Time-Coherent Streamline Placement

Bachelor's Thesis

Alexander Baucke

Supervisor
Prof. Dr. Filip Sadlo

Heidelberg, Germany, February 6, 2025

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

Common elements in visualizing steady vector fields are integral lines that show flow dynamics, called streamlines. Many possible criteria can define the quality of a set of streamlines when visualizing a vector field. Frequently 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 the most relevant criterion for visualizing *unsteady* fields using e.g., animations, is largely unaccounted for. Currently, most algorithms can only animate a vector field frame-by-frame without time-awareness. The result is many visually pleasing images lacking coherence between each other, as said algorithms place streamlines merely to fulfill the frame-limited optimum. We overcome this problem by introducing an algorithm capable of time-coherent streamline placement, effectively minimizing streamline movement between frames. We implement an image-guided approach based on the work by Greg Turk and David Banks [TB96] to generate initial streamline placements according to the above criteria. To add time coherence, several key changes to different components are made. The first change revolves around the energy measure, a function defining the uniformity of streamline placement by comparing it with a target brightness. Another change is the kernel Turk and Banks use to blur the image, which we adapt to include time coherence, as well as introducing 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

Ziel dieser Arbeit ist das Erreichen von Zeitkohärenz bei Stromlinien in zeitabhängigen Vektorfeldern, um deren Darstellung durch z.B. Animationen zu verbessern. Wir beginnen mit einer Definition für Zeitkohärenz, die im Wesentlichen auf die Bewegung einer Stromlinie von einem Zeitschritt zum nächsten gestützt ist. Wenn diese Bewegung groß ausfällt bzw. sich die Stromlinie viel bewegt hat, bezeichnen wir dies als "schlechte Zeitkohärenz". Wir zeigen, warum zeitkohärentes Verhalten wichtig ist für z.B. Animationen, und entwickeln eine prototypische Implementierung, deren Parametrisierung wir dann anhand einiger Beispiele bestimmen. Unser Algorithmus basiert auf einer durch Bild-geleiteten Implementierung von Greg Turk und David Banks, den wir durch einige Änderungen an den zentralen Komponenten zeitkohärent gemacht haben. Im Anschluss werden einige Datensätze damit ausgewertet und mit anderen einfachen Algorithmen verglichen; außerdem beschreiben wir einige Limitierungen sowie Ideen für Verbesserungen. Zum Schluss wird eine kurze Komplexitätsanalyse durchgeführt und die Performance untersucht.

Contents

1 Introduction

Vector field visualization is a ubiquitous component encountered in many processes used by science and industry alike, such as aerospace engineering, geology, fluid dynamics, material science, or the biomedical sector. Frequent use cases for vector fields include weather systems, rotor design, analysis of marker movement inside a cell, or predicting the magnetic field of a star. Displaying such fields in a human-focused way is invaluable when analyzing the underlying systems' dynamics, allowing for faster and more accurate data analysis and subsequent decision-making. Contemporary research has a primary focus on continuous and steady vector fields. Streamlines are usually optimized w.r.t. a single frame, allowing for e.g., a spatially uniform and therefore visually pleasing streamline placement.

A problem arises when small changes over time are introduced, i.e., an unsteady vector field is used instead of a steady one. While current algorithms still produce good images frame by frame, it is unlikely for these algorithms to prefer streamlines similar to the ones in the previous frame. Due to this, streamlines move about and shift between frames a lot more than the time-induced changes to the field would warrant, with the sporadic movement making it very difficult to perceive the flow change of the field. Creating an animation of an unsteady vector field's behavior thus requires frequent human intervention, which makes good animations very tedious, time-consuming, and ultimately expensive. Most vector fields represent dynamic systems, which makes many areas susceptible to this limitation.

We believe that by introducing a new criterion that describes how much streamlines move from one frame to another, we can quantify the visual fidelity of streamline placements generated for different frames. In this thesis, we will refer to this criterion as *time coherence*, indicating that the objective is to constrain streamline movement from one frame to the next to ensure that the human eye perceives the "movement" of a streamline as one fluid motion through several frames.

This paper will focus on three core aspects revolving around time coherence:

- A motivation for—and definition of—time coherence by deriving a measure from simple cases lacking time coherence.
- The implementation and underlying ideas of an image-based algorithm generating evenly-spaced long streamlines, which we use as the basis for our implementation.
- Adaptations of the base algorithm to allow a controlled bias between similarity to a previous time step, and optimality w.r.t. the spatial layout for the current step. We show how time coherence can be added in a non-intrusive, compatible way to combine it with spatial coherence.

We have chosen an image-based approach (as opposed to a feature-guided one, see Section 2 for a disambiguation) for our work because the movement of streamlines between time frames is an appearance-focused problem, and therefore better suited by 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: In Section 2, we briefly note and classify some groups of algorithms, and describe how our algorithm fits into these surroundings. The fundamentals, mostly in the areas of vector field visualization and image processing, are provided in Section 3. Section 4 starts with a brief segment about a discontinued heuristic algorithm, followed by a close examination of the image-guided algorithm we then decided to use as a foundation. Time coherence is motivated and explained in greater detail, with initial ideas regarding its implementation laid out, then executed in Section 5. Section 6 contains our findings and ideas for future work, with Section 7 concluding this thesis.

2 Related Work

Most of the work published in the field of streamline placement algorithms can be divided into three categories, each having different focuses, strengths, and drawbacks. Each category will be briefly addressed in the following sections.

2.1 Image-Guided Algorithms

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

One of the first and most prominent examples in this category was created by Greg Turk and David Banks [TB96] in 1996, 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 streamline spacing. Adding/moving/removing/resizing streamlines is done randomly, at every step the energy function is used to determine whether a proposed change gets accepted or discarded. The algorithm is finished when the energy function reaches a lower energy bound or the user stops it. This idea was extended by [SLCZ09] to allow streamline generation on 3D surfaces. Another image-guided approach, which is also view-dependent for 3D fields, was created by [LS07].

2.2 Feature-Guided Algorithms

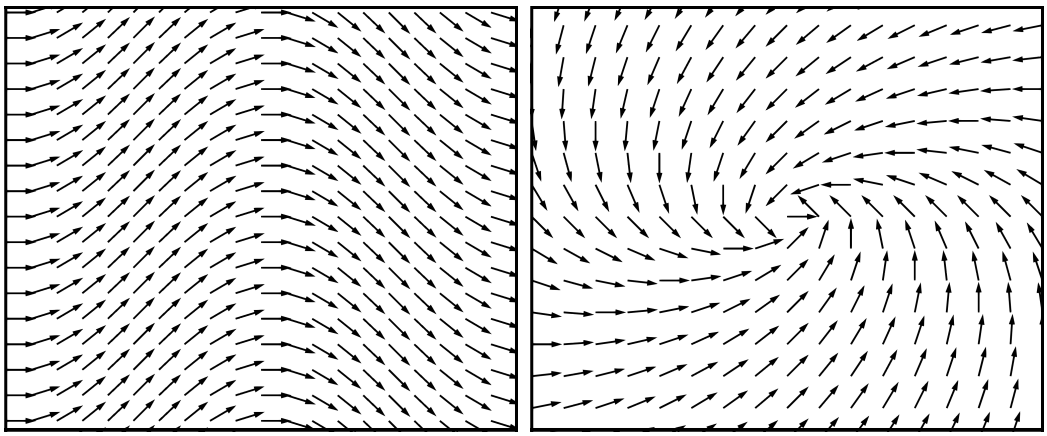
Feature-guided algorithms examine the underlying field structure when placing seeds. They search for critical points or patterns in the field and then seed around them, capturing them in much higher detail. The resulting images inherently represent the critical points much better, sometimes at the cost of visual appeal com-

pared to image-guided algorithms. [VKP00] is one of the first algorithms in this category, extracting field topology and then applying a fixed pattern of seeds for the different structures to capture them as accurately as possible. [MAD05] developed a farthest-point seeding strategy using Delaunay triangulation for fast seed positioning. [WLZM09] picks up on this idea, improving it to generate better images still adhering to the underlying topology.

2.3 Greedy Algorithms

Greedy algorithms do not use a global guiding principle for streamline placement. We used an approach similar to [JL97] as a prototype, placing seed candidates along a generated line and attempting to grow streamlines from them until the space is filled. [LMG06] later improved this algorithm's speed by an order of magnitude. [MTHG03] extended this principle into 3D, adding several features to make it more suitable for user interaction and 3D rendering.

3 Fundamentals



(a) (b)

Figure 3.1: Two vector fields visualized using a vector plot. The field in (a) is defined by $u(x, y) = (1, \sin(x))^T$, the field used for (b) is $u(x, y) = (-0.5x - y, 0.5x - y)^T$.

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 *vector field visualization* in Section 3.1. Our algorithm uses an image-guided base, so we also need 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 we use them to develop a prototype in Section 4.1.2.

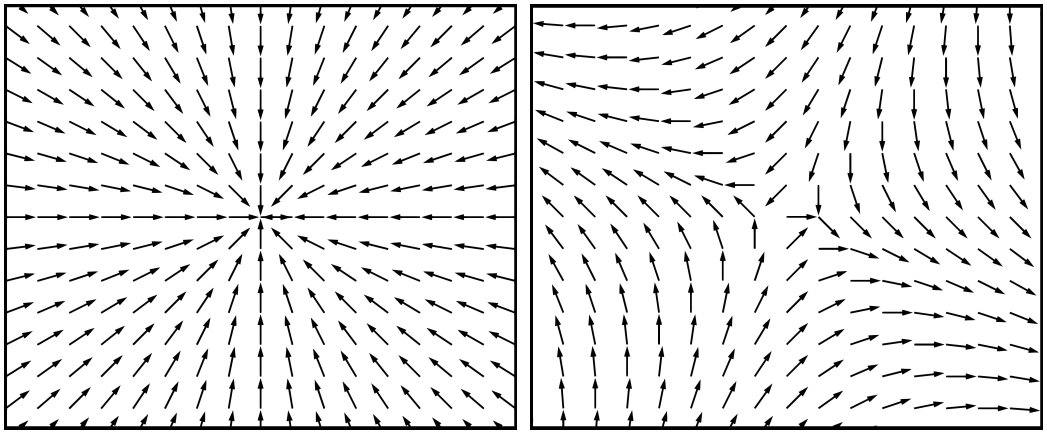
3.1 Vector Field Visualization

This field revolves around how to make spatial scalar data more accessible to human understanding. We focus on the visualization of two-dimensional fields, two examples can be seen in Figure 3.1.

3.1.1 Vector Fields

A vector field represents how vectorized elements act over a spatial domain. Concerning vector fields representing flow, for every point in a domain we can obtain the flow velocity 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 3.1. If we want our velocity to not only depend on spatial input but also on another scalar like a time component t, 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.



(a) (b)

Figure 3.2: (a) Vector plot of a sink defined by $u(x,y) = (-x,-y)^T$. (b) Vector plot of a saddle defined by $u(x,y) = (x-y,-x-y)^T$.

3.1.2 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. Hence, we refer to such points as *sources*.

Sink Similarly, $u(x,y) = (-x,-y)$ would give us a *sink* at $(0,0)$, equivalent to destroying liquid flowing inward. This is shown in Figure 3.2.

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*.

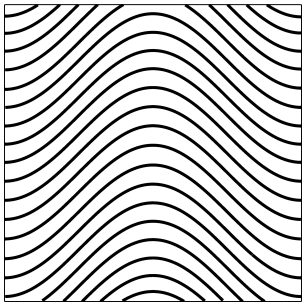


Figure 3.3: A set of streamlines generated for the field in Figure 3.1(a).

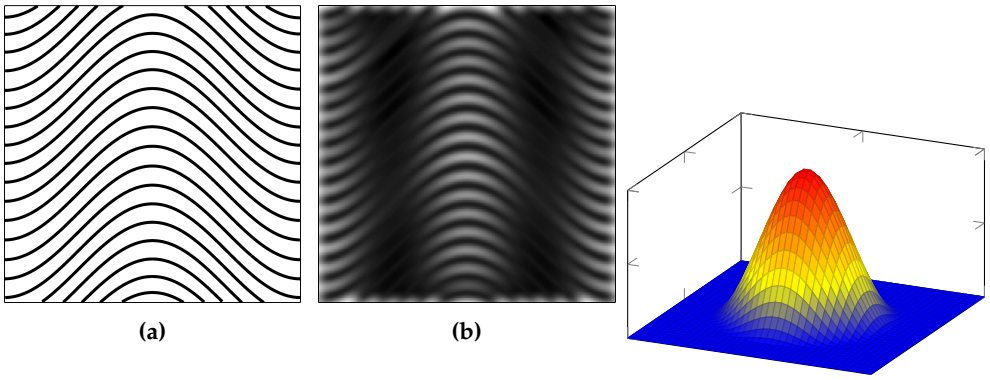
3.1.3 Streamlines

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.

3.1.4 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. Figure 3.3 shows strong spatial coherence.



(a) (b) (c)

Figure 3.4: A set of streamlines (a) generated for the field in Figure 3.1(a). (b) Low-pass version of (a) after convolving it with a kernel like the one shown in (c).

3.2 Image Processing

In this thesis, we need to convolve an image with a cubic Hermitian kernel in order to achieve a consistent blur with a specified falloff distance. We therefore use some concepts from the field of image processing we briefly describe here.

3.2.1 Convolution

Convolution is a process often found in image- or signal processing. For this thesis we focus on the former: A kernel (Figure 3.4(c)) is applied to every pixel in an image, affecting it and other surrounding pixels by adding or subtracting its value at that position.

3.2.2 Blurring

Blurring refers to a special type of convolution, making edges in an image less sharp. Note the difference between black and white contrast for Figure 3.4(a) and (b).

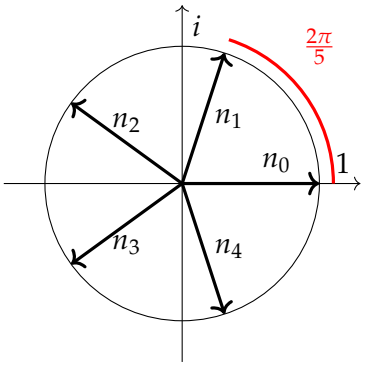


Figure 3.5: The 5th roots of unity $n_0 ... n_4$ partition the unit circle equally

3.3 Roots of Unity

To extend our initial prototype from 2D to 3D, we need to be able to place uniformly spaced points on a normal plane around a line segment. To do this easily, we make use of the field of linear algebra and use the 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 seen in Figure 3.5, 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

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

4 Method

In this chapter, we describe the motivation and concepts used for developing an algorithm supporting time coherence. We start with an implementation we ultimately deemed infeasible in Section 4.1, replacing it with the image-guided version described in Section 4.2. Section 4.3 contains the motivation, i.e. why time coherence is beneficial, and provide an example for undesirable effects that can appear without it. The definition is provided in Section 4.4, which we then adapt to our use case and implement in Section 4.5.

4.1 Initial Greedy Algorithm

> **Motivation:** Since time coherence only exists as a connection between one set of streamlines to another, we start by creating an algorithm capable of generating such streamlines. We 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, we developed the algorithm for the 2D case; Section 4.1.2 contains the extension to 3D.

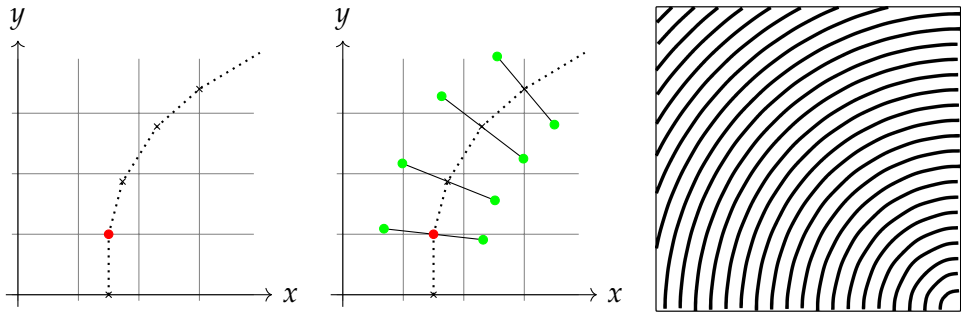
Our first approach uses two operations, streamline traversal and seed filtering, which it executes round-robin until there is no more room for new streamlines. The result is a space-filling set of streamlines with an even spacing in 2D and 3D vector fields.

We use two essential parameters to control image generation. The first one, d_s, is the neighbor search distance, which controls the distance from the current streamline where new streamlines start, allowing us to fill the whole space. In order to add longer streamlines 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 $\geq 0.75\, d_s$, many 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) Streamlines with equal distance generated by our algorithm in a field defined by $u(x, y) = (y, -x)^T$.

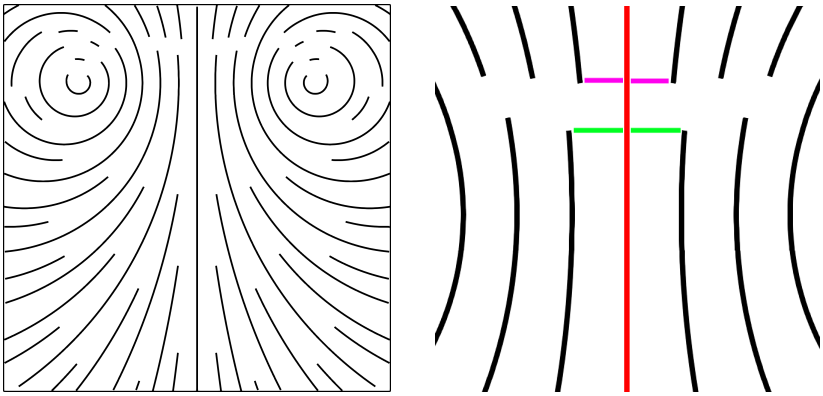
Streamline Traversal

First, we choose a seed point from a list of candidates (initially an arbitrary point from the dataset) and remove it from the list. We then integrate forward and backward (Figure 4.1 (a)) to obtain the other points on the streamline, until a given number of steps is reached, we leave the vector field's domain, or get too close to another streamline. If the total length of the streamline is too short, 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 (Figure 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. We use two filtering conditions to remove seeds that cannot produce a good streamline. The first condition arises from roughly half the seeds generated by a streamline L_1 during the traversal process lying close to—if not precisely on—the preceding streamline, L_0, because the seeds are placed a distance d_s away, and the streamlines are mostly that same distance apart. 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: seeds generated during the traversal process and those present in the entire field.



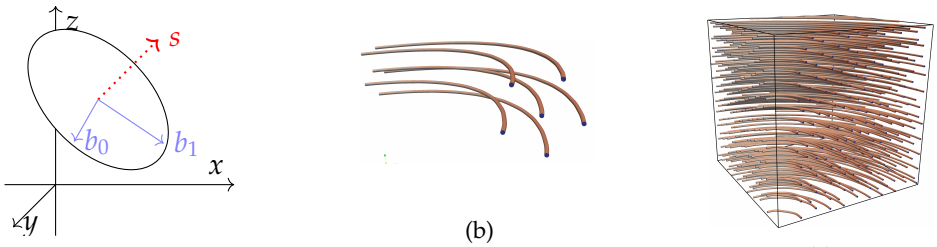
(a) (b)

Figure 4.2: (a) Streamlines created in a double gyro dataset. (b) Top center zoom of (a). New streamlines created from the initial 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 pairs of streamlines (black). The lower streamlines continue at a distance of d_S (green).

Line Cutoff and Optimization

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 within range, 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 eight neighboring cells as a key to points in the area. If we find a seed closer than d_c in those cells, we remove it, allowing for an easy way to end the integration when getting too close to another streamline (see Figure 4.2) as well: We need only check 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)

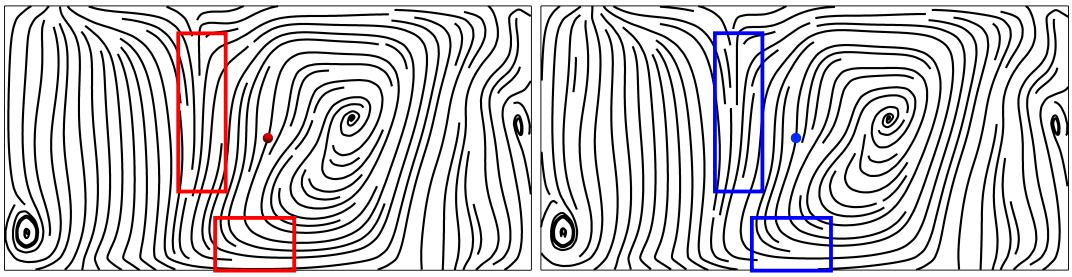
Figure 4.3: (a) Normal plane of a streamline segment s with two orthonormal basis vectors $\mathbf{b}_0, \mathbf{b}_1$. (b) Clear view of the center streamline with five neighbor streamlines evenly distributed at distance d_S. (c) Filled cube containing 254 streamlines. Notice the resemblance to Figure 4.1 (c).

The most important part of generating streamlines in 3D is obtaining the seed locations. In three dimensions, a vector has infinitely many normals, all lying on its normal plane. 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 $\mathbf{b}_0, \mathbf{b}_1$ (see Figure 4.3 (a)). Having found $\mathbf{b}_0$ and $\mathbf{b}_1$, we can use the roots of unity as described in Section 3.3 to obtain k evenly spaced points on the complex unit circle n_j, $j = 1, ..., k$. Since the magnitude of n_j is always one and $\mathbf{b}_0, \mathbf{b}_1$ are orthogonal, we do a simple basis transformation into the 3D frame of reference. We obtain the j-th 3D-vector $\mathbf{v}_j$ from the j-th root using our basis vectors $\mathbf{b}_0$ and $\mathbf{b}_1$:

$$\mathbf{v}_j = \mathrm{Re}(n_j) \cdot \mathbf{b}_0 + \mathrm{Im}(n_j) \cdot \mathbf{b}_1$$

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

Shortcomings

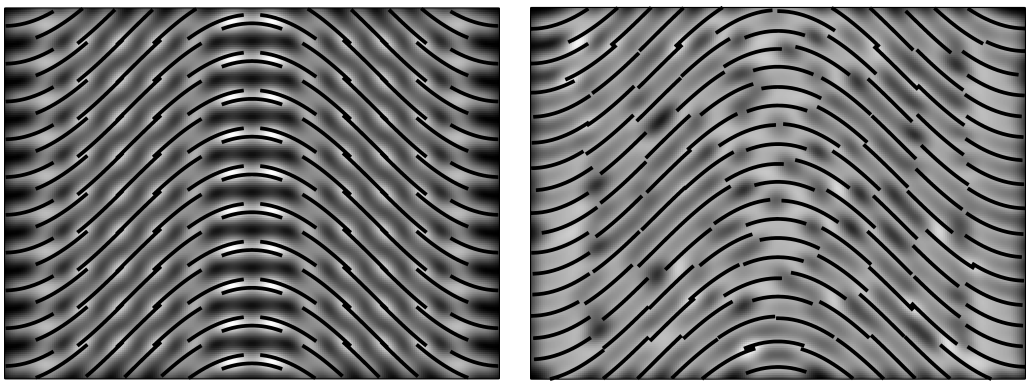


(a) (b)

Figure 4.4: Two different streamline placements for the same vector field. (a) hast its initial seed (red dot) directly at the center, (b)'s seed (blue dot) is moved by 0.01 % of image width, causing a completely different line image. The images look similar at first glance due to the density and rough shape being perceived the fastest by the human eye, but when following the individual lines the differences are easily noticeable. The stronger changes are indicated by the red and blue boxes, though differences visibly persist throughout most of the image.

While this algorithm can quickly fill a space with streamlines according to the criteria mentioned above, the main problem is the inability to react to small changes to streamlines, which is primarily due to the strong hierarchical nature: Since every streamline after the first comes from its predecessor, a change at the "root" has drastic consequences for the succeeding streamlines (see Figure 4.4). In the context of time coherence, the algorithm thus becomes unsustainable, as an unsteady field is very likely to change at least slightly over 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 in order to compensate the small, field-induced streamline movements. With this approach, changing the streamline's position after the generation of its neighbors causes a re-evaluation of every streamline it is a predecessor of and leading to massive computational overhead. Due to this problem, we do not see an effective way to implement time coherence with this approach, and supersede it in favor of an image-guided one.

4.2 The Image-Guided Algorithm by Turk and Banks



(a) (b)

Figure 4.5: (a) Short streamlines placed with seeds on a regular grid. (b) Same streamlines after optimization through seed shifting. Notice the more even grayscale image in the center and on the sides. Both images contain 130 streamlines, 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 [TB96]. The algorithm forms a more suitable basis for time coherence due to how it reacts to changes to single streamlines. It retains a good amount of control regarding the streamline placement and length, while still allowing the streamlines 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 streamlines themselves, the low-pass filter, and the energy measure. The filter creates blurred images of the streamlines, where pixels close to the streamline are brighter, and empty spaces darker (Figure 4.5(a)). An optimal streamline 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 streamline placement effectively in order to minimize it. This is achieved using various randomized changes to the position (Figure 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 blurred image (fundamentals, Section 3.2) from a uniform grayscale target.

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

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

2. The second component is the low-pass filter L. It uses a kernel to generate the filtered image of a streamline. 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 streamline section is considered. Conversely, the only way to get a brightness greater than one is either via a curve in the streamline or two streamlines being too close to each other. This is immensely useful, as it allows two streamlines to get close to each other from the ends, but not from the sides (Figure 4.5(b)).

3. In order to determine the brightness, also referred to as the *energy* of the image generated by the kernel application, the following expression is used as the *energy function*:

$$E(I) = \int_x \int_y [(L * I)(x,y) - t]^2 \, \mathrm{d}x \, \mathrm{d}y \tag{4.1}$$

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

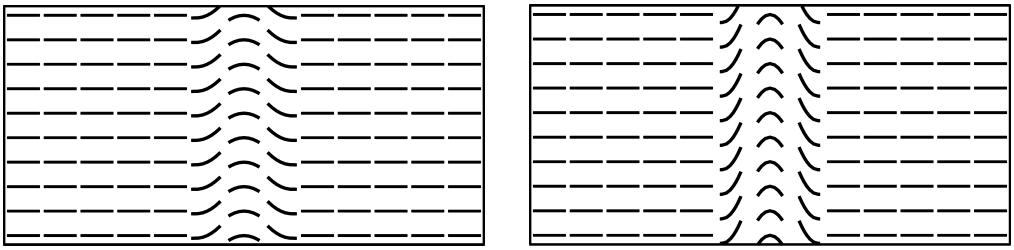
It is pivotal for this energy measure to react to minimal changes for the algorithm to successfully converge. Therefore, this algorithm does *not* use a simple rasterization and blur technique, as the initial discretization/rasterization of the streamline would remove too much information, even when 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 streamline 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 streamline, leading to much higher sensitivity.

4.2.3 Randomized Optimizations

Turk and Banks define six actions:

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

These actions are selected randomly with random parameters, then applied to a random streamline. 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 streamlines toward a more uniform energy level. Naturally, this depends heavily on the choice of t. If t were to be chosen closer to two, the image would become crowded by many streamlines to reach the increased target gray level.



(a) (b)

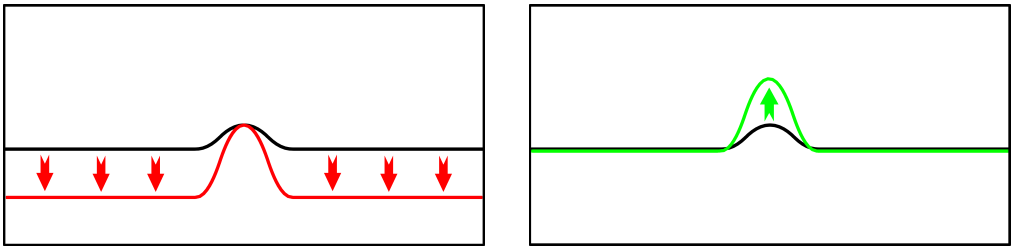
Figure 4.6: (a), (b): A vector field undergoing a change over time, increasing the amplitude at its center, visualized using a hedgehog plot. It is described by $u(x, y, t) = (1, \sin(5t\pi x)$ if $0.4 \leq x \leq 0.6$ else $0)^T$ for $t = 1$ and $t = 3$ in (a) and (b) respectively.

4.3 Motivation for Time Coherence

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.

We can illustrate this behavior using the field in Figure 4.6, taking a closer look at $u(x, y, t)$ at two time steps $t = 1$ 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.



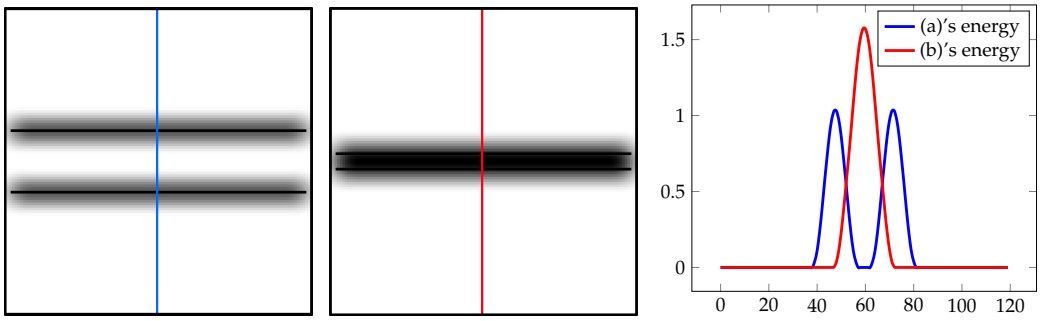
(a) (b)

Figure 4.7: A streamline (black) seeded directly at the center for both cases. The colored arrows represent the perceived direction of motion. (a) The streamline mostly drifting downward from time $t = 1$ to $t = 3$ (red). (b) The same initial streamline at $t = 1$, but keeping same height at time $t = 3$ (green) because the algorithm shifted the seed.

We now show how this behavior can lead to a visual artifact when using streamlines as the means of visualization. In Figure 4.7 (a), we see how the increase in ridge height leads to movement of the whole streamline, 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 are looking for is a line placement strategy that allows large portions of the image to stay the same between two time steps, i.e. that achieves the largest possible "streamline overlap" between the two images, as shown in the transition in Figure 4.7 (b). Here, large portions remain stationary, with only segments close to the center rising upwards, reflecting the localized change much better.

An important factor for the generation of such images is the seed choice. We have chosen the same centered 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*.



(a) (b) (c)

Figure 4.8: How the proximity of streamlines changes the brightness of their respective footprints in a 120x120 px image. (a) Two streamlines approx. three times the blur radius apart. (b) Two streamlines only 3/4 the blur radius apart. (c) Energy of the more distant streamlines (a, blue) and the closer ones (b, red) is shown in the y-axis. The x-axis shows 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. We base this measure on the spatial energy function E from the Turk and Banks algorithm.

> **Notation:** To avoid confusion, we from now on write the spatial components (former E and L) as E_s, L_s to better separate them from their temporal counterparts E_t and L_t. t has two further uses: If we talk about time steps, t refers to the time, e.g., when describing vector fields $u(x, y, t)$. In the context of the Turk and Banks algorithm, t is the target brightness.

Instead of the comparison between a low-pass image of streamlines and a constant target brightness used in E_s, the temporal energy E_t now depends on two sets of streamlines (I_0, I_1) and uses a different low-pass filter (L_t):

$$E_t(I_0, I_1) = \int_x \int_y [(L_t * I_0)(x, y) - (L_t * I_1)(x, y)]^2 \, \mathrm{d}x \, \mathrm{d}y$$

This measure tells us the sum of squared difference between the energy of two different sets of streamlines. 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 streamline

positions, which can be seen in Figure 4.8. This means that by using E_s instead of E_t, we would change the number or position of streamlines by drawing them into the center of two former streamlines, thereby intently worsening time coherence. Conversely, this would also prohibit streamline creation at darker spots, giving rise to holes in our streamline 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 streamlines, moving them in the direction of their former footprints.

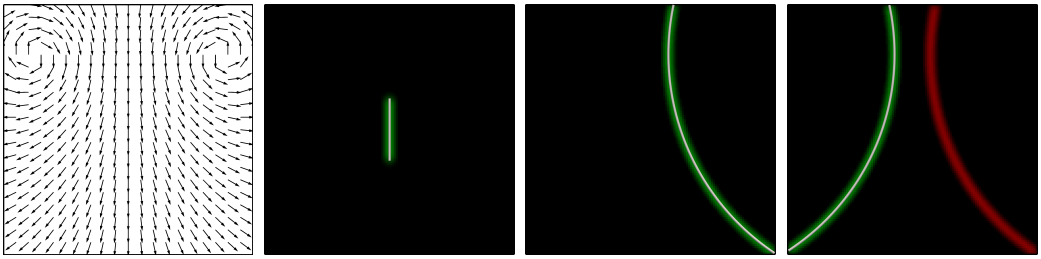
A second addition we call *shattering* will be introduced as well, where streamlines are split into smaller segments (*fragments*) to be used as the starting layout for the next time step. This can increase coherence as well by seeding streamlines at the same place, and works especially well in combination 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 good leverage regarding how streamlines are placed. In order to make the algorithm favor previous streamline positions, we therefore rewrite the previous energy function E 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 generated using the time kernel $L_t * I_0$ and the current image as $L_t * I_1$, we use:

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

We have found values for α in the range $[0.4, 0.8]$ to be effective. Choosing a higher value causes very few streamlines to be drawn, and only yields sporadic segments



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

Figure 4.9: Streamlines 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) Initial seed and starting streamline length. (c) Random move and lengthen steps reach a local minimum energy by increasing the streamline length, converging to one side at random. (d) A different result may occur with the same starting conditions.

due to the inhibitory effect on the lengthen and join operations when leaving the previous streamline'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.

Since we compare many images and footprints from this section onward, it is useful to include a distinction using different color channels. For the rest of this thesis, we use a consistent coloring to show streamline movement between time frames. 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 or change their length slightly due to this inherent randomness.

We now take a closer look at the energy development for different streamline positions seen in Figure 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) shows the two likely outcomes of how the starting streamline develops under the specified starting conditions.

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

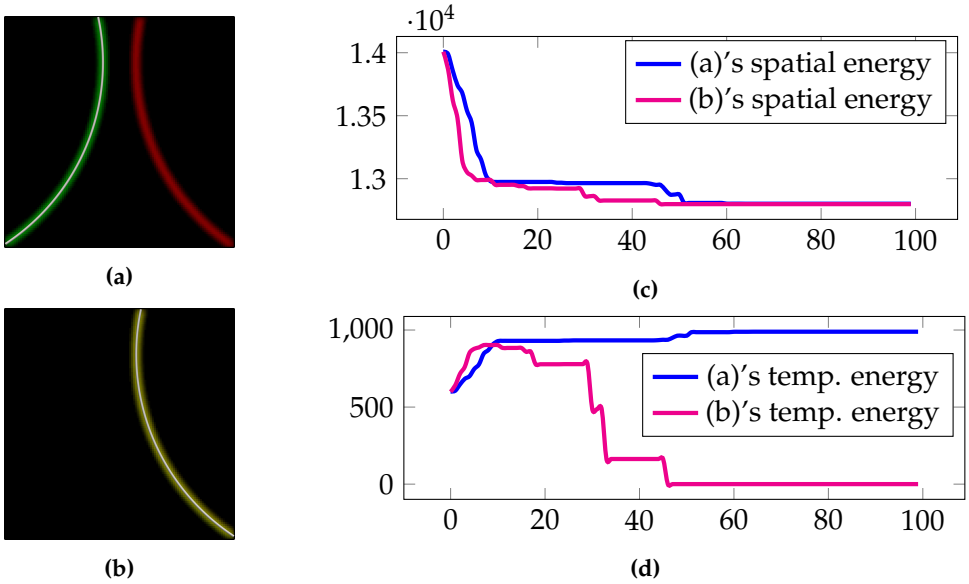


Figure 4.10: (a) Two divergent streamlines resulting from the same initial streamlet. (b) The line overlap in favor of temporal energy. (c) Spatial energy of (a) and (b) vs optimization steps. (d) Temporal energy of (a) and (b) vs optimization steps.

In Figure 4.10 (c), we see a plot of the spatial energy vs the current optimization step. The maximum spatial energy equals the image resolution at 120x120=14400. Initially, we see a decline in spatial energy for both curves, caused by the comparatively fast lengthening process. On average, the streamlines grow about 3 % image size per step on both sides. A plateau phase is reached after $\approx$ 10 steps, as the streamlines approach the domain boundaries and cannot lengthen further. Due to random movements, the streamline ends drift along the upper and lower domain borders, slowly lengthening due to the field's curvature increasing the further outward they move. A final stronger decline happens at 30 and 50 steps for (a) and (b) respectively, as the curvature is strong enough to allow larger regions to be filled by lengthening again. The delay between (a) and (b) is caused by the random nature of the algorithm, and is equal if both runs use the same randomization setup. Our streamlines reach the minimum possible energy of about 12800, with a reduction of $\approx$ 1600. Since the streamline length lies at roughly 140 px and the filter is about 14 px wide (note: L_t as shown is half the size of L_s), these spatial energy measures aren't surprising. The temporal energy is shown on plot (d), where we can see a stark difference in the development of (a) and (b). Many features regarding local rate of change are recognizable in both plots, e.g. the plateau of (b) centered around the 40 step mark. The final temporal difference lies at 1000 due to the reduced filter radius.

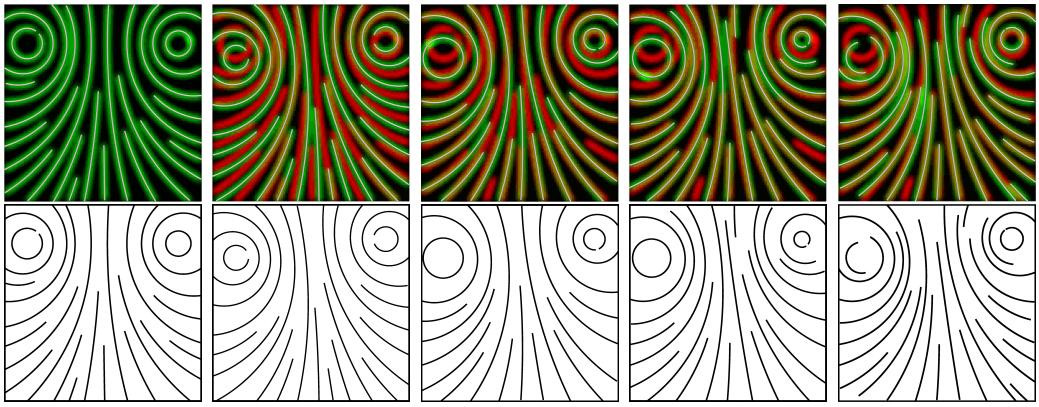


Figure 4.11: Comparison of different α values for a fixed $r_t = 0.5$ between the different second frames in the double gyre field from Figure 4.9. The top row contains the footprints, the bottom row only the streamlines for visual clarity. (a) Field at $t = 0$, the same first frame is used for (b) to (e). (b) $\alpha = 0$, no coherence. (c) $\alpha = 1/3$, some coherence on the image borders. (d) $\alpha = 2/3$, good coherence in most regions, some strong changes remain. (e) $\alpha = 1$, maximum coherence w/o regard for spatial placement.

4.5.2 Parameter Study for α and L_t

Since L_t drives the temporal energy measure, and α determines its weight, we show how different values for each parameter affect the images generated for two different datasets.

> We first introduce some variables for brevity and clarity: The radius of L_t is written as r_t, and defined as a factor compared to the radius of L_s. To indicate that we use a radius for L_t 1/3rd the size of L_s's, we simply write $r_t = 1/3$.

The first set of images from Figure 4.11 uses $r_t = 0.5$ to not introduce artifacts from the footprints overlapping as discussed in Section 4.3. In (a), we see the baseline image of the first time step $t = 0$, and (b) to (e) are all images of the second time step.

Figure 4.11(b) With $\alpha = 0$, the streamlines moved without regard for time coherence, only achieving it by chance or seed choice.

Figure 4.11(c) Using $\alpha = 1/3$, we immediately notice some improvements as the outer regions near the bottom and on the left are more stationary. The footprints overall gain a more compact appearance, as green and yellow parts aren't as far

apart as they were previously, and we can see more regions where no streamlines were drawn before remain empty.

Figure 4.11(d) $\alpha = 2/3$ causes even the center to remain nearly the same, and the image seems to exhibit good time coherence all over. An artifact at the top center was introduced due to the temporal preference, with a very short streamline appearing.

Figure 4.11(e) The image degradation from the previous step is exacerbated; with $\alpha = 1$ we can see the spatial quality drop decisively when looking at the streamline representation compared to (c). This is to be expected, as setting $\alpha = 0$ causes the algorithm to disregard E_t entirely, and setting $\alpha = 1$ does the same to E_s.

The optimal range therefore lies between $[1/3, 2/3]$, with the former possessing better spatial quality. We move to the next set of pictures with $\alpha = 0.5$.

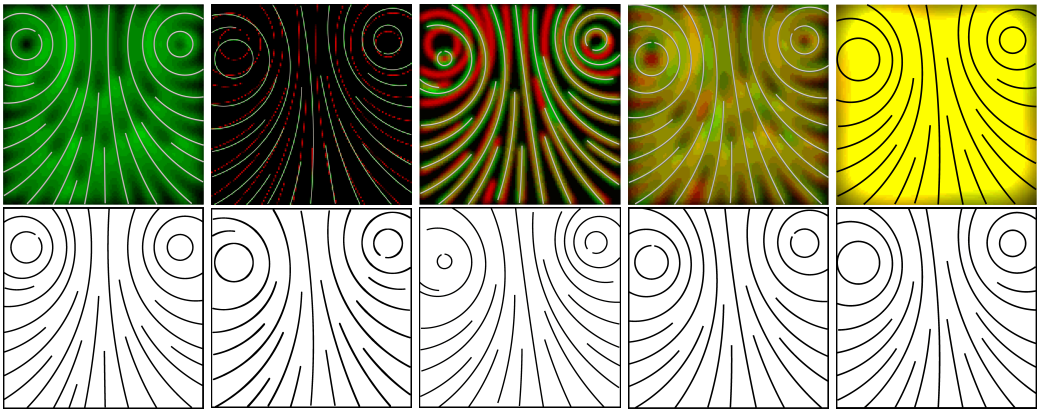


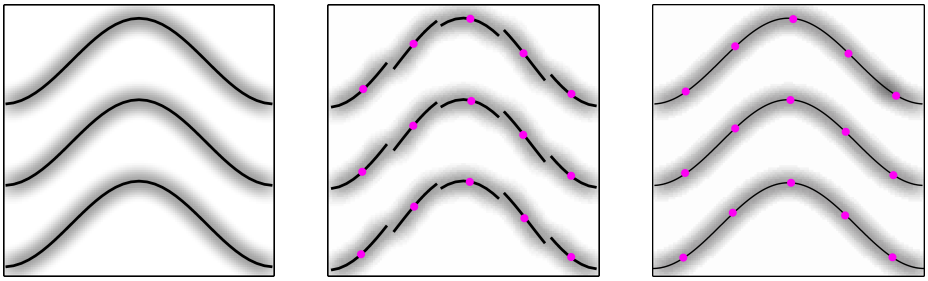
Figure 4.12: Comparison of different α values between the first five steps for an unsteady version of the double gyre field from fig. 4.9. Each row represents five time steps and for each step, the left vortex moves down by 3% of the image height. (a): With $\alpha = 0$ most streamlines have low coherence 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.

For the second part of this section, we look at how changes to r_t affect the image as seen in Figure 4.12. We start with an extreme case for $r_t = 0.05$, giving us footprints that are almost invisible in the rendered version of the low-pass image. Accordingly, the footprints have almost no effect on the streamline placement, and using low values for r_t effectively removes time coherence altogether. As the next example, we use an r_t equal to the radius of L_s. Here, we can see a lot of yellow regions, however, when using such a large radius, they do not necessarily mean that time coherence was achieved. We conclude search for a good r_t with a final extreme case of $r_t = 3$. While almost the entire image is rendered as yellow due to the energy being capped at 2.0 when rendering to fit the range [0, 255], the energy measure itself can use higher values. In fact, the streamlines themselves exhibit time coherence regardless, as their energies can still be compared and keep reacting to small changes in position as long as they happen within the falloff distance due to the rasterization routine. This reduces the accuracy however, as contributions from one streamline can fade and be replaced by those from a different streamline more easily.

We thus conclude that there is a lower bound for r_t after which time coherence is not reliably achieved. An upper bound was not noticed, just a slow degradation of time coherence after the $r_t = 1$ mark. We also note the partial similarity between α and r_t, as placements generated using lower values for r_t are virtually indistin-

guishable from those created with low αs, as both lead to a diminished final time coherence component in the total energy measure.

We therefore choose $\alpha = 0.5$ and $r_t = 0.5$ as good starting values that introduce a bias toward time coherence, while not regressing image quality too much and remaining local enough to be effective and mostly guide the streamlines next to them.



(a) (b) (c)

Figure 4.13: (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) Shards quickly rejoin when redrawn in the same field they originate from (a). We used a slightly reduced thickness to make it more visible that the shards do not simply overlap but actually rejoin into their single former streamline.

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 streamline length-wise into sections which equal the length of the starting streamlines. The shards are assigned a seed in the middle (lengthwise) of these intervals, and their length equals the streamline start length. If the parent streamline 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. 4.13 (b). The shards then act as the initial seeding strategy for the subsequent time-frame; 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 streamline they came from, as can be seen in fig. 4.13 (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 streamlines. This results in changes being limited to parts where a streamline change is necessary, providing extra streamlines in these areas while not affecting streamline trajectory too much on a global level.

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 streamline 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 streamlines whenever possible while still allowing relaxation to ensure good spatial distribution.

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)" provides a broad spectrum of available algorithms. The most important components used are briefly listed in the following section, the final algorithm design is delivered afterwards in Section 5.2.2, and we conclude this chapter with a complexity analysis in Section 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, we try to summarize it 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 → Screen 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 velocity as a 3D vector. In our case, we use it exactly in this way: Points are assigned velocities, 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 Effectively a wrapper for an image, allowing easier access to some properties. It contains the brightness information of the image that guides our algorithm.

Painter This is the modifying actor for FilterTargets. Painters use a configuration (line brightness, blur size, etc.) and draw poly lines to the FilterTargets accordingly. This is how we distinguish between L_t's and L_s's radii.

Filter These objects contain a list of lines that make up the vector field image, and orchestrate the assigned Painters and their respective configuration objects. They also provide methods for adding/removing/modifying lines, using a given energy function to determine their success. They act as the binding agent for the logic modifying line placement and actually performing the change.

FilterStack This class is best used (though not enforced) as a singleton; it manages two filter objects, one for the current, and one for the last 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 depending on how long ago it was created, e.g. to not only use time coherence w.r.t the last frame, or to allow effects like onion skinning of older frames' low pass images.

The entry point for this algorithm is, as with any vtkPythonAlgorithm, the "RequestData" method (we leave out the other Request() methods for brevity). We are provided the vector field via the vtkImageData object as input, and start to set up our low-pass filter stack.

By setting up a filter with a config (the standard config uses similar values as Turk and Banks' implementation), we create 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 returned from the vktStreamTracer. 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

Line Integration We heavily rely on the vtkStreamTracer (see Section 5.2.1), which we configured to use the Runge-Kutta-4 solver. RK4 uses a complexity of $O(n)$, with n being the number of integration steps taken. Hence, obtaining the sample points of a single streamline of length n is of complexity $O(n)$.

Drawing a Line Segment When computing the footprint of a streamline, we do so segment by segment. A segment footprint's pixel count f is defined by two values: The filter radius r and the maximum step length l. The maximum cost of drawing a segment is therefore a constant, and we write this cost as c.

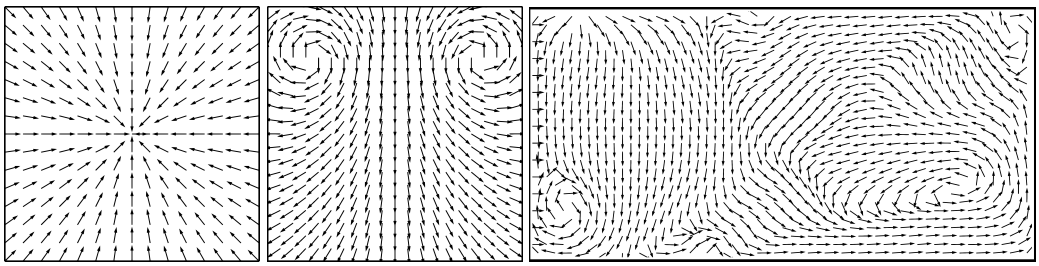
Drawing a Line If the line consists of n segments each costing a maximum of c, the cost is simply $O(n)$. This remains unchanged when including the cost of integration, as we still get $O(2n) = O(n)$.

Generating a Streamline Lines start at very low length, and need to grow and shift, causing them to be redrawn over and over. Generating a single line by letting it grow means we re-evaluate it as often as necessary. We need to draw $O(0.5 \cdot n^2) = O(n^2)$ segments to reach a length of n, combining this with the cost

of drawing a length-n line, we arrive at $O(cn^2) = O(n^2)$.

The randomized operations all cause a line to be re-generated, but since its length remains, this is done with $O(n)$. However, we do this fairly often, making the complexity w.r.t. segment size and iteration count behave as if it were quadratic.

6 Results



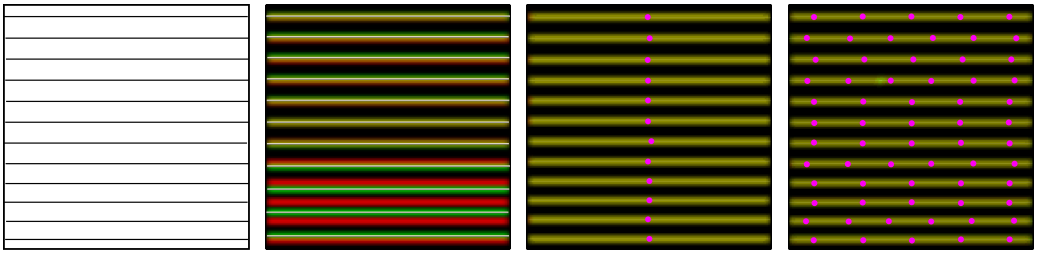
(a) (b) (c)

Figure 6.1: Vector plots of fields used to compare different algorithms in this chapter. (a) A trivial sink $u(x, y, t) = (x - t, y - t)$. (b) Double gyre. (c) HotRoom dataset.

In this chapter, we start by running the full algorithm on various datasets while discussing the different outcomes compared to other strategies. Then, we show some performance metrics and the hardware configuration. We finish this chapter with limitations and some ideas for future work regarding the functionality and choice of parameters of our algorithm. For the entirety of this chapter, we use $0.5 \cdot L_s$ as the radius of L_t and a temporal weight of $\alpha = 0.5$, unless stated otherwise.

6.1 Comparing Behaviors for Different Fields

The comparisons for the various fields are made between Turk and Banks' algorithm without coherence, a slightly modified version that simply redraws the lines at their seed for subsequent frames, and our implementation using coaxing and shattering. We start this section with three simple fields visualized using straight lines, continue with the other fields already used in this thesis, and conclude with images from a 60 x 30 dataset. In Figure 6.1, we see one field from each group: (a) as a base case, (b) to compare behavior on a smaller scale, and (c) as the final benchmark.



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

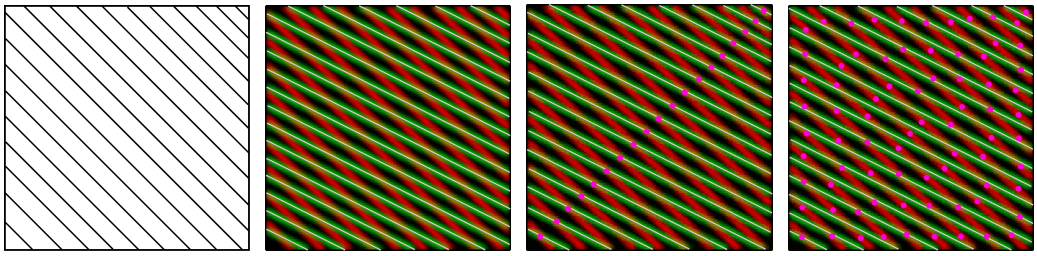
Figure 6.2: (a) The initial state. (b–d) Footprints for the field $u_1(x, y) = (1, 0)^T$ generated by (b) the base algorithm, (c) the constant algorithm, (d) our implementation. Seeds are drawn in magenta.

6.1.1 Baseline Fields

> **Note**: We refer to the implementation by Turk and Banks as the *base algorithm*, and call the modified version placing constant seeds the *constant algorithm*. Seeds, if applicable, are drawn in magenta.

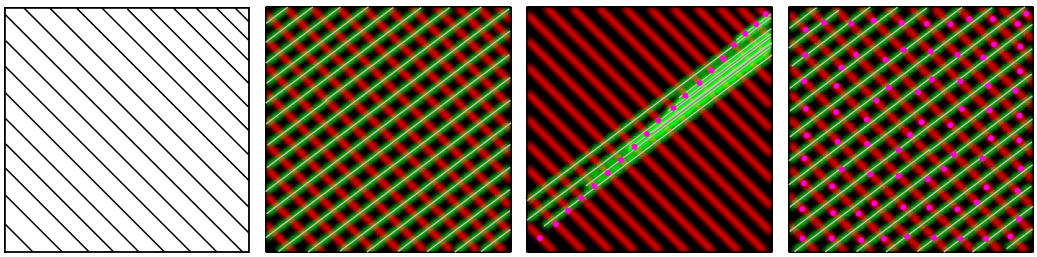
We compare streamline placements for three baseline cases exhibiting inherently trivial coherence, no coherence, and partial coherence. We start with the first case by using a constant field $u_1(x, y) = (1, 0)$, which is seen in Figure 6.2. Coherence is trivially achieved due to it being steady, i.e. time coherence is already achieved by not moving the seeds.

One notable difference in Figure 6.2(b) compared to (a) and (c) is the removal of a line in the lower half for the base algorithm. This happens due to the lines being added and removed at random, and an early removal has left more room for relaxation so that it could not be back-filled again. Since (b) does not have any time coherence, we can conclude that it has not converged on one placement at the 1500-step mark, as the result of two of its executions is different. While this effect also exists in our implementation, it is essentially neutralized by the temporal weight, thus producing matching lines.



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

Figure 6.3: Streamline placement for $u(x, y, t) = (1, t)^T$. (a) u at $t = -1$. (b) u at $t = -0.5$, base algorithm. (c) u at $t = -0.5$, constant seeds. (d) u at $t = -0.5$, our implementation.



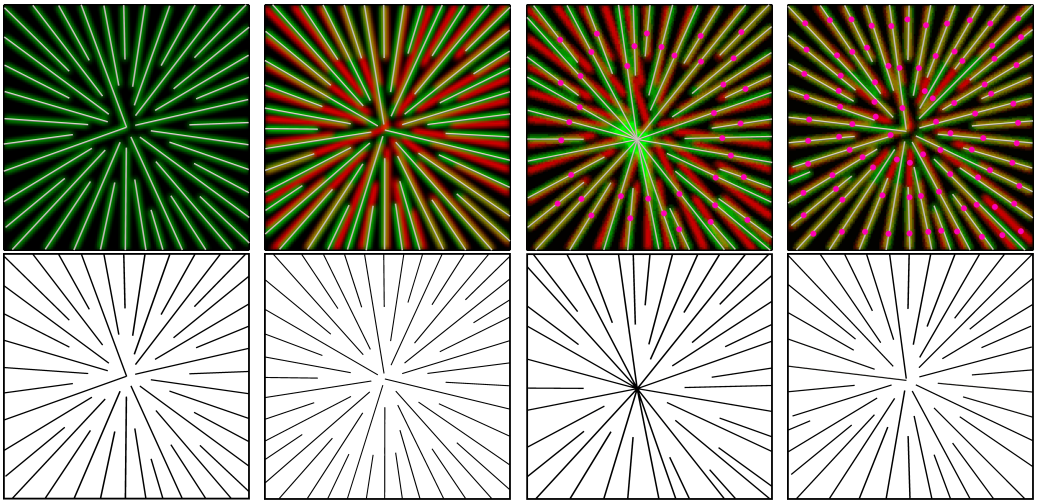
(a) (b) (c) (d)

Figure 6.4: Streamline placement for $u(x, y, t) = (1, t)^T$. (a) u at $t = -1$. (b) u at $t = 0.75$, base algorithm. (c) u at $t = 0.75$, constant seeds. (d) u at $t = 0.75$, our implementation.

The second case uses a field with a rotating set of parallel streamlines defined as $u_2(x, y, t) = (1, t)^T$, which is seen in Figure 6.3 and Figure 6.4. Due to the nature of this field, time coherence is difficult to achieve consistently along the entire line. Hence, the only useful measure is the spatial energy, and all three algorithms generate images with a uniform spatial distribution of streamlines. This shows that our implementation can effectively deal with this case too when choosing α accordingly, as stated at the beginning of this chapter, allowing E_s to take over when E_t cannot produce a consistent bias.

The case where $t = 0.75$ in Figure 6.4 makes algorithms relying too much on the seed placement perform badly. We can see how the constant algorithm in (c) tries to rectify the high spatial energy by shortening the streamlines, as it cannot move the seeds apart.

Our algorithm is able to mitigate such effects due to allowing sufficient relaxation for the streamlines created from the individual shards. Since we only allow new lines to be created from seeds obtained during the shatter process when they improve the image, the effect of "over-seeding" as seen in (c) is avoided.

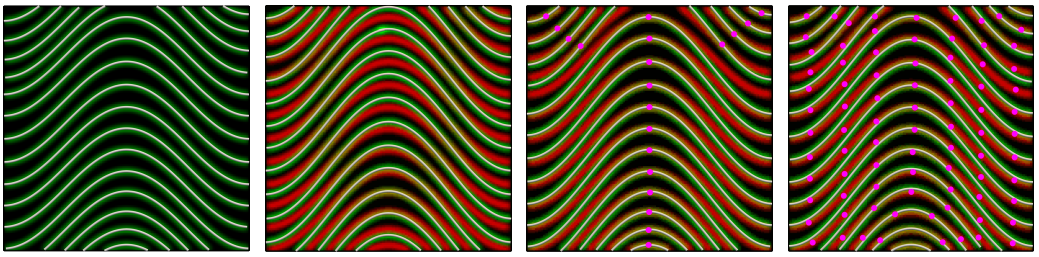


(a) (b) (c) (d)

Figure 6.5: Streamline placement for $u(x, y, t) = (-x - t, -y - t)^T$. (a) u at $t = 0$. (b) u at $t = 0.05$, base algorithm. (c) u at $t = 0.05$, constant seeds. (d) u at $t = 0.05$, our implementation.

For the last trivial case, we look at the sink defined by $u_3(x, y, t) = (-x - t, -y - t)$ in Figure 6.5. The base algorithm (b) places its streamlines with good spatial uniformity, but very little overlap between the frames. While streamlines from the constant algorithm (c) exhibit better coherence, their spatial quality quickly decays. Our implementation (d) creates evenly spaced streamlines as well, and behaves similar in terms of placement to (c) due to the shatter process.

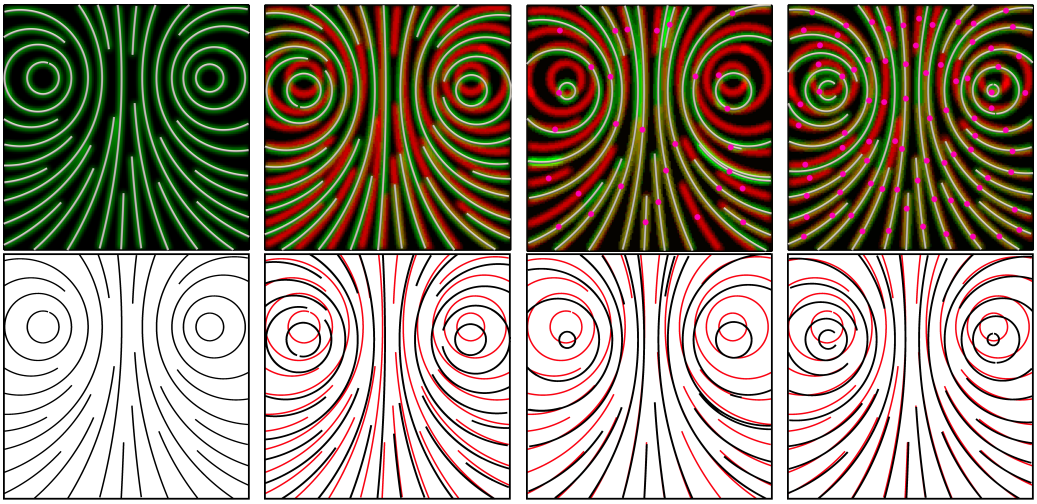
6.1.2 More complex Fields



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

Figure 6.6: Streamline placement for $u_1(x, y, t) = (1, \sin(x)t)^T$. Seeds are drawn in magenta. (a) u at $t = 1$. (b) u at $t = 1.25$, base algorithm. (c) u at $t = 1.25$, constant algorithm. (d) u at $t = 1.25$, our implementation.

We now look at some more interesting cases allowing for stronger differences in temporal and spatial placement quality. The first field in Figure 6.6 uses a sine pattern defined by $u_1(x, y, t) = (1, \sin(x)t)^T$. Here, we can see slight reduction in image quality for our implementation (c) compared to the base algorithm (b), as ours produces a gap in the top half. We note a strong similarity between (c) and (d) because our algorithm favors the time coherence provided by the curved regions in the center and near the edges, as opposed to the straight sections between them. Coherence is therefore focused on the curved regions, as the overlap is simply higher. Furthermore, when compared with the image from the motivation in Figure 4.7, we see that we have reached the case we wanted to avoid at first. This is expected, because the algorithm can no longer determine which change to consider the "local" one since we do not have the stationary sections on the sides.



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

Figure 6.7: Streamline placement for the double gyre. The first row shows the footprints, the second row the lines for visual clarity, with the red lines being the starting frame (a), and the black lines the respective algorithm's placement. (a) base case, with vortices at 70% image height. (b) vortices at 65% image height, base algorithm. (c) vortices at 65% image height, constant algorithm. (d) vortices at 65% image height, our algorithm.

As the second medium complexity case, we show the double gyre with a slightly different movement pattern. Moving both centers downward instead of just one allows for better time coherence as the field does not undergo a rotation but rather a downward translation. The lower row in Figure 6.7 clearly shows how (c) and (d) perform significantly better than (b), with (c) exhibiting very inconsistent line spacing. Our implementation handles this case better, with deviation from the previous frame only noticeable at the gyre centers, which is expected since they produce the strongest field distortion when moved over time.

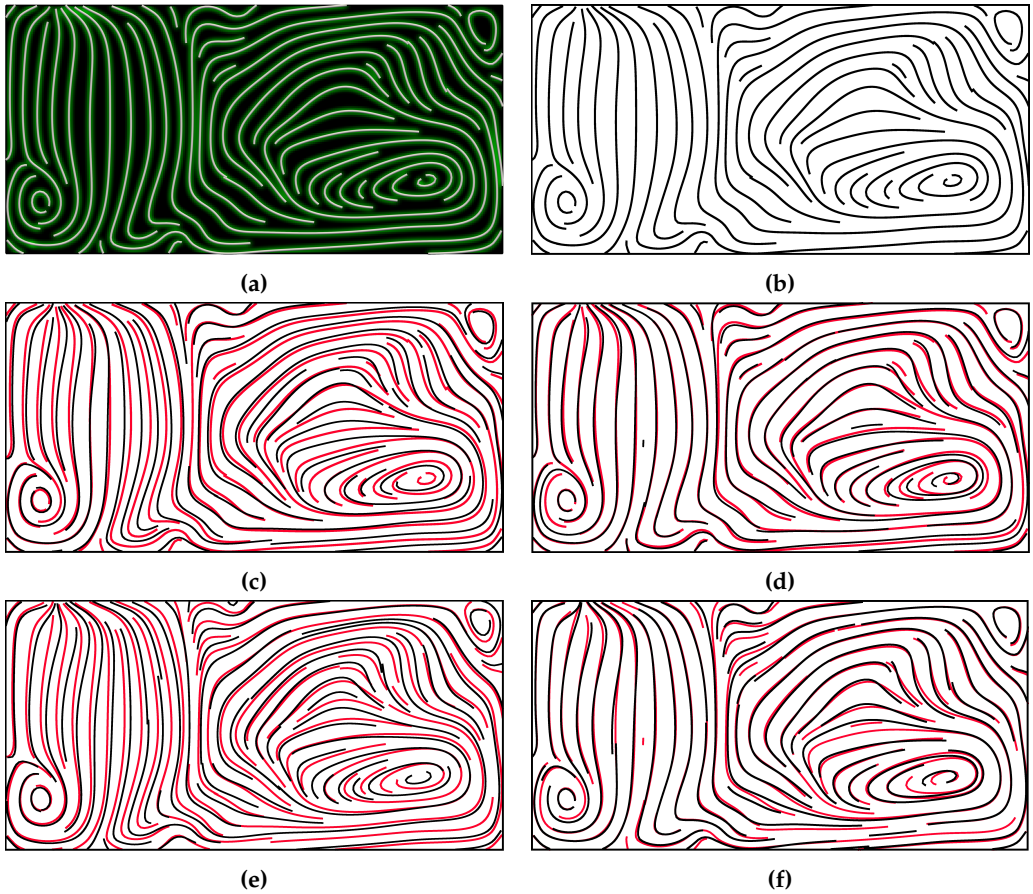


Figure 6.8: A comparison to how similar each algorithm's streamline image is compared to its previous one. (a, b) are the initial frame's footprint (left) and streamline placement (right) for both algorithms at time step 450. (c) shows the base algorithm's streamline placement (black) at step 451 compared to (b)'s streamlines (red). (e) compares step 452 (black), to 451 (red). (d, f) work similarly with our implementation.

6.1.3 Hot Room Dataset

The final vector field we examine contains 60x30x60 data points. Since the dataset is 3D and our algorithm only works in 2D, we slice the dataset parallel to the X, Y-plane at half height with a Z-index of 30. Because the field is more chaotic than the algebraic cases we used until now, we decided on an iteration limit of 5000 steps compared to 1500 for the other images. The comparison is made between our implementation and the base algorithm only, as the constant algorithm does not produce useful images beyond the first, and we will show several.

We start with the dataset at timestamp 450, the individual changes shown in Figure 6.8. It is easy to see the general trend of our implementation and the base

algorithm producing images of similar quality w.r.t. spatial placement continuing, with our version creating images exhibiting much better time coherence.

On the next page, we evaluate the behavior of our implementation when we skip a large amount of frames. We start at frame 700 and generate the subsequent frame, jump to 1000, and generate the next frame again, comparing the results.

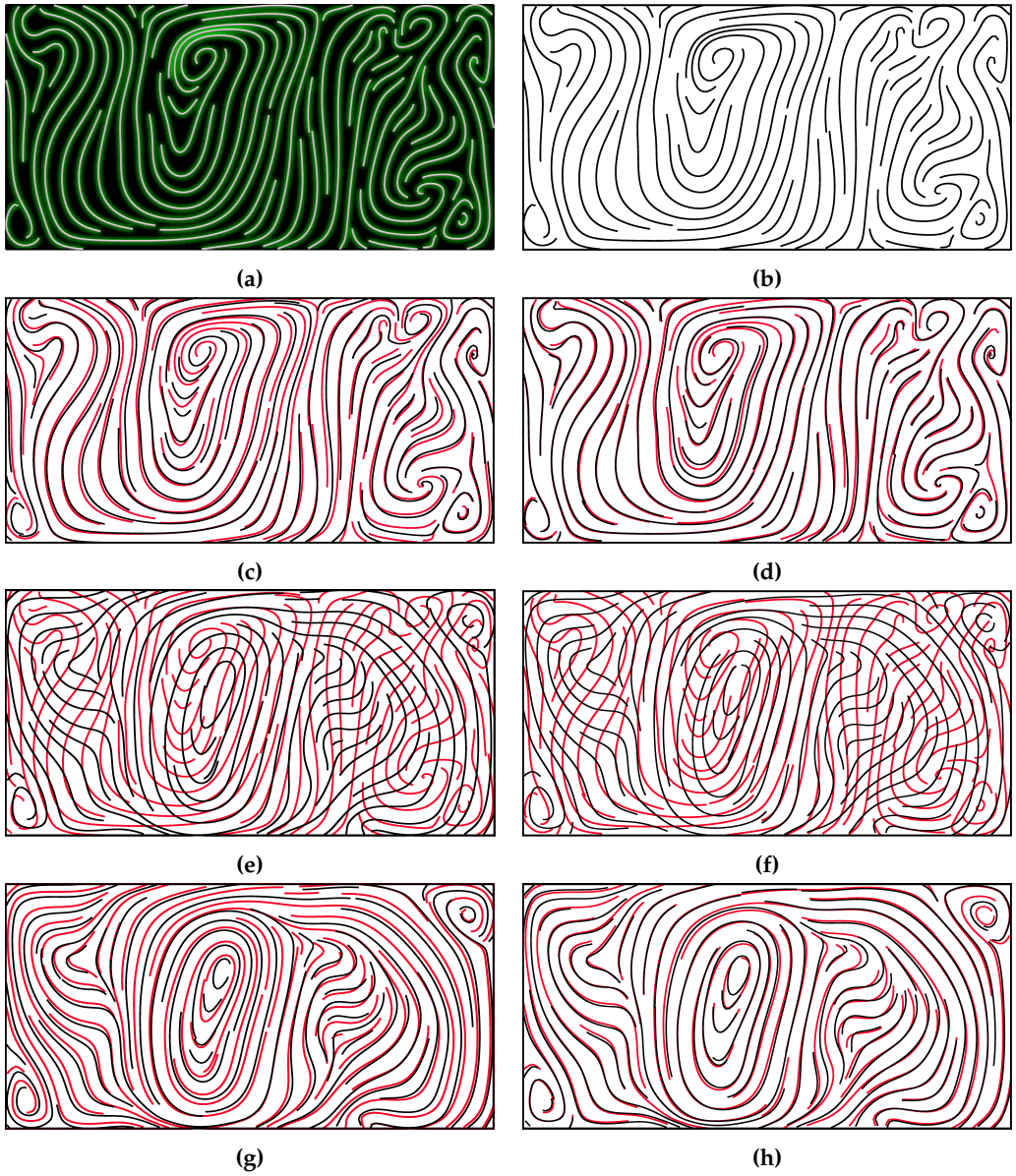


Figure 6.9: Same image layout as Figure 6.8, with a 300 time step jump between the top and bottom four images.

We see that the jump in time steps occurred between Figure 6.9 (c) and (e), and (d) and (f), as the currently shown streamline after image does not match the afterimage at all. Even for this case, our algorithm creates lines adhering well to the spatial criteria, with time coherence not achieved in most regions, as is to be expected since most of the field is vastly different due to the large time step differ-

ence. The algorithm quickly recovers from this discrepancy, as the next frame has regained most of its time coherence.

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. Images from the HotRoom dataset took about 13 minutes per frame when using coaxing (and half the time otherwise) due to the convolution with two filters being twice as expensive. For comparison, the initial greedy algorithm took about 10 seconds. The size of the low pass image is of little to no importance for speed, we tested filter radii between 2 and 32, both of which took about the same time to complete.

6.3 Limitations and Possibilities for Future Work

Restriction to 2D An obvious constraint is the restriction to two-dimensional data. This is a result of the image-guided nature from Turk and Banks' algorithm, as its principle only makes sense with a "bird's eye" view onto the line image. Extending this principle into 3D could be done by extracting a view-dependent surface, and then using the algorithm on that surface.

Sensitivity to Field Changes Another important aspect is the sensitivity of the algorithm regarding α and the distance lines move from one time step to the next. If this distance is too large, high α values will quickly lead to degradation of image quality due to the high coherence weight. This can be counteracted by using more fine-grained interpolation of the field movement, or reducing α to allow the lines to move and relax more freely.

Adaptive Choice of α and r_t The coherence weight parameter is configurable prior to a run, but for different datasets (or even different frames) it should prove useful to vary this parameter. This could perhaps be added via an adaptive method that generates n line placements for the next frame using different α and r_t values. It then moves α and r_t closer to the parameters from the image causing the least amount of relative energy fluctuation. Of course, this would increase the computational cost by a factor of n, but potentially allows more effective automation by removing the need for fine-tuning in some cases.

Memory Usage 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 grows $O(n \cdot m)$, with n being the number of lines and m the image pixel count.

7 Conclusion

We have shown a motivation for and definition of time coherence, and discussed its importance when animating vector fields. Two basic algorithms capable of spatial coherence were presented, one of which we adapted to become capable of time-coherent streamline placement. Examples for different configurations and use cases were provided and compared, with conclusions regarding optimal parameter choices.

By running our algorithm on a multitude of different vector fields and datasets, we have shown that it drastically improves the time coherence of images generated for subsequent time steps in unsteady fields, and that it can be used to animate them effectively.

Some ideas for future work include choosing different base algorithms and approaches, especially regarding performance to allow for interactive use were presented. An implementation for 3D vector fields in combination with view-dependent line choice may also prove to be quite interesting and useful with many areas of application.

Bibliography

[JL97] B. Jobard and W. Lefer. "Creating Evenly-Spaced Streamlines of Arbitrary Density". *Proceedings of the eight Eurographics Workshop on visualization in scientific computing* 7, 1997 (cit. on p. 4).

[LMG06] Z. Liu, R. Moorhead, and J. Groner. "An Advanced Evenly-Spaced Streamline Placement Algorithm". *IEEE Transactions on Visualization and Computer Graphics* 12:5, 2006, pp. 965–972 (cit. on p. 4).

[LS07] L. Li and H.-W. Shen. "Image-based streamline generation and rendering". *IEEE Transactions on Visualization and Computer Graphics* 13:3, 2007, pp. 630–640 (cit. on p. 3).

[MAD05] A. Mebarki, P. Alliez, and O. Devillers. "Farthest point seeding for efficient placement of streamlines". In: *VIS 05. IEEE Visualization, 2005.* 2005, pp. 479–486 (cit. on p. 4).

[MTHG03] O. Mattausch, T. Theußl, H. Hauser, and E. Gröller. "Strategies for Interactive Exploration of 3D Flow Using Evenly-Spaced Illuminated Streamlines". In: 2003 (cit. on p. 4).

[SLCZ09] B. Spencer, R. S. Laramee, G. Chen, and E. Zhang. "Evenly Spaced Streamlines for Surfaces: An Image-Based Approach". *Computer Graphics Forum* 28:6, 2009, pp. 1618–1631 (cit. on p. 3).

[TB96] G. Turk and D. Banks. "Image-guided streamline placement". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. Association for Computing Machinery, New York, NY, USA, 1996, pp. 453–460. ISBN: 0897917464. URL: https://doi.org/10.1145/237170.237285 (cit. on pp. iv, 3, 16).

[VKP00] V. Verma, D. Kao, and A. Pang. "A flow-guided streamline seeding strategy". In: *Proceedings Visualization 2000. VIS 2000 (Cat. No.00CH37145)*. 2000, pp. 163–170 (cit. on p. 4).

[WLZM09] K. Wu, Z. Liu, S. Zhang, and R. J. Moorhead II. "Topology-aware evenly spaced streamline placement". *IEEE Transactions on Visualization and Computer Graphics* 16:5, 2009, pp. 791–801 (cit. on p. 4).