

Visualizing 2D Flows with Animated Arrow Plots

Bruno Jobard¹, Nicolas Ray² and Dmitry Sokolov³

¹ LIUPPA laboratory, University of Pau, France, bruno.jobard@univ-pau.fr

² ALICE Team, INRIA Nancy Grand-Est, France, nicolas.ray@inria.fr

³ University of Lorraine, France, dmitry.sokolov@loria.fr

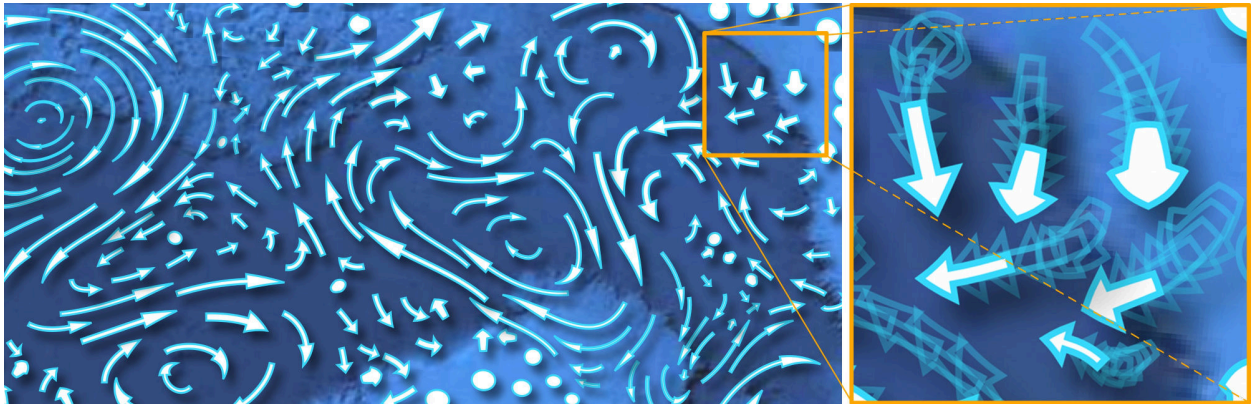


Figure 1: **Ocean currents visualized with a set of dynamic arrows.** (Left) The domain is filled with arrows aligned with the flow. The length is proportional to the velocity magnitude. The arrow density is controlled by a custom map to better capture local turbulences. (Right) Close-up showing the arrow trajectories and the morphing of their glyphs.

Abstract

Flow fields are often represented by a set of static arrows to illustrate scientific vulgarization, documentary film, meteorology, etc. This simple schematic representation lets an observer intuitively interpret the main properties of a flow: its orientation and velocity magnitude. We propose to generate dynamic versions of such representations for 2D unsteady flow fields. Our algorithm smoothly animates arrows along the flow while controlling their density in the domain over time. Several strategies have been combined to lower the unavoidable popping artifacts arising when arrows appear and disappear and to achieve visually pleasing animations. Disturbing arrow rotations in low velocity regions are also handled by continuously

morphing arrow glyphs to semi-transparent discs. To substantiate our method, we provide results for synthetic and real velocity field datasets.

Introduction

Arrow plots are standard static representations for 2D vector fields. They are intuitive and thus often used to present flows mixed with a contextual background image to non-expert public. The goal of this work is to provide a simple algorithm that produces clean animated arrow plots for presentation purpose.

This aim greatly differs from the traditional objective as evidenced by the most recent 2D vector field visual-

ization techniques, where the efforts have focused on the interactive exploration of the data. For the purpose of exploration, image-based techniques such as flow textures (LIC and its animated extensions) allow for interactive visualization of flow details by using every pixels of the display device to communicate dense information. However, flow textures present two major drawbacks in our context: blending them with an additional color map (a background image or a dynamic field) might greatly deteriorate its details, and such representations are quite sensitive to the quality of the display device and cannot be controlled while broadcasting videos or during a public presentation. For the purpose of presentation, a sparse set of moving arrows can easily convey the desired information. The arrows do not deteriorate the background information (only occlude it temporary), and is robust to low quality display device.

This work proposes a method producing sparse and smoothly animated representations of a flow with moving arrows (Figure 1). We list below the general properties and constraints such an algorithm should satisfy.

Arrow trajectories: to intuitively convey its dynamic nature, the arrow trajectories should follow the flow.

Local flow depiction: the arrow shape should depict the local orientation and velocity magnitude of the flow at any time.

Uniform domain coverage: the representation should provide an uncluttered information of the flow everywhere in the domain at any time.

Smooth animations: the arrow movement should be as smooth as possible to avoid distraction.

We propose an algorithm that generates intuitive arrow plot animations by advecting and bending arrows over time while guaranteeing that arrows will not occlude each other and ensuring a complete coverage of the domain. Moreover, the method is able to adapt the density of arrows to arbitrary density field.

However, keeping a uniform coverage of the domain with no occlusion involves inserting new arrows to fill the empty places and removing some arrows in places that get too crowded. This necessary insertion and deletion of arrows introduce strong popping artifacts when they appear and disappear, deteriorates the smoothness of the animation. Our algorithm has been designed to minimize it, both when generating arrows and at rendering time.

The main contributions in this paper are :

- an efficient algorithm that controls the density of arrows and manages their life span while maintaining low popping artifacts,
- a rendering algorithm that further reduces popping by both fading arrows in and out, and a morphing strategy that handles transitions between high and low velocity regions,
- and experimentations on both real and synthetic flows to evaluate how much the (unavoidable) popping artifacts can be maintained acceptable for visualization purposes.

The rest of the paper is organized as follows: after reviewing the state of the art, our arrow representation is introduced (Section 1), the arrow generation algorithm is provided (Section 2), the capacity to adapt the arrow density to any density field is explained (Section 3), the rendering method is presented (Section 4), and results are presented (Section 5) and discussed (Section 6).

Related work

Most of the recent work on visualizing 2D vector field have targeted the ability for scientists to interactively explore their flow datasets. This task is very efficiently achieved with texture-based techniques [9] which offer a dense representation of the fine details of a vector field. They are inexpensive to compute and can produce smooth animations of unsteady flows. However, when combined with a background image, these texture-based representations sometimes fail in displaying both the fine details of the flow and the background (see Section 6.3 and Figure 11). In the context of presenting flow behavior in an animated way to non-expert public, simpler and more schematic alternatives are more attractive.

Numerous such geometric based flow visualization methods were invented over the two last decades [12]. We focus below on the techniques closely related to visualization of a flow field by animated geometric primitives.

Vector plots: The simplest vector field visualization method consists in drawing straight segments originating from the nodes of an underlying mesh [2] (Possibly a Cartesian grid) to indicate the local flow direction and possibly its orientation by placing an arrow tip at its other

end. Its magnitude might be conveyed by the segment length. The main drawback comes from the origin of arrows being unable to change over time, leading to occlusions [8] and confusing animations when the vector field is a flow.

Arrow placement: In flow visualization, clutter and occlusion problems have been mainly addressed in the context of streamline placement methods. These algorithms apply here since an arrow can be carried by a small streamline (streamlet) to better depict the local flow. Any of these numerous methods [18, 6, 13, 10] can be used since they guarantee that no streamlet will be placed within a distance d_{sep} to its neighbours. It is also possible to adapt the streamline density [16]. An animated streamline placement has been proposed by Jobard and Lefer [7]. This later work renders the streamlines with an animated texture that looks like particle trails advected along the streamline. Restricting these particle trails to be aligned on streamlines makes it impossible to advect them in the flow, and constrains all particles of a streamline to be born and die together. Moreover, the lifetime of streamlines is more sensitive to flow evolution e.g. a flow with constant rotation in time will create spinning arrows with our algorithm whereas streamlines would have very short lifetime.

Other methods have been proposed to nicely distribute glyphs. Hiller *et al.* [5] minimizes Lloyd’s energy to evenly distribute glyph’s positions and other works aim to place a minimal number of glyphs [17, 11] to represent the flow. However, these works do not extend nicely to unsteady flows. An error diffusion approach has been proposed [3] to distribute glyphs in unsteady flows, but it exhibits both high popping and the distribution is not convincing.

Particle tracing: The dynamics of the flow can be revealed by visualizing particles advected in the domain. Contrary to arrows, the small size of particle glyphs minimizes the occlusion problems. Inter particle distances has not to be checked and the density is mainly controlled by the seeding strategy. Bauer *et al.* use tiles of Sobol quasi-random positions to regularly seed particles in region of interest of unsteady 3D flows [1]. Since they deal with incompressible flows, the initially constant density of injected particles remains constant during the advection process. Following the same framework, Helgeland and Elboth [4] enhanced the rendering with anisotropic

diffusion to better represent the flow. It is even possible to represent particules with arrows that are advected by the flow [19]. However, arrows will suffer shearing and could only be used when the support is given by a pathlet i.e. advecting a streamlet do not produce a streamlet at the new frame.

Our algorithm better covers the domain and avoids arrow overlaps thanks to more complex creation and deletion strategies, including backward propagation. Somehow, it requires relaxing the realtime feature of particle tracing.

1 Moving arrow representation

During the animation, the arrows are represented with glyphs mapped on rectangular supports. The supports are warped according to the local flow orientation and magitude. Each arrow is initiated from a so-called *handle point* and its support is then warped according to a short streamline integrated backward and forward from the handle point (see figure 2). The integration length of the streamlets is such that their length is proportional to the local velocity magnitude of the flow.

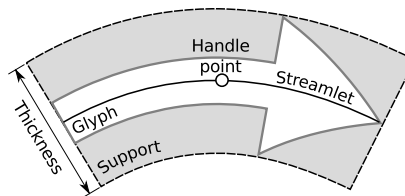


Figure 2: **Arrow Anatomy.** An arrow glyph is mapped on a rectangular support of a given thickness warped along a streamlet integrated from a central handle point.

More formally, given a 2D time-dependent vector field $\mathbf{v}(\mathbf{x}, t) = (v_x, v_y)$, a streamline S is a parametric curve $S(\tau)$ defined at time t and initiated from an handle point p . $S(\tau)$ is given by the equation:

$$\frac{dS}{d\tau} = \mathbf{v}(S(\tau), t) \text{ with } S(0) = p$$

The streamlets have a constant integration length L and

any sample point of the streamlet S is given by:

$$S(l) = S(0) + \int_0^l \mathbf{v}(S(\tau), t) d\tau \text{ with } l \in [-\frac{L}{2}, \frac{L}{2}]$$

A standard Runge-Kutta integration scheme is used to sample the streamlets backward and forward from their handle point.

To prevent having severely distorted arrows in high velocity regions, streamlets might be clamped if the support aspect ratio (arrow length over thickness) becomes superior to a given user threshold.

To intuitively convey the dynamics of the flow, it is preferable for the arrows to be transported along the flow. Since their shape is determined at any time step by the streamlet integration, only the handle point is advected from one step to the next. This particular point follows a pathline trajectory P initiated from a seed point p_s at time t_s :

$$\frac{dP}{dt} = \mathbf{v}(P(t), t) \text{ with } P(t_s) = p_s$$

An arrow will then move along this trajectory between its birth time t_b and its death time t_d ($t_b \leq t_s \leq t_d$). Both t_b and t_d are determined by the arrow reaching the boundaries of the space-time domain or by a lack of empty space required to place its support as discussed in the following section.

2 Uniform placement of moving arrows

To uniformly place moving arrows over the domain, our algorithm successively fills each time step with as many arrows as possible such that a minimal "separating" distance d_{sep} is respected between arrows. The d_{sep} parameter controls the tradeoff between the competing objectives of avoiding cluttering and covering the whole domain.

Animation frames are populated with arrows by the Algorithm 1, which consists in three stages.

- **S1.** The first stage fills the first time step with evenly-spaced arrows (see Section 2.1 and the red arrows in Figure 3).
- **S2.** The second stage iterates over the time steps. First, the arrows that can be propagated from the

previous time step are inserted into the current one (see Section 2.2 and the orange arrows in Figure 3). Second, the current time step is completed with new arrows inserted in the previous stage (see the red arrows in Figure 3, bottom).

- **S3.** The third stage iterates from the last time step to the first one, and advances arrow's birth when possible, thus increasing arrow's lifetime (see the "backward" paragraph in Section 2.2 and the green arrows in Figure 3).

Algorithm 1: PlaceMovingArrows

Output: arrows // set of arrows
Data : tmax // max vector field time step

```

S1 CompleteTimeStepWithArrows (0, arrows);
for time  $\leftarrow$  1 to tmax do
S2    $\Delta t \leftarrow$  1; // forward advection
      PropagateArrowsOneStep (time,  $\Delta t$ , arrows);
      CompleteTimeStepWithArrows (time, arrows);
for time  $\leftarrow$  tmax - 1 downTo 0 do
S3    $\Delta t \leftarrow$  -1; // backward advection
      PropagateArrowsOneStep (time,  $\Delta t$ , arrows);

```

The two next sections explain how individual animation frames are populated with new arrows (see Section 2.1) and how these arrows will be propagated to first populate the next time step (see Section 2.2).

2.1 Completing a time step with evenly-spaced arrows

Evenly distributing arrows over a 2D domain could be addressed by Lloyd's relaxation [5]. However, in the dynamic case, it is sufficient to use a faster algorithm as the distribution quality will decrease rapidly due to arrow displacements. Therefore the problem can be reduced here to the well studied placement of streamlets. We implemented a quite standard greedy approach that works as follows (see Algorithm 2): from a sufficiently dense sampling of the domain, streamlets are successively integrated

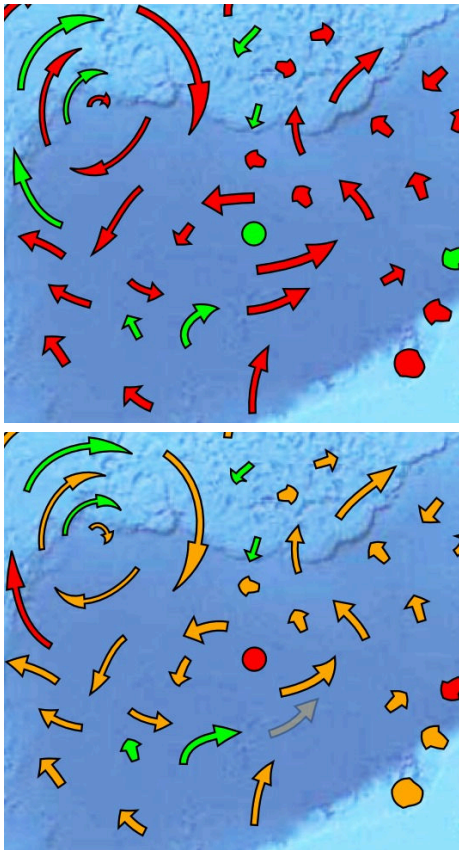


Figure 3: **First and second time step of an animation.** (Top) The empty domain is first filled with d_{seed} -separated arrows (red). (Bottom) The arrows from the previous time step are propagated into the current one (orange) and new arrows are inserted. (Right to left) Then empty spaces are filled with backward propagated arrows (green) from future time steps. The semi-transparent arrow on the right image is removed to preserve the minimal separating distance.

from these sample positions and are inserted into the representation if their distance to already placed streamlets is superior to a seeding distance d_{seed} (with $d_{seed} > d_{sep}$ as explained in Section 2.1.2).

Algorithm 2 requires evaluation of distance between a new candidate arrow and the previously placed arrows (Section 2.1.1), and a seeding strategy that determines

Algorithm 2: CompleteTimeStepWithArrows

Input : time // current time step
In/Out : arrows // set of arrows
Data : d_{seed} // seeding distance

Fill a vector `seedPositions` with domain sampling positions;
foreach position \in `seedPositions` **do**
 newArrow \leftarrow *CreateArrow* (position, time);
 if *Distance* (newArrow, arrows, time) $>$ d_{seed}
 then
 | arrows.*Insert* (newArrow);

where to place candidate arrows (Section 2.1.2) and when to stop trying.

2.1.1 Evaluating the distance between arrows

The distance between arrows is approximated by the minimal distance between their streamlets. In our implementation, the distance from any point of the domain to the existing arrows is stored in a discretized distance map, which is filled with a fast marching algorithm. The distance requests are then fast to process since they only require accessing the distance map at the requested locations. The distance map resolution is defined with respect to d_{sep} as illustrated in Figure 6. This approach also remains efficient in the presence of adaptive arrow density (see Section 3.2).

2.1.2 Choosing where to seed the arrows

The purpose of the seeding strategy is to cover the whole domain with arrows as close as possible to the maximum authorized density. A simple solution is to create a shuffled list of positions in a Cartesian grid, and always select the next position in the list.

In Algorithm 2 the arrow density is controlled by the parameter d_{seed} , which would be equal to d_{sep} for static representations. Since arrows will be propagated to the next step (see Section 2.2), we introduce a security distance by taking $d_{seed} > d_{sep}$. This way, individual arrows have a higher probability to be propagated more steps ahead before their distance to surrounding arrows falls under the d_{sep} threshold. We observe that setting $d_{seed} =$

$2d_{sep}$ leads to satisfying results. The ratio d_{seed}/d_{sep} manages the tradeoff between the arrows life-time and the uniform domain coverage.

2.2 Arrow propagation to the next step

For each time step t , new arrows are iteratively introduced by Algorithm 2 but first, it is tested if it is possible to increase the life span of the arrows that exist at time step $t - 1$ (forward) or $t + 1$ (backward) with Algorithm 3.

During the forward stage, arrows evolve through advection (of its handle point) and may enter into conflict with one another. Therefore, some arrows are to be deleted and the choice is made by a greedy approach: all arrows alive at time $t - 1$ are taken in a heuristic order and propagated one by one to time t . If an arrows conflicts with already propagated arrows, it is discarded. This method is fast and allows to favor arrows by defining priority criteria. In practice, sorting arrows by decreasing streamlet length in screen space allows to keep long arrows as much as possible, and therefore minimizes the popping artifacts. The benefits of this strategy are illustrated in Figure 4.

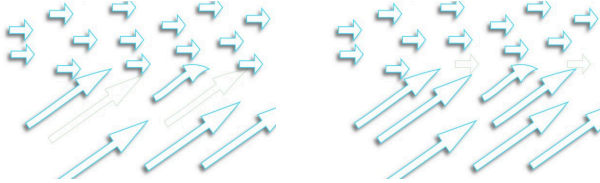


Figure 4: **Arrow propagation with priority.** Giving priority to short arrows (left) would kill long arrows and therefore waste a lot of space and create noticeable popping artifacts. Our strategy to propagate long arrows first (right) resolves this issue by removing small arrows.

The backward stage is similar to the forward one, except that it does not try to introduce new arrows because the empty spaces have already been filled during the forward stage. However, since the arrows have been seeded at least at the d_{seed} distance to the surrounding ones, some of these arrows might find enough place to propagate back until they reach the d_{sep} separating distance. These cases are illustrated with the green arrows on Figure 3.

To obtain progressive appearance and disappearance of the arrows along the borders of the frame, it is necessary

to extend the domain with a buffer zone where the vector field is extrapolated. The size of this hidden buffer zone is related to half the maximal length of the arrows. All the operations are performed on this extended domain.

Algorithm 3: *PropagateArrowsOneStep*

```

Input : time           // current time step
Input :  $\Delta t$         // 1 is forward, -1 is backward
In/Out : arrows       // set of arrows
Data : vf             // velocity vector field
Data :  $d_{sep}$          // separating distance

prevTime  $\leftarrow$  time -  $\Delta t$ ;
growingCandidates  $\leftarrow$   $\emptyset$ ;           // empty list
foreach arrow  $\in$  arrows do
  if arrow.IsAlive(prevTime)
    and not arrow.IsAlive(time) then
       $\lfloor$  growingCandidates.Insert(arrow);
SortByPriority(growingCandidates);
foreach arrow  $\in$  growingCandidates do
  arrow.PropagateTo(time);
  if Distance(arrow, arrows, time) <  $d_{sep}$  then
     $\lfloor$  arrow.RemoveTimeStep(time);

```

3 Introducing an adaptive density of arrows

The algorithm defined in the previous section computes a sparse set of moving arrows that keeps a uniform density over time. However, it is often interesting to adapt the density of arrows to the local features of the flow (see Figure 5). This can be done by introducing a density map which measures the scale at which the phenomenon needs to be captured.

The density map is a scalar field that gives the local "zoom factor" of the field: in particular, a value of 1 means that the arrows are generated as with the previous algorithm, and in general a value *scale* means that the algorithm generates arrows as such that a close-up (of factor *scale*) of the region gives the same appearance as the previous algorithm. We first discuss possible ways to automatically define the density map, then present the modifications of the algorithm required to add this feature.

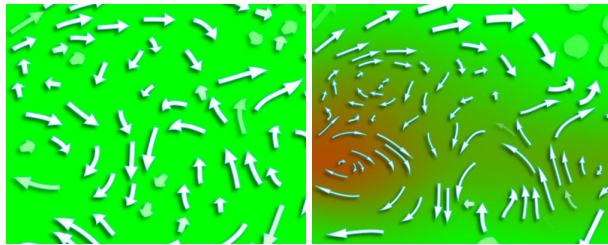


Figure 5: **Uniform vs. adaptive density of arrows.** (Left) A uniform density of arrows will fail at revealing the turbulent areas if the separating distance is too high – or would overpopulate the domain with a small separating distance. (Right) Adapting the separating distance to a density field (green to red background) will adapt the number of arrows necessary for depicting the details of the flow.

3.1 Choice of a density map

The density map can be used to add more details where the flow has more variations. It is therefore natural to estimate it by a differential quantity derived from the vector field. In our experiments, we estimate it by the Frobenius norm of the Jacobian matrix i.e. $\sqrt{(\partial v_x/\partial x)^2 + (\partial v_x/\partial y)^2 + (\partial v_y/\partial x)^2 + (\partial v_y/\partial y)^2}$, at the position x, y with the velocity field $v(v_x, v_y)$. This quantity allows to focus on flow features as it is correlated with both divergence and curl. However, other density maps may be more appropriate in particular cases, such as the velocity or vorticity magnitude as discussed in Schlemmer *et al.*'s work [16]. Regardless the way for estimating the density map, its values are set in the range $[1, scale_{max}]$ so that some regions can exhibit an arrow placement $scale_{max}$ times denser than others. Most frequently we set $scale_{max} \leq 10$.

3.2 Adapting the algorithm to handle a density map

The only thing we need to change in Algorithm 1 in order to take the density map into account is the distance between arrows: the new distance is just scaled by the density. When comparing this new distance with d_{sep} and d_{seed} in the algorithm, the spacing between neighboring arrows becomes proportional to the inverse of the density.

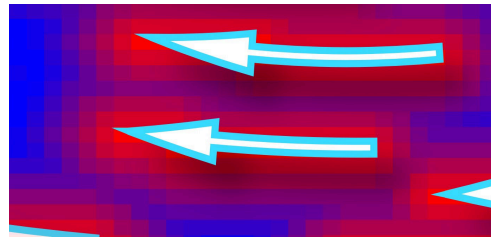


Figure 6: **Resolution of the distance map.** Setting the pixel width to be a quarter of d_{sep} achieves a good trade-off between the update time of the distance map and its accuracy.

The new distance is therefore the weighed distance with respect to the density. A weighed distance is formally defined in [14], and is commonly used in images applications such as image segmentation [15].

As introduced in section 2.1.1 the distance of each point to previously placed arrows is stored in a distance map. The distance map is now considered as a weighted graph where each pixel is connected to its 8 closest neighbors and the weights corresponds to the edge length (1 or $\sqrt{2}$ for diagonals) times the density map evaluated at this position. The distance between two points (pixels) is then the cost of the shortest path in this graph.

Evaluating the distance to a new arrow from all previously placed arrows only requires reading the distance map at the sampling points of the new arrow.

Updating the distance map is a bit more difficult. It first requires to be initialized to ∞ , then for each new inserted arrow, the distance field is updated by a n -seed Dijkstra algorithm (placing one seed for each point touched by a rasterization of the arrow's streamlet). The very specific nature of the graph (bounded weights between 1 and $\sqrt{2} \times scale_{max}$) makes it possible to use simpler and more efficient algorithms such as [20].

As illustrated in Figure 6, setting the pixel size of the distance map to be a quarter of the separating distance is enough to discretize the distance map.

4 Rendering arrows

To draw all the arrows, the rendering algorithm determines the mapping of the arrow glyph onto the screen

(Section 4.1), morphs the arrows to new symbols when the flow magnitude becomes too low (Section 4.3), and adds transparency to reduce the popping artifacts (Section 4.2).

4.1 Arrow mapping

The glyphs are centered on the handle point of the arrows. Our algorithm precomputes these positions at each time step. For frames in-between time steps, the smoothness of arrow displacements is ensured by a cubic hermite interpolation (in time) of the handle point. From the interpolated handle points, a streamlet is integrated and a support is computed by thickening it. For adaptive density, the thickness of the support is divided by the local density to avoid occlusion with closest arrows.

Notice that it would be possible to integrate (in time) the handle point position with Runge–Kutta, but we prefer to interpolate them due to the asymmetry of the integration scheme.

4.2 Fade-in and fade-out of arrows

Popping effects come generally from the insertion and deletion of arrows. To attenuate this effect, the arrows are rendered with an opacity coefficient that smoothly decreases near birth and death time steps. When the number of rendering frames is higher than the simulation time step, all arrows born (resp. died) at time t have the same transparency coefficient, leading to a visual artifact. This can be solved by adding a random delay (shorter than a time step) before applying the fading effect.

4.3 Arrow morphing

Arrows are standard symbols to represent flows, however it may become misleading where the flow magnitude becomes too low. Inspired by meteorologists, we decided to draw discs to indicate calm regions.



Figure 7: **Glyph morphing sequence.** The arrows are smoothly morphed to discs in regions of low velocity.

To render dynamic flows, a smooth transition between arrows and discs (see Figure 7) allows the avoidance of distracting symbol switches. The glyph to use is determined by the streamlet length over glyph thickness ratio: this ensures that arrows length is always greater than arrows thickness.

The support of the glyph also requires a special treatment when its streamlet is shorter than the arrows thickness: the support is no longer obtained by thickening the streamlet, but by drawing a square centered on the handle point and oriented by the vector between the streamlet extremities.

Using a symbol with rotational invariance (disc) prevents the user from being distracted by the frequent rotations when the flow velocity is almost null.

5 Results

Real datas

We have tested our method on a water flow (gulf of Mexico), two winds data (Europe’s Storm in 1999 and Ocean winds) and a simulated velocity jet. Snapshots of the resulting animations can be found in Figure 8, and videos in the accompanying material. We tried to reflect the capabilities of our method. The image of the 1999 Europe’s storm shows coarse structure of the cyclone, the arrows morph smoothly between zones with high and low velocity. The flow in the Gulf of Mexico and the velocity jet have well established currents as well as plenty of small turbulencies. To avoid overpopulation with plenty of small arrows, we used high variation of underlying densities and thus we capture tiny details while the well established flows are shown with longer and thicker arrows.

Synthetic data

The real data come from simulated or acquired flow fields, and they have a limited divergence (due to the limited compressibility of the fluids). As a consequence, the behavior of our method in extreme cases of divergence can not be observed on such data, so we rely on synthetic fields to evaluate the limitation of our method. Even in such cases, our algorithm is able to evenly distribute arrows in the field, as illustrated in Figure 9.

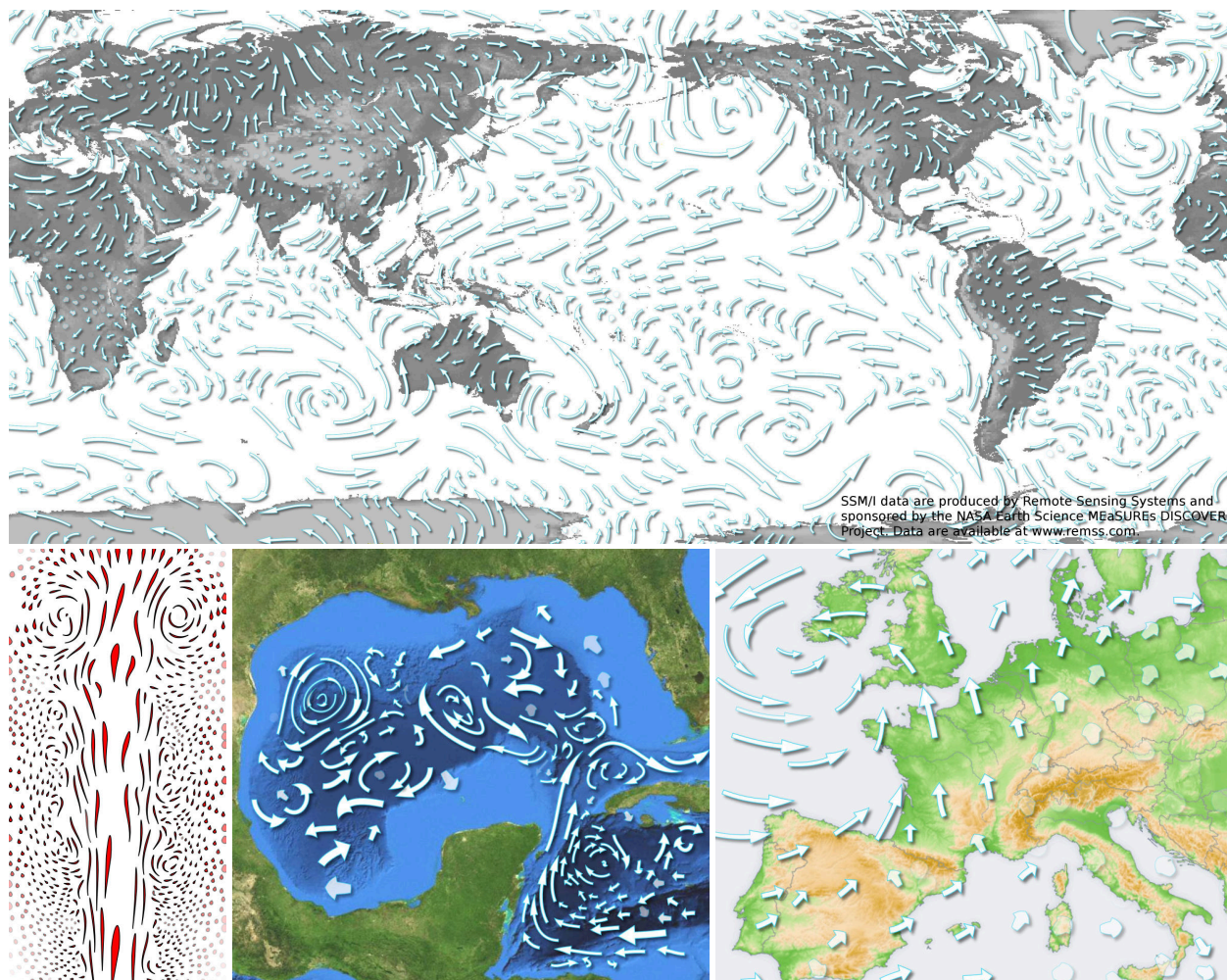


Figure 8: Frames of animated arrow plots of different datasets.

Timings

The table below shows the time necessary to pre-compute various animations used in this paper.

| Dataset | processing (seconds) | frames | vector field resolution |
|-------------------|----------------------|--------|-------------------------|
| Storm dec 1999 | 4.1 | 48 | 385×325 |
| Ocean winds, 1987 | 56.7 | 40 | 1440×628 |
| Velocity jet | 45.5 | 500 | 128×256 |
| Gulf of Mexico | 19.25 | 183 | 352×320 |
| Dipole | 0.29 | 40 | 64×64 |

6 Discussion

6.1 Accuracy and resolution

As stated in the introduction, a sparse set of arrows cannot represent all details of the flow. To avoid high frequencies from perturbing the algorithm, it is convenient to filter the data. A convolution with a Gaussian having the minimal arrow size as standard deviation is sufficient to maintain undistorted arrows. As a consequence, sampling the flow with a resolution higher than one pixel corresponding to

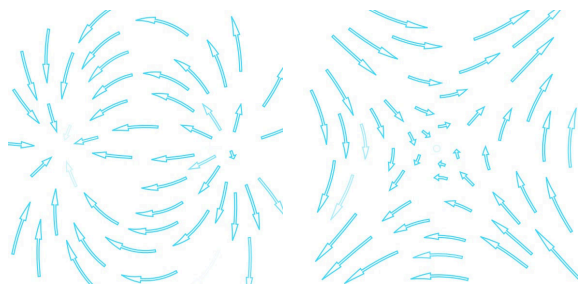


Figure 9: **Stress tests** with synthetic vector fields having extreme divergence.

the minimal arrow size is not useful for the algorithm.

The time resolution of the flow do not impact the result quality as long as the distance between neighbor arrows at each time step is a fair approximation of this distance between two consecutive time steps. In practice, the displacement of an arrow handle point between two frames should not be greater than the arrow length.

6.2 Streamlets vs. pathlets

As mentioned in the section 1, the arrow shapes follow streamlines, while their centers move on pathlines. This means that the arrows in the visualization can point into different directions than the arrows move to. It may seem more natural to use pathlets for arrow support. On the positive side, arrows move into the direction they point to. On the negative side, pathlets show history of the underlying field and not the current state of the flow.

In the case of steady flows or short arrows, using streamlet or pathlet is almost equivalent. In other cases, using pathlines creates confusing effects as illustrated in Figure 10. Thus, we choose streamlets to carry arrows in our animations.

6.3 Comparison

As stated in the introduction, our objective is to produce pleasing and easy to understand animations to represent a 2D flow field. As illustrated here with flow textures and fixed position arrows, previous methods only partially satisfy these constraints.

Fixed position arrows (see top middle image of figure 11) can overlap and their alignment can be distract-

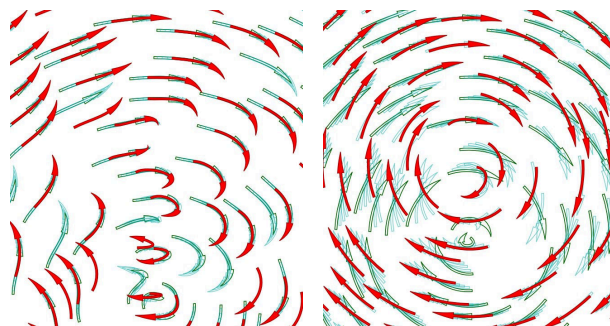


Figure 10: **Arrows as pathlets or streamlets**. Both images represent the same vortex moving from bottom to top. The arrows are bended along (Left) pathlets and (Right) streamlets. Green hollow arrows show the field at timestep $t - 4$, red arrows at timestep t . Visually, the animation of streamlet-based arrows is more intuitive than pathlet-based one.

ing. It is a fair solution to visualize fields in a realtime context, or for vector fields where advection would be meaningless such as electromagnetic fields that are not flows. However, our algorithm offers a more uniform coverage and a natural displacement of the arrows.

Flow texture methods cannot represent flow orientation and its magnitude without deteriorating global rendering quality. It is also difficult to combine it with other sources of information such as a detailed background as illustrated in figure 11.

Moreover, texture based methods may suffer from bad rendering device (gamma or resolution) and video compression. Our method does not provide an as accurate representation of the flow details, but does not suffer from these drawbacks.

Acknowledgments

We wish to thank MeteoSwiss for the dataset of the winds over Europe, the Center for Ocean-Atmospheric Prediction Studies (COAPS) for the dataset of the ocean currents in the Gulf of Mexico, Remote Sensing System for the dataset of the winds around the world and Christoph Garth at UCDAVIS for the dataset of the high velocity jet.

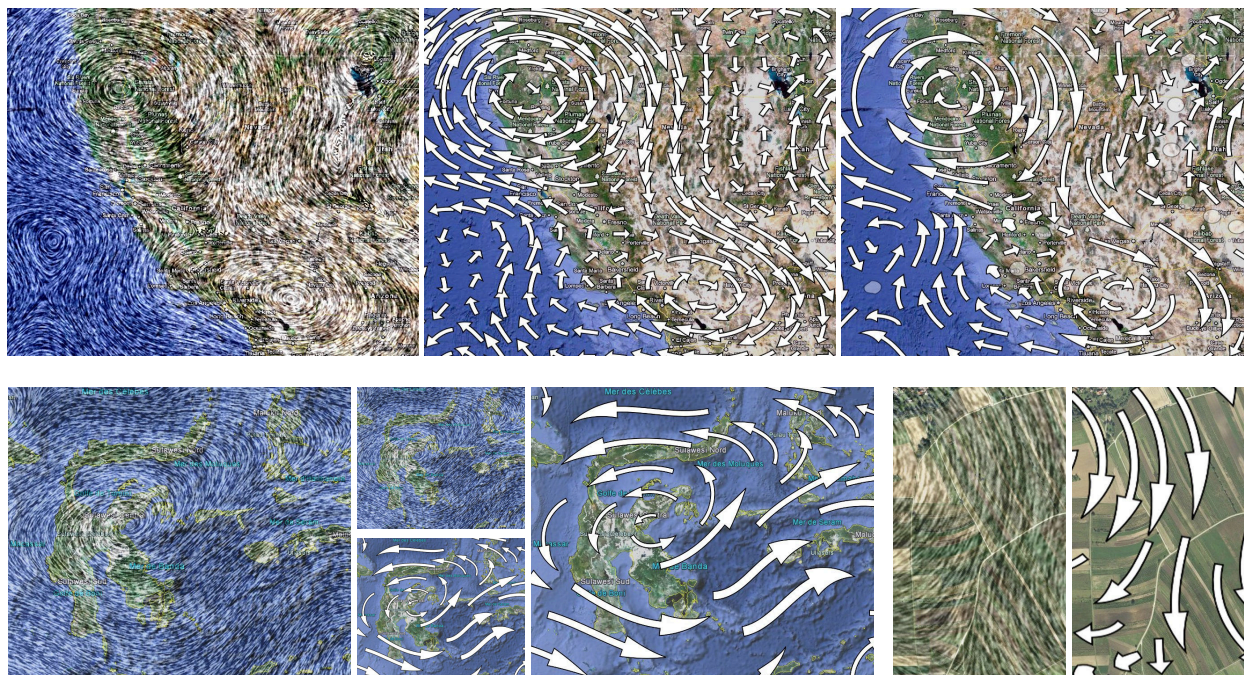


Figure 11: **Visualizing a flow with a background image.** Top row, left-to-right: Flow texture, bent arrows on a uniform grid and with our method. Compared to flow textures, arrows temporary occlude the background but do not deteriorate it. Bottom row, left images: arrow plots are less sensitive to the display resolution; two right images: certain anisotropic textures such as aerial photographies interfere with flow textures.

Conclusion

We have developed a representation of dynamic vector fields by moving arrows. Despite the intuition that the divergence will always create incessant popping effects, we were able to produce convincing videos. It was achieved by coupling arrow generation and rendering strategies. Moreover, the extreme cases of divergence (source, sink) that first come to mind as counter-examples that could challenge our method are correctly handled by our algorithm and are not likely to occur often in real flows.

References

- [1] D. Bauer, R. Peikert, M. Sato, and M. Sick. A case study in selective visualization of unsteady 3D flow. In *Proceedings of the conference on Visualization'02*, pages 525–528. IEEE Computer Society, 2002.
- [2] D. Dovey. Vector plots for irregular grids. In *Proceedings of the 6th conference on Visualization'95*, page 248. IEEE Computer Society, 1995.
- [3] A. Hausner. Animated visualization of time-varying 2D flows using error diffusion. In *Proceedings of the working conference on Advanced visual interfaces*, pages 436–439. ACM, 2006.
- [4] A. Helgeland and T. Elboth. High-quality and interactive animations of 3d time-varying vector fields. *Visualization and Computer Graphics, IEEE Transactions on*, 12(6):1535–1546, 2006.
- [5] S. Hiller, H. Hellwig, and O. Deussen. Beyond stippling—methods for distributing objects on the

- plane. In *Computer Graphics Forum*, volume 22, pages 515–522. Wiley Online Library, 2003.
- [6] B. Jobard and W. Lefer. Creating evenly-spaced streamlines of arbitrary density. *Visualization in Scientific Computing*, 97:43–56, 1997.
- [7] B. Jobard and W. Lefer. Unsteady flow visualization by animating Evenly-Spaced streamlines. In *Computer Graphics Forum*, volume 19, pages 31–39. Wiley Online Library, 2000.
- [8] R. V Klassen and S. J Harrington. Shadowed hedgehogs: A technique for visualizing 2D slices of 3D vector fields. In *Proceedings of the 2nd conference on Visualization'91*, pages 148–153. IEEE Computer Society Press, 1991.
- [9] R. S Laramee, H. Hauser, H. Doleisch, B. Vrolijk, F. H Post, and D. Weiskopf. The state of the art in flow visualization: Dense and Texture-Based techniques. In *Computer Graphics Forum*, volume 23, pages 203–221. Wiley Online Library, 2004.
- [10] Z. Liu, R. Moorhead, and J. Groner. An advanced evenly-spaced streamline placement algorithm. *IEEE Transactions on Visualization and Computer Graphics*, pages 965–972, 2006.
- [11] A. McKenzie, S. V Lombeyda, and M. Desbrun. Vector field analysis and visualization through variational clustering. In *Eurographics-IEEE VGTC Symposium on Visualization*, volume 2005, 2005.
- [12] T. McLoughlin, R. S Laramee, R. Peikert, F. H Post, and M. Chen. Over two decades of integration based, geometric flow visualization. In *Computer Graphics Forum*, volume 29, pages 1807–1829. Wiley Online Library, 2010.
- [13] A. Mebarki, P. Alliez, and O. Devillers. Farthest point seeding for efficient placement of streamlines. In *Visualization, 2005. VIS 05. IEEE*, pages 479–486. IEEE, 2005.
- [14] F. Mémoli and G. Sapiro. Fast computation of weighted distance functions and geodesics on implicit hyper-surfaces. *Journal of Computational Physics*, 173(2):730–764, 2001.
- [15] A. Protiere and G. Sapiro. Interactive image segmentation via adaptive weighted distances. *Image Processing, IEEE Transactions on*, 16(4):1046–1057, 2007.
- [16] M. Schlemmer, I. Hotz, B. Hamann, F. Morr, and H. Hagen. Priority streamlines: A context-based visualization of flow fields. In *Eurographics/IEEE-VGTC Symposium on Visualization*, page 227–234. Citeseer, 2007.
- [17] A. Telea and J. J Van Wijk. Simplified representation of vector fields. In *Visualization'99. Proceedings*, pages 35–507. IEEE, 1999.
- [18] G. Turk and D. Banks. Image-guided streamline placement. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 453–460. ACM, 1996.
- [19] J. J Van Wijk. Image based flow visualization. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 745–754. ACM, 2002.
- [20] L. Yatziv, A. Bartesaghi, and G. Sapiro. O(N) implementation of the fast marching algorithm. *Journal of Computational Physics*, 212(2):393–399, 2006.