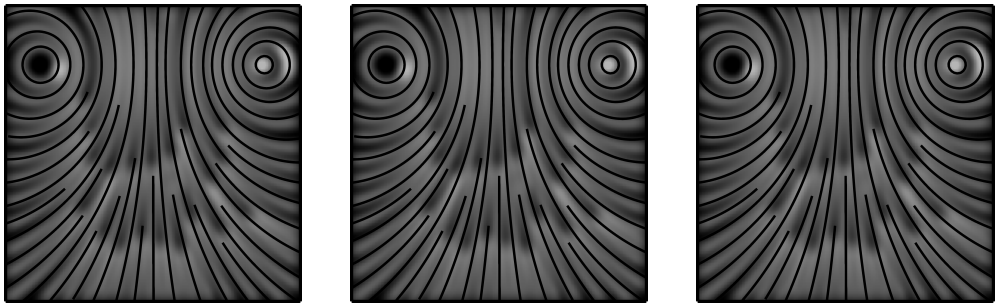


(a) (b) (c)

Figure 4.12: (a): Three streamlines after being optimized. (b): To make the individual shards visible, we change the field's amplitude slightly. The shards' seeds are the center of the streamlines in (b), and all lie on one of the streamlines in (a). (c): The lines quickly rejoin when re-drawn in the same field used in (a). We used slightly thinner lines to better show the seeds (thin black dots on the streamlines), and to make it more visible that the lines do not simply overlap but actually rejoin.

4.5.3 Shattering

At the end of a time step's optimization phase, we break every streamline apart into smaller streamlets we refer to as *shards*. We start by dividing the parent line length-wise into sections which equal the length of the starting lines. The shards are assigned a seed in the middle (lengthwise) of these intervals, and their length equals the line start length. If the parent line has some length remaining because it was not perfectly divisible by the start length, the last shard's length will be shorter. This leaves each former streamline with the appearance of being dashed with each fragment having its own seed, and can be seen in fig. 4.12 (b). The shards then act as the initial seeding strategy for the subsequent timeframe; the regular grid is only used for the first frame. This way, we obtain many seeds that, if the field does not change too much, will merge back into the line they came from, as can be seen in fig. 4.12 (a) and (c), saving iterations that would be needed for new seeding and lengthening in these regions. If the field *does* change, some segments will still reconnect and therefore keep their temporal coherence, whereas areas of strong fluctuation will connect to different seeds. This results in changes being limited to parts where a streamline change is necessary, providing extra lines in these areas while not affecting streamline trajectory too much on a global level.



(a) (b) (c)

Figure 4.13: (a): Two time steps without shattering. (b): Two time steps with shattering. (c): Total energy vs iteration step for (a) and (b).

4.5.4 Combining Shattering and Coaxing

Combining shattering and coaxing, we obtain a more reliable way of generating streamlines according to the footprint left behind by the last frame. The seeds created during the shatter process all lie inside the footprint left behind by the previous streamline path. Due to the coaxing function of the modified energy measure, it is unlikely that they will leave this valley solely due to the random movements of relaxation. A change in the field is necessary in order to overcome the weight of the time coherence, making the line move or grow outside the previous footprint. Due to the seeds being held in place in this way, it is very likely for them to rejoin to form the same lane they originated from. If the field changes drastically in this region, the seeds can not fully connect to each other anymore, and will instead gravitate to a different footprint, forming long patches of coherent lines whenever possible while still allowing relaxation to ensure good spatial distribution.
In fig. 4.13 (c), we see that joining shards to form the streamlines is significantly faster than generating them anew.

5 Implementation

This chapter briefly mentions the used libraries, and focuses on the initial implementation of - and iterative additions to - the proposed algorithm.

5.1 Libraries

The algorithm is implemented in Python3.10, and heavily relies on three libraries which are not part of the Python3.10 standard library:

- ParaView v.5.12.0: A Scientific visualization software, combining data science and interactive visualization while providing custom algorithm support via the VtkPythonAlgorithm base class.
- VTK v.9.3.20231030 : The library used to manage anything related with the data to be visualized in ParaView.
- NumPy v.1.23.4: Widespread data manipulation/scientific computing library, which is used to edit the data encapsulated by VTK's objects.

5.2 Time Coherent Algorithm Implementation

Using the "Visualization Tool Kit (VTK)" and the "Parallel Viewer (ParaView)" comes with some caveats, but also several benefits due to the broad spectrum of available algorithms. The most important components used are briefly listed in the following chapter, the final picture of the algorithm design is delivered afterwards in subsection 5.2.2, and we conclude this chapter with a complexity analysis in subsection 5.2.3.

5.2.1 Vital External Software Components

VTK Pipeline While an extremely involved topic with dozens of hours to be spent on reading, this can be summarized as follows: The VTK pipeline consists of

three types of objects: Sources, filters, and sinks. Sources create data, filters modify it, and sinks display it. Each object has input and output ports, how many of which it has decides its membership of one of the above groups. A simple workflow to visualize a vector field would be: Vector Field Source → Line Geometry Generation → Line Rendering.

vtkPythonAlgorithm This class is intended as a base class for writing custom algorithms in either of the three categories. We use this twice: Once for a group of vector field sources to test our implementation with, and for the algorithm itself. In the former case, we use it as a source, in the latter as a filter. The relevant method in this class is called "RequestData", which is passed an information object. We can modify this object in order to pass data forward (and backward, though we do not need this) through the pipeline.

vtkImageData Objects of type vtkImageData hold a grid defined by 2 3-vectors: The extents (number of points in each direction), and the spacing (how far points are apart in X/Y/Z direction). Every point on this rectilinear grid can have scalars or other objects assigned, like a 3D vector containing velocity. In our case, we use it exactly in this way: Every point is assigned a velocity, which are then interpolated as needed.

vtkStreamTracer The vtkStreamTracer class is a filter with two inputs: We provide it with a vector field (vtkImageData) object and a point, which it then integrates through the field. The relevant output for us is the list of points making up the streamline that it returns.

5.2.2 Algorithm Design

Since we need two filters (L_s, L_t), and want them to act on different time steps, we have decided to implement the filtering and drawing subsystem as follows:

FilterTarget s are effectively images, something with 2D pixel data that can e.g. contain the brightness of the image that guides our algorithm.

Painter s act as a modifying actor for FilterTargets. Painters use a configuration (line brightness, blur size, etc.) and draw polylines to the FilterTargets accordingly.

Filter objects contain a list of lines that make up the vector field image, and orchestrate the assigned Painters and their respective configs. They also pro-

vide methods for adding/removing/modifying lines, using a given energy function to determine their success.

FilterStack : This class is best used (though not enforced) as a singleton; it manages two filter objects, one for each time step. It also provides the energy methods as lambdas to the new filter added every time step.

This change compared to the original was necessary, because we now want to manage multiple filters from different time frames at the same time. It is even possible to make the filter change the further back in time it was created, e.g. to not only use time coherence w.r.t the last frame, or to allow onion skinning of older frames' low pass images. The entry point for this algorithm is, as with any VTKPythonAlgorithm, the "RequestData" (we leave out the other Request... calls for brevity) method. We are provided the vector field via the VTKImageData object as input, and start to set up our low-pass filter stack. By providing a filter with a config (the standard config uses similar values as Turk and Banks' implementation), we set up the E_s part. If we are not interested in time coherence, this is all that is necessary for a line to be drawn filter-wise. Otherwise, we simply add another config specialized to work well with E_t, so our filter now has two configs, painters, and targets: One for E_s, one for E_t. Drawing the lines themselves is done using NumPy's vectorization, since we can use the NumPy-compatible vtkDataSetAdapter (DSA). We use this to quickly obtain and transform the coordinates obtained from the vktStreamTarcer. The drawing process is handled entirely by the Painter objects: For a line L containing n seeds, we calculate the bounding boxes of $n-1$ segments, padded by the filter radius. Each pixel inside this rectangle has a number of vectorized calculations performed in order to determine its brightness. The brightness is evaluated using a precomputed grayscale table which we interpolate via SciPy's RegularGridInterpolator, as this also supports vectorized access. Once each segment's pixels are computed, we simply add them to the global line image.

Having finished the drawing process, we now look at the energy measure. The filter stack hands over a lambda to the respective filter, with some arguments bound to their respective FilterTarget. This way, we can dynamically change how the filter calculates values based on the gray scale values form the bound targets. If, e.g., we do not have an old filter yet, we cannot use the *coaxing* strategy. Therefore, we simply leave the argument bound to "None" when passing it to the first filter.

5.2.3 Complexity Analysis

The whole algorithm depends on a single parameter called the *separation distance*, which we write as d. $5/6 \cdot 2/d$ is used to calculate the size of the low pass filter, with typical ranges of d being $[0.01, 0.04]$, referring to 1-10% of the screen width. For simplicity, we assume the screen shape to be square.

This probably terrible. How to do it properly?

We can therefore write the total number of pixels in the low pass filter as $25/36\dot{4}/d^2 \in O(d^{-2})$.

For a single time step, we draw L lines. Each line contains S segments, or $S + 1$ points. Let B be the typical size of a bounding box in pixels, then we have to compute about $L \cdot S \cdot B^2$ pixels. Note that "pixels" in the low-pass filter have nothing to do with those used to represent the final lines. Due to the delayed rasterization, it is sufficient to use a low pass image of size 32x32px to compute an image about 12x12 line spacings across.

Assuming an image size of one, and the typical line length to be one with a step size of 0.005, we get about $S = 200$ segments for a single line.

With a filter of size 32x32 and a filter radius of 6px (giving us a spacing of about 12 lines per side), so let $B = \lceil 0.005 * 32 \rceil + 2 \cdot 6 = 13$. This means we end up with $200 * 13^2 = 33,800$ pixels to be calculated for a single line. Every move operation causes the entire line to be re-evaluated, and the most relevant operations are moves, as they are what relaxes the image and makes lines grow past each other. A lengthen operation can increase the length by a maximum of 4% of the image width, averaging at about 2%. Therefore, at least 25 redraws are needed, with earlier redraws being much cheaper, since the line is shorter.

Generating simple images like the ones used in this thesis can take up to 5 minutes, twice as long if we use multiple painters like when enabling coaxing. The strongest factor impacting runtime is the number of integration steps, as the application of the individual footprints is still implemented "loopy", without vectorization.

6 Results

In this chapter, we start by running the full algorithm on various datasets while discussing the different outcomes depending on the coherence strategy, kernels, and coherence weights. Then, we show some performance metrics and the hardware configuration. We finish this chapter with problems and limitations regarding the functionality and choice of parameters of our algorithm.

6.1 Visualizing different fields

The comparisons for the various fields are made between Turk and Banks' algorithm without coherence, a slightly modified approach simply reusing the line's seeds for successive frames, and our modified version using coaxing and shattering enabled. We start this section with three simple fields.

$$(a) : u(x,y) = (1,0) \qquad (b) : u(x,y,t) = (1,t) \qquad (c) : u(x,y,t) = (x-t, y-t)$$

Next, we use the algebraically defined fields from previous parts of this thesis.

Finally, a dataset of size 60 x 30 is used.

6.2 Performance

The algorithm was run on a system with the following relevant hardware. CPU: AMD Ryzen 5 5600X 3.70 GHz, RAM: 16 GB DDR4 ($\approx$ 400 MB used by ParaView). Most of the images with a black background color were generated in about 3-4 minutes each. Image ??? from the hot room dataset took ??? h m s. For comparison, the initial greedy algorithm took about 10 seconds. The size of the low pass image is of little no importance for speed, we tested filter radii between 2 and 32, both of which took about the same time to complete. *Some better numbers/tables*

6.3 Issues and Limitations

Strong changes lead to drastic degradation of image quality when using high coherence weights. This can be counteracted by either interpolating the field movement, or reducing α to allow the lines to move more freely. Memory usage scales with the low pass size and line count, as every line saves its contributions to the energy as an array of the same size. When using large amounts of lines with a high low-pass resolution, memory usage can increase drastically. We found that the low-pass resolution has very little effect on the resulting line placement, we used the 120x120 resolution solely for a smoother appearance The coherence weight parameter is configurable prior to a run, but for different datasets (more specifically: different time strides) it is often useful to vary this parameter. This could perhaps be added via an adaptive method, generating different images from one previous image using different weights, and selecting the one with the least total energy. Of course, this would increase the computational cost drastically, and we have therefore decided not to add this feature for now.

7 Conclusion

Bibliography

[96] "Image-Guided Streamline Placement". In: 1996 (cit. on p. 3).

[Han+19] K. Hanser, S. Meggendorfer, P. Hügel, F. Fallenbüchel, H. M. Fahad, and F. Sadlo. "Energy-Based Visualization of 2D Flow Fields". In: *Proceedings of International Conference on Information Visualization Theory and Applications (IVAPP)*. 2019, pp. 250–258 (cit. on p. 4).

[HS19] L. Hofmann and F. Sadlo. "The Dependent Vectors Operator". *Computer Graphics Forum* 38:3, 2019, pp. 261–272 (cit. on p. 4).

[Jun+17] P. Jung, P. Hausner, L. Pilz, J. Stern, C. Euler, M. Riemer, and F. Sadlo. "Tumble-Vortex Core Line Extraction". In: *Proceedings of SIBGRAPI WVIS*. 2017 (cit. on p. 4).

[SJMS19] A. Sagristà, S. Jordan, T. Müller, and F. Sadlo. "Gaia Sky: Navigating the Gaia Catalog". *IEEE Transactions on Visualization and Computer Graphics* 25:1, 2019, pp. 1070–1079 (cit. on p. 4).

[SRS18] K. Sdeo, B. Rieck, and F. Sadlo. "Visualization of Fullerene Fragmentation". In: *Short Paper Proceedings of IEEE Pacific Visualization Symposium (PacificVIS)*. 2018, pp. 111–115 (cit. on p. 4).

[ZRLS19] B. Zheng, B. Rieck, H. Leitte, and F. Sadlo. "Visualization of Equivalence in 2D Bivariate Fields". *Computer Graphics Forum* 38:3, 2019, pp. 311–323 (cit. on p. 4).